TUCS

Jesús Carabaño Bravo

A Compiler Approach to Map Algebra for Raster Spatial Modeling

Turku Centre for Computer Science

TUCS Dissertations
No 243, August 2019

A Compiler Approach to Map Algebra for Raster Spatial Modeling

Jesús Carabaño Bravo

To be presented, with the permission of the Faculty of Science and Engineering of Åbo Akademi University, for public criticism in Auditorium XX, the Agora building, on August 30th, 2019, at 13:00.

Åbo Akademi University
Faculty of Science and Engineering
Agora, Vattenborgsvägen 3
20500 Åbo, Finland

2019

Supervisors

Jan Westerholm
Faculty of Science and Engineering
Åbo Akademi University
Agora, Vattenborgsvägen 3, 20500 Åbo
Finland

Mats Apsnäs
Faculty of Science and Engineering
Åbo Akademi University
Agora, Vattenborgsvägen 3, 20500 Åbo
Finland

Reviewers

Oliver Schmitz
Department of Geosciences
University of Utrecht
PO Box 80.115, 3508 TC Utrecht
The Netherlands

William Spataro
Department of Mathematics and Computer Science
University of Calabria
Ponte Bucci, Cubo 30B Arcavacata di Rende, Cosenza
Italy

Opponent

Oliver Schmitz
Department of Geosciences
University of Utrecht
PO Box 80.115, 3508 TC Utrecht
The Netherlands

ISBN 978-952-12-3853-6
ISSN 1239-1883

Abstract

Modeling and simulation enables the study of spatial phenomena that are otherwise impossible to reproduce in the physical world. On the one hand, digital models replicate the shape and state of the Earth with bits and bytes that we can store and manipulate. On the other, computer models turn mathematical equations into instructions that computers can run to simulate events of interest. Together, they provide a digital laboratory where to virtually isolate and experiment with spatial phenomena.

Recently, advances in remote sensing technologies have brought unprecedented volumes of spatio-temporal data. Simultaneously, computer architectures have developed into parallel and heterogeneous designs. More data enables better models, but also entails more calculations, which in turn requires faster computers. Parallel computers promise higher performance, but their complex programming impacts both productivity and portability. This raises a conflict between the execution speed, ease of development and portability of computer models. However, these three qualities are necessary for an efficient practice of modeling and simulation.

In this thesis we propose a compiler approach to map algebra that provides a productive, performant and portable environment where to model spatial phenomena and simulate their dynamics. Modelers write sequential scripts in the map algebra formalism, a language for raster spatial analysis. Then the compiler parses their scripts to generate parallel codes that run efficiently on modern heterogeneous architectures. Moreover, scripts are written once and automatically translated to the target computer architecture, attaining portability.

To test our hypothesis, we build a prototype compiler and evaluate typical map algebra workloads. Our experiments produced three clear outcomes: (1) the compiler approach can handle large volumes of data even when they exceed the computer memory, (2) the performance is competitive as it provides large speed-ups over interpreted map algebras, and (3) the compilation process is fully transparent to modelers and therefore user-friendly. In conclusion, a map algebra compiler meets the necessary qualities for a practical modeling and simulation of spatial phenomena with raster data, and as a result it becomes a valuable tool for modelers.

Abstrakt

Modellering och simulering möjliggör studier av rumsliga fenomen som annars inte kan reproduceras i den fysiska världen. Å ena sidan representerar digitala modeller jordens form och tillstånd med bitar och byte som vi kan lagra och manipulera. Å andra sidan gör datormodellerna matematiska ekvationer till instruktioner som datorer kan utföra för att simulera händelser av intresse. Tillsammans skapar de ett digitalt laboratorium där man kan virtuellt isolera och experimentera med rumsliga fenomen.

Nyligen har framsteg inom fjärranalysteknik gett upphov till mångdubblade volymer av spatio-temporala data. Samtidigt har datorarkitekturer utvecklats till parallella och heterogena system. Mer data möjliggör bättre modeller, men medför också fler beräkningar, vilket i sin tur kräver snabbare datorer. Parallella datorer utlovar högre prestanda, men deras komplexa programmering påverkar både produktivitet och portabilitet. Detta ger upphov till en konflikt mellan exekveringshastigheten, enkel utveckling av och portabiliteten hos datormodeller. Dessa tre egenskaper är emellertid nödvändiga för en effektiv användning av modellering och simulering.

I denna avhandling föreslår vi en kompilatormetod för kartalgebra som ger en produktiv och portabel programutvecklingsmiljö med hög prestanda där man kan modellera rumsliga fenomen och simulera deras dynamik. Modellerare skriver sekventiella skript i kartalgebraformalism, ett språk för rumslig analys med raster. Kompilatorn analyserar sedan skripten för att generera parallella koder som löper effektivt på moderna heterogena arkitekturer. Noteras bör att skripten skrivs en gång och översätts automatiskt till måldatorarkitekturen, genom vilket man uppnår portabilitet.

För att testa vår hypotes bygger vi en prototypkompilator och utvärderar typiska kartalgebraiska arbetsmängder. Våra experiment gav tre klara utfall: (1) Kompilatormetoden kan hantera stora datamängder även när dessa överstiger datorminnet, (2) prestandan är konkurrenskraftig eftersom kompilatorn ger stor prestandaökning jämfört med tolkad kartalgebra och (3) kompileringsprocessen är enkel för modellerare och därför användarvänlig. Sammanfattningsvis uppfyller en kartalgebrakompilator de nödvändiga egenskaperna för en praktisk modellering och simulering av rumsfenomen med rasterdata, och blir följaktligen ett värdefullt verktyg för modellerare.

Acknowledgements

This could only start by acknowledging Jan Westerholm, who casually found Jesús around the university, and now the rest is history. Jan provided me with the hottest* GPUs on the market to support my master thesis, all without asking a thing, or that I thought. Back then I was a carefree exchange student and soon after, without really knowing how, I voluntarily enrolled for a not-so-carefree PhD under Jan's supervision. The PhD did not ruin my life, well almost, but I finished it thanks to the support of innumerable people, Jan foremost.

Literally a 350W-hot Nvidia GTX, which later caught fire...

I thank those who were technically involved in this thesis, from Tapani Sarjakoski's group (J Oksanen, J Kovanen, V Mäkinen...) for providing the Geographical side of this venture, to the Åbo Akademi bunch for supporting the Computer Science part: starting with Mats Aspnäs and his enlightened advices, with lots of former colleagues in the middle (V Timonen, E Yurtesen, D Eranen, F Robertsen...), and inevitably finishing —don't ask why— with Jani Sainio*.

This can't be mentorship if you always get me in trouble...

Let's not forget those colleagues involved in collateral* ways, from other PhDs (M Kamali, J Wiik, S Kanur, W Lund, S Holmbacka, H Rexha, B Byholm, T Ahmad, J Iqbal, V Popescu...), to staff (M Barash, K Rönnholm...), lecturers (J Ersfolk, S Lafond, J Lilius, M Neovius...) and fellows (A Domi, J Martini, L Nwaogo, A Morariu...). Most likely I still owe you a coffee. Well, today wine and coffee are on me.

I was possibly detrimental to your work; collateral damages of this thesis.

Without revealing their secret identities —hidden by a hacker hoodie— I shall also thank my industry colleagues, who unknowingly supported me when the PhD got me to my knees. They provided some oxygen and a break from academia, enough to stand up and give this thesis the final blow. Equally important were my plants*, or probably much more important.

Forever in debt to Silo.AI, for watering my plants as well as my career.

To my dear international friends, who helped me balance work in life and keep it together. I should not name one of you without naming you all, but printing such long thesis would get expensive. In particular to those in the ESN family: it's true, "once erasmus, always erasmus". Do pay me a visit in the future, the PhD is over, I am finally free* and got time for you.

Be quick before work, family or adulthood imprison me!

Immense gratitude to my actual family back in Spain, who don't see me much and wondered all these years whether I would return. Sorry, I will stay abroad some time more, but I will visit more often from now on. To them, who missed me a lot, yet never turned their backs, I am grateful the most*. If today I made it here, it is surely because of their support.

Their oil, wine and ham deliveries might have influenced this opinion.

Last but not least, to those I didn't mention.
You know who you are, you were vital.
Muchas Gràcias.

Åbo, August 2019
Jesús Carabaño Bravo

List of original publications

1. Carabaño, J., Sarjakoski, T., and Westerholm, J. (2015). Efficient implementation of a fast viewshed algorithm on simd architectures. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Disturbed, and Network-Based Processing*, pages 199–202. IEEE

2. Carabaño, J. and Westerholm, J. (2017). From python scripting to parallel spatial modeling: Cellular automata simulations of land use, hydrology and pest dynamics. In *Proceedings of the 25th Euromicro International Conference on Parallel, Disturbed, and Network-Based Processing*, pages 511–518. IEEE

3. Carabaño, J., Westerholm, J., and Sarjakoski, T. (2018). A compiler approach to map algebra: automatic parallelization, locality optimization, and gpu acceleration of raster spatial analysis. *GeoInformatica*, 22(2):211–235

4. Carabaño, J. and Westerholm, J. (2019). A compiler and runtime approach to parallel spatial modeling. Technical Report 1200, TUCS. ISSN 1239-1891, No 1203

List of abbreviations

AES	Advanced Encryption Standard
AGU	Address Generator Unit
ALU	Arithmetic Logic Unit
AST	Abstract Syntax Tree
BMI	Bit Manipulation Instruction
CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CU	Control Unit
DAG	Directed Acyclic Graph
DEM	Digital Elevation Model
DSL	Domain-Specific Language
DSP	Digital Signal Processor
FMA	Fused Multiply-Add
GPL	General-Purpose Language
GPU	Graphics Processing Unit
HBM	High-Bandwidth Memory
HDD	Hard Disk Drive
I/O	Input/Output
ILP	Instruction-Level Parallelism
ISA	Instruction Set Architecture
ISP	Image Signal Processor
LOS	Line-of-Sight
LRU	Least Recently Used
LSU	Load Store Unit
NAS	Network Attached Storage

NINE	Non-Inclusive Non-Exclusive
PPP	Productivity Performance Portability
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SAN	Storage Area Network
SCSI	Small Computer System Interface
SFC	Space Filling Curve
SHA	Secure Hash Algorithm
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multi-Threading
SQuaRE	Software product Quality Requirements and Evaluation
SRTM	Shuttle Radar Topography Mission
SSA	Static-Single Assignment
SSD	Solid-State Drive
TCP	Transmission Control Protocol
UAV	Unmanned Aerial Vehicle
USACE	United States Army Corps of Engineers
VPU	Vision Compute Unit

Contents

Chapter 1

Introduction

Ordinary minds discuss people,
interesting minds discuss events,
but great minds... those discuss ideas,
and today we are here to discuss theories and ideas.

...paraphrasing ~ **Eleanor Roosevelt**

1.1 Context and Motivation

The world is a complicated place. Since we are born, we make sense of it by observing, experiencing and learning. We conceptualize what we perceive to understand the world out there. In our minds, we form ideas that model our interpretation of the environment. We exercise those ideas to predict our surroundings, and when they fail, we learn. We model the world to navigate it, and at the same time we make of those models our world.

1.1.1 Computer Modeling and Simulation

*What are **models**?* If theories are abstractions that reduce phenomena to their essentials, then models puts such abstractions into forms that we can manipulate. Mental models are the most common and valuable models we have. Our mind builds them to anticipate events and plan accordingly. Thus, as we cross the street we know it is safe, because our mind predicts the traffic. Mathematical, logical and physical models are other examples of models. They are less intuitive, more formal and exist outside our minds. We use them to predict the weather, design safe planes or explain the financial crisis. With the invention of computers we have acquired new types of models, the digital and computer models, on which this thesis focuses.

Model is a very overloaded word. Here we discuss the physical model, *"a three-dimensional representation of a person, thing or structure, typically on a smaller scale than the original"* , the mathematical model, *"a simplified description, i.e. a mathematical one, of a system or process, to assist calculations and predictions"* , and their digital counterparts. Henceforth we designate **digital model** as the digital version of the physical model, *"a digital representation of a person, thing or structure"* , and **computer model** as the digital version of the mathematical model, *"a digital description of a system or process that can be simulated by a computer to assist calculations and predictions"*. In cases where the semantic context determines the meaning, we might use the substantive *model* alone.

Computers are revolutionizing the way we work. They enable us to store, query, modify, visualize, analyze and simulate about anything we can digitalize, and they do so drastically faster than anything known to man before. For instance, it is now more convenient to design the prototype of a vehicle as a digital model, than it is to actually build and rebuild consecutive physical models of the prototype. Likewise, it is more efficient to simulate the aerodynamics of such vehicle with a computer model, than it is to test the physical prototype in a wind tunnel or in the actual roads. Besides engineering, other fields that now rely on computers are finance (e.g. risk modeling), meteorology (e.g. weather forecast), energy (e.g. consumption analysis) and medicine (e.g. drug discovery), to name a few.

Computers are transforming the way we do research, too. Recall the scientific method (Figure 1.1). Scientists observe the environment, question the reasons behind some phenomena, formulate hypothesis that explain their assumptions, perform experiments to prove or refute their hypothesis, analyze the experimental results, and finally draw some conclusions. Of course, research is rarely that straightforward. Often some obstacle stops the progress and forces bifurcations and iterations on the method. This is where computers stand out, as they help us bypass long-standing *research obstacles*, such as the observation of unmeasurable events (i.e. nano-structures formation, protein folding); the experimentation with uncontrollable processes (i.e. plasma physics, nuclear reactions); or the analysis of unmanageable volumes of data (i.e. human genome, satellite imagery).

A crucial obstacle that computers enable us to overcome is that of our *human limits in the experimentation.* Physical experiments are necessary reality checks: they synchronize our theories with the real world. However, sometimes experimenting with the actual phenomena is not a viable option. For example, studying the lifecycle of galaxies and stars by observing the night sky too slow; testing aerospatial materials and jet engines with throwaway rockets is too expensive; developing nuclear weapons by trial and error in the laboratory is too dangerous; or experimenting with living human brains is too risky and possibly unethical.

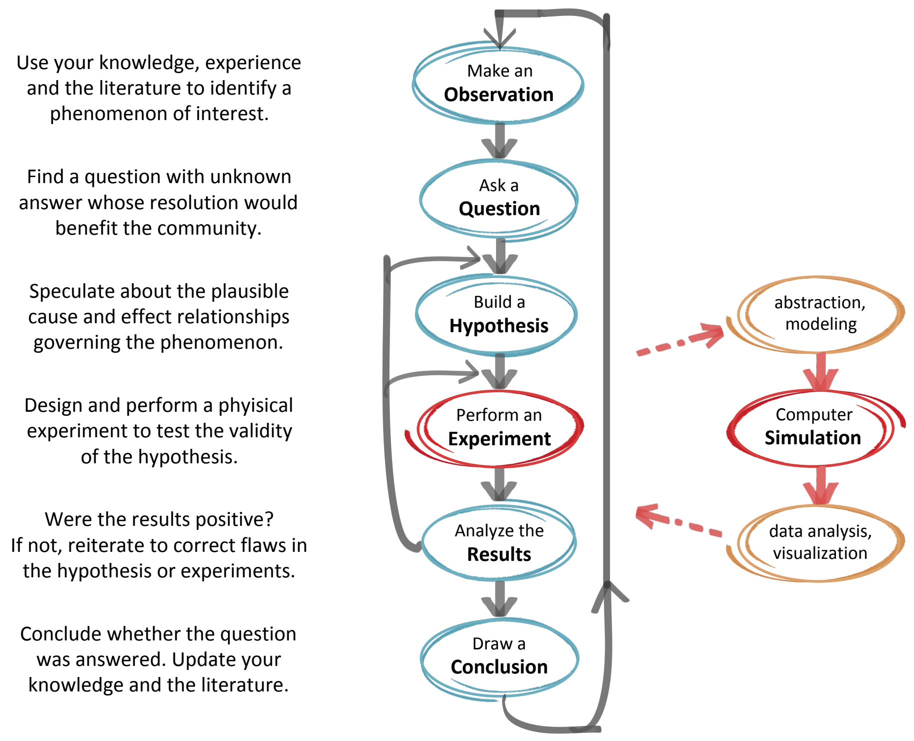

Use your knowledge, experience and the literature to identify a phenomenon of interest.
Find a question with unknown answer whose resolution would benefit the community.
Speculate about the plausible cause and effect relationships governing the phenomenon.
Design and perform a phyisical experiment to test the validity of the hypothesis.
Were the results positive? If not, reiterate to correct flaws in the hypothesis or experiments.
Conclude whether the question was answered. Update your knowledge and the literature.
Make an Observation
Ask a Question
Build a Hypothesis
Perform an Experiment
Analyze the Results
Draw a Conclusion
abstraction, modeling
Computer Simulation
data analysis, visualization

Figure 1.1: The scientific method: an iterative process for discovery. Computer simulations pose an alternative where physical experiments fail.

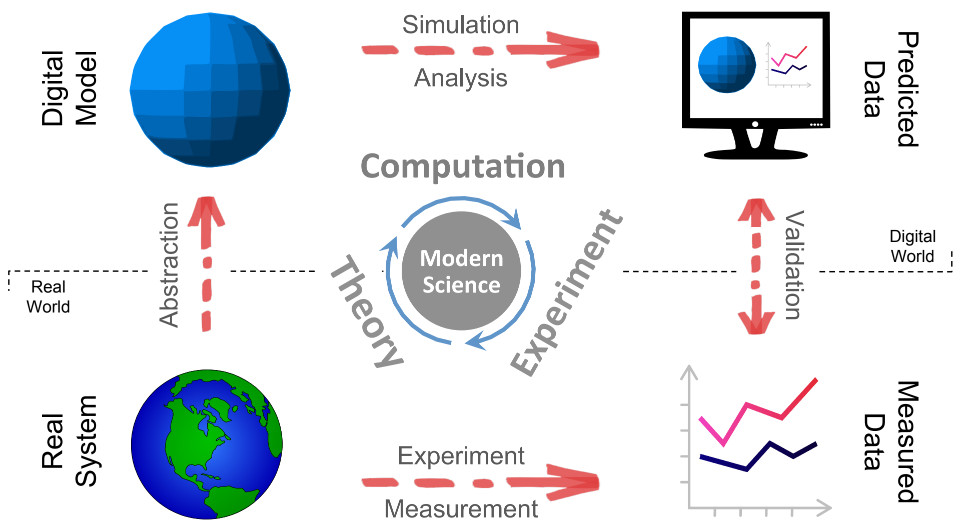


Figure 1.2: Modeling and simulation is enabled by computation, a pillar of moder science together with theory and experimentation [23].

Today, computers enable the study of phenomena that are otherwise impractical to reproduce in the physical world. Computers have become the digital laboratory where to perform rapid, cost-efficient and safe simulations. As a result, **modeling and simulation** has turned into a fundamental tool for the scientific method, and it is acknowledged as a third pillar of modern science, together with theory and experimentation. Computers not only allow us to test theories beyond our physical limitations, but are also useful to narrow down the experimental configurations of most interest, and serve as a validation tool for the experimental methodologies (Figure 1.2).

This thesis revolves around the modeling and simulation of large spatial phenomena. Spatial phenomena are of our interest because of two trends: the advances in data collection technologies, which have outpaced our capacity to process new spatial data, and the developments in computer architectures, which cannot deliver more performance without laborious parallelism. The next subsections set the context and motivate the reasons behind the research that makes up this thesis.

1.1.2 Modeling Large Spatial Phenomena

Space is an important factor in many natural and artificial events. Geography, hydrography, ecology, meteorology and other so-called Earth sciences all deal with natural events occurring across space. Commerce, transport, communications, urban planning and other fields of human origin also tackle events characterized by spatial relationships. Two examples are the evolution of geomorphological processes and the organization of human settlements. These and other spatial events are phenomena of great interest that science wishes to predict and explain. However, their large scale limits their physical experimentation, which in turn complicates their research.

As an example, imagine a densely populated city in an area at risk of 1000-year floods [1]. Suppose that, due to climate change and global warming, the risk intensifies to become a 10-year flood. Because of the dangers it entails for the population, soon governments call for scientists to tackle the crisis. Scientists know that, given how limited control humans have over the natural forces, little can be done beyond placing defensive hydraulic constructions. Scientists and hydraulic engineers will then question whether a levee system could prevent the flood. With the hypothesis that said system would work, they simply need to perform an experiment. Unfortunately, such a physical experiment is not that simple, if at all feasible.

How could they gather and manipulate the volumes of water that would drown a large city? How could they consistently repeat this feat as the hypothesis is refined and iterated? An approximated solution could be to replicate the event at a much smaller scale. However, even if smaller, most of

[1] A 1000-year flood risk refers to a 0.1% chance of flooding occurring in any given year

the challenges are not eliminated, just scaled down. Large volumes of water are still needed, as well as physical prototypes of the levee system. Hence the experiment becomes expensive, slow and too impractical to iterate more than a few times. This was at least the conclusion of the USACE[2], who in the 1950's dared such experimentation with a small-scale Mississippi River Basin model (Figure 1.3).

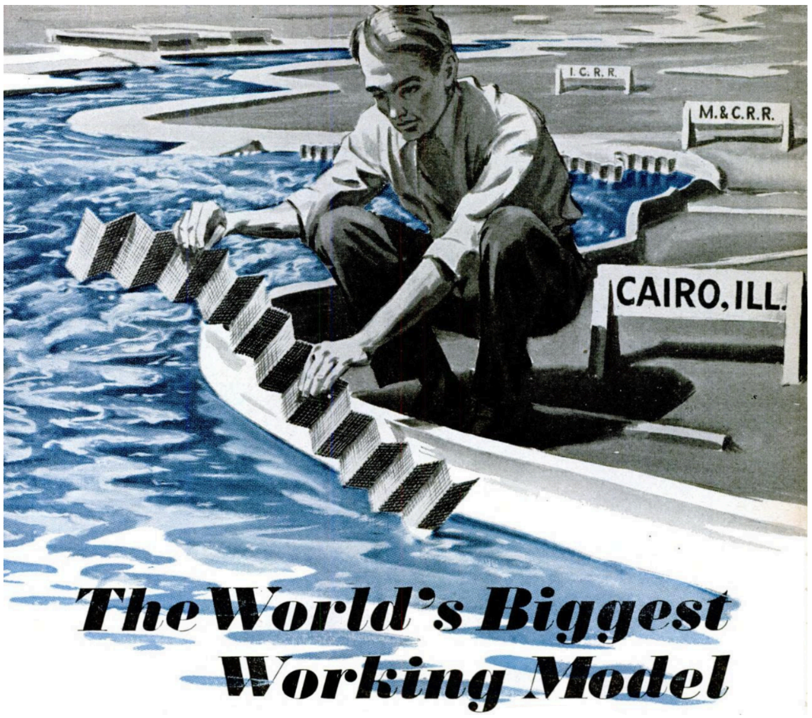


Figure 1.3: Mississippi River Basin Model, illustrated in an article in Popular Science (April 1948 issue, page 115).

After the Great Mississippi Flood in 1927 [20], the USACE built an extensive system of levees and floodways across the Mississippi basin to prevent future floods. These measurements targeted single sites and were planned locally, without considering the global river network. Years later scientists discovered that the local measurements not only did not work as expected, but they might even intensify future floods due to changes in the water currents [58]. Hoping to better understand the issue, the USACE built a hydraulic model of the Mississippi basin [12] where to physically simulate the entire system at small scale. Constructing and operating the $1km^2$ model required the work of a thousand laborers and a hundred hydraulic engineers, and although the physical model produced satisfactory results during its 20

[2]The USACE, or U.S. Army Corps of Engineers, is a federal and militar agency dedicated to public engineering, design, and construction management.

years of operation, it was remarkably impractical. In 1971, the project was decommissioned and substituted by more manageable, economical and efficient computer models.

With the popularization of computing and the commercialization of microprocessors, it did not take long for the USACE to recognize the value of computers for **spatial modeling**. They permit the study, planning and optimization of problems with economical, social and environmental interests. Besides flooding, computer spatial models help us: understand complex ecological processes, like the sustainability of ecosystems; analyze threatening future scenarios, like climate change and global warming; estimate the damages of spontaneous natural hazards, like tropical cyclones; or optimize the distribution of infrastructures, for example, in a city.

Nonetheless, a straightforward computer model is not enough to master the complex dynamics of large spatial processes. To be valuable, computer spatial models require:

- Carefully devised mathematical **equations** capable of reproducing the dynamics of complex phenomena. Often alternative formulations are available, each with different tradeoffs in accuracy and computational cost.

- Methods to quantify the **uncertainty** of the model, which at best produces vague approximations of reality. Sensitivity analysis and the Monte Carlo method are typical strategies to estimate the volatility of the output.

- Proper **calibration** with ground data to find the parameters that best connect simulation with reality, because even the most accurate equations can be meaningless if they are not backed by the right parameters.

- Independent **validation** with separate ground data from the calibration data, otherwise the model might overfit the calibration data, creating a false impression of accuracy.

- Continuous **iteration** of the design by testing, inspecting and adjusting until the model works. Poorly designed models will not only err but, worse still, they might misguide gullible decision-makers.

Concluding, computers enable the modeling and simulation of large spatial phenomena that are otherwise too complicated, expensive and time-consuming to experiment with. This causes a fundamental shift in the way modeling is done, from field work to office work. Note that, although the nature of the work changes for the better, the work does not disappear. Modelers still require long hours so as to formulate, quantify, calibrate, validate and iterate their models. However, because this is done with computers, the cost is now dependent on the quality of the modeling software. Given that modelers are not computer developers, their software should strive to be simple and intuitive. Therefore, and as we discuss later, a fundamental requirement of modeling software is **productivity**.

1.1.3 Advances in Spatial Data Collection

While the spatial phenomena occurring in the real world are continuous in nature, computers can only work with discrete data made of bits and bytes. The process by which we capture analog data into digital snapshots is called sampling. Spatial sampling is thus the process of collecting observations in a two-dimensional space. Temporal sampling, or just sampling, refers to the discretization of data in the time dimension. When the sampling rates in the different dimensions are high enough, this process results in reasonable approximations of reality that we call digital models.

In the last decades, and as a result of the improvements in data collection techniques, the sampling rate has rapidly multiplied in space, time and spectrum, which has brought unprecedented volumes of spatial data. The main contributors to this trend is the family of remote sensing technologies [49], a variety of onboard sensors designed to capture the earth's surface and atmosphere. These sensors are typically mounted in spacecrafts, aircrafts or UAVs (i.e. drones), from where they emit and measure radiation at multiple electromagnetic wavelengths in order to produce digital images of different spectra and resolutions [10]. The digital images then undergo a series of preprocessing steps that finally result in digital models of the Earth.

A typical digital model of the Earth (or any other celestial body) are Digital Elevation Models (DEM). DEMs are rectangular grids of cells, each of which holds a single elevation measurement. Together, these cells approximate a surface with varying accuracy depending on the spatial sampling. Figure 1.4 displays a DEM whose cells have been tinted and shaded for better visualization. This rectangular organization of the data is called **raster data format** and it is used to store continuous spatial information, like temperature or rainfall. Rasters, and particularly DEMs, are used extensively in the experimental section of this thesis. An alternative format is the vector data format, which is not covered in this thesis.

The number and size of grid cells determine the extension and resolution of a raster model. More cells lead to larger extensions, while smaller cell area imply higher spatial resolutions. The size of the smallest object that can be resolved in the model is determined by the resolution. Therefore, higher resolutions are preferable as they enable finer-grained simulations, but excessive resolution will increase the computational requirements beyond our capacity. Consider for example Finland, a northern European country with a land area of 338.424 km^2. A low-resolution DEM of the country (i.e. km scale) only occupies few megabytes of data, whereas a very high-resolution DEM (i.e. meter scale) increases that size to the terabytes.

One might question whether all that high-resolution spatial data being produced by remote sensors is needed. In fact, sometimes it is not. Current resolutions are sufficient to detect coarse-grained events such as deforesta-

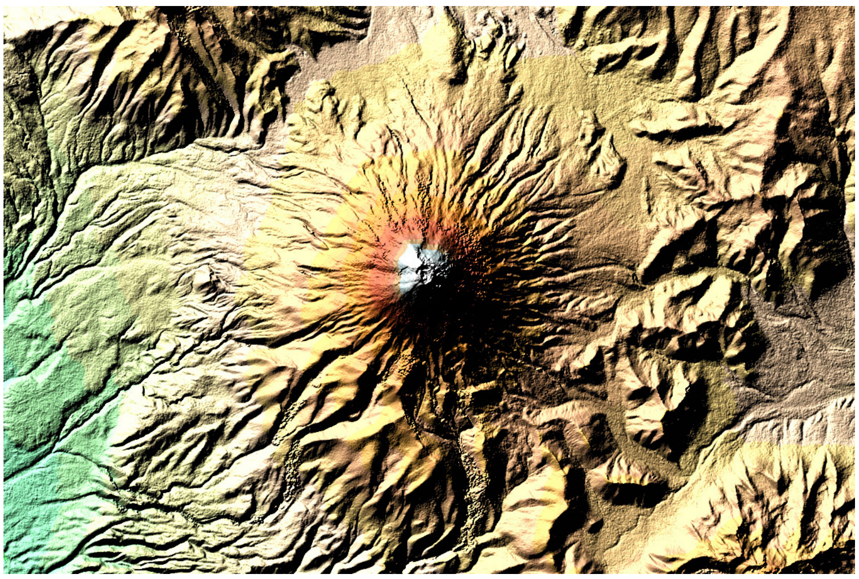


Figure 1.4: Mt. Cotopaxi, Andes Mountains. Image by NASA Earth Observatory [59], generated from the DEM acquired by the SRTM [26]

tion, pollution, great wildfires, ice caps melting, or land-use change, to name a few [60]. However, fine-grained models do require higher spatio-temporal resolutions to better resolve their dynamics. For instance, hydrological models yield deficient results if the resolution cannot register small streams, or if the extension of the area does not contain complete drainage basins. Another example are urban models, whose dynamics arises from a myriad of interconnected components. Housing, transportation, energy or infrastructure, all interact in complex ways and need to be tackled jointly. Therefore no single data source is sufficient and more spectrums, higher resolution and extensive areas are necessary to better comprehend the behavior of such complex systems.

Concluding, the increasing volumes of spatial data enable larger and more accurate simulations than any time before. However, more data entails more storage and calculations, and too many calculations become computationally prohibitive. Quickly, the cost of simulations escalates from tolerable minutes, to hours, to days, to unbearable weeks. At this point, the main obstacle to modelers is not the productivity of their modeling software anymore, but the time their laptop, workstation or cluster takes to execute their models and output some answer. Given that modelers are not computer developers, they have little knowledge on how to speed up their simulations. Therefore, and as we discuss later, another fundamental requirement of modeling software is **performance**.

1.1.4 Developments in Computer Architectures

Computer architectures have evolved from sequential, to parallel and to heterogeneous. The improvements in sequential processors, which had remained uninterrupted for half a century, came to a sudden halt in beginning of the 21st century because of three figurative walls. Due to the *power wall*, processors could not keep up the increases in clock frequency, because higher clocks generate more heat than standard cooling technologies can dissipate. Due to the *memory wall*, data could not be brought any faster to the processor, because higher bandwidth requires more pins than fit the chip and lower latency necessitates faster signal propagation than electrons permits. Due the *ILP wall*, processors could not execute more than a few instructions per cycle, because pipelining, out-of-order and speculation techniques have past their point of diminishing returns. Together, the three walls meant that the rapid improvements in sequential processors would soon plateau.

In 1974, Dennard observed that power density stays constant as transistors get smaller [3]. In other words, smaller transistors require quadratically less power to implement the same circuitry, and as a result, reductions in transistor lithography enable straightforward increases in frequency. By 2007 the trend known as **Dennard scaling** began to break down as transistors shrank below 65 nm [40]. This event put an end to the steady frequency increases (Fig. 1.5) and gave rise to the power wall. Hoping to return to the performance pace formerly driven by the frequency increases, CPU makers took an unconventional shift: from single-core to multi-core processors.

The idea was that heat would disrupt any 2.5 Ghz architecture if it was clocked to 5 Ghz, but a pair of 2.5 Ghz cores could be placed together to provide the performance of a 5Ghz core. Multiple cores would also aggregate more memory bandwidth and instructions per cycle, and so, the parallel shift would overcome the three walls, at least ideally. Unfortunately, parallelism is not the ultimate pass through the walls, as it brings new challenges along: many codes are inherently sequential and for the most part cannot run in parallel; even with plenty of parallelism, their parallel execution implies extra communication costs; and parallel programming is remarkably more complicated than writing sequential codes.

In 1965, Moore predicted that the transistor count on chips would double every year, and later extended the prediction to two years, in what became the most cited law of computing, **Moore's Law** [55]. Moore's Law is the most important trend after Dennard scaling because it enables more parallel circuitry. By 2012, as CPU makers struggled to push the transistor lithography below 22 nm [27], it became obvious that the pace of advancement had started to slow down in what could be the beginning of the end for Moore's law. In an attempt to return the advancements, CPU makers took another architectural shift: from homogeneous to heterogeneous architectures.

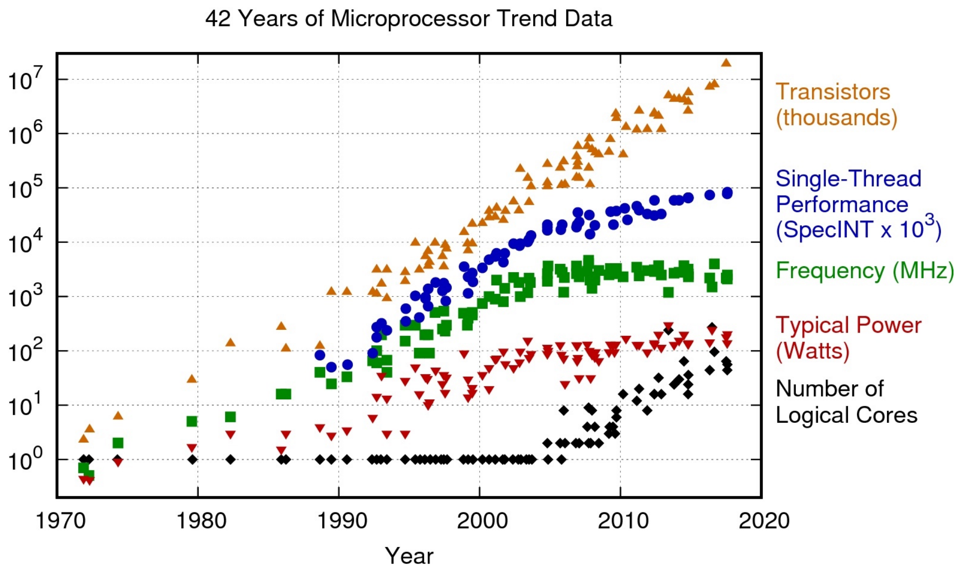


Figure 1.5: Four decades of microprocessor trend data [70] that expose the ILP and power walls, the shift toward parallelism, and the steady increase in transistor count predicted by Moore's Law.

The theory is that, as both power and transistor budgets become the limiting factor, it gets increasingly profitable to tailor the architecture to target critical workloads. Thus, if a specialized processor can run the critical workloads more efficiently than a regular CPU, then pairing both would lead to a more performant heterogeneous design, again, in the ideal case. Unfortunately, the new shift is not a final solution either, and also brings more problems: the devices require manual orchestration (e.g. memory transfers, kernel invocations); the codes need to be strategically mapped and scheduled to the different devices; and every architecture requires a bespoke version of the code to be performant.

To sum up, computer architectures have become parallel and heterogeneous to secure continuous performance improvements, but in exchange they have overwhelmed us with complexity. To make matters worse, not every problem benefits of this change; only those workloads with abundant parallelism, little need for communication and typical algorithmic structure do. As it turns out, spatial phenomena meet such requirements. They are inherently parallel, as they integrate numerous independent events dispersed across large spaces. They communicate little, since most events develop locally or only interact with their immediate neighborhood. They resemble the typical GPU workload, since the raster models are essentially images from the Earth surface. Consequently, as long as the programming complexity is dealt with, they have good prospect for performance.

Concluding, the nature of spatial models enables performance improvements with which to offset the data escalation. However, because attaining good performance out of modern architectures requires bespoke codes, codes written for a dual-core laptop will not make the most of, say, an octa-core workstation with GPUs. A productive development of computer models requires codes that scale automatically to better machines, because if new hardware requires new codes, the porting process becomes costly and impractical. Given that modelers are not computer developers, they have little knowledge on how to effectively port their codes. Therefore, and as we discuss later, a last fundamental requirement of modeling software is **portability**.

1.2 Research Focus

Throughout the previous section we have introduced the theme of computer modeling and simulation, we have made the case for simulating large spatial phenomena where physical experimentation fails, and we have discussed the trends and developments in spatial data collection and computer architectures. In between the lines, three fundamental requirements of spatial modeling software have been identified: *productivity*, *performance* and *portability*. Our research revolves around these three qualities and how they restrain the modeling of spatial phenomena. The next subsections state the research problem, the research goals and the research methodology.

1.2.1 Rationale

Before going any further, it is necessary to define the meaning of productivity, performance and portability (hereafter **PPP**). PPP are umbrella terms. They are clusters of desired *software qualities*. There are two types of software qualities, functional and structural qualities. Functional qualities measure how well a software fits its purpose. They relate to *what* the software does. Structural qualities refer to the remaining non-functional traits. They describe *how* the software works. A widespread classification of software qualities is the ISO/IEC 25010 [45], also known as SQuaRE (Fig. 1.6).

The classification used here is a tailored subset of the SQuaRE model, with a focus on performance. We disregard the Functionality, Reliability, Compatibility and Security groups of qualities, and only deal with the Usability, Maintainability, Performance and Portability types of qualities. Because these software qualities overlap and interact in complex ways, we have clustered them in the three PPP concepts, as seen in Figure 1.6. The Usability and Maintainability groups are merged into Productivity, while the Portability group acquires performance-related qualities.

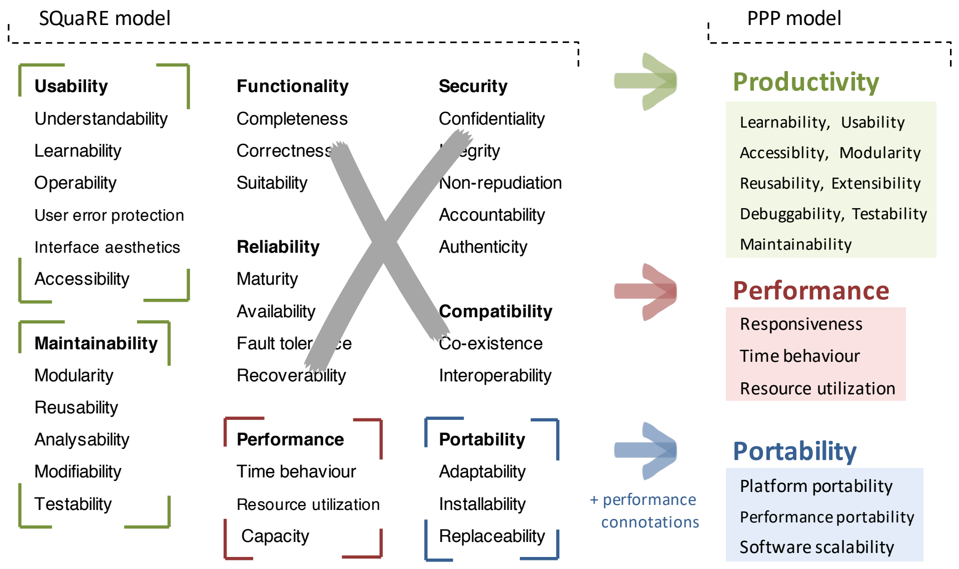


Figure 1.6: Software product Quality Requirements and Evaluation (SQuaRE) model [45] to the left, and our subset model to the right, which groups those qualities of interest into the PPP concepts.

Productivity is, in broad terms, the ability to rapidly develop and maintain computer models. If the modeling software is not productive, few models will be created and supported to begin with. Productive software presents a gradual learning curve for the unexperienced user. It prioritizes usability, while being flexible in the construction of solutions. It permits the composition of large solutions out of smaller, reusable modules. Because models are complex to get right, productive software is also testable. It assists the tracing of errors and facilitates their step-by-step debugging. Finally, productive software makes it viable to extend and adapt existing solutions.

Performance is the capacity of the computer models to execute efficiently in the underlying architecture. Performance is essential to handle large volumes of data, and its absence restricts the functionality of models. For instance, regional weather models need high resolution to reproduce fine dynamics such as tropical cyclones. Performant software responds quickly to the user commands and queries. It achieves as high throughput and as low latency as the underlying hardware permits. Performant software is time-efficient, since it minimizes the time users wait for a simulation. At the same time, it does so without wasting unnecessary memory or other hardware resources. Lastly, performant software is not only efficient to run, but also to test and debug.

Portability is the ability of the computer models to seamlessly scale to larger and newer hardware. Note that portability has performance connotations here and goes beyond typical machine and platform portability. This is a quality of increasing importance due to the shifts toward parallel and heterogeneous architectures. Portable software is developed once but runs anywhere. To achieve this, it abstracts away the low-level details of computer architectures and execution models. Portable software can process moderate workloads on a laptop, and scale to a workstation when the data increases. It can be submitted to a cluster to execute a large workload, or deployed to the cloud as part of a service. Finally, portable software enjoys a long-life cycle, reducing expenses related to its deployment.

PPP are critical qualities that, when missing, hinder the development of computer models. From the moment a computer model is envisioned until it is put into operation, it goes through a development process that hereinafter we call the **modeling loop** (Figure 1.7). This process is highly iterative and is meant to derive the model pragmatically. It starts simply, with straightforward equations that are extended gradually. The data, too, should be clear at first before moving to larger, more detailed datasets. This approach makes it easier to develop the model, correct its errors and understand why it works.

The modeling loop is not a strict process, but rather a series of recurrent steps (Figure 1.7). It starts with the conceptual design of the model (1), which is translated to source code computers can run (2), that is then tested and debugged until it is operative (3). Note how these steps are restrained by the productivity of the modeling software, because they require constant interaction and manipulation of the code.

Next the sensibility and uncertainty of the model are quantified (4), followed by its calibration with historical data of relevance (5), and by the model validation against a different set of data (6). These steps are constrained by the performance of the modeling software, because they all need to run repeated executions of the model.

Once the model is functional for small inputs, it can be scaled to larger datasets (7). When the increases in data undermines the performance, the model is transferred to a faster machine. Note that, without portability, this step might require a reimplementation of the code for the new machine. Finally, the model is deployed for use in analysis and simulations (8).

Apart from being iterative, the modeling loop is also a very dynamic process. The modeler could at any moment return to a previous step to correct a flaw, and therefore the three bottlenecks may alternate during any stage of the development. Consequently, the software should be designed to always maximize the three qualities. However, there is a conflicting connection between the three, namely, a **tradeoff** (Figure 1.8).

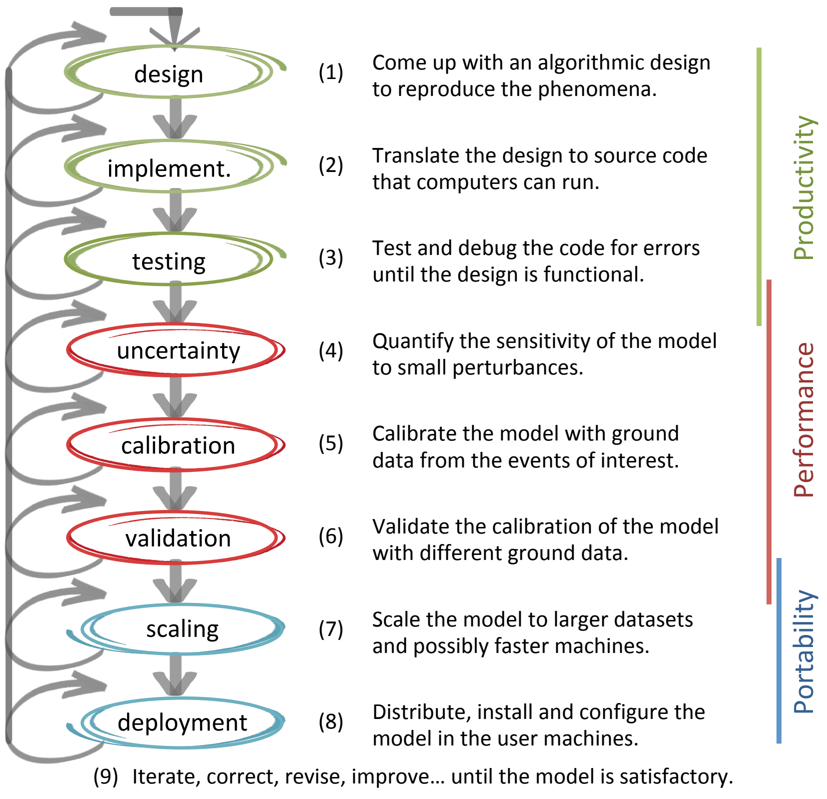

design
(1) Come up with an algorithmic design to reproduce the phenomena.
implement.
(2) Translate the design to source code that computers can run.
testing
(3) Test and debug the code for errors until the design is functional.
uncertainty
(4) Quantify the sensitivity of the model to small perturbances.
calibration
(5) Calibrate the model with ground data from the events of interest.
validation
(6) Validate the calibration of the model with different ground data.
scaling
(7) Scale the model to larger datasets and possibly faster machines.
deployment
(8) Distribute, install and configure the model in the user machines.
(9) Iterate, correct, revise, improve… until the model is satisfactory.
Productivity
Performance
Portability

Figure 1.7: Modeling loop: a pragmatic process to develop models.

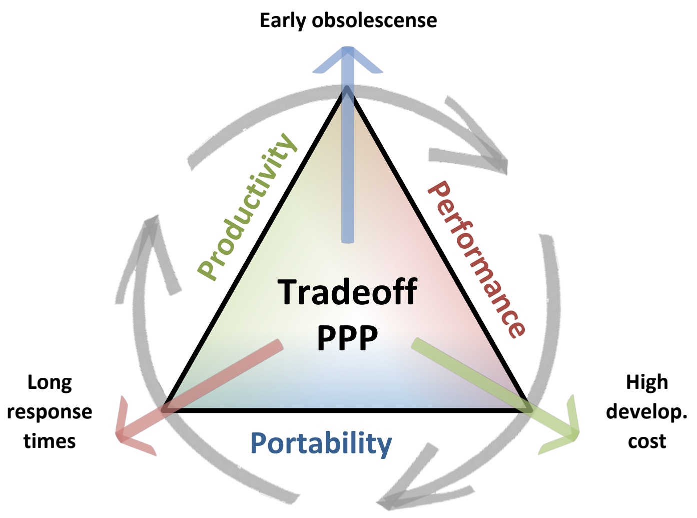


Figure 1.8: Due to the Productivity-Performance-Portability (PPP) trade-off, approaching any of the qualities distances the others.

For example, productivity requires writing simple, sequential and generic codes, but this contrasts with the parallel and heterogeneous architectural shifts, decreasing the performance. Performance requires parallel programs tuned for the architecture, but this implies writing low-level codes in machine-dependent semantics, giving up the portability. Portability requires bespoke codes for the different architectures, but this implies multiple designs, implementations and testing of the model, halving the productivity. Nonetheless, the three qualities are important, and overlooking any one has important consequences. This tradeoff between the PPP qualities, hereafter the **PPP tradeoff**, is the basis of our work.

In conclusion, there is a **problem**: *we need PPP, but they conflict with each other.* Thus, we have a clear research **question**: *how can we get PPP without major compromises?*

1.2.2 Aims and Objectives

The research that makes up this thesis focuses on one general goal: *to solve the exposed PPP tradeoff in the context of raster spatial modeling.* This is important because spatial modeling is a valuable tool against the problems of the developing world. More specifically, *we aim to attain productivity, performance and portability in the analysis and simulation of large spatial phenomena with computer and digital models built upon raster data.* Although straightforward to comprehend, the PPP tradeoff is no simple problem to solve. It raises multiple questions, requires complex solutions and permits different approaches. Consequently, the research scope has been narrowed in the following ways:

On the computing side, we...

+ Utilize shared-memory machines (e.g. laptops, workstations).
+ Focus on multi-core CPUs and many-core GPUs.
− Omit distributed systems (e.g. clusters, supercomputers).
− Omit other types of heterogeneity (e.g. big.LITTLE).

On the data side, we...

+ Target models built upon raster data (e.g. DEMs)
+ Test data sizes fitting into shared-memory, of up to the terabyte.
− Omit models based on vector data (e.g. TINs).
− Omit datasets requiring distributed-memory, beyond the terabyte.

On the modeling side, we...

+ Propose and prototype a solution to the PPP tradeoff.
+ Investigate and experiment quantitatively with performance.
− Only perform a qualitative evaluation of productivity.
− Do not evaluate and only discuss about portability.

For reasons that become apparent later, we will tackle the problem with techniques from programming languages and compiler theory. On the one hand, we will employ an intermediate representation (IR) like compilers do, because this enables the translation from productive, high-level programming abstractions to performant, low-level machine instructions. On the other hand, we will target a subset of applications like domain-specific languages (DSL) do, because this narrows the optimization possibilities and reduces the compilation complexity.

In particular, we will restrict the application domain to spatial models built upon raster data. At the same time, we will adopt the **map algebra** formalism, a DSL for raster spatial analysis. The idea is to integrate the architecture and techniques of a compiler into map algebra, so that the execution is decomposed into a front-end and a back-end connected by an IR. In this way, only the front-end requires productivity, the back-end performance and the IR portability. This ***breaks the cyclic tradeoff*** displayed in Figure 1.8, and now each component can maximize one quality without affecting the other two. We call this strategy **a compiler approach to map algebra**.

Having introduced the approach that will be used to answer the research question, the specific objectives of this research are:

- *To design a map algebra compiler that brings productivity, performance and portability to raster spatial modeling.*

- *To identify the performance bottlenecks of map algebra workloads and determine code optimizations to relieve these bottlenecks.*

- *To collect experimental data from typical map algebra workloads and analyze the effects of the optimizations in the performance.*

- *To develop a prototype implementation of the compiler approach that can be used to evaluate the functionality of the design.*

1.2.3 Methodology

A research methodology is a systematic process that guides the creation of new knowledge. It is like an algorithm, made of well-defined steps and transitions, but for scientists. Methodologies are meant to prevent the degeneration of the research process, by which scientists get lost in the complex research flow and lose direction. In this research we have followed a mix between scientific method and engineering design process. The scientific method involves observations, hypothesis and experiments (Fig. 1.1), whereas the engineering process is concerned with designing, building and testing solutions. The former discovers new knowledge, while the latter applies that knowledge to solve problems.

This combination of scientific and engineering approaches obeys two reasons. First, we are computer engineers, not spatial modelers. Our goal is to provide solutions to their problems. This is what engineers do, they identify and solve problems. They ask: "**who** needs **what** because **why**?" In our case: "spatial modelers need a map algebra compiler because of the increasing volumes of spatial data." Second, coming up with the compiler architecture and the code optimizations is not straightforward. It requires a more scientific approach, by which we hypothesize that organizing the execution of map algebra scripts in a certain way brings higher performance, and subsequently run experiments to verify the claim.

More specifically, the methodology employed in this research is an iterative process that involves the following steps:

1. Rational / Observation.

   Modeling and simulation of spatial phenomena is a useful tool. As the volumes of data increase, this tool requires more performance. Computers are improving too, but harnessing their power is unproductive and non-portable.

2. Problem / Question.

   For spatial modeling software to be useful, we need productivity, performance and portability. However, getting the three qualities poses a tradeoff.

3. Design solution / Hypothesis.

   A map algebra compiler with certain architecture and optimizations might be the solution. *Architecture*: productive front-end, portable IR, performant back-end. *Optimizations*: parallelism, reordering, specialization, sparsity.

4. Build prototype / Experiment.

   Implementation: Python interface, C/C++ back-end, OpenCL parallel code. *Benchmarks*: representative map algebra workloads and spatial models.

5. Test prototype / Analyze results.

   Benchmark the prototype and its different optimizations. Analyze the results and identify possible improvements.

6. Iterate or Conclude.

   Evaluate the state of affairs. If more improvements are possible (or necessary), go to point 3. If the design solves the problem acceptably well, conclude.

Briefly, after the problem has been properly motivated and identified, we continually design, build and test a prototype of the solution. Some ideas will work while others will not, but this is not known in advance. Therefore it is necessary to test a hypothesis and, if it yields bad results, update the design. A cycle ends and makes way for the next iteration when we have learnt enough to improve the design. These steps are repeated until a satisfactory state is reached.

1.3 This Thesis

The previous section has presented the research problem (i.e. the PPP tradeoff in raster spatial modeling), stated the research goal (i.e. to solve the tradeoff with a map algebra compiler), and justified the research methodology (i.e. a scientific-engineering method). This thesis is not a chronological narrative of our research (except for subsection 1.3.2), but it presents the –at the time– latest results achieved via the research methodology. The following subsections list the scientific contributions of the thesis, present the original scientific publications, and outline the organization of the thesis.

1.3.1 Thesis Contribution

This thesis makes the following contributions to the body of knowledge:

First contribution. We combine the map algebra formalism with compiler techniques to achieve a productive, portable and performant software for raster spatial modeling. Thereby, modelers simply write sequential map algebra scripts that, when executed, get automatically translated into parallel codes that run on modern computer architectures. The approach targets shared-memory machines with multi-core CPUs and many-core GPUs, and can deal with up to terabytes of raster data without running out of memory.

Second contribution. We device several code optimization techniques for the map algebra compiler. After investigating the behavior of map algebra workloads we find that memory, and not computation, is the major bottleneck when processing large spatial datasets. This motivates the development of four groups of optimizations targeting: the parallelism of the spatial dimension, the reordering of the computation, the specialization of algorithmic patterns, and the exploitation of the sparsity in data.

Third contribution. We analyze the effects of the optimization techniques, individually and collectively, on different map algebra scripts with varying combinations of spatial operations. This shows that memory is the major bottleneck and that no single optimization excels all others, but their efficiency depends on the configuration of the script and the given input data. We first benchmark four simple workloads, and then make the case for the map algebra compiler with two real-world problems, viewshed analysis and a cellular automaton for urban development.

Fourth contribution. Finally, to verify our claims we develop a prototype implementation of the compiler approach. We choose the Python language for the front-end interface because of its simplicity and readability.

The IR is formulated as a directed graph where nodes represent operations and edges represent data. We implement the back-end and runtime system in C++ because of its high performance and low-overhead. The multi-core CPUs and the many-core GPUs are instructed with the OpenCL kernel language. This is moreover transparent to the users, who only interact with the map algebra interface.

1.3.2 Original Publications

This subsection lists the scientific publications fruit of our research:

The first paper, with title *"Efficient implementation of a fast viewshed algorithm on SIMD architectures"* [6], was devised as an exploration of the affinity of map algebra operations for modern computer architectures. The article focuses on viewshed analysis, a raster operation with complex parallelization. This paper provided valuable feedback that made some points clear to us: that the spatial dimension in raster models is highly parallelizable and a source of performance; that complex map algebra operations like viewshed can be parallelized in quite a few ways; and that writing parallel map algebra codes is nontrivial even for expert programmers. With this work, our concerns about performance and parallelism were augmented with questions about productivity. It became apparent that modelers should not struggle with parallelism, but they needed transparent parallelism.

The second paper, titled *"From python scripting to parallel spatial modeling..."* [7], investigates the use of map algebra for spatial modeling, with a focus on cellular automata models. The article introduces a productive Python interface with sequential semantics yet parallel execution. This work is the inception of the compiler approach, where we realized that interpreting scripts is inefficient. The evaluation of the performance in this work, while simple, already shows high speed-up numbers. More importantly, it reveals that for large datasets considerable time is spent just moving memory around. During this work, our concerns about performance were extended with the identification of the memory bottleneck.

The third paper, with title *"A Compiler Approach to Map Algebra..."* [9], exposes the performance issues of interpreting map algebra scripts and proposes the compiler approach. It shows that parallelism is not enough to get high performance, but data locality is critical too. Consequently, multi-core CPUs or GPUs bring no advantage unless the memory bottleneck is dealt with. The article tests five map algebra scripts for which it achieves speed-ups of one to two orders of magnitude. Compared to traditional map algebras, the compiler approach is equally productive but considerably more performant. Moreover, it supports multiple computer architectures, meaning it is portable too.

The fourth paper, entitled *"A compiler and runtime approach to parallel spatial modeling"* [8], is the first article to identify and discuss the productivity, performance, portability tradeoff. It also provides a clearer decoupling of the compiler components and of the groups of optimizations. The paper introduces control flow in the IR, permitting the use of branching and loop structures. This enables richer map algebra scripts, particularly those whose computation depends on the data. An example of this is a water basin catchment simulation that keeps running until all water reaches a ground state. The article also performs a more exhaustive evaluation of the code optimizations and their interactions. It tests a cellular automata model for urban development and shows that no single optimization is responsible for most of the speed-up, but they all contribute to relieve the different bottlenecks. After this work we concluded that the PPP tradeoff was indeed solvable in the domain of raster spatial modeling.

1.3.3 Organization of the Thesis

This thesis is a compendium of peer-reviewed articles that consists of two parts. **Part I** introduces and overviews the doctoral research in a concise but detailed manner. **Part II** compiles the original publications described in subsection 1.3.2. Part I is structured as follows.

Chapter 1 has introduced the context (raster spatial modeling) and motivated the problem (PPP tradeoff). **Chapter 2** presents the three background fields upon which the research builds (computer architectures, computer cartography and programming languages). **Chapter 3** describes the solution to the problem (map algebra compiler), which is also the main contribution. **Chapter 4** tests a prototype implementation of the solution with three experiments to verify its adequacy. Finally, **Chapter 5** concludes this thesis and discuss its limitations and future possibilities.

Chapter 2

Background

"If I have seen further is because I stand on the shoulders of giants"

~ **Isaac Newton**

Three fields of knowledge are crucial to understand this thesis: *computer architectures* for their hectic development during the last half of a century, *computer cartography* for its enabling role in the modeling and simulation of spatial phenomena, and *programing languages and compilers* for their effect in the productivity and performance of programs. The next sections describe these three topics in that order.

2.1 Computer Architectures

Primitive computers like Colossus and ENIAC had to be physically rewired to perform different tasks, making the development of programs a tedious and error-prone process [16]. These computers were also expensive, fragile, power-hungry and needed trained manpower to be operated. The von Neumann architecture made away with the programming-by-rewiring by storing programs in memory. In the core of this design lays a control unit that decodes and dispatches instructions to arithmetic logic units. Although more programmable, few organizations could afford the early von Neumann machines built with vacuum tubes. The invention of the transistor (1947), followed by the integrated circuit (1959), gave born to modern computers. A decade later, the Intel 4004 [25] became the first commercial microprocessor, and many others followed. Since then, computer architectures have developed into very advanced designs, while still maintaining the general principles of the von Neumann architecture.

Computers have historically improved in two ways: via architectural innovations and via developments in semiconductors [39]. The first microprocessors were relatively simple, static in frequency, and executed scalar instructions sequentially. Much of their early improvements came from the advances in the manufacturing technologies, which brought steady increases in transistor count and operating frequency for a few decades. With time, it became remarkably harder to scale frequency (*end of Dennard's scaling*), and to shrink and pack more transistors (*deceleration of Moore's Law*). Thus, to continue the pace of development, most improvements had to come now from the architectural innovations. As a result, architectures transitioned toward more complex, dynamic, superscalar and parallel designs. This evolution of architectures can be classified in three periods: the sequential, parallel, and heterogeneous eras. The next subsections overview these periods and the effects they brought onto the PPP qualities.

2.1.1 Sequential Era

The sequential or single-core era was driven by improvements in ILP[1] and increases in frequency. Notably, pipelines went from scalar to superscalar and their clock raised from kHz to GHz speeds. The fast pace of development in this era made most codes double their performance by simply buying the next generation microprocessor.

Hereafter we focus on the x86 ISA[2], as this is the architecture of the CPUs we employed. Basically, x86 is a register-based, CISC[3], dynamically scheduled and performance-oriented architecture: based on registers as opposed to stack-based architecture where data is pushed and pulled into a stack; CISC and not RISC[4], since the x86 ISA includes complex instruction and diverse addressing modes; dynamically scheduled because the hardware, and not the compiler, determines the execution order of the instructions; and performance-oriented in contrast to energy-efficient architectures designed for the mobile market, like ARM.

The x86 ISA, like any von Neumann architecture, requires a minimal set of functional units to operate (Fig. 2.1). The control unit is in charge of decoding the instructions that are fetched from memory. The arithmetic logic unit reads operands from registers or memory, operate on them, and write back the results. Few special purpose registers are also needed, like

[1]Instruction-Level Parallelism (ILP) refers to the possibility to execute multiple instruction per clock cycle when said instructions present no dependencies.

[2]An Instruction Set Architecture (ISA) is an abstract model of a computer, a logical blueprint not tied to any specific implementation.

[3]Complex Instruction Set Computer (CISC) architectures employ large ISAs that include high-level non-primitive operations (e.g. substring search).

[4]Reduced Instruction Set Computer (RISC) architectures present small ISAs made of basic instructions that map directly to hardware (e.g. register movement).

the program counter that points to the next instruction. The memory unit mediates the load and store of data residing in the memory addresses requested by the CPU. Finally, these units are not isolated but communicate with the exterior via I/O ports connected to devices like disks.

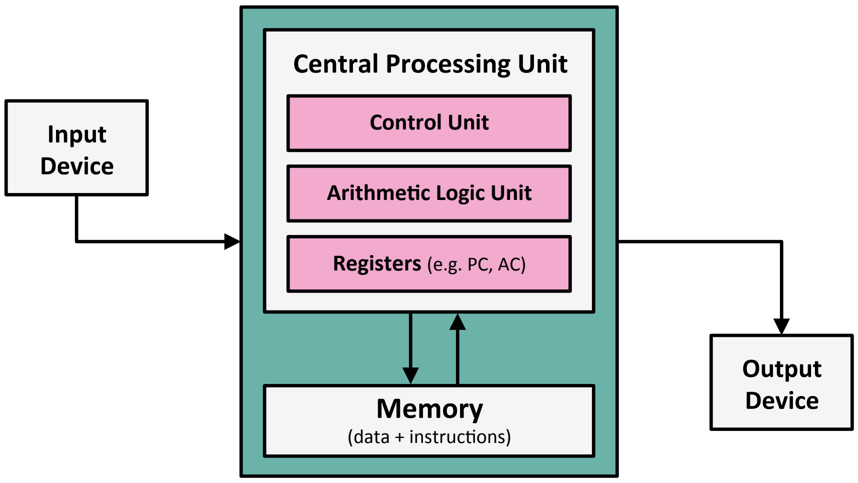


Figure 2.1: Abstract diagram of a von Neumann machine.

While Figure 2.1 is conceptually functional, x86 implementations were more complex from the beginning. The Intel 8086, released in 1978 as the first x86 model, could already overlap the fetch and execution of instructions. This mechanism, known as pipelining, was thoroughly extended to overlap more stages in following x86 models. For instance, the Pentium microprocessor (1993) could fetch an instruction, decode another, solve an address, look up the cache, execute an instruction, and write back the result of another, all in one clock cycle. This was the first x86 superscalar pipeline, which could complete up to two instructions per cycle via two independent ALUs. On the other hand, the Pentium pipeline worked in-order, meaning instructions had to complete in the program order. Such design is highly affected by pipelining hazards[5], whereby the next instruction cannot execute in the following cycle because its input is still being computed by a previous instruction in the pipeline.

The average computer program is riddled with hazards that constantly stall the execution of in-order pipelines. To circumvent this issue, the Pentium PRO (1995) introduced the first out-of-order pipeline in the x86 family. These pipelines are able to reorder the stream of instructions to eliminate many data and structural hazards. Data hazards are avoided by is-

[5]Three types of pipelining hazards can delay the execution: **data** hazards due to data dependencies, **structural** hazards due to limited functional units, **control** hazards due to branches making the next instruction unpredictable.

suing other instruction from the stream whose inputs are ready instead of stalling, while functional hazards are minimized by replicating critical functional units (e.g. ALUs, address generators). Control hazards, on the other hand, are not preventable with these techniques alone, but require a more aggressive class of ILP optimizations known as speculative execution.

Although the first Pentium had an in-order pipeline, it could already execute instructions speculatively. To do so, it makes use of a branch prediction unit that guesses the target branch based on the program history. As long as the guess is right, this removes the control hazards and allows the continuous pipelining of instructions. However, when the guess is wrong, the execution needs to be retaken from the correct branch, incurring a penalty. Speculation can also apply to data, for example, by prefetching memory addresses that might be accessed in the future. Prefetching is common in modern CPUs, which can deduce simple memory access patterns.

Another important ILP technique was the addition of single instruction multiple data features. The Pentium MMX (1993) was the first x86 model to include a 64-bit wide vector ALU that could operate on either eight 8-bit integers, four 16-bit integers, two 32-bit integers, or one 64-bit integer. These SIMD instructions targeted specific workloads (e.g. multimedia, graphics, scientific codes) and required manual programming with intrinsics[6]. Nonetheless, they were well received and progressively extended in width (128b SSE, 256b AVX, 512b AVX-512) and functionalities.

Perhaps the most radical evolution of x86 was its internal reorganization into a RISC microarchitecture (μarch). CISC instructions are in general too complex to achieve a highly clocked pipeline [39], but moving to a RISC ISA was not an option since backward compatibility was a main selling point. Since the P6 family, the x86 core was split into a stable CISC front-end and a malleable RISC back-end connected by a decoder that turns the x86 instructions into micro-operations. Thereby, the pipeline could pursuit higher clocks and ILP without impacting the established x86 interface.

It goes without saying that many other improvements not mentioned here were developed during that time. For example, simultaneous multithreading (SMT) was introduced with the Pentium 4 (NetBurst μarch, 2000), and later μarchs introduced the Bit Manipulation Instruction (BMI), the Fused Multiply-Add (FMA), the Advanced Encryption Standard (AES) and the Secure Hash Algorithm (SHA) instruction sets extensions. Nonetheless, while it is important to understand the origin and complexity of modern computers, henceforth we only need to remember that this was an era of increasing and uncompromising performance.

[6]Intrinsic functions provide direct access to advance instructions (e.g. SIMD), which is necessary when compilers cannot generate these instructions on their own.

2.1.2 Parallel Era

The parallel or multi-core era started with the shift toward multi-core processors[7]. As single cores would not get much faster, more cores had to be used together. However, now only those codes designed with parallelism would attain more performance when buying a new processor.

From the 8086 to the Pentium 4, x86 codes boasted effortless performance from the increasing ILP and frequencies. Beginning of the 21st century, **Dennard scaling** started to break down as transistors approached the atomic level. Consequently, power would not continue to shrink linearly with transistors size, leading to the power wall [43]. At the same time, most pipelining optimizations were bringing diminishing returns, prefacing the ILP wall [89]. Lastly, memory bandwidth and latency suffered their own limitations too, in what is called the memory wall [96]. The walls meant that *the free lunch was over* [78] and performance would not come without effort. This began an architectural shift toward parallelism, not at instruction level[8], but at thread level.

In x86, the parallel shift manifested with the discontinuation of NetBurst in favor of the Core μarch[9]. NetBurst had a remarkably long pipeline in an attempt to reach 10 GHz in future models, but the excessive power consumption of such design impeded scaling its frequency beyond 4 GHz. The new Core μarch returned to the lower clock, high efficiency small pipeline of the P6 μarch. Most importantly, it divided the processor die into multiple cores connected by a unifying last-level cache. These independent but interconnected cores would collaborate to run programs faster than a single power-limited core. Thus, provided that x86 codes are parallelized, this architectural shift set a new path for continued performance.

Figure 2.2 shows the die shot of a modern Intel x86 consumer architecture (Coffee Lake, 2017). This is a symmetric multiprocessing design, where cores share a main memory and the I/O devices. The x86 architecture implements a strong memory model and is also cache coherent, therefore caches are actively synchronized to provide all cores with an up-to-date view of the shared memory. Maintaining this coherency for increasing core numbers puts strong pressure in the internal topology, which evolved from buses (e.g. Core) and crossbars (e.g. Nehalem) to rings (e.g. Sandy Bridge) and meshes (e.g. server parts). Another development was the addition of dynamic voltage and frequency scaling techniques[10] to achieve a finer power

[7]The conventional definition of a multi-core CPU is such that it presents from two to tens of cores, cache coherence and a shared memory system.

[8]Note that while the sequential era presented parallelism at instruction level, this was oblivious to programmers who wrote sequential codes. However, the thread level parallelism of this era implied the explicit parallelization of codes.

[9]Although other multi-core processors were already in the market (e.g. Power 4, 2001), it was the discontinuation of NetBurst what most strongly marked the multi-core shift.

management of the cores and optimize the power budget.

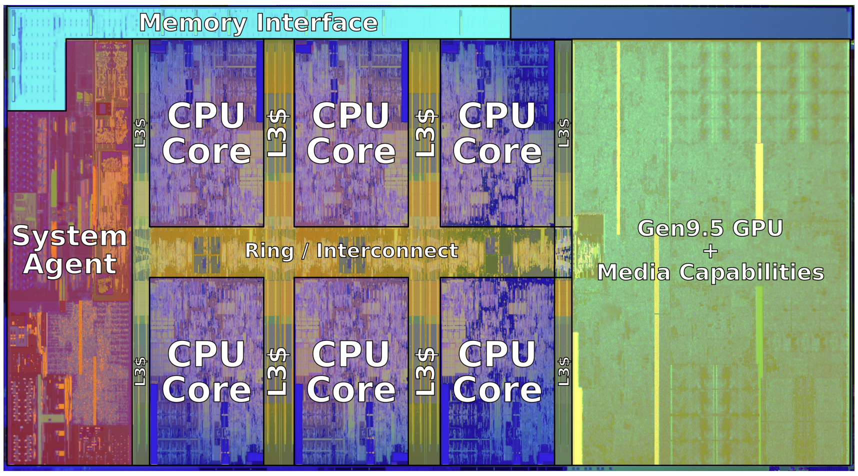


Figure 2.2: Annotated die shot of Intel's Coffee Lake architecture (2017) with six cores and ring interconnection [91].

Unfortunately, programming these multi-core processors turned out to be complex and laborious. Unlike instruction-level parallelism, which is automatically handled by the processor and compilers, thread-level parallelism is a complex problem that requires manual intervention from the developers. This generates new programming issues for the developers, which in turn impacts their productivity.

First, many algorithms are inherently *sequential* and for the most part cannot run in parallel. This effect is characterized by **Amdahl's Law** [1], which states that even small fractions of sequential execution have large impact on the speed-up of parallel codes. Examples are cryptographic algorithms like Cypher Block Chaining or iterative solvers like Newton-Raphson, where the computation of the next step cannot be started until the results from the previous step are available.

Second, even if a problem displays plenty of parallelism, its parallel execution may imply extra *communication* costs. This occurs because the separate cores need to cooperate and synchronize data in order not to step on each other's work. Due to the Memory wall, communication becomes a bottleneck that limits the scalability of any parallel codes. An example are graph algorithms, for instance Dijkstra's single-source shortest-path algorithm, which permits the parallel exploration of unvisited edges, but requires constant synchronization of the visited edges.

Third, parallel *programming* is remarkably more complicated than writ-

[10]Dynamic frequency scaling reduces the clock speed of circuits with low workload, while dynamic voltage scaling lowers the voltage when the frequency is low.

ing sequential codes. Parallel codes are more error-prone, harder to debug, less maintainable and not very portable. Furthermore, an unsophisticated parallelization is not enough, otherwise the speed-ups will be worthless. Programmers need to carefully design their codes to minimize synchronization, split the work wisely, balance the load fairly, coordinate the parallel I/O and sort out a number of other parallel issues.

Lastly, writing parallel programs while ensuring their *correctness* adds yet more complexity. Often, seemingly correct codes hide bugs that only manifest under rare circumstances. Unlike with ILP, where the processor (or compiler) takes care of the hazards, here it is the programmer who must take responsibility for the coordination of the execution. Failing to do so leads to race conditions, deadlocks and other concurrency issues.

Despite all these problems, multi-core processors are here to stay. Programmers have embraced parallelism when performance is a requirement, as this is now the main source of performance in modern x86 chips. Nowadays, x86 consumer processors include up to eight cores (e.g. Intel Core), x86 workstation/enthusiast processors have up to sixteen cores (e.g. AMD Threadripper), and the x86 server processors comprise up to thirty-two cores per socket (e.g. AMD Epyc). Additionally, the server parts are coupled into multi-socket systems to double or quadruple their cores. Extending the core number beyond this point requires distributed-memory solutions called clusters, in which multi-socket systems are connected by a high-bandwidth low-latency communication network. Although it is important to mention these distributed-memory systems and their inter-chip parallelism, hereafter we focus on the shared-memory architectures and intra-chip parallelism employed in this work.

2.1.3 Heterogeneous Era

The heterogeneous or many-core era commenced with the introduction of specialized architectures, such as GPUs[11]. As semiconductors approach physical limits, it becomes difficult to further shrink their size and increase their numbers. In face of this growing shortage, specializing the circuitry for typical workloads becomes a promising alternative.

Even though no one can confidently say **Moore's Law** has ended, it has clearly slowed down [40]. This puts unprecedented pressure on the transistor budget, which like the power budget might soon reach a limit. As a result, packing more symmetric cores into multi-core processors can only continue so far, and now more than ever it becomes profitable to tailor the finite circuitry toward selected workloads. This tailoring leads to what is called a

[11]Graphics Processing Units (GPUs) are many-core processors, i.e. with hundreds to thousands of cores, primarily targeted at graphics workloads, but compatible with other data-parallel domains, such as linear algebra and machine learning.

heterogeneous architecture, where the processors can be asymmetrical and targeted at certain compute patterns.

Like the parallel era was consolidated with the transition from NetBurst to Core, the heterogeneous shift was apparent with the evolution of GPUs into general-purpose processor[12]. GPUs became programmable first with CUDA [62] and later with OpenCL [46], and since then much software has being ported to heterogeneous systems combining GPUs and CPUs. Another prevalent example of heterogeneity is the ARM big.LITTLE architecture, which pairs slower low-power cores (LITTLE) with faster power-hungry cores (big) to maximize battery and performance.

Nowadays the heterogeneity is spreading where power or performance are of great concern, such as in embedded systems, image processing, computer vision, artificial intelligence or machine learning. For instance, modern smartphone chipsets typically incorporate along their mobile CPU an Image Signal Processors (ISP), a Digital Signal Processors (DSP) and a mobile GPU. Another example is Nvidia's Xavier System-on-Chip (SOC), shown in Figure 2.3, which includes a Deep Learning Accelerator (DLA) and Programmable Vision Accelerator (PVA).

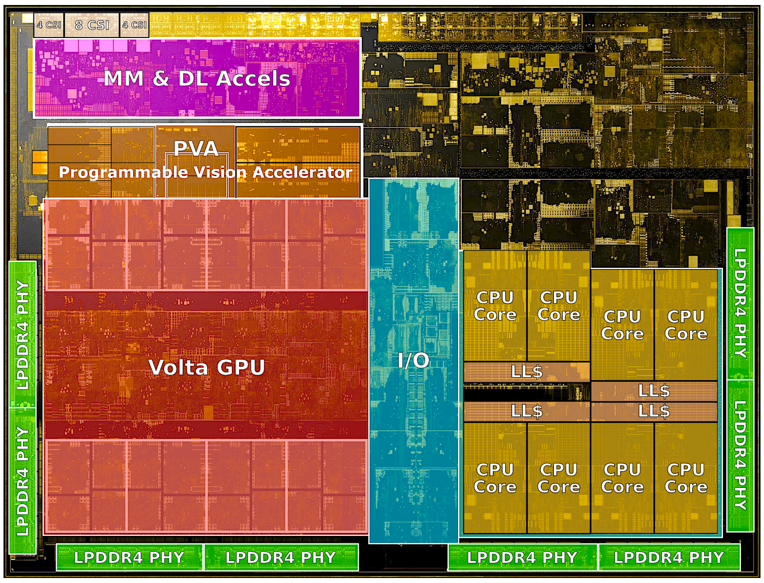


Figure 2.3: Annotated die shot of Nvidia's Tegra Xavier architecture (2018) with a multi-core CPU, an integrated GPU and other accelerators [92].

Unfortunately, the heterogeneous shift comes with tradeoffs again. Compilers, programming languages and libraries are not able to abstract away the

[12]Early heterogeneous processors like the Cell [28] entered and left the market with little success, while the polyvalence of GPUs for graphics and compute allowed them to stay relevant.

heterogeneous architectural details. This leads to new programming challenges that impact the productivity and the portability of high-performance codes.

First, attaining peak performance requires *code versions* tailored to the architecture. GPUs can run straightforward ports of CPU algorithms, but that is often very inefficient. For example, ray tracing on CPUs is implemented differently than on GPUs due to how they handle branches [68].

Second, heterogeneous architectures often lack coherency and a strong *memory model* unlike x86. Consequently the management of memory is exposed to the programmer, who needs to orchestrate it manually. For instance, in CPU-GPU systems the CPU acts as a host that commands the memory transfers for the GPU via API calls.

Third, heterogeneous processors need heterogeneous *programming models* and languages. For example, GPUs use kernel languages (e.g. CUDA, OpenCL) while FPGAs employ hardware description languages (e.g. Verilog, VHDL). This can lead to a complex mix of languages and models within a single high-performance program.

Lastly, programmers also face problems like the scheduling and mappings of codes to architectures. For example, in big.LITTLE processors choosing the big or the small cores brings performance or energy savings, while in CPU-GPU systems it is the type of parallelism that will determine which processors works best.

Despite these problems, heterogeneity is increasingly exploited where performance is mandatory. A good example is the training and inference of large neural network models. Attaining human-like accuracy in tasks like image recognition, language translation or the game of Go [75] requires supercomputing level of performance (i.e. beyond petaflops) sustained for several days [19]. Lacking this level of performance would strongly constrain the development of these models, therefore the motivation to develop the accelerator architectures mentioned above. After acknowledging the increasing diversity in heterogeneous architectures, henceforth we focus on the heterogeneous CPU-GPU systems employed in this work.

2.2 Computer Cartography

Cartography, the study and practice of making maps, has developed dramatically with the invention of computers. The first maps were carved on stone, cut in wood, pigmented in animal skin and drawn in papyrus. Though rudimentary, they were critical for commerce, sailing, war and the development of civilizations overall. Their strategical value soon motivated the advancement of the mapping, drawing and printing techniques. However, while the implementation of maps improved, their format essentially remained physical and static. Recently, computers have enabled the virtualization of maps, turning them from physical to digital models. At the same time, remote sensing has accelerated the collection of spatial data around the Earth. Together, these technologies have revolutionized every aspect of cartography: from the creation, to the distribution, to the interaction with maps [4].

Figure 2.4 illustrates an antique physical map to the top and a modern digital counterpart to the bottom. The former map was produced more than a century ago with technical drawing and craftmanship. It possibly required earlier field work to obtain precise measurements of the urban geometries. The latter map is less than a decade old and was designed with computers and CAD (Computer-Aided Design) tools. Quite possibly, it only needed office work since the data can be conveniently derived from existing databases. Moreover, it can be visualized at multiple levels of detail and can be edited with precision. Most importantly, the digital format means that computers can now manipulate and analyze the data, enabling the automated discovery of new knowledge at a much faster pace that humans ever could. Next we expand this topic, discuss the analysis of raster data, and introduce the map algebra language.

2.2.1 Geographic Information Science

In a context where computers expand the boundaries of cartography emerges Geographic Information Science (GIScience) [50]. GIScience sits at the intersection of cartography, remote sensing, statistics and computer science, and is enabled by suites of computer software known as Geographic Information Systems (GIS). Information systems are designed to produce answers, deduce knowledge and support decision making. GIS does precisely so, but with a focus on Geography and other spatial-dependent sciences. It can answer questions like: *where is the most profitable place for opening a business?* Such questions are typically simple yet pose difficult analysis due to the many factors influencing them. For instance, the demand for a business depends on demographics, economy, accessibility, trends, marketing and more.

In its most basic form, GIS serves as a computer cartography toolset for the making of digital maps [64]. More advanced features include the query of attributes, reclassification of data, reconstruction of topologies, overlaying of datasets, or the connectivity analysis between points of interest (e.g. flats, schools, hospitals). Over time, GIS has become a full-fledged software for the storage, edition, analysis and presentation of spatial data, and is nowadays used to track and uncover spatial patterns, trends and relationships in numerous fields[13]. Some applications are the tracking of population growth, the study of traffic patterns, or the mapping of crime

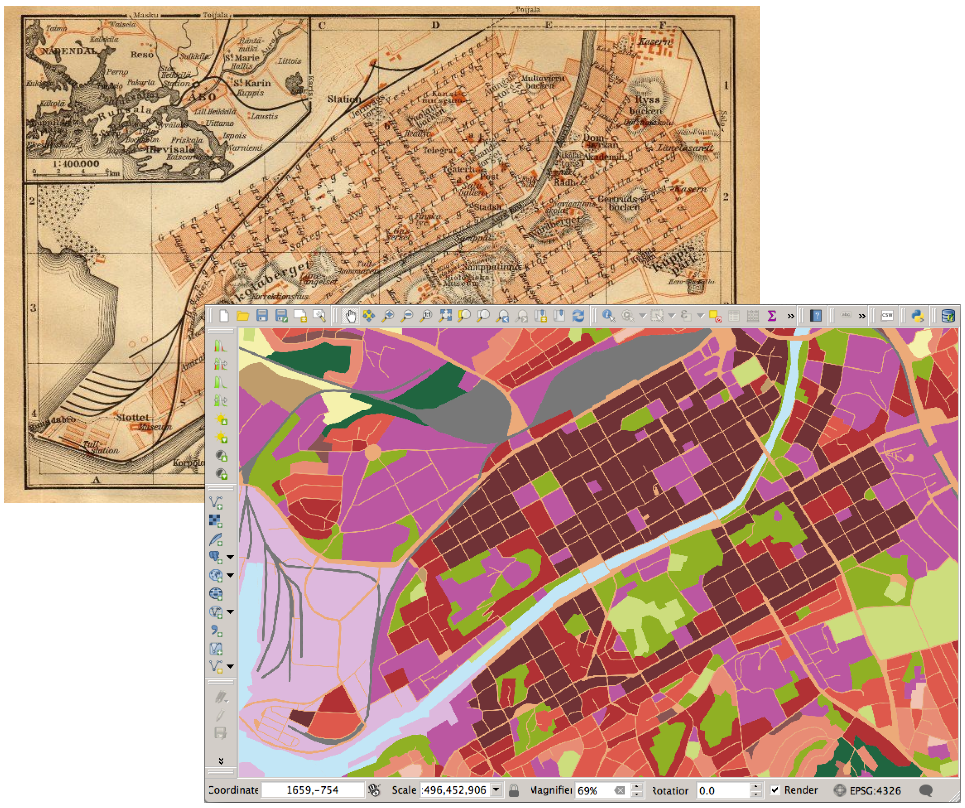

Coordinate 1659,-754 Scale 496,452,906 Magnifier 69% Rotation 0.0 Render EPSG:4326

Figure 2.4: Antique physical map (1912) and modern digital counterpart (2006) of the city of Turku / Åbo.

and diseases. At present, GIScience represents a very active area of research and new usages and functionalities continue to emerge.

GIS employs two data formats to model the space, the **raster** and **vector** formats. In the literature, rasters and vectors are also called fields[14] and objects[15] respectively [34]. Raster data is made of regular cells arranged as a rectangular grid, while vector data consist of points, polylines and polygons arranged in any order. Rasters are well suited for the continuous and smooth data typically found in the natural environment, whereas vector data is more appropriate for the discrete and pronounced features found in artificial structures. Thus, raster models store elevations (i.e. DEMs), rainfall, chemical concentration or land-cover, while vector models store entities

[13]GIS extends to numerous fields, including earth sciences (e.g. agriculture, geology, meteorology, hydrology), natural resources (e.g. oil, gas, forestry, mines), governmental (e.g. defense, intelligence, national security), public safety (e.g. health care, emergency planning, criminology), or urban development (e.g. land-use, real state, city planning).

like cities (as points), streets (as polylines) or regions (as polygons).

GIS is built on the principle of layering information. Datasets, of raster or vector type, are partial layers of reality that are stacked on a spatial basis. For that, datasets must be georeferenced[16], so that their features have well defined spatial positions. Figure 2.5 depicts this principle, where five layers compose a simple model of the real world. Say, for example, that we want to determine the travel distance from a point A to a point B in the model. Then we would need both the street and the elevation layers (since sloping streets present longer distances). Finding the elevation of a street after georeferencing is trivial, because the layers are spatially aligned. The layering principle is therefore fundamental for the analysis of raster data.

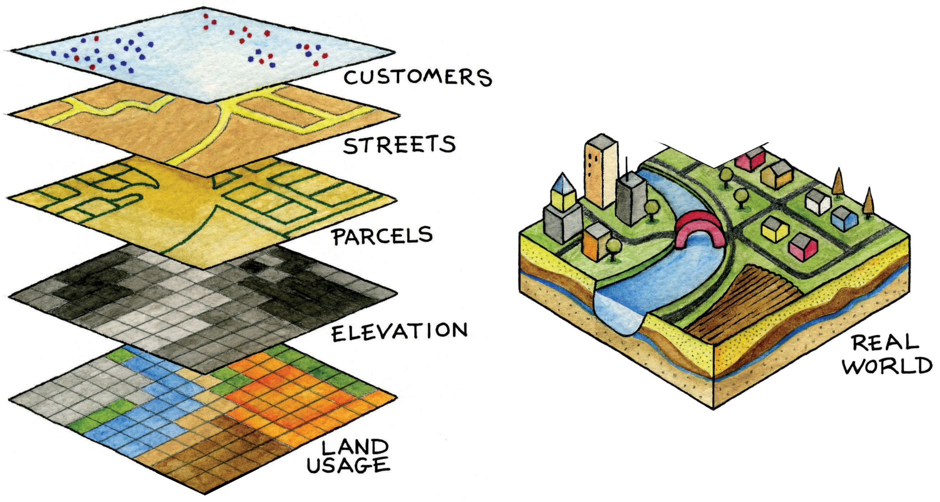


Figure 2.5: GIS layered model [4].

2.2.2 Raster Spatial Analysis

Spatial analysis predates GIScience and was first used in life sciences such as ecology, geology and epidemiology. A notable example is the London Broad Street cholera outbreak uncovered in 1854 by John Snow [52]. Though valuable on its own, it was the emergence of GIScience that made spatial analysis thrive, as GIS provides with the data infrastructure that is the foundation for the analysis. Today the true power of GIS lies in its analysis capability, which goes beyond its original cartographic role [50]. In this work

[14]Fields model natural, continuous attributes with varying values across space.

[15]Objects model man-made, size-limited features with concrete spatial coordinate.

[16]Georeferencing is the process of assigning real-world coordinates to each pixel/feature of a raster/vector dataset, so that they can be located.

we employ raster data, therefore we head the discussion toward raster-based spatial analysis.

Raster spatial analysis is the process whereby spatial-dependent problems are modeled with raster data, which is then processed on a per cell basis to derive new insights and understand spatial developments. For example, comparing the present and past version of a dataset cell by cell can uncover change. Inspecting the cells of a small neighborhood can reveal local correlations, distributions or variance. Collecting statistics (e.g. mean, count, frequency) of large regions of cells can expose underlying behaviors. More complex mathematical methods can also serve to derive distances, optimal routes or suitable locations. In general, spatial analysis covers any technique aimed to describe, explore, explain or optimize spatial phenomena.

Progress in raster spatial analysis is driven by the success of its applications in solving real world problems. A popular application is suitability analysis, in which multiple factors are overlaid to find optimal locations [51]. Examples are finding a highly accessible district for a hospital, or locating profitable areas for a business. Another application is proximity analysis, whereby geographical structures are evaluated to find well-connected areas, shortest paths across surfaces or optimal corridors between two locations [32]. Examples are the planning of transportation networks and the conservation of natural habitats. A third application is terrain analysis, in which topographic features are interpreted to explain the flow of water, infer the visibility of an area, or discover convenient routes along a landscape [56]. Figure 2.6 exemplifies a visibility analysis, made possible by the digitalization of Figure 2.4.

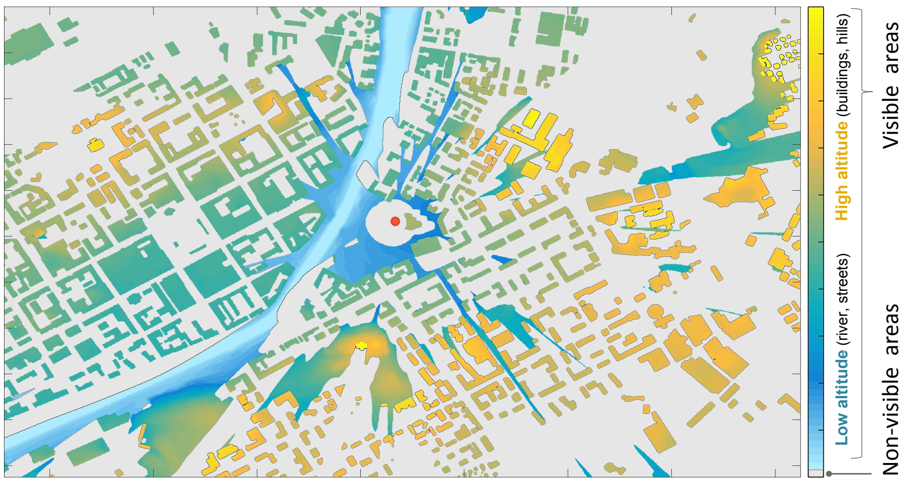

Low altitude (river, streets)
High altitude (buildings, hills)
Visible areas
Non-visible areas

Figure 2.6: Standard two-dimensional viewshed

Previous applications describe what are called static or steady state models. **Static models** have no time component, because they describe the state of a system in equilibrium. For example, terrain analysis employs static models because the topology of the Earth remains constant[17]. On the other hand, problems where time plays a principal role require dynamic or transient models. **Dynamic models** execute forward-in-time, where the next state of the system is based on the state at the previous time step. For instance, wildfire risk analysis [97] requires dynamic models because fire changes rapidly with time.

Raster analysis techniques, while more often employed for static modeling, can deal with dynamic models too. To do this, the techniques must be combined with general programming constructs like loops, branches and variables. Loops are needed to repeat the process equations and thereby simulate the advance of time in small steps. Branches enable the equations to express different behaviors depending on the runtime state of the process. Intermediate variables hold the current outputs of the equations in order to feed the inputs of the next iteration. Surprisingly, these additions turn raster spatial analysis into more than just an analysis tool. It becomes a powerful language for raster modeling and simulation.

2.2.3 Map Algebra

Map algebra was originally conceived as a mathematical formalism for cartographic modeling [81], and soon it became the standard language for raster spatial analysis. Map algebra was innovative and well received by the emerging GIScience community, who later developed analogous languages (e.g. R.mapcalc [73], PCRaster [90], GeoAlgebra [79], MapScript [66]) These packages extended map algebra with new features, most of them targeted at the modeling of dynamic processes. Nonetheless, they all derived from Tomlin's implementation [82] and followed similar algebraic principles. Eventually, the principles of map algebra became part of every GIS software [83] and today they are extensively used to perform from basic postprocessing to complex analysis.

Map algebra is a collection of raster primitives that are combined to perform spatial analysis and modeling. Its building blocks can be classified as objects (e.g. rasters, numbers, variables), operators (e.g. +,*,<,=), functions (e.g. sin, sqrt, max, slope) and grammatical rules (language structure). Objects store data that can be scalar or spatial, discrete or continuous. A special value of data is *null*, used to indicate missing, non-categorizable, or non-meaningful values. Operators cover the elementary assignment, arithmetic, relational and logical operations. Functions cover less common oper-

[17]Note that while the physical geography appears invariant on human time scales (e.g. days, years, decades), that is not the case on geological time scales (e.g. periods, eras, eons) where the Earth surface is steadily changing.

ations like trigonometrics, as well as key spatial functions, like slope. Grammatical rules define what is a legal expression and how expressions are combined together. See Figure 2.7 for one possible grammar proposed by Tomlin in [84].

```
NEWLAYER = LocalFUNCTION of 1STLAYER
  [and NEXTLAYER] ETC.

NEWLAYER = ZonalFUNCTION of 1STLAYER
  [within 2NDLAYER]

NEWLAYER = FocalFUNCTION of 1STLAYER
  [on SURFACELAYER]

Where LocalFUNCTION, ZonalFUNCTION and
FocalFUNCTION can take different forms
according to the functionality they perform.

NEWLAYER = FocalEXTENDED of 1STLAYER
  [at DISTANCE] ETC.
  [by DIRECTION] ETC.
  [spreading
    [in FRICTIONLAYER]
    [on SURFACELAYER]
    [through NETWORKLAYER]]
  [radiating
    [on SURFACELAYER]
    [from TRANSMISSIONLAYER]
    [through OBSTRUCTIONLAYER]
    [to RECEPTIONLAYER]]
```

Figure 2.7: Example of grammatical rules of a map algebra language [84]

Map algebra operations (i.e. operators and functions) are classified into four types according to their *spatial reach*[18]. **Local operations** apply a function for each cell in the input raster to generate an output raster of the same dimensions. This is done with spatial independence, meaning the output cell only depends on the input cell in the same spatial location. **Focal operations** compute output cells as a function of a neighborhood. Each output cell depends on a localized area of regular shape and small extent around its input cell. **Zonal operations** apply to large groups of cells within a common zone, like a watershed or a district. They require two input rasters, one defining the zones and other with the values to be operated on. **Global operations** consider the entire input raster when computing each individual cell of the output raster. They impose full spatial dependence, are typically complex and present irregular execution patterns.

While the four types of operations are important, some are more common than others. Local operations are pervasive across many domains, but especially abundant in optimization problems (e.g. suitability analysis), data preprocessing (e.g. atmospheric correction), and time series analysis (e.g. historical weather and rainfall). Focal operations are mostly used to find spatial derivatives, for example in surface analysis (e.g. slope, curvature), soil erosion estimation and wind direction analysis. Zonal operations almost exclusively serve as statistical tools, for example to assemble regional statistics in order to find outliers, underlying trends, frequency distributions et cetera. Global operations are the most unique and typically characterize the whole application, for example in proximity and cost-distance analysis,

[18]Spatial reach refers to the maximum spatial extent a map algebra operation might reach in order to compute the value of a single output cell.

hydraulic analysis and visibility analysis.

Lastly, it shall be noted that the classification of map algebra operations is not definite or rigorous. Certain works exclude the Global class, while others extended the Focal type beyond simple neighborhoods [84]. In our research we incorporated the **Radial** class, used for example to implement viewshed [9]. However, extending the classification could become overwhelming because quite many algorithmic patterns can be identified. As a result, the Global group is used as a miscellaneous class where to place these recurrent algorithms. More details can be found in papers 3,4 and in the literature [85].

2.3 Programming Languages and Compilers

The human brain has evolved to be robust to ambiguity and efficient in the reasoning of abstract concepts. Computers were designed to be rapid in the handling of binary information, but cannot handle ambiguity. As a result, humans talk flexible languages that developed naturally and evolved to be practical, whereas computers talk formal languages constructed on purpose and designed for programming simplicity. In a fraction of a second computers can input, process, and output large series of structured digital data, while humans take even seconds to decipher sounds or images, to reason at symbolic level, and to react with voice or actions. These cognitive and linguistic discrepancies pose a barrier to the conversion of human thoughts into computer programs [86]. The field of programming languages and compilers attempts to address such a communication gap.

Two pieces are necessary to bridge the gap (Figure 2.8): (1) programming interfaces that are free from ambiguity yet comfortably operated by humans, and (2) program translators that convert human orders into machine instructions. Programming languages address the first requirement. They define formal interfaces where to express programs. Thus, in the human-to-computer communication, languages are the medium while programs are the message. Compilers (and interpreters) meet the second requirement. They translate human orders into computer instructions. At the same time, compilers report errors in the programs and attempt to optimize them to execute faster. The next subsections outline these two subjects and the tradeoffs involved in their design.

2.3.1 Language Theory

When the von Neumann architecture made away with programming-by-rewiring, it sparked the development of programming languages [72]. The first generation of programming languages were *machine languages*, where

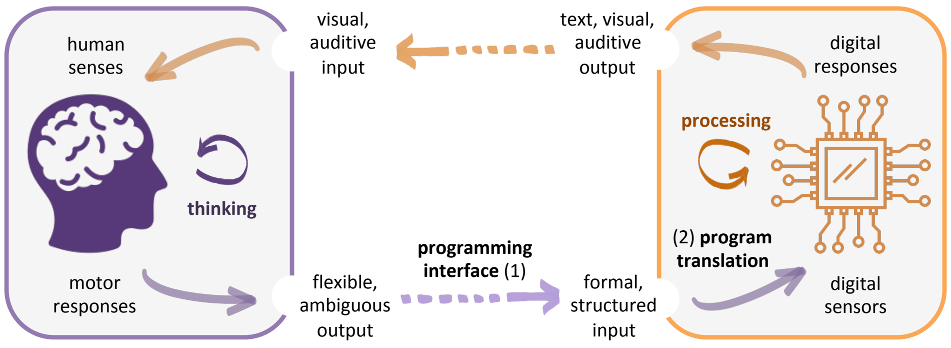


Figure 2.8: Programming interfaces (i.e. languages) and program translators (i.e. compilers) address the human-computer communication barrier.

instructions map directly to hardware. Machine codes are sequences of hexadecimal opcodes and data, and are remarkably arduous to read and write for humans. The second generation, called *assembly languages*, raised the level of abstraction and became more symbolic. Assembly codes abstract opcodes and addresses with mnemonics and labels in order to simplify the programming. The third generation, or *high-level languages*, brought machine independence and structured programming [17]. They support expressions, branches, loops, functions, and constructs to make the programming more human-friendly. Examples of early high-level languages are Fortran, Algol and Cobol, while more modern examples are C/C++, Java and Python.

In turn, the level of abstraction achieved by later generations enabled new programming paradigms and execution models. *Imperative* languages instruct the machine what to do and how. They dictate order, change state and allow side-effects. *Procedural* languages, an imperative subtype, group instructions into procedures to attain modularity and abstraction. *Object-oriented* languages, another subtype, encapsulate state and procedures into objects for further abstraction. On the contrary, *declarative* languages dictate their intent, but not how to achieve it. They disallow state, shared data and side-effects. *Functional* languages, a declarative subtype, state their intent as a succession of mathematical functions. *Dataflow* languages, a second subtype, declare programs as directed graphs where nodes produce data that flows across edges. Note that these are just a few examples, and we refer the interested reader to the literature [72].

Besides paradigms, another important distinction is that of general-purpose and domain-specific languages. *General-purpose languages* target a broad domain of applications, but are rarely the best tool for the job. They are flexible, multi-paradigm, complete[19], and often extensively used. Examples are C/C++ (for systems programming), Java (for mobile and web) and Python (for scripting and scientific codes). Conversely, *domain-specific* *languages* are tailored to some domain, which they serve very well [88]. They abstract the core concepts of the domain to bring closer the programming and thinking processes (Fig. 2.8). Examples are MATLAB for matrix operations, SQL for database queries, and LaTeX for typesetting documents.

Regardless of domain, all programming languages are described by their syntax (form) and semantics (meaning). The *language syntax* is the set of rules that specify the structure of a valid program. It defines what is a valid keyword, variable name, expression form, function call, loop structure, etc. As an analogy, the syntax of the English language defines the valid words and the structure of well-formed sentences. The *language semantics* is a second set of rules constraining the interpretation of well-formed programs. It ensures that variables are declared, accessed from valid scopes and only passed to compatible functions. Another English analogy is the sentence "Colorless green ideas sleep furiously" [13], which is syntactically correct but semantically absurd.

Before a high-level program can be executed it must be translated to machine code. There are two main strategies to do so, and a gradient of options in between. The whole program can be translated at once and later run by the user, in what is called *compilation* and execution. Alternatively, the program can be translated and executed statement by statement, in what is labeled *interpretation*. One midway option is to compile the program into bytecode, a portable intermediate form that is later interpreted. Another alternative is to interpret the whole program but compile critical parts as the code runs (i.e. just-in-time). The next subsection describes the compilation process and the design of a typical compiler.

2.3.2 Compiler Theory

A compiler is a computer program that translates code from a source language to a target language [15]. Note that this definition allows many interpretations for what "source" and "target" languages could be[20]. Here we refer, respectively, to high-level (i.e. readable by humans) and low-level (i.e. executable by computers) languages. Typical compilers are designed as a translation pipeline consisting of front-end, middle-end and back-end. From front to middle they analyze the source code, while from middle to back they synthesize the target code. Figure 2.9 outlines this process, where each end transforms the input and pass the output along the pipeline. Unlike interpreters, compilers analyze the code to infer the big picture and uncover possible optimizations. Analysis is a precondition for optimization, and code

[19]Computationally complete, or Turing-complete, languages can implement any algorithm there is through the combination of their primitive constructs.

optimization is the essence of compilation.

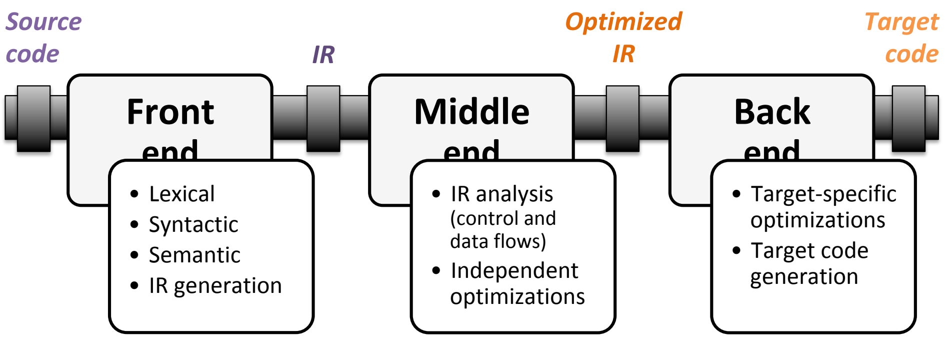


Figure 2.9: Compiler architecture, designed as a pipeline of transformations.

The *front-end* validates the input code according to the syntax and semantics of the source language. First, the lexical analysis scans the text for tokens (or words in the English analogy). This is done with regular expressions and preserves the linear structure of the code. Then, the syntactic analysis parses the tokens to build a syntax tree, where nodes represent language constructs. This is done with context-free grammars, which at the same time detect errors in the program form. Next, the semantic analysis fills a symbol table with the declared variables, functions and types. The table is then looked up to catch undeclared usages, incompatible types, accesses outside the scope, etc. Finally, the syntax tree is walked to generate an intermediate representation of the code, the IR.

The *middle-end* splits the compilation pipeline in order to decouple the source from the target language. Thereby, any source only needs to be translated into IR, while any target only needs to be derived from IR. This reduces the number of front-end and back-end implementations from $N \times M$ to $N + M$. The stage begins with the analysis of the IR. This step depends greatly on the domain of the compiler, but typical analyses will look into the IR's control-flow, data-flow, dependences and aliases. The IR also enables the refactorization of many target-independent optimizations out of the back-end. Examples of optimizations are dead code elimination, common subexpression elimination or loop-invariant motion. Importantly, the transformations must not alter the semantics of the IR.

The *back-end* synthesizes and optimizes the output code that will later execute on the target hardware. Synthesis and optimization can be done in an intertwined manner, like the *LLVM* compiler does [48]. When not intertwined, it becomes fundamental to rewrite the synthesized instructions with peephole optimizations. Important target-dependent optimizations applied

[20]Depending on the *source* and *target* languages, compilers are classified as source-to-source, bytecode, just-in-time, assembler, disassembler, decompiler, rewriter...

now are instruction selection, to choose optimal instructions per IR statement, instruction scheduling, to find optimal orders for those instructions, and register allocation, to optimally bind variables to registers. The backend normally generates assembler, which is later translated to machine code that processors understand. Additionally, the target code might need to be accommodated to a runtime before its execution.

Each and every programming language requires a *runtime system* to support their execution model. This is because high-level abstractions have no direct equivalent in the low-level instructions run by the machine, therefore it becomes necessary to maintain extra data structures that assist these functionalities at execution time. For example, automatic memory management requires a garbage collector to keep track of the allocated memory. Runtime systems vary in form and responsibilities. They can be as simple as in C/C++, where the compiler generates the runtime logic and integrates it with the code. On the other extreme, they can constitute the entire execution environment, like the Java virtual machine. Once the runtime is settled the code can be finally run by the user.

2.3.3 Design Tradeoffs

Designing programming languages is all about tradeoffs. All-round languages are impractical because maximizing one quality tends to minimize another. Productivity impacts performance, simplicity decreases versatility, safety impedes flexibility, writability reduces redability, extensibility conflicts with compatibility, et cetera. Language design is therefore an engineering problem. To maximize their overall purpose, languages must balance a large array of qualities. However, this balance of tradeoffs is not straightforward and it affects every aspect of the language.

At the interface level, the design should strive for programming friendliness. To do so, a first tradeoff is to resolve the application domain. Contained domains lessen the language complexity and allow more precise symbolic abstractions, but too narrow domains have limited applicability and could trap the software in a future dead-end. Another tradeoff is to choose a matching paradigm. For instance, imperative languages are versatile at expressing stateful algorithms, but functional languages are easier to optimize and parallelize with compiler techniques. Next in the list is a suitable execution model. Memory management, concurrency and exceptions all add complexity, but might be essential to certain domains. Lastly, syntax and semantics need proper balance too. Exotic features can provide good abstractions, but very unusual constructs will confuse programmers.

Regarding the compiler design, the goal is to efficiently manipulate and optimize codes. The foremost tradeoff in compilers is the design of the IR. Depending on the language purpose, the IR should target different class and semantic level. IRs can be *structured* as graphs or trees, attaining a representation closer to the source program (e.g. Clang AST). IRs can also be *linear* sequences of tuples, matching the assembly and machine code representations (e.g. LLVM IR). IRs composed of *stack* operations lay somewhere in the middle, providing conciseness and portability (e.g. Java bytecode). Additionally, these classes of IRs can be designed at high, medium or low semantic levels. At a high semantic level the operands work with abstract data types objects, like arrays and structs. The medium level works with registers and memory, while remaining independent from particular machines. At low semantic level the operands are very close to the target language, almost like x86 instructions. Note that compilers are not restricted to one single IR form. Combining multiple IRs along the pipeline facilitate the discovery of optimizations, but at the same time increases the compiler complexity considerably.

At the ecosystem level, the design should promote the diffusion of the language. On the one hand, standalone languages employ their own compiler/interpreter and development toolchain. This provides maximum freedom, but requires an active community backing their progress. If the community is inactive or divided, the language loses momentum and lags behind its competitors. On the other hand, small languages can be hosted into the ecosystem of a standalone language. One way to do so is with a library (although it is arguable whether to consider that a language). Libraries are easily integrated across platforms, but are limited in syntax and control-flow[21]. Additionally, a middle ground option is to embed a DSL into a powerful GPL with metaprogramming support[22]. Certain languages provide algebraic data types, first-order functions and even access to the abstract syntax tree. Embedded languages can exploit these features to adapt syntax, semantics and control-flow to their will.

A last important tradeoff, and a topic of increasing relevance, is parallelism. In domains where performance is vital, languages must adopt designs that interact graciously with parallelism. Again, there are multiple routes to address the parallel issue: from low-level control, to abstracted constructs, to automatic methods. System languages like C/C++ expose the parallel hardware to the users and provide them with low-level access. While this offers maximum control, it is remarkably difficult to write codes that are parallel and correct. Modern languages like Go provide lightweight threads, communication channels, concurrent actors and others parallel constructs. These abstractions interface and facilitate the parallelism, while protecting

[22]Libraries are interfaced with functions, hence their syntax is restricted to sequences of calls and cannot seamlessly employ control structures like branches or loops.

[22]Examples of metaprogramming languages are C++, for its templates, Python, for its inspection features, and Haskell, for its algebraic data types and higher order functions that enable the deep embedding of languages.

users from typical concurrency pitfalls. DSLs can augment their domain abstractions with intrinsic parallelism in order to free programmers from this effort. For example, a linear algebra library or DSL can run matrices operations in parallel without users ever knowing.

Chapter 3

Methods: A Compiler Approach to Map Algebra

"Trying to outsmart a compiler defeats much of the purpose of using one"

~ **Kernighan and Plauger**,
in The Elements of Programming Style

Chapter 2 overviewed the three background fields upon which the research builds. This chapter presents the main contribution of the thesis: *a map algebra compiler that solves the PPP tradeoff for raster spatial modeling.* To that end, we first explain the limitation of interpreters for large raster datasets, then introduce the compiler architecture and the roles of its parts, next describe the spatio-functional decomposition of the execution, and finally present the code optimizations at several hierarchical levels. The following sections cover these four topics.

3.1 The Pitfall of Interpreters

Map algebra has become ubiquitous. Over the years, the map algebra functionalities has been integrated across GIS software to the extent that its borders have blurred and disappeared. This success derives from a design conceived by and for modelers, which abstracts the spatial semantics to simplify the programming. While productive, map algebra cannot be said to be performant in face of the data escalation. As the resolution of spatial data doubles, the number of cells to process multiplies by four. As new spectrums are collected, more detailed equations requiring extra computation are possible. Consequently, the performance becomes insufficient to quantify, calibrate and validate large spatial models.

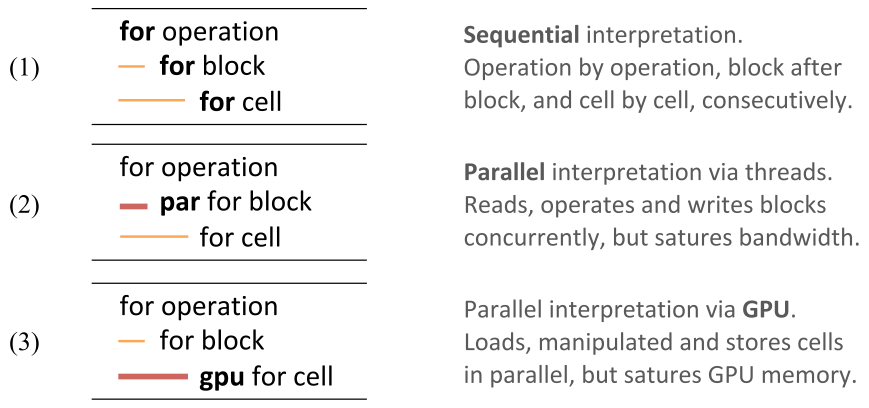
With the shift toward parallel and heterogeneous computer architectures (section 2.1), performance can mainly be obtained through parallel and specialized codes. Several works have attempted the parallelization of map algebra in different parallel systems. For instance, some studies opted for distributed systems via the message-passing interface (MPI) [44, 35, 11, 69]. Other works focused on shared-memory systems via OpenMP or other parallel abstractions [74, 69]. Lastly, some studies investigated heterogeneous systems combining CPUs and GPUs [95, 76, 69]. However, despite multiple attempts, the success of this line of research has been moderate.

Even if parallel, the unforeseen cost of moving data limits the performance of raster operations. In some cases, the performance was constrained by the I/O operations with disk [35]. In other cases, the communication between processors played a limiting role [11]. In some works, the finite memory bandwidth saturated the performance [95, 74]. Lastly, the data transfers from CPU to GPU memory affected some approaches [76]. This reinforces the hypothesis that parallelism is not enough. Performance is also influenced by how far data has to travel, i.e., by *data locality*.

Traditional map algebras neglect data locality because they are implemented as **interpreters**. They process scripts in order and execute operations one by one. Thereby, they never start the next operation before finishing the current one, and only attempt to optimize one operation at a time, if they optimize at all. They also ensure that the result of every operation is immediately consistent on memory. For map algebra this means that operations always load, manipulate and store the full rasters. Therefore, at every step of the execution the data is constantly being moved in and out from the storage. If the raster is too large to fit into memory, that storage will be a slow disk or an even slower network storage.

Figure 3.1a.1 illustrates the interpreter approach. Note that to enable the processing of datasets larger than the size of memory, rasters are decomposed into blocks. Thus, the interpretation of map algebra scripts consists of a triple loop structure where operations, blocks, and cells are iterated in sequential order. For every operation, input rasters are moved from disk to main memory, on a block-by-block basis. For every block, cells are moved from main memory to the processor registers and are operated on. At the same time, the resulting output cells are moved into new blocks residing in memory. Similarly, these blocks are moved down to disk, where the complete output raster resides. Once the operation has finished, the interpreter moves on to the next operation and repeats. As a result of this scheme, most of the execution time is spent waiting for data to travel from disk, to CPU/GPU memories, to registers, and back (Fig. 3.2).

Figure 3.1a.2 shows the addition of coarse-grain parallelism through the use of multi-threading. This is common in the literature [35, 11, 69], as it enables the processing of blocks in parallel. Now the blocks can be read, com-

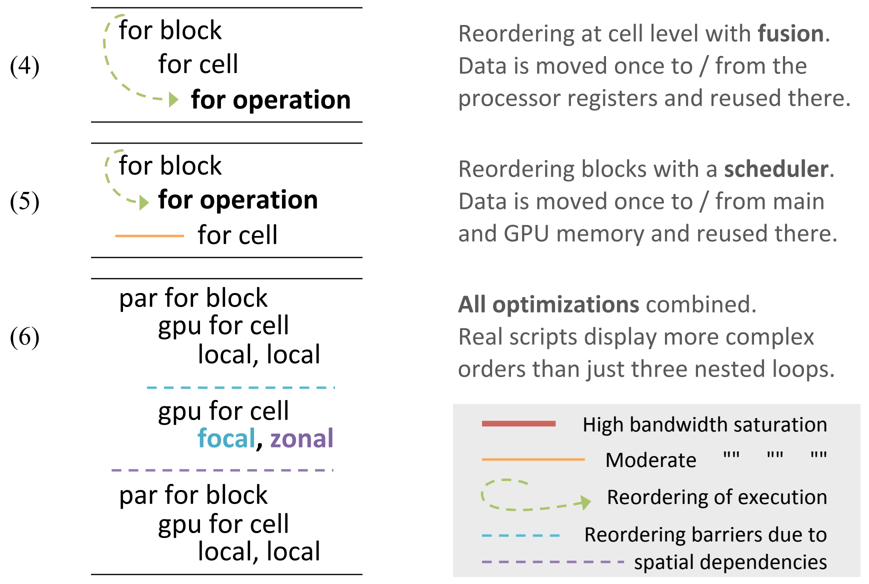


Sequential interpretation.
Operation by operation, block after block, and cell by cell, consecutively.

Parallel interpretation via threads.
Reads, operates and writes blocks concurrently, but satures bandwidth.

Parallel interpretation via **GPU**.
Loads, manipulated and stores cells in parallel, but satures GPU memory.

(a) Execution strategy of a map algebra **interpreter**.



Reordering at cell level with **fusion**.
Data is moved once to / from the processor registers and reused there.

Reordering blocks with a **scheduler**.
Data is moved once to / from main and GPU memory and reused there.

All optimizations combined.
Real scripts display more complex orders than just three nested loops.

(b) Execution strategy of a map algebra **compiler**.

Figure 3.1: Interpretation vs. compilation of map algebra workloads.

puted and written concurrently, bringing immediate performance benefits. However, there is only a limited amount of disk's bandwidth, which saturates soon and stagnates the performance. Other works also employ GPUs [95, 76, 69] to add fine-grain parallelism to the cells loop (Fig. 3.1a.3). Unless the PCIe transfers from CPU to GPU becomes a bottleneck, this brings performance benefits too. However, the GPU memory bandwidth saturates again and the GPU threads mostly wait idle. Although parallel, both solutions are still interpreters that neglect the locality between raster operations, since the data produced by one operation is often consumed by the next, ergo moving it is unnecessary.

Unlike interpreters, our map algebra **compiler** optimizes across raster operations. For that, it gathers a global view of the whole script before the execution. Knowing the operations and their interrelationships, it plans an execution order with better data locality. In this way, most raster data can be reused before being evicted from their memory level, leading to fewer data movements throughout the hierarchy, hence requiring lower bandwidth. Like the parallelism, this reordering technique can occur at cell and block level, too. The first variant is called *fusion*, while the latter is achieved with a locality-aware *scheduler*.

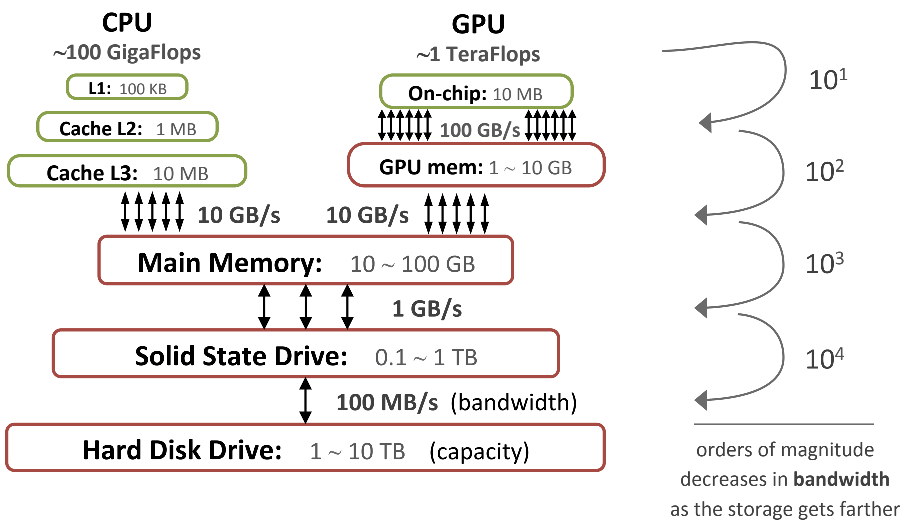


Figure 3.2: Memory hierarchy and its bandwidth in a typical heterogeneous system with a multi-core CPU and a many-core GPU.

Figures 3.1b.4 and 3.1b.5 show ideal situations where all operations can be reordered by one single loop transformation. However, this is rarely the case for real-life map algebra scripts. Mixing multiple operations together with loops and branches restricts the possible orders in complex ways. This result is an out-of-order execution that cannot be modeled with three loops anymore. Figure 3.1b.6 shows the complexities of the reordering problem for a slightly more complicated script. For example, focal operations present small barriers in their input direction that can disable fusion, while zonal operations create larger barriers in their output direction that also inhibit the scheduler.

Fusion and the scheduler are important optimizations enabled by the compiler approach, but not the only ones. Section 3.4 discusses the four main groups of optimizations. The next section introduces the compiler approach and details its architecture.

3.2 Architecture of a Map Algebra Compiler

Previous section has motivated the need of compilation (as opposed to interpretation) in order to achieve a performant map algebra that mitigates the memory bottleneck. Recall from subsection 1.2.1 that to maximize performance we need to break the PPP tradeoff, otherwise attaining performance reduces the portability, which diminishes the productivity, et cetera. This is done with a compiler architecture that splits the programming interface at the *front-end* from the program execution at the *back-end* via an intermediate representation at the *middle-end*. Additionally, there is a *runtime system*, related but decoupled from the back-end, that guides the execution. The roles of these four components are summarized in Table 3.1 and illustrated in Figure 3.3. After a brief advance below, the four next subsections describe these components in detail.

	Front-end	**Middle-end**	**Back-end**	**Runtime**
Component Function	User Interface	Optimization Framework	Parallel Execution Model	
			Programming Model	Runtime System
Program Paradigm	Imperative	Functional	Parallel	Concurrent
Linguistic Level	Source Language	Intermediate Language	Target Language	Executable Language
Software Quality	Productivity	Portability	*Static* Performance	*Dynamic* Performance
Implementation	Python DSL	Graph IR	Tasks, OpenCL	Work Pool

Table 3.1: Components of the compiler architecture.

The front-end acts as the user interface (Figure 3.3a). With it, modelers write sequential map algebra scripts as Python code. The scripts follow the imperative model, with variables and statements executed in order. The interface is expressive and versatile, resembling tools like MATLAB and NumPy. This first component provides a productive development environment.

The middle-end serves as a generic framework where to optimize the scripts (Figure 3.3b). To get there, the map algebra operations are parsed into a functional graph-based IR. The IR is independent from the scripts while also lossless in their representation. Here the IR is transformed with machine-independent and domain-specific optimizations. This second component enables the portability of codes to different architectures.

The back-end is the entry point to a parallel execution model (Figure 3.3c). It exploits the task and data types of parallelism via the OpenCL and a task programming models. For that, the IR is translated into executable tasks and these are spatially decomposed into jobs. Jobs are specialized for algorithmic patterns (e.g. map, reduce) and architectures (e.g. CPU, GPU). This third component brings static performance, i.e. it makes decisions ahead of execution for better performance.

The runtime is the active part of the parallel execution model (Figure 3.3d). It is in charge of orchestrating the jobs generated by the back-end, and to do so it spawns worker threads that concurrently request jobs from a work pool, resolve their data dependencies, acquire memory entries from a cache, perform the necessary I/O, issue kernels for execution, notify the dependent jobs, and repeat. This last component brings dynamic performance, i.e. it makes decisions at execution time for better performance.

3.2.1 Front-end: Python DSL

The first component of the map algebra compiler is the front-end. The front-end consists of a productive interface through which users express their programs. This is the only part of the four layers exposed to the users. Therefore all configurations, equations, data and other commands are inputs to the front-end. Similarly, the errors messages, simulation results, visuals and other outcomes are output via the front-end. When users run their models, the front-end first verifies the program for static errors, and only when the program is semantically correct, it is translated into IR and passed to the middle-end. The goal of the front-end is to support and stimulate the early stages of the modeling loop (Fig. 1.7), where modelers iteratively design, implement and test their models until they work.

In our prototype, the front-end is implemented as an **imperative spatial DSL embedded in Python** (Fig. 3.3a). This design aims to:

– Focus on the *modeling*, abstract the implementation.
– Enable the *structured*, modular and hierarchical design of models.
– Maximize *usability*, so that modeling becomes intuitive and easy.
– Aid the *integration* of models into larger ecosystems and ensembles.
– Assist the *testability* of models by tracking and reporting errors.

Spatial semantics and modeling focus

The spatial DSL is founded on the map algebra formalism introduced in subsection 2.2.3. This design facilitates the programming by abstracting spatial concepts like the raster datatype. For instance, Local operation act on full rasters without having to explicitly iterate the cells with a loop.

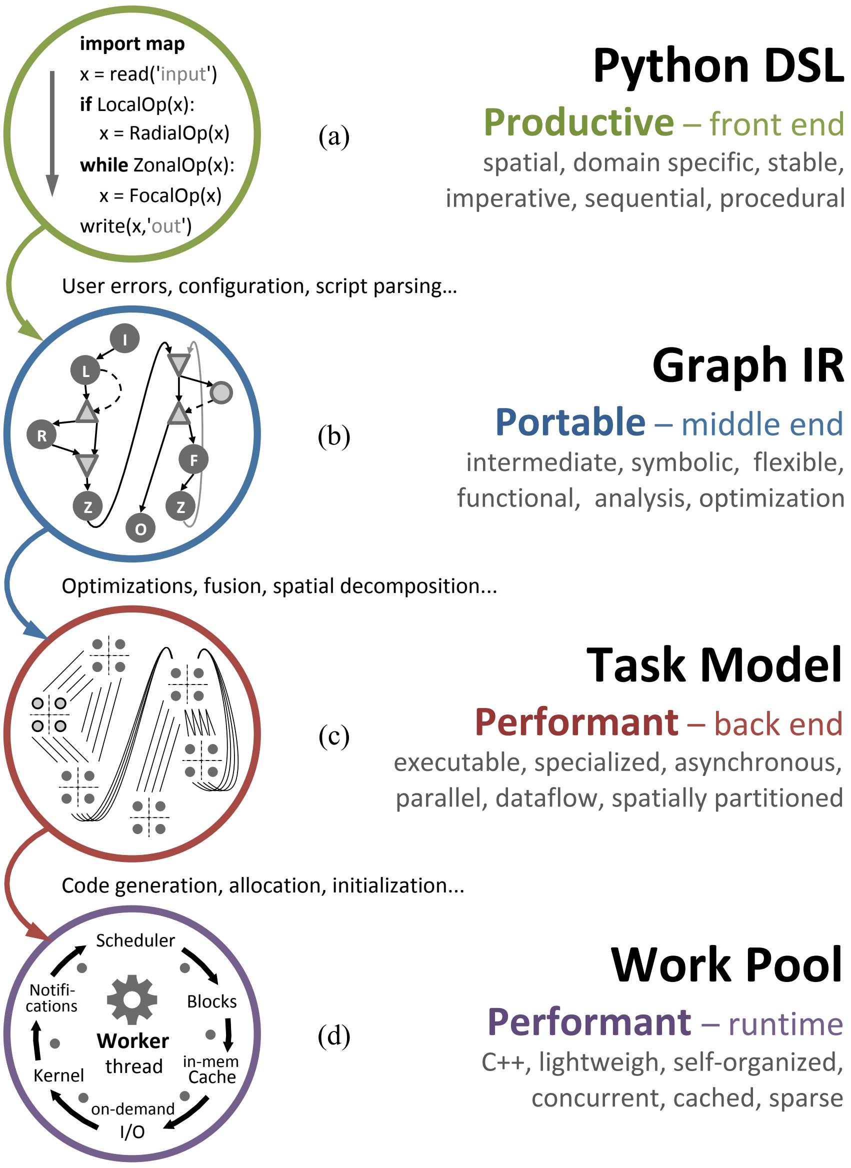

import map
x = read('input')
if LocalOp(x):
x = RadialOp(x)
while ZonalOp(x):
x = FocalOp(x)
write(x,'out')
(a)
Python DSL
Productive – front end
spatial, domain specific, stable, imperative, sequential, procedural
User errors, configuration, script parsing...
I
L
R
Z
O
F
Z
(b)
Graph IR
Portable – middle end
intermediate, symbolic, flexible, functional, analysis, optimization
Optimizations, fusion, spatial decomposition...
(c)
Task Model
Performant – back end
executable, specialized, asynchronous, parallel, dataflow, spatially partitioned
Code generation, allocation, initialization...
Scheduler
Notifi-cations
Blocks
Worker thread
Kernel
in-mem Cache
on-demand I/O
(d)
Work Pool
Performant – runtime
C++, lightweigh, self-organized, concurrent, cached, sparse

Figure 3.3: Architecture of a map algebra compiler.

Defining and accessing neighborhoods and spatial zones in Focal and Zonal operations is also straightforward. Even complex algorithms that would take dozens of lines of code are embedded into Global operations. Moreover, these operations combine with each other without breaking the spatial semantics.

The interface is devised to be productive, not performant. Therefore it hides all detail regarding parallelism (e.g. threads, intrinsics), architectural features (e.g. memory hierarchy), abstract data types (e.g. graphs, queues) or low-level logic (e.g. I/O, memory management). Instead, it provides spatial objects (i.e. rasters), spatial operators (i.e. map algebra) and basic functionalities for composing models (i.e. read, write, loops, branches, functions, import). This design narrows the domain and therefore reduces the programming complexity. For example, the lack of parallel and heterogeneous semantics frees the user from the programming issues described in subsections 2.1.2 and 2.1.3.

Imperative and structured programming

The DSL follows the imperative programming paradigm. Thereby, scripts are composed of sequential statements that run separately, one after another. Statements can be expressions, control flow, function definitions or data I/O. Expressions assign the result of an operation or function to a raster variable. Control flow permits branching, i.e. selection one of two code paths, and iteration, i.e. returning to a previous point in the code. Functions encapsulate multiple statements and need to be defined at some point before their usage. I/O operations communicate with the external world to take rasters in or let them out.

The imperative design endows map algebra with structure. For instance, functions behave like modules that can be reused to compose large models in short code spans. Doing this hierarchically, i.e. functions calling functions, creates multiples levels of abstraction. An example is an urban development model composed of smaller traffic, real estate and population modules. Each of the small modules can later be reused in other models or extended into a larger, standalone model. Moreover, the hierarchical structure permits to quickly activate, deactivate or replace any of the modules. This imperative, procedural and structured organization of the code facilitates its design, development and maintenance.

Python interface and ecosystem

The first reason for Python is its philosophy: simple, concise and readable. Python provides an intuitive interface that is easy to grasp for new and hobbyist programmers. It is well documented, supported by a large online community, and actively discussed in the forums. Being a dynamic language, models can be compacted into concise scripts that require little more than one file. This dynamism makes the language customizable too, as most Python constructs can be redefined during execution. Its retrospection capabilities provide powerful mechanisms to create expressive embedded DSLs. For instance, Python can access, modify and execute the abstract syntax tree of the currently running program.

The second reason for Python is its ecosystem, because computer models rarely work in vacuum. Usually they are integrated into larger software suites, like a decision-making supporting tool. Python has become a universal language that runs almost everywhere, which eases the integration. In addition, the most prevalent spatial packages already support Python, like in the case of ArcGIS, QGIS, GRASS and other geographic information systems. This enables a workflow where other tools generate inputs that feed a map algebra script, and the other way around, where the results of a script are passed to other spatial packages.

```python
from map import * # "Parallel Map Algebra" package

def pitFill(dem,stream):
    acti = stream or border(dem) # streams/ borders are active
    init = con(acti, dem, +inf)  # init = acti ? dem : +inf
    ngbh = [[1,1,1],[1,0,1],[1,1,1]] # spatial neighborhood
    orig = dem  # original dem, not modified in the loop
    ceil = init # minimum filling ceiling, updated in the loop

    while zonal_or(acti):           # loops until done
        minc = focal_min(ceil,ngbh) # min. ceiling in the ngbh
        maxc = local_max(minc,orig) # never lower than original
        acti = maxc < ceil          # activate lowered cells
        ceil = maxc                 # updates the ceiling
    return ceil                     # returns the filled dem

dem = read('elevation.tif')      # digital elevation model
stream = read('streams.tif')     # water streams (e.g. rivers)
fill = pitFill(dem,stream)       # calls 'pit filling' function
write(fill,'filled.tif')        # writes the filled dem to disk
```

Figure 3.4: Sample map algebra script of a pit filling algorithm [65].

Sample map algebra script

Figure 3.4 lists the map algebra script of a pit filling algorithm based on Planchon [65]. Pits are local minima elevation cells of natural or artificial origin that are present in most DEMs. Pits trap the outflow of surface water, preventing the application of common water flow algorithms. Pit filling produces depressionless DEMs, where all cells belong to a monotonically decreasing path. As a result, any surface water simulated on the DEM can always flow toward some stream or border point.

Figure 3.5a shows a one-dimensional DEM with three exit points and three pits in between them. Planchon's algorithms proceeds inversely to the flow of water, from exits to hills (Fig. 3.5b). As it advances, it keeps a minimum ceiling elevation to which it raises all pits it finds (Fig. 3.5c). At the end, all pits are minimally filled so that water can flow toward its natural exit (Fig. 3.5d). We refer to paper 4 for an extended explanation on how the script implements the algorithm.

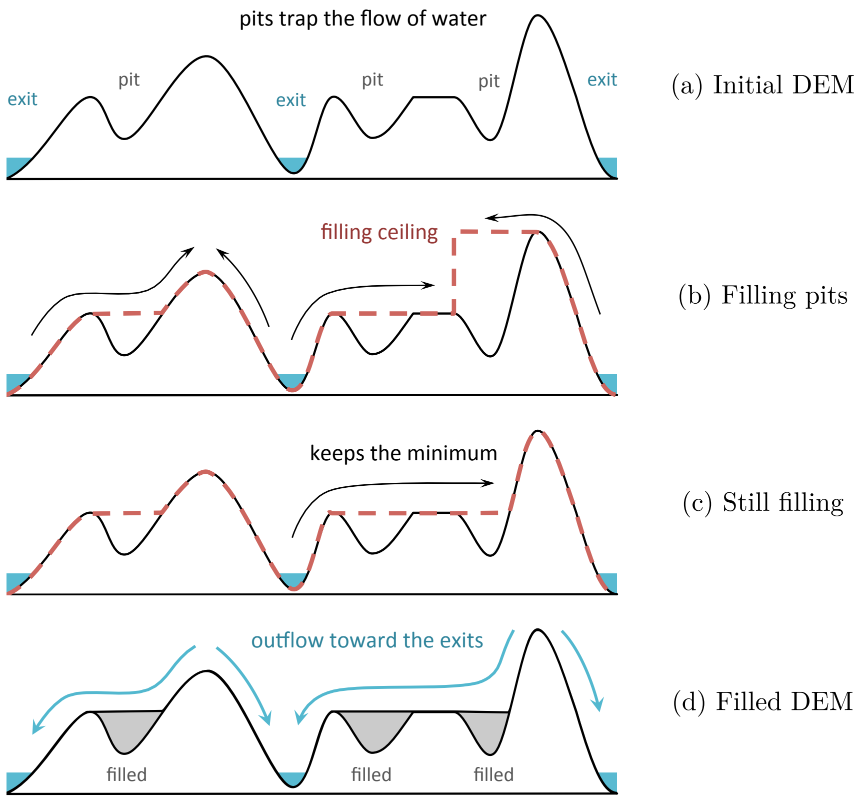


Figure 3.5: One-dimensional illustration of the pit filling algorithm.

Running the Python code

Running the Python code initiates the translation from Python code to graph-based IR. Before the IR is handed to the middle-end, the code is checked for errors. Syntactic errors, like incorrect calls to functions or use of uninitialized variables, are handled by Python. Semantic errors, like wrong type of arguments or incompatible raster dimensions, are captured by the DSL. Some runtime errors, like reading or writing to protected files, are detected before the actual execution. Other runtime errors, like out-of-memory accesses, are not possible because of the language restrictions. The remaining runtime errors will occur during execution and halt the process.

The map algebra models are translated by a combination of *running* and *parsing*. To do this, the Python operators are overloaded to create and connect IR nodes as they run. In this way, running the script simply accumulates a computation graph (akin to a syntax tree) that gives the compilers a big picture of the operations, variables and their connections. However, when control flow is involved the non-taken branches will never execute. Python introspection capabilities grant access to the bytecode and syntax tree of the script, through which the problematic control flow structures can be walked and parsed into IR nodes. Finally, when an I/O operation attempts to output some variable to the external world, the front-end forwards the accumulated graph to the middle-end to beginning the execution.

3.2.2 Middle-end: Graph IR

The second component of the compiler approach is the middle-end. The middle-end provides a framework where to apply portable optimizations to the scripts. The middle-end is the host of an intermediate representation, a transitional language that connects the source language, i.e. Python script, with the target language, i.e. parallel code. The IR must be well-defined and agreed-upon, so that different front-ends can generate similar, unambiguous IR. After the optimization, the IR is given to the back-end to generate parallel code that the underlying machine can run. The goal of the middle-end is to decouple the modeling interface from the actual implementation, so that models become portable and the modeling loop is not disrupted when moving to newer hardware.

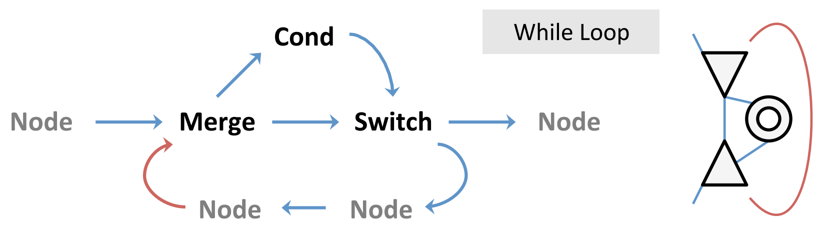
In our prototype, the middle-end consists of a **functional graph-based intermediate representation** (Fig. 3.3b). It is designed to be:

- *Independent* from source and target languages, to achieve decoupling.
- *Accurate* yet minimal in the representation of map algebra scripts.
- *Extensible* to new functionalities, yet stable and backward compatible.
- *Flexible* to manipulate, while supporting rich transformations.
- Suitable for machine-independent, domain-specific *optimizations*.

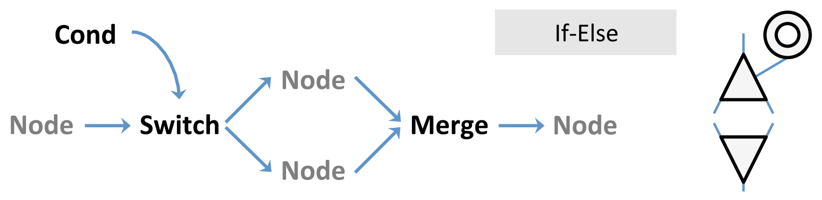
Graph IR and dataflow model

The graph representation belongs to the high-level, structured type of IRs (subsection 2.3.2). This type of IR is close to the source language and well-suited for domain-specific optimizations. The graph is directed, meaning that edges only flow in the forward direction. It is not acyclic, because cycles are necessary to capture loops in the script. Nodes in the graph represent raster operations, while edges model the flow of raster data. Nodes carry their class, type, metadata, statistics, spatial reach, parents, and descendants. Edges need no tags, because their configuration is determined by their unique source node. This IR is simple and easy to manipulate, yet accurate in the representation of map algebra codes.

Unlike most compilers that model data flow with basic blocks[1] and control flow with CFGs[2], we express both flows with a single flat graph, similar to dataflow models. The flow of data is intrinsically captured by the shape of the graph, whereas the flow of control is explicitly modeled with special nodes. The Merge node combines two streams of data into one, the Switch node reroutes a stream in one of two directions, and the Condition node determines the direction of a Switch. Together, these three special nodes serve to express branches and loops (Fig. 3.6).



(a) While Loop = Merge + Condition + Switch



(b) If-Else = Condition + Switch + Merge

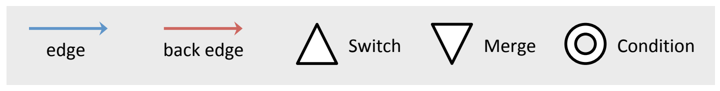


Figure 3.6: Switch, Merge and Condition nodes used to model control flow.

Functional SSA form

The graph IR is structured in SSA[3] form. Thereby nodes are defined once and never modified. The SSA form combined with the dataflow style endows the IR with functional proprieties. Nodes are pure functions (i.e. stateless, side-effect free) that transform and forward data. Such a functional scheme has no variables, sequential statements, or explicit order of execution. This makes the IR very suitable for parallelization, reordering and other optimizations.

[1]A basic block is a sequence of machine instructions with single input and output, so that the sequential instructions always execute together.

[2]A control flow graph (CFG) is a representation of a program where nodes are basic blocks and edges model all the possible directions of the flow during execution.

[3]Static Single Assignment (SSA) is a property of intermediate representations whereby variables are assigned only once to remove ambiguity and simplify the analysis.

On the other hand, the functional paradigm of the IR contrasts with the imperative style of the DSL. Variables in the script do not correspond to nodes in the graph. Instead, they behave like tags. The map algebra operations create the nodes, while the assignment operator tags the node with the variable name. If the variable is reassigned with a different expression, the tag is moved to that new node. Note that the IR is hidden from users by default, therefore this mismatch causes no confusion.

Sample intermediate graph

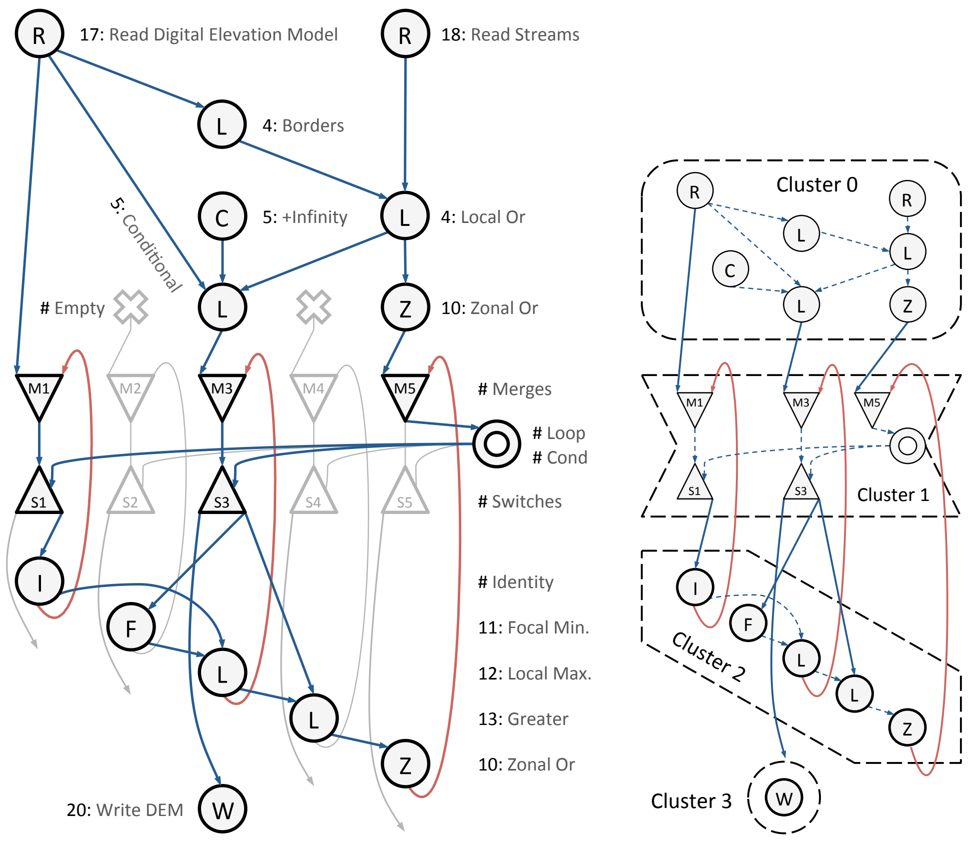
Figure 3.7a illustrates the IR after parsing the pit filling script in Figure 3.4. In the figure, nodes are tagged with their line of code and type of operation to their right. The special nodes have no line because they do not correspond to operations in the script. Nodes found in at least one cyclic path between a Switch and a Merge are called loop nodes. Edges flowing from a loop node to a Merge node are called back edges. Only back edges can create cycles. Every loop node needs a Switch and a Merge node connected to each other. Every Switch links to the Condition. If a loop node has no father outside the loop, an Empty node is created and connected to its Merge. Constant variables, i.e. used but not updated in the loop, require an Identity node. While the Condition node is true, the Switches feed the loop nodes through their true side. When the condition fails, the loop stops and the Switches feed the nodes in their false side.

Optimization framework

The IR provides a framework where to apply machine-independent and domain-specific optimizations. Optimizations are transformations in the graph that speed up the execution while preserving the results. The graph is transformed with forward, backward or complex (i.e. no strict direction) passes. Examples of machine-independent forward and backward passes are presented in paper 4. Fusion, described in subsection 3.4.2, is the best example of domain-specific complex pass. Fusion groups the IR into clusters of nodes with compatible data dependencies. The clustering rules are described in paper 3 and implemented in the code repository [5]. After fusion the IR becomes two-level graph of clusters of nodes, and it is given as input to the back-end.

3.2.3 Back-end: Task Model

The third component of the map algebra compiler is the back-end. The back-end contains the programming model that serves as entry point to the parallel execution model. The programming model abstracts the computer architectures to ease the development of the parallel codes. The execution



(a) Intermediate Representation (b) Clusters

Figure 3.7: Pit filling IR (a), merged into clusters (b) after fusion.

model defines traits like the unit of work, the order of execution, or the levels of parallelism. As the back-end receives the IR from the middle-end, it first decomposes the spatial domain, then generates target parallel code, and last compiles the target to executable code that the runtime will run. The goal of the back-end is to effectively map the operations to the architecture, so that they execute fast and do not delay the quantification, calibration and validation stages of the modeling loop (Fig. 1.7).

In our prototype, the back-end consists of a **parallel task model over OpenCL skeletons** (Fig. 3.3c). It is designed to:

– Focus on the *implementation*, not on the modeling.

– *Decompose* the spatial domain to expose its parallelism.

– Identify and utilize the freedom in the *order of execution*.

– Exploit the multiple levels of *parallelism* in modern architectures.

– Employ the specialized architectural features of *heterogeneous* devices.

Parallel skeletons

Skeletons are code templates that encapsule classes of recurrent algorithms with broad applicability [14]. Users simply need to configure the parameters exposed by the skeleton, which then generates the algorithmic code. Parallel skeletons are, therefore, templates that encapsulate classes of recurrent parallel algorithms [33]. They avoid the coding of repetitive implementation details and favor the parallel analysis of the problem. Parallel skeletons can be parameterized by granularity, processors number, topology or other parallel criteria. For example, a finite difference[4] skeleton will generate code out of some differential equations, and a parallel version of such skeleton will also handle the communication between the parallel processors.

In this work, the classes of algorithms to skeletonize coincide with the map algebra classes (subsection 2.2.3). Local operations follow the map pattern, Focal operations are stencils, Zonal operations apply reductions, Radial operations are 2-dim scans and Global operations exhibit their own algorithmic patterns. Whereas the parallel analysis of non-Global operations is straightforward, their parallel implementation is laborious. Depending on the architecture, the sizes of data and memory, the different heterogeneous devices and other details, the amount of boiler-plate code that needs to be written can be drastically large for a rather simple functionality. It is for this reason that skeletons are a good fit for a map algebra compiler.

OpenCL kernels

OpenCL is the open, royalty-free standard for cross-platform, parallel programming of heterogeneous processors. It enables the portable programming of a range of devices: CPUs, GPUs, FPGAs, DSPs, VPUs, IPUs and other accelerators. Compute Kernels lay at the core of OpenCL. Kernels are data-parallel routines intended for high-throughput processors. An OpenCL kernel is a program based on extended-restricted C99 that is run by a multi-dimensional grid of workers. Workers share an off-chip global memory and execute a single stream of operations in parallel, with no particular order. Workers cooperate in small blocks that share an on-chip local memory and present limited synchronization capabilities.

OpenCL is not portable from the performance point of view. Consequently, attaining high-performance often requires specialized codes for each of the parallel devices. For instance, the OpenCL local memory maps to on-chip scratchpad memory on GPUs but to the off-chip main memory on CPUs. Because these memories have different profiles, optimizations that improve performance for GPUs may reduce it on CPUs. In this work we test

[4]Finite difference is a method for the numerical solution of differential equations by discretizing the domain into units that are approximated with derivatives.

CPU-GPU systems, and thus require multiple skeletons to generate kernels for both devices. Although OpenCL skeleton libraries exist [77], we designed the skeletons ourselves for maximum control.

Implementation focus

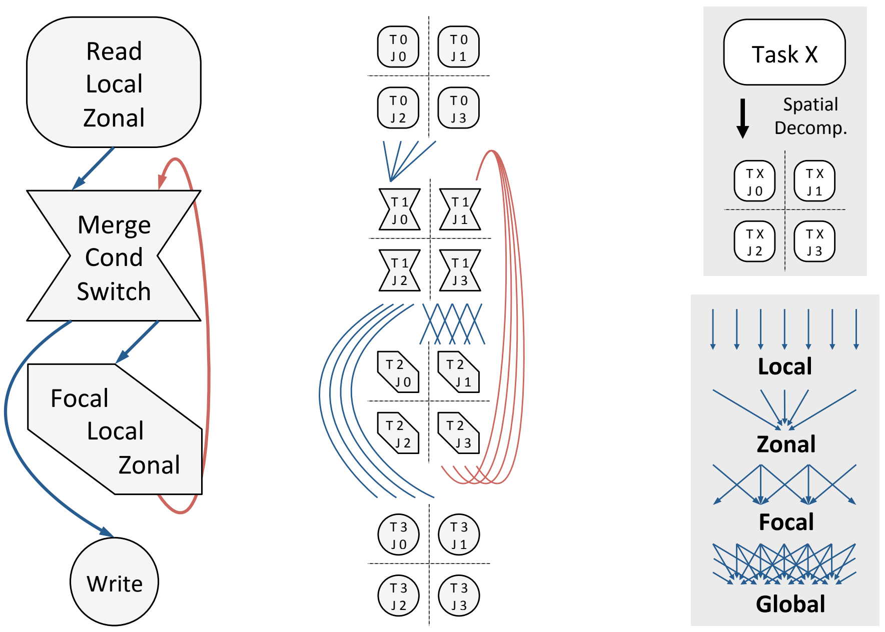
The back-end is devised to be performant, not productive. Therefore it can no longer rely on abstractions and must deal with the peculiarities of the hardware. This means that all the details regarding parallelism, architectural features or memory hierarchy, together with problems such as memory management, deadlocks or race conditions, are now exposed in the back-end. Note, however, that it is not the modelers that face these challenges, but the developers of the map algebra compiler.

On the other hand, OpenCL provides a somewhat intermediate abstraction that relieves some work from the developers. It abstracts several low-level APIs, like intrinsic instructions, OS threads or the GPU runtime driver. Nonetheless, as OpenCL is not performance portable, skeletons still need to be specialized for the devices and map algebra classes. For example, GPU skeletons of Focal and Zonal operations should exploit the on-chip scratchpad memory, while CPU skeletons should utilize multiple code paths that differentiate borders and inner cases. Specialization is a broad optimization that we describe in section 3.2.

Compilation and tasks model

The kernels generated by the skeletons need to be compiled to obtain machine code the OpenCL devices can run. The compilation is entirely handled by the OpenCL driver, thus developers have little control over this process. For example, one cannot activate, deactivate or reorder the low-level optimizations applied to the kernel. Nevertheless, the most important optimization comes from generating large kernels, since this rises the ILP and reduces the memory movements from the off-chip memories. That is precisely the purpose of the fusion pass, which merges nodes into cluster to attain larger kernels.

While the IR provided a generic framework where to apply machine independent passes, the compiled code is specific and can only run on the device it was compiled for. For every cluster of nodes, a **task** is created that wraps its device code. Tasks are work entities that consume some inputs to produce some outputs. They inherit the input dependencies of their clusters and can only execute once those are met. Tasks follow a chained execution where completed tasks activate further tasks until all have completed. Tasks may execute in parallel if they do not depend on each other, commonly known as task parallelism. Because tasks contain data-parallel kernels, they combine both types of parallelism.



(a) Tasks (b) Jobs Legend

Figure 3.8: The spatial decomposition of tasks (a) generates standalone jobs (b) that can execute as soon as their input dependencies are met.

Spatial decomposition, jobs and blocks

The advancements in remote sensing have brought unprecedented volumes of spatial data (subsection 1.1.3). Data has become so large that nation-wide raster models do not fit in the memory of a single computer anymore. To go around this limitation, rasters are decomposed in their spatial dimension and are processed in blocks. This technique will be described in section 3.3, and is only mentioned now to support the narrative.

The spatial decomposition not only affects the data, but also the computation. As a result, tasks are divided into smaller execution units, called **jobs**. Jobs are spatial portions of tasks confined to the area of a block. They are standalone executable units that only depend on their input blocks, and once started, they execute until completion.

Jobs are the unit of scheduling, too. The order of execution of jobs is restricted by the spatial extent of their input dependencies, which derives from the number of non-Local nodes clustered into the parent task of the job. Specifically, Focal operations create bounded dependencies according to their neighborhood, Zonal operations create large dependencies according to the extension of their zones, and Global operations potentially generate full dependencies on whole rasters.

Continued example and dependency DAG

Figure 3.8 shows the spatial decomposition of Figure 3.7b. Applying spatial decomposition unravels the two spatial dimensions contained by the tasks. Visually, a job is a spatially independent portion of a task that depends on previous jobs. The further their dependencies extend in the spatial dimension, the more they restrict the schedule. For example, in Figure 3.8 all Zonal jobs must complete before advancing in the operation dimension.

The number of jobs depends on the size of the data, hence it could become very large. Since jobs occupy memory, maintaining a large number of them would deplete this resource. For that reason, jobs are not instantiated until their inputs are ready, and are deleted soon after. In Figure 3.9, only the first row is initially active as it depends on data readily available on disk. When these jobs complete, their memory is released and the dependent jobs are activated. Due to this gradual activation of jobs, the dependency graph always remains acyclic. As the chores of the back-end are over, this directed acyclic graph (DAG) becomes the input to the runtime.

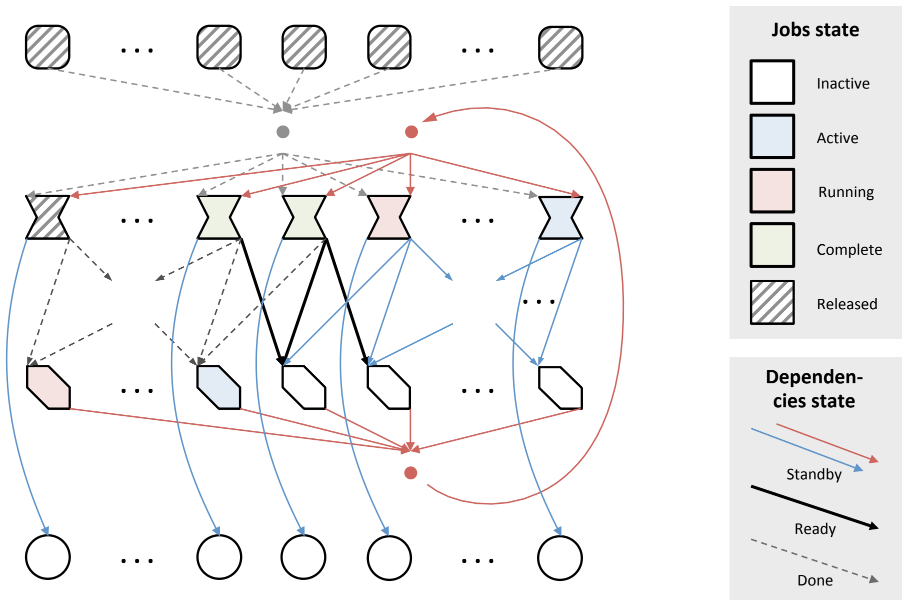


Figure 3.9: Directed acyclic graph of jobs that supports their scheduling.

3.2.4 Runtime: Work Pool

The fourth and last component of the compiler approach is the runtime. The runtime is the active part of the parallel execution model that orchestrates the execution. This component is in charge of the machine resources, such as memory, cores and heterogeneous devices. The runtime runs a cyclic routine where it selects the work, finds the necessary data, acquires resources, conducts I/O and memory transfers, issues the execution, notifies the completion, and gathers statistics. The runtime is not concerned about the IR anymore, but about the parallel code generated by the back-end, and together the two make up the parallel execution model. The goal of the runtime is to efficiently orchestrate the work at execution time, so that the resources are well utilized and the performance is satisfactory.

In our prototype, the runtime consist of a **concurrent work pool written in C++** (Fig 3.3d). It is designed to be:

– *Lightweight*, to steal little processing time from the actual work.

– *Asynchronous*, so it can handle other chores while waiting for I/O.

– *Concurrent*, to manage multiple streams of work simultaneously.

– *Runtime-aware*, to apply optimizations only discovered at execution time.

– *Scalable*, so that it can effectively utilize large numbers of cores.

Work pool model

A work pool is a parallel execution model in which worker entities collaborate to tackle some large problem. This model is applicable when both problem and solution can be expressed as a partitionable data structure. Such data structure is shared by the worker entities and constitutes their source of parallel work. Initially, the large problem is decomposed into smaller problem pieces that are pushed into the work pool. Each problem piece is to be solved by applying a procedure to it, which makes up a piece of work. The procedure is run by the workers, who repeatedly take one work piece, process it, and request another one. Thereby, the workers progress toward a partitioned solution, which is completed once all work is finished.

The work pool is a flexible parallel model. For instance, the pool can be shaped as a queue, list, hash table, tree or as a hybrid form. Another parameter is the work generation strategy, which can be static or dynamic. The former strategy has a fixed set of work, while the latter can generate and add new work to the pool. The mapping of work can be dynamic too, so that any process can perform any piece of work. Additionally, the mapping can be centralized via a master entity, or decentralized with self-organized workers. On the other hand, if the work is generated dynamically and a decentralized mapping is used, then some termination detection logic is required for the workers to discern the end of the process.

Continued example

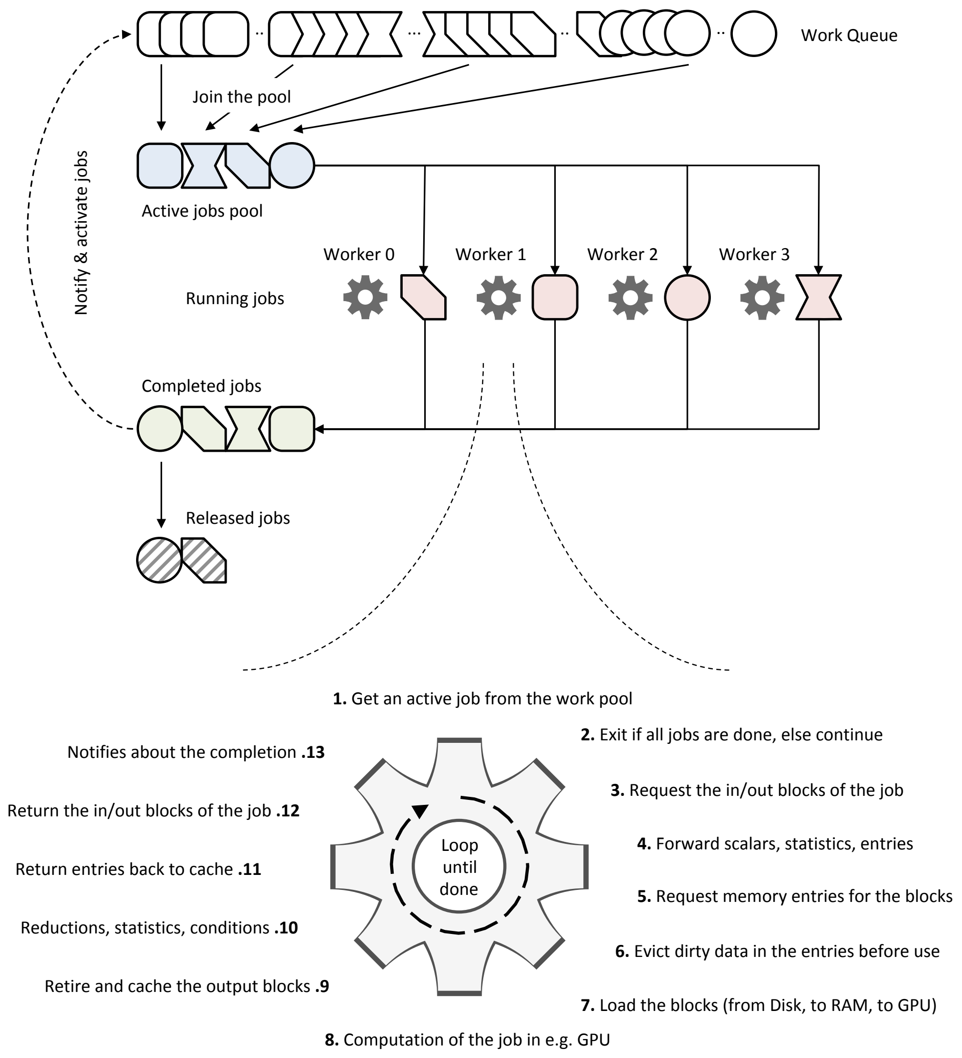
In the pit filling example (Fig. 3.5), the problem to be solved is the input DEM with pits, the solution is the depresionless DEM, and the partitioning is achieved via spatial decomposition. Therefore the problem pieces are the raster blocks, the procedures are the tasks generated by the back end, and the work pieces are the jobs, which are handled one at a time by a group of worker threads. The work generation must be dynamic because the duration of the pit filling loop is only known at runtime. The mapping could be centralized but that presents poor scalability, hence self-organized workers are preferable. Thereby, the work pool creates a cyclic procedure where jobs are generated and run until all pits are filled.

Life of a Worker

The central actor of the work pool model is the worker. Workers are CPU threads that coordinate the execution of jobs, and their behavior is depicted in Figure 3.10. From birth to death, workers continuously execute a procedure that can be summarized as follows:

1. request an active job from the pool,
2. exit if the halt condition is detected,
3. resolve and request the blocks of the job,
4. attempt to optimize away unnecessary work,
5. request available memory entries from the cache,
6. evict the dirty data stored in the memory entries,
7. load the input blocks into their memory entries,
8. run the task on the input blocks to produce the output blocks,
9. retire the output blocks and add their entries to the cache
10. update the statistics and other maintenance work,
11. return the memory entries to the cache,
12. return the blocks of the jobs,
13. notify the dependent jobs to activate and join the pool,
    and loop back to step 1.

Before the work can take place, the runtime needs to set the stage for the workers. For example, the tasks are initialized and their OpenCL code compiled, the memory entries of the cache are allocated, the jobs free of dependencies are added to the pool, and the threads hosting the workers are instantiated. Likewise, the runtime needs to clean the stage after the work. For example, the cached data is written to disk, the memory entries are released, the workers are discharged, the C++ structures are released, and the control is returned to the Python interface.



Work Queue
Join the pool
Notify & activate jobs
Active jobs pool
Worker 0
Worker 1
Worker 2
Worker 3
Running jobs
Completed jobs
Released jobs
1. Get an active job from the work pool
2. Exit if all jobs are done, else continue
3. Request the in/out blocks of the job
4. Forward scalars, statistics, entries
5. Request memory entries for the blocks
6. Evict dirty data in the entries before use
7. Load the blocks (from Disk, to RAM, to GPU)
8. Computation of the job in e.g. GPU
Retire and cache the output blocks .9
Reductions, statistics, conditions .10
Return entries back to cache .11
Return the in/out blocks of the job .12
Notifies about the completion .13
Loop until done

Figure 3.10: Work pool model, where workers concurrently process jobs.

Runtime optimization

While the back-end attains static performance, the runtime pursues dynamic performance. It optimizes the program as it executes, by taking into account all information that is known. For example, if a raster is full of zeros, then adding it to another raster will not change the values. The runtime can perform this type of optimizations because it keeps statistics of the data, whereas the back-end lack such information and can only optimize for constant factors like the hardware. On the other hand, the back-end has more time to work ahead of execution, while the runtime must be quick in its decisions to not bottleneck the execution.

The runtime optimizes the execution in multiple ways. It reorders the execution of jobs to improve their data locality, as will be detailed in subsection 3.4.2. This is done by a scheduler that rearranges the jobs in the pool according to a space-filling curve. The runtime also caches recently read blocks to avoid reloading them in future requests, and likewise, it does not immediately evict recently computed blocks in case of future uses. More information on the cache is found in Paper 3. Additionally, the runtime exploits the sparsity in data to avoid memory movements and computation, as will be covered in subsection 3.4.4. For example, blocks whose content is homogeneous can be compressed into a scalar value, and operations on homogeneous blocks can sometimes be avoided altogether.

Concurrent and scalable C++

The runtime needs to track the active jobs, their blocks, dependencies and associated resources during execution. However, very large datasets lead to very many entities to keep track of, which may bottleneck the execution. Consequently, the runtime demands a low-overhead language supporting threading and concurrency, like C/C++. By executing workers in different threads, they can overlap processes that would otherwise stall their advance. For example, when one worker stops to perform some I/O it must wait for milliseconds before it completes. This time is not wasted because the OS sleeps the worker and wakes up another to perform its own work.

Note that using a low-level language like C++ is not enough. The code needs to be carefully designed so that memory, data structures and algorithms are efficient. For example, memory must be managed manually (i.e. no garbage collector) and pre-allocated whenever possible. Failing to do so destroys the concurrency of workers, as reallocating is an expensive and non-scalable process. Data structures must provide quick access to recurrent data, even when incurring high memory cost. Nodes, jobs, blocks and other entities need to hold pointers to reach each other in one step. Algorithms need to prioritize time over space, too. For example, the memory cache, the fusion routine and most other functionalities should employ hash tables extensively.

Finally, as the self-organized workers access shared data, they need to watch out for typical concurrency problems. The pool, the cache, the blocks, the tasks and other data structures accessed by workers need some type of regulation. In our implementation, workers acquire a mutex before accessing shared resources and release it after. However, a straightforward locking strategy is again insufficient to attain good performance. Congested sections like the cache require finer locking, since a single mutex would lead to workers contention. There is a difficult balance to be found here between the complexity and the efficiency of the locking strategy.

3.3 Hierarchical Decomposition

Computers are designed hierarchically. Their memory, for example, is composed of multiple horizontal layers of different technologies. From closest to farthest to the processor, some of the layers are: processor registers, on-chip memories (i.e. cache, scratchpad), off-chip memories (i.e. RAM, HBM), secondary storage (i.e. SSD, HDD) and network storage (i.e. NAS, SAN). These layers vary in density, latency, bandwidth, power consumption, volatility, resilience, price, etc. Registers are the smallest memory, built into the processor core upon a few logical gates made of transistors. On-chip memories lay close to the registers and often present a hierarchy on their own (e.g. cache L1/L2/L3) Off-chip memories have their own package, reside nearby the chip, and represent the main type of computer storage. Secondary storage is external, removable, non-volatile and cannot be accessed directly by the processor. Network storage is decoupled from the processor box and is accessed via network protocols (i.e. TCP, SCSI).

Computer's processors, too, are built somewhat hierarchically. Whereas the memory subsystem consists of connected but separate memory levels, each processing level is made up of elements from the immediate previous level. From smallest to largest in scale, these levels are: processing elements (i.e. ALU, LSU, CU), compute units (i.e. CPU core), processors (i.e. GPU, DSP), computer systems (whole computer) and networks of computers (i.e. grid, clusters). The simplest processing circuitry in a typical von Neumann architecture are the ALUs and CUs. These elements are coupled to form compute cores that are Turing-complete and can run any program. Cores are connected to compose versatile multi-core CPUs that are part of most computing systems nowadays. CPUs are clustered into large computer networks that run the IT and internet services of the world.

The above is the typical computer platform where raster spatial models are run. Therefore, for a map algebra implementation to be performant, it must comply with such hierarchical design. Our map algebra compiler applies a hierarchical decomposition to both the spatial and the functional dimensions. Thereby, *spatial decomposition* breaks the data from rasters to blocks, to groups, to cells, while *functional decomposition* breaks the control from programs to tasks, to sections, to instructions. Table 3.2 shows the hierarchical levels resulting from this decomposition.

Spatial Decomposition

Raster data is spatially decomposed to match the hierarchy of computer memory (Fig. 3.11). High resolution rasters are possibly larger than main memory and might only fit in secondary or network storage. To be able to process such rasters, the compiler splits them into *blocks*. Blocks must fit

Data : / Control :	Raster / Program	Block / Task	Group / Section	Cell / Instruction
User Model	Script	Function	Scope	Operation
Intermediate Representation	Full Graph	Cluster	Section	Node
Memory	Storage (Disk, NAS)	Off-chip (RAM, HBM)	On-chip (Cache, Scratch.)	Registers
Processor	System (host + devices)	Device (CPU, GPU)	Compute Unit (Core, SM)	Proc. Element (ALU, LSU)
Work Actor	Runtime	Worker	Work-group	Work-item
Work Load	—	Job	—	—

Table 3.2: Hierarchical decomposition of data and control, and how this relates to the hierarchy of memory, processors and other aspects.

into main memory and occupy from kilobytes to a few megabytes. During execution, blocks are loaded from memory in small portions, called *groups*. Groups must fit into on-chip memory and occupy from bytes to a few kilobytes. Groups are made of *cells*, the basic spatial unit in raster data. Cells must fit into the processor registers and occupy from one bit to a few bytes. Once loaded to the registers, the processor can act on the input cells to compute the output cells.

The spatial decomposition technique is *static* and *regular*: all blocks from all rasters share the same size, which is kept constant throughout the execution. Thereby, any two blocks can always be operated on (e.g. added) without having to mix and match their size. This avoids overhead and simplifies several parts of the runtime, like the in-memory cache. Groups are more flexible and only share size within their block, not with groups from other blocks. This permits the specialization of groups by algorithmic pattern, which is useful for Global classes. The block and group sizes can vary within reasonable limits. However, too small a size leads to high scheduling overhead, whereas too large a size diminishes the opportunities for optimizations.

Functional Decomposition

Like data, the computer program is functionally decomposed following the processing hierarchy. *Instructions* are the smallest unit of computation, executed by the processing elements. *Sections* are series of instructions executed by compute units and synchronized at group level. *Tasks* wrap

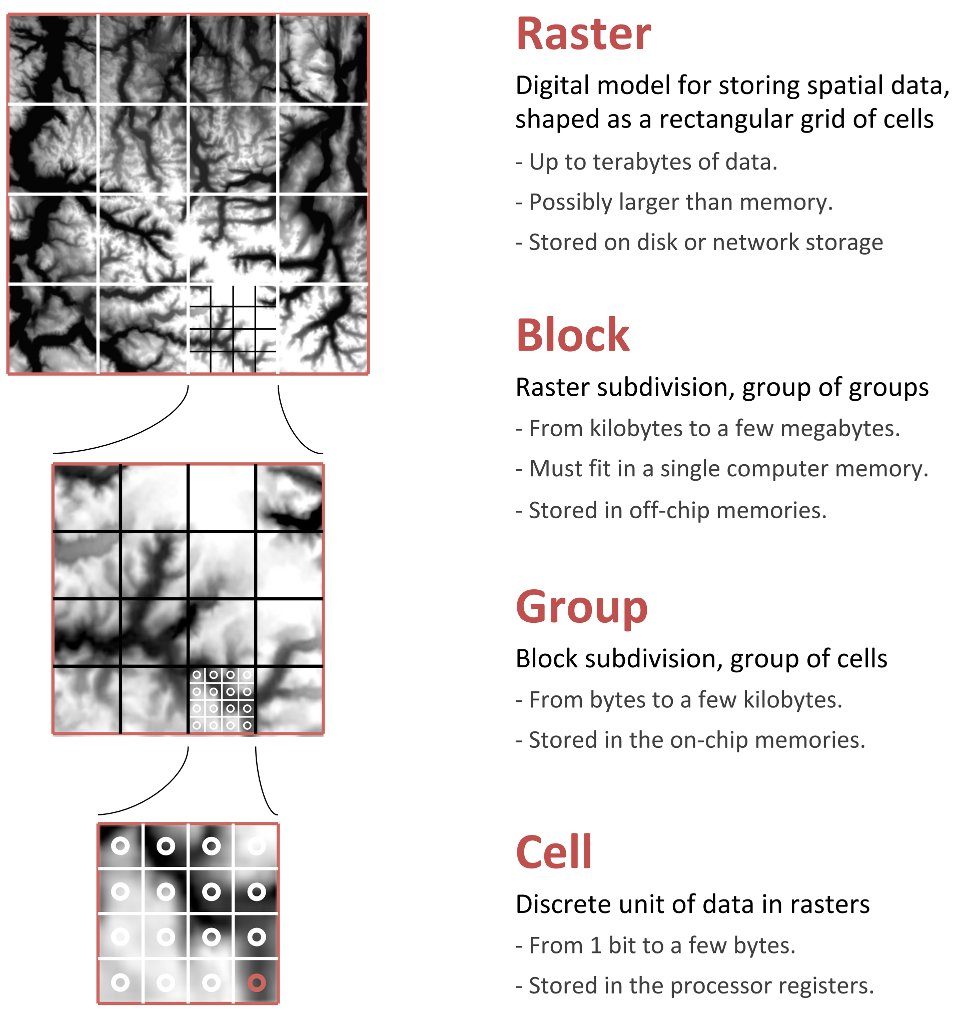

Raster
Digital model for storing spatial data, shaped as a rectangular grid of cells
- Up to terabytes of data.
- Possibly larger than memory.
- Stored on disk or network storage
Block
Raster subdivision, group of groups
- From kilobytes to a few megabytes.
- Must fit in a single computer memory.
- Stored in off-chip memories.
Group
Block subdivision, group of cells
- From bytes to a few kilobytes.
- Stored in the on-chip memories.
Cell
Discrete unit of data in rasters
- From 1 bit to a few bytes.
- Stored in the processor registers.

Figure 3.11: Spatial decomposition of rasters into blocks, groups and cells.

kernels, which contain lists of sections executed by devices and synchronized at block level. Lastly, the program is a graph of tasks executed by the system and synchronized at raster level. Recall that the program derives from the IR graph, which is parsed from the user script. These prior representations of the program are somewhat hierarchical too, with the graph IR composed of clusters, regions and nodes, whereas the script is formed by functions, scopes, and operations.

The functional decomposition is dictated by the spatial reach of the map algebra operations. Global operations with complex or unpredictable spatial dependencies always split the graph into clusters. Focal and Zonal operations with simpler dependencies fuse to a certain extent but require separate sections. Operations with trivial spatial dependencies like Local and I/O fuse all the way into a single section. Typical scripts are composed by a majority of I/O and Local, a few Focal and Zonal, and one or none Global operations. Hence, typical programs can be exhaustively fused and memory movements can largely be avoided.

3.4 Code Optimizations

Section 3.2 detailed the compiler architecture by which to achieve PPP. Productivity comes from the map algebra interface, while portability is enabled by the IR. Performance is more scattered and arises from multiple optimizations across the back-end and runtime. This section focuses on such code optimizations, which we classify in four groups: the parallelization, the reordering, the specialization and the sparsity groups of optimizations.

The parallelization group is based on the data parallelism extracted from the spatial dimension. To a lesser extent, it also exploits the task parallelism from the functional dimension. The degree of parallelism is constrained by the algorithmic patterns of the map algebra classes.

The reordering group is enabled by the freedom of execution exposed with the dataflow model. This freedom is inversely proportional to the spatial dependencies of the map algebra operators. The main benefit is data locality, which is critical to relieve the memory bottleneck.

The specialization group is founded on the heterogeneity of computer architectures and map algebra classes. Each algorithmic pattern is implemented with a bespoke code version. Furthermore, any complex algorithm worth specializing can be encapsulated into its own Global class.

The sparsification group is intended to exploit the homogeneity in some raster data. Large areas of cells with the same value can be packed together, and their computation can be combined. Thereby these optimizations avoid computing duplicated results and moving compressible data.

None of these groups guarantees consistent improvements as their efficiencies depend on the script. For instance, some scripts display abundant parallelism, while others present a high degree of sparsity. Note that for the optimizations to be valid and beneficial, their computational cost must be low, the execution must improve in speed or resources usage, and the output must remain unchanged. Additionally, the optimizations occur across the four hierarchical levels detailed in section 3.3. The next subsections detail the groups of optimizations.

3.4.1 Parallelization of the Work

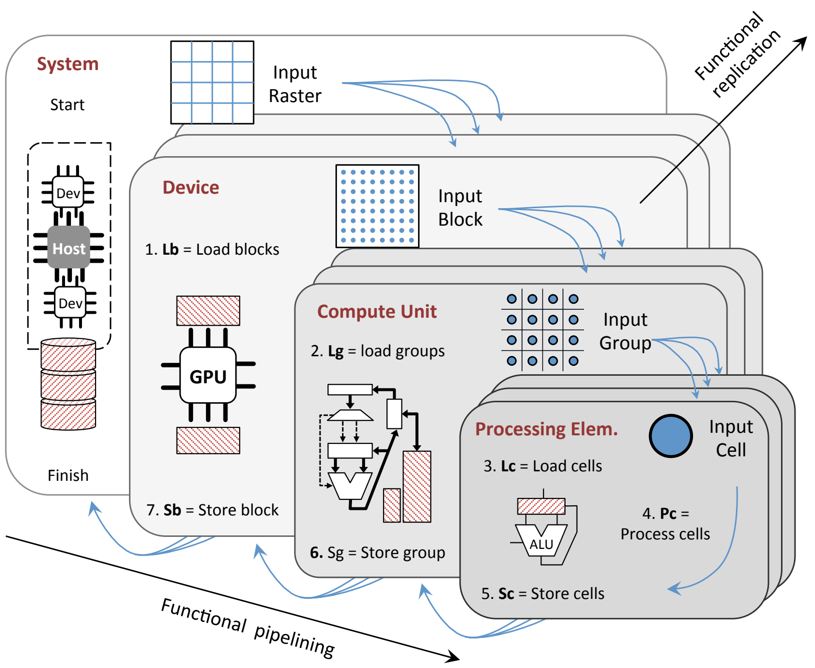
The parallelism in spatial modeling arises from two sources and can be exploited at multiple hierarchical levels. It arises from the *spatial dimension*, since one phenomenon can progress independently in two distant areas. It arises from the *functional dimension* too, as two separate phenomena can progress independently in one area. Moreover, the two dimensions intertwine in multiple ways, leading to abundant but intricate parallelism.

The spatial dimension provides **data parallelism**. For example, the runoff of stormwater is not concentrated to a unique area of a terrain, but water flows freely in numerous directions toward several lakes and streams (Fig. 3.5d). As a result, the waterflow equations can be run concurrently for each of the cells of a DEM. Some spatial phenomena present more data parallelism than others. Whereas modeling rainwater is perfectly parallel, modeling a wildfire is partly sequential because fire does not emerge simultaneously everywhere, but it spreads slowly from one area to another. This variation is captured by the map algebra classes, from perfectly parallel Local operations to largely sequential Global operations.

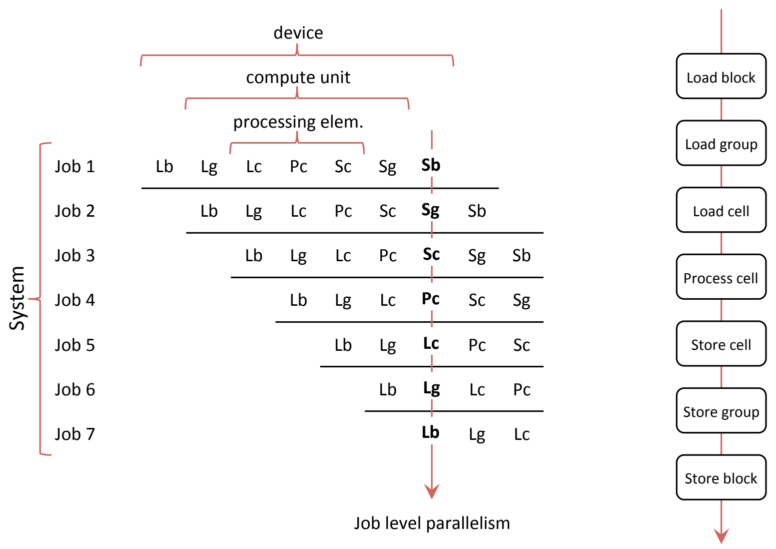
After the spatial decomposition, the data parallelism can be exploited at block, group and cell levels. Two blocks of a raster can be loaded, processed and stored in parallel by two workers. Note that the workers do not directly process the blocks, but they issue the kernels that will do so. Two groups of a block can be loaded, processed and stored in parallel by two work-groups. OpenCL kernels are executed by work-groups, which map to compute units (e.g. CPU cores). Two cells in a group can be loaded, processed and stored in parallel by two work-items. OpenCL work-groups are split into work-items, which map to processing elements (e.g. ALUs).

The functional dimension provides **task parallelism**, also known as control or functional parallelism. In urban modeling, the developments in transportation and housing are synergistic but separate processes. The equations modeling one process depend on the past state of the other, but not on the current state. Hence, the transportation and the housing equations can be run concurrently for a given time step. Some compound phenomena like urban development present plenty of task parallelism, whereas simpler phenomena like waterflow only display data parallelism. This is directly reflected on the equations modeling the phenomena, which will present more or less independent map algebra operations.

After the functional decomposition, the task parallelism can be exploited at task, section and instruction levels. Two tasks of a program can have their kernels executed in parallel by two OpenCL devices, or even in a single device if it supports the concurrent execution of kernels. Two sections of a kernel could potentially execute in parallel in two compute units, but the OpenCL model does not contemplate this option. Two instructions of a section can



(a) Diagram of the pipelined and replicated execution.



(b) Schematic execution of seven jobs in the pipeline.

Figure 3.12: Pipelined, replicated execution to exploit data, task parallelism.

execute in parallel in two processing elements. For this, the processor needs to be superscalar, which is typical of modern CPUs.

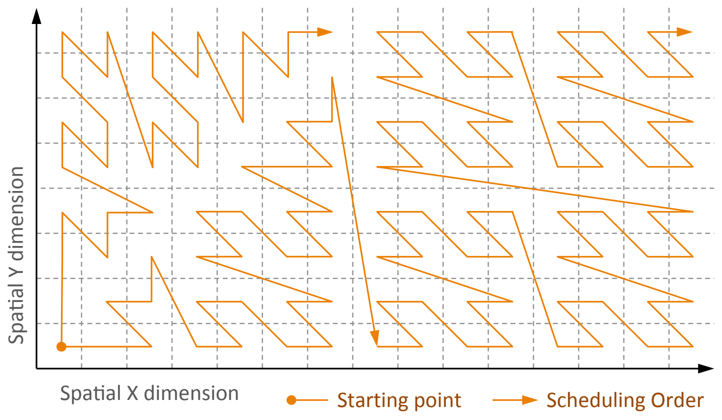
The combination of data and task parallelism leads to a *pipelined* and *replicated* execution. This design exploits the duplicated functional units (e.g. cores) and the fact that they all work concurrently. Thereby, the execution of a job can be seen as the succession of seven main stages: block loading (*Lb*), group loading (*Lg*), cell loading (*Lc*), cell processing (*Pc*), cell storing (*Sc*), group storing (*Sg*) and block storing (*Sb*). At any point in time, different parts of different jobs are being partially executed in the pipeline. This process is illustrated in Figure 3.12a, and more schematically in Figure 3.12b. In one direction, the same functional units are replicated to handle the data parallelism, on the other, different functional units are pipelined to exploit the task parallelism.

3.4.2 Reordering for Data Locality

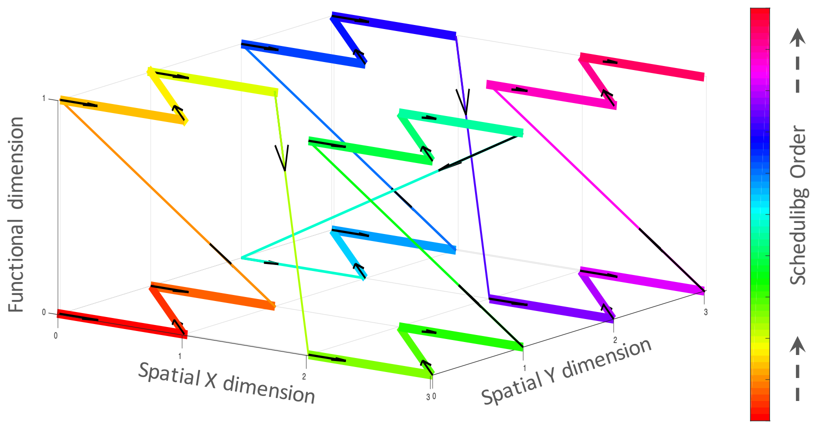
As was motivated in section 3.1, parallelism is not enough to get performance. Modern computer architectures present considerably higher parallel throughput than memory bandwidth. Often, the memory subsystem cannot keep up with the many parallel requests and the processors have to wait idle. Avoiding this bottleneck requires data to reside as locally as possible to the processors, because closer memory provides higher bandwidth (Fig. 3.2). However, the higher the bandwidth the lower the capacity, which leads to memory thrashing. Therefore, as memory cannot accommodate the whole raster, blocks are constantly swapped in and out.

The reordering optimizations exploits the freedom in the order of execution to maximize locality. This freedom arises from the lack of data dependencies in the spatial and functional dimensions. Interpreters, which can only see one operation at a time, fail to acknowledge and exploit this freedom. They execute in a *space-first order*, whereby the full spatial dimension is iterated for every operation. Conversely, the compiler approach analyzes the IR to find better execution orders. When possible, it imposes a *functional-first order* that increases locality and diminishes the memory thrashing. This reordering is done at block level with a scheduler, at group level with sections and at cell level with fusion.

A **scheduler** reorders the active jobs in the work pool to save block movements from storage to off-chip memory. To do this, the pool is shaped as a priority queue that rearranges the jobs as they become active. The priority is measured as the distance of a space filling curve, specifically the Z-order curve [57]. More formally, the SFC maps the functional and spatial dimensions to a single dimension where to rank the jobs. Figure 3.13a shows how the X, Y spatial dimensions are projected to a contiguous Z-curve, while Figure 3.13b also includes the functional dimension into this projection. At the same time, the raster blocks are kept in an in-memory cache with LRU and NINE[5] policies (Paper 3). The goal is to schedule together those jobs sharing same inputs or with producer-consumer relationships, so that their blocks are more likely to still reside in the cache and less block movements are needed.



(a) Radial operation scheduled with Z-curve order.



(b) Scheduling of consecutive Focal operations.

Figure 3.13: Scheduling of jobs based on the Z-order SFC.

At group level the reordering is attained by structuring the kernels in **sections**. Work-items in a section execute instructions in functional-first order until they hit a barrier. OpenCL barriers synchronize the work-groups, forcing them cover their spatial domain before continuing. This effectively changes the execution order from functional to spatial and enables the reuse of data. For example, now only one of multiple work-items has to load common input cells to the work-group, decreasing the memory traffic between

[5]Non-Inclusive Non-Exclusive (NINE) multi-level caches find balance somewhere in between inclusive policies (i.e. higher levels are fully contained by lower levels) and exclusive policies (i.e. no overlap is allowed between cache levels).

the off-chip and on-chip memories. All map algebra operations with spatial dependencies benefit in one way or another. Focal operations avoid common memory loads in their neighborhood (Listing 3.2), while Zonal operations avoid memory stores beyond the on-chip memories (Listing 3.3).

At cell level the reordering is achieved with **fusion**. Fusion merges IR nodes into clusters, which later derive into tasks and into OpenCL kernels. Work-items in a kernel execute in a functional-first order until a barrier is found. Thereby, the intermediate results can be streamed at register level, avoiding cell movements from on-chip memory. While this is the most beneficial reordering optimization, it is also the most easily disrupted. Non-Local map algebra operations with spatial dependencies will prevent or severely impact fusion. For instance, Zonal operations might reduce a full raster and involve the whole memory hierarchy doing so, forcing the following operations to wait for the reduction and thus blocking their fusion. For more information on fusion and the reordering optimizations we refer to Paper 3.

3.4.3 Specialization by Algorithmic Pattern

Specialization refers to the design of exclusive codes to accelerate recurrent workloads. This is as opposed to generalization, namely, approaching all workloads with generic codes. It turns out that a loop of Focal operations suffices to model most spatial phenomena [66]. However, this generic approach is inefficient because it often performs more operations than are needed. Attaining competing performance necessitates of specialized codes for important recurrent workload. Specialization is based on the map algebra classes and the device architectures, and is applied at block and group levels via bespoke codes for tasks and skeletons.

At block level, the specialization is embedded in the **task logic** executed by workers. Some tasks require more input blocks, employ supporting data structures or run multiple kernels. For instance, Focal jobs require access to the neighboring input blocks in a short radius. Zonal jobs perform partial reductions that are temporarily accumulated in a supporting data structure. Radial jobs employ specialized equations for each compass direction, leading to eight different kernels. Global jobs might access very disperse input blocks and require any non-trivial data structure and algorithmic logic.

At group level, the specialization is applied to the **skeletons design**. A skeleton is designed for each combination of map algebra class and OpenCL device. For example, Focal skeletons on GPUs make use of shared memory for the neighborhood, while on CPUs they employ two separate code paths for the border cells and for the central region. Zonal skeletons on GPUs perform a parallel reduction that revisits the N cells $NlogN$ times, whereas on CPUs the reduction of groups is sequential and only visits the N cells N times. On the other hand, trivial workloads like Local operations run well without specialization.

The Listings below shows pseudocode versions of the OpenCL kernel codes generated by these skeletons. Kernels are split in sections, which may synchronize the work-group to alter the execution order. The Local kernel in Listing 3.1 is simple enough to be used in both CPUs and GPUs. It bypasses the on-chip memory, presents no barriers, and follows a functional-first order. The Focal GPU kernel in Listing 3.2 receives additional blocks from where to load the neighbor cells. It uses barriers to synchronize the on-chip memory and makes the work-items access the neighboring cells. The Zonal GPU kernel in Listing 3.3 reduces the result from on-chip to off-chip memory with atomics. It also performs a complicated synchronization within a loop, where it reduces the group by half per iteration.

While not done in this work, it is interesting to mention that Global operations can take the specialization one step further. This is the case of Figure 3.4, which reproduced pit filling by a loop of Local, Focal and Zonal operations. Alternatively, the operations could be integrated into a "Pit-Filling" class exclusively optimized for Planchon's algorithm. For example, Figure 3.5c shows how pits are to be filled with the minimum possible elevation. This means that any previous filling above the minimum will be discarded (Figure 3.5b), and implies that, had the algorithm found the minima first, it could stop trying to further fill the pits. As a result, scheduling the jobs from lower to higher altitude can reduce the algorithm complexity. At group level, the pit filling skeleton could also follow a sequential iteration of cells ordered by altitude. This unique characteristic of Planchon's algorithm can only be exploited with a specialized Global class.

Listing 3.1: Local class OpenCL skeleton for CPU/GPU

```
kernel Local-Skel (
 offchip input_block,
 offchip output_block,
 ... )
{
  { // init section
    g = get_global_id( )   // cell id within block
  }
  { // input section
    in = load(input_block[g])   // load input cell
  }
  { // local section
    tmp = map_op(in)   // local ops follow the map pattern
    out = map_op(tmp)   // another op, consuming the first
  }
  { // output section
    store(output_block[g],out)   // store output cell
  }
}
```

Listing 3.2: Focal class OpenCL skeleton for GPU

```
kernel Focal−GPU−Skel (
  offchip input_block,
  offchip neighbor_block,
  ...
  offchip output_block,
  ... )
{
  { // init section
    g = get_global_id( )
    l = get_local_id( ) // cell id within group
    onchip group [_group_size_]
  }
  { // input section
    if _border_case_:
        in = load(neighbor_block[g])
    else : // _central_case_
        in = load(input_block[g])
    store(group[l],in)
    barrier( ) // group−level synchronization
  }
  { // focal section
    acu = _neutral_element_
    for c in _neighborhood_ :
        cell = load(group[c])
        tmp = map_op( cell ,_mask_)
        acu = red_op( acu , tmp )
  }
  { // output section
    store(output_block[g],acu)
  }
}
```

Listing 3.3: Zonal class OpenCL skeleton for GPU

```
kernel Zonal−GPU−Skel (
  offchip input_block,
  offchip output_block,
  ... )
{
  { // init section
    g = get_global_id( )
    l = get_local_id( )
    b = get_group_id( ) // id of group
    onchip group [_group_size_]
  }
  { // load section
    if (g < _data_size_) :
        group[l] = input_block[g]
    else
        group[l] = _neutral_element_
  }
  { // zonal section
    while group.size() >= 2 :
        barrier( ) // group-level synchronization
        if (l >= group.size()/2)
            continue; // only half workers
        left_group, right_group = split(group)
        left_cell = load(left_group[l])
        right_cell = load(right_group[l])
        tmp = red_op( left_cell , right_cell )
        store(left_group[l],tmp)
        group = left
  }
  { // output section
    if (l == 0) :
        tmp = load(group[l])
        atomic_red( output_block[b] , tmp )
  }
}
```

3.4.4 Sparsification of the Computation

Raster data is organized as a rectangular grid of cells, with each cell holding an independent value. This type of dense format fits well those models with continuously varying data, like DEMs. However, rasters are a wasteful format when the concentration of data is low, like in road network models. Another example is the stream layer in pit filling, where only few narrow areas contain flowing water. Large portions of the stream raster are empty and their cells all known to hold the same null value. As a result, memory movements of such empty areas are redundant, because all they transfer are the null values. Likewise, operations on the empty areas are redundant too, since identical inputs produce identical outputs. The goal of the sparsification is to avoid this unnecessary work all together.

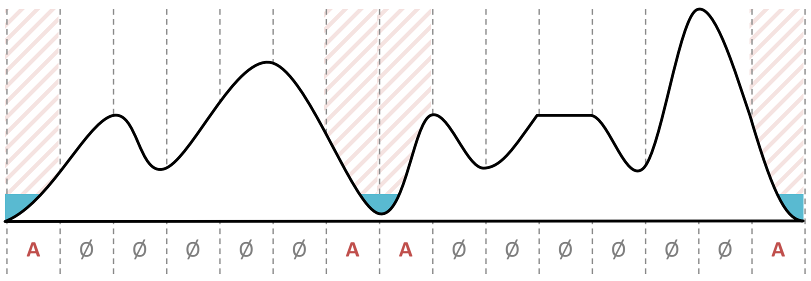
The sparsity is exploited by actively summarizing the data and predicting the computation. Rasters are summarized to obtain a series of statistics for each of their blocks and groups (e.g. maximum, minimum, average and deviation values). Thereby, blocks can be compressed into scalars when their statistics show they hold redundant values (i.e. maximum equals minimum). This avoids not only the movement of redundant memory, but also the costly interaction with the cache. Additionally, the use of statistics sometimes enables the prediction of the computation. For example, instead of operating on two blocks whose cells are known to be uniform, it is enough to operate on their statistics to avoid the redundant computation across cells.

Exploited wisely, the sparsification of the computation can remarkably accelerate certain raster models. In a wildfire simulation it is the fire fronts that change quickly, while the rest remains mostly unaltered. A sparsified execution could conveniently ignore the stable areas and focus on the changing ones [21]. The pit filling algorithm presents a similar behavior, where only a reduced front of cells change per iteration. These are the active cells (Figure 3.4 lines 4,13), which propagate from lower to higher elevations. Figure 3.14 illustrates this example for the same DEM previously showed in Figure 3.5. Here, the sparsification saves a large portion of memory movements and computation per iteration, since the active cells are considerably fewer than the inactive, and they also diminish over time.

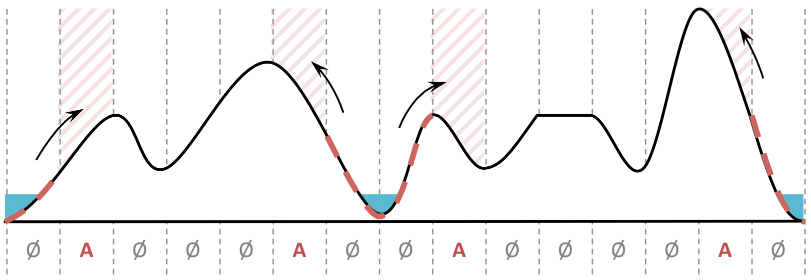
At block level, the sparsification is embedded into the **workers logic**. In Figure 3.10, steps 4 and 10 in the worker chores handle the statistics and prediction. Before the actual work is done, the worker attempts to predict the job outputs by operating on the statistics. If an input block is compressed into a scalar, the worker skips its loading in the 7th step. If an output block is predicted to be uniform, the worker skips its storing in the 9th step. In both cases the 5th, 6th and 11th steps are unnecessary. Finally, the worker skips the computation in step 8 if all output blocks are predicted. As a result, jobs on sparse areas run almost instantly as their heavy chores are skipped all together.

At group level, the sparsification is encoded in the **kernel skeletons** and is directed by the work-groups. The kernel sections are equipped with extra logic that checks whether groups are sparse and predictable. Strategic if-else branches avoid the loading of sparse input groups and the storing of predicted output groups. If all the outputs are predictable, the work-group can promptly exit the kernel without working any further. The sparsification is finer at group level and can therefore optimize areas not reachable at block level. On the other hand, the overhead is also larger and can incur a noticeable penalty in predominantly dense models.

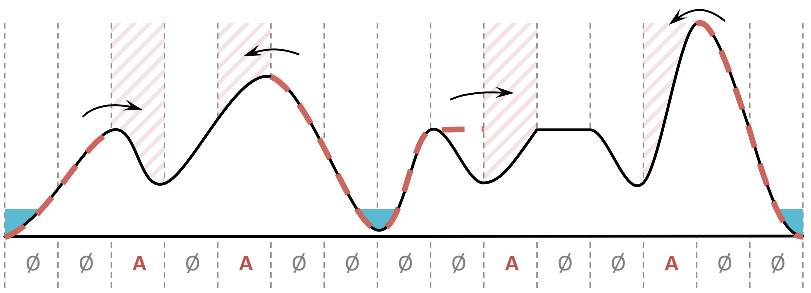
Finally, it is important to note that a balance has to be found between exhaustive and superficial sparsification. Keeping updated statistics throughout the execution is reasonably inexpensive, but not free, hence the data can only be summarized every so often. Likewise, very complex sparsification logics benefit sparse models, but put extra overhead on dense models.



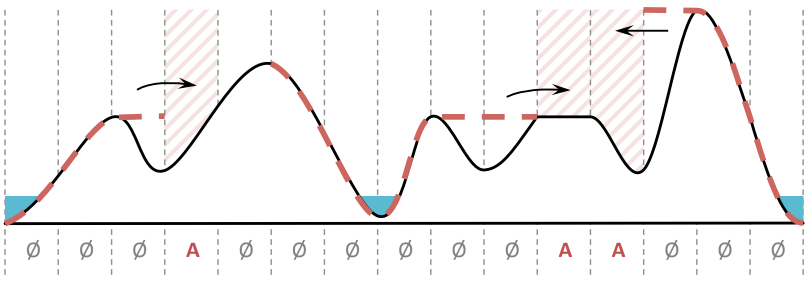
(a) Initially only the exit cells are active (A).



(b) The activity spreads toward the neighboring cells.



(c) Meanwhile the rest of the cells stay inactive (Ø).



(d) Some active fronts dissapear as they merge with each other.

Figure 3.14: Sparsification of pit filling (Fig. 3.5), where only the active cells need to perform computation and move memory.

Chapter 4

Experiments

*"No amount of experimentation can ever prove me right;
yet a single experiment can prove me wrong"*

~ **Albert Einstein**

Chapter 3 reasoned the limitations of map algebra interpreters, proposed a compiler approach to solve the PPP tradeoff, and detailed the hierarchical decomposition and code optimizations of such compiler. In this chapter a prototype implementation of the compiler is put to test with simple workloads that highlight the need for locality beyond parallelism, a viewshed script that showcases and justifies the specialization techniques, and a more complex urban development model that employs the sparsity optimizations. The following sections present these experiments.

4.1 Workloads characterization

This section aims to demonstrate the compiler's ability to mitigate the memory bottleneck. To that end, it tests four map algebra scripts (Table 4.1) of increasing arithmetic intensity[1]. Thus, the first script is mostly memory-bound while the last is largely compute-bound, with a progression in between. This sets a convenient scenario where to observe the effects of parallelism and locality on the performance. Besides the Python scripts, the intermediate graphs are also presented with their clusters identified by colors. The parallel kernel codes are not included due to space constrains, but can be found in the code repository [5]. Lastly, the effect of the optimizations and the contribution of more powerful machines are analyzed.

[1]Arithmetic intensity refers to the ratio of the work to memory traffic [93].

Script	Description	Operations Type	Inputs	Outputs
W. Sum.	Weighs and adds four rasters on a cell by cell basis	7 Local	4	1
Statistics	Computes statistical values: mean, max, min, std.dev...	4 Zonal, 9 Local	1	0
Hillshade	Generates a self-shadowing light effect on a DEM	2 Focal, 36 Local	1	1
Life (x16)	Game of Life with random start and 16 iterations	16 Focal, 64 Local	0	1

Table 4.1: Tested scripts representing typical map algebra workloads.

The *Weighted Summation* script is the first and the most memory-bound workload of the four. It consists of multiple input rasters whose values are scaled and summed into a single output raster. This is done with computationally inexpensive Local operations, whose execution cost is virtually just moving memory. Listing 4.1 shows the uncomplicated Python code, while Figure 4.1 displays the resulting IR graph. The shared color of the nodes implies that all the operations are merged into a single cluster. Although simple, this is a common workload in multicriteria problems such as site selection and suitability models [51].

Listing 4.1: Weighted summation

```
from map import * ## Map Algebra Compiler package

snow = read('snow.tif') * 0.4
rock = read('rock.tif') * 0.1
soil = read('soil.tif') * 0.3
sand = read('sand.tif') * 0.2

wsum = snow + rock + soil + sand

write(wsum,'weighted_sum.tif')
```

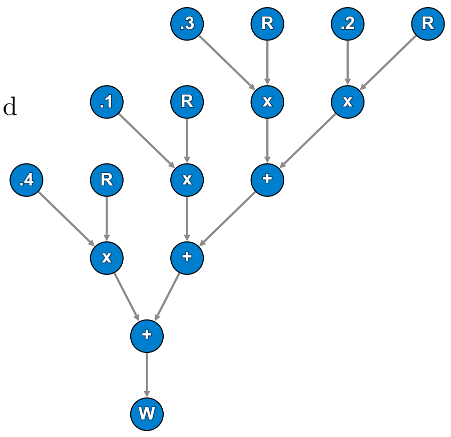


Figure 4.1: Weighted summation graph IR, fused into one cluster.

The *Statistics* script computes typical summary values like the maximum, average and deviation. It comprises Zonal and Local operations that perform reductions and simple arithmetic on the input raster. This script is still memory-bound, even though it requires five time less I/O movements than the previous one. Listing 4.2 shows the Python code and Figure 4.2 displays the IR graph. Since no zone is explicitly given as an argument, the Zonal operations act on the whole input raster. As a result, their large spatial reach causes fusion to split the graph into six clusters of different color. This script is interesting because Zonal operations are a recurrent primitive in spatial statistical analysis [54].

Listing 4.2: Statistics

```
from map import * ## Map Algebra Compiler package

raster = read('raster.tif')
N = prod(raster.datasize())

maxv = zmax(raster)
minv = zmin(raster)
rang = maxv - minv

mean = zsum(raster) / N
geom = zprod(raster) ** (1.0 / N)
harm = N / zsum(1.0 / raster)
quam = sqrt(zsum(raster ** 2) / N)
cubm = cbrt(zsum(raster ** 3) / N)

dev = (raster - mean) ** 2
var = zsum(dev) / N
std = sqrt(var)
norm = (dev - minv) / rang
nstd = sqrt(zsum(norm) / N)
```

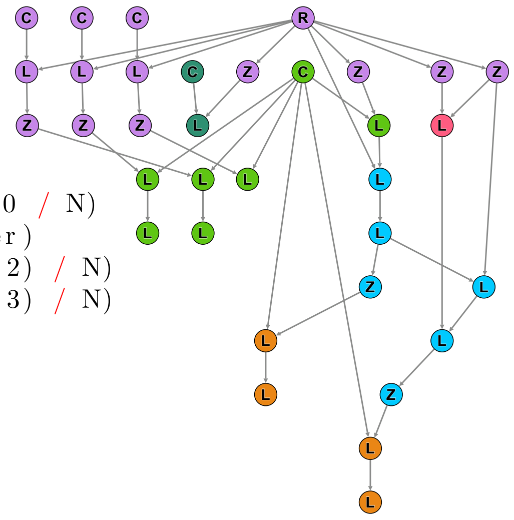


Figure 4.2: Statistics graph IR, fused into six clusters.

The *Hillshade* script generates a self-shadowing effect on an input DEM to create a sense of topographic relief. The equations employed here are based on Horn's formulation [42], which builds upon the slope and aspect functions. Both slope and aspect need to access the vertical and horizontal neighboring cells by means of two Focal operations. Listing 4.3 shows these operations, while Figure 4.3 displays the IR graph after several transformations. For example, each Focal operation (i.e. horizontal and vertical convolutions) is executed twice in the script, but in the IR this repetition is eliminated by the *common subexpression elimination*[2] routine. Additionally, the transcendental and floating-point operations in the script increase its arithmetic intensity considerably. This, together with the seamless fusion into a single cluster, make the OpenCL kernel computationally intensive. Focal operations are common to surface analysis, where topographic relief influences the dynamics of e.g. runoff water [67].

[2]Common subexpression elimination is a frequent optimization that avoids duplicated expressions by computing them once and reusing the result thereafter.

Listing 4.3: Hillshade

```
from map import * ## Map Algebra Compiler package

def hori(dem, dist):
  h = [[-1, 0, 1],
       [-2, 0, 2],
       [-1, 0, 1]]
  return convolve(dem,h) / (8 * dist)

def vert(dem, dist):
  v = [[-1,-2,-1],
       [ 0, 0, 0],
       [ 1, 2, 1]]
  return convolve(dem,v) / (8 * dist)

def slope(dem, zf=1, dist=1):
  x = hori(dem,dist)
  y = vert(dem,dist)
  z = atan(zf * hypot(x,y))
  return z

def aspect(dem, dist=1):
  x = hori(dem,dist)
  y = vert(dem,dist)
  z1 = (x != 0) * atan2(y,-x)
  z1 = z1 + (z1 < 0) * (pi * 2)
  z0 = (x == 0) * ((y > 0) * (pi / 2)
        + (y < 0) * (pi * 2 - pi / 2))
  return z1 + z0

def hillshade(dem, zenith, azimuth):
  zr = (90 - float(zenith)) / 180 * pi
  ar = 360 - float(azimuth) + 90
  ar = ar - 360 if (ar > 360) else ar
  ar = ar / 180 * pi
  hs = ( cos(zr) * cos(slope(dem)) +
         sin(zr) * sin(slope(dem)) *
         cos(ar - aspect(dem)) )
  return hs

dem = read('elevation.tif')
hill = hillshade(dem,45,315)
write(hill,'hillshade.tif')
```

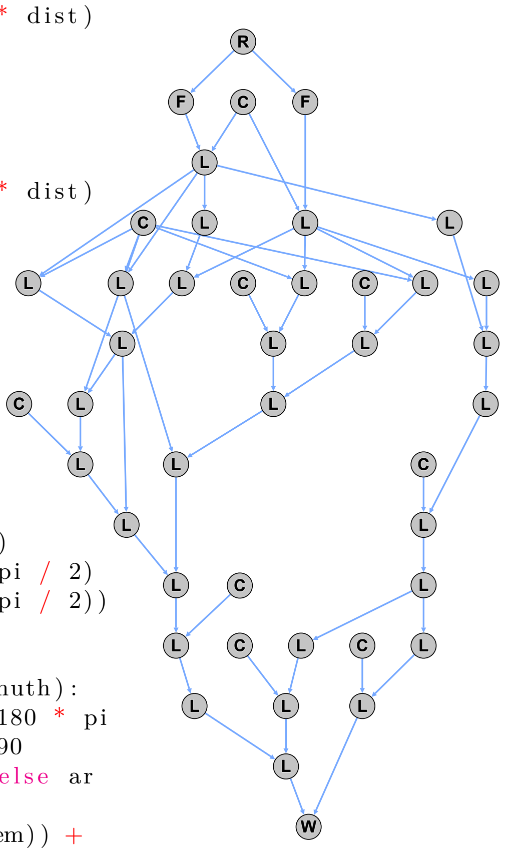


Figure 4.3: Hillshade graph IR, fused into one cluster.

The *Life* script computes the classic Game of Life [66], where cells live or die according to their neighborhood. This is the most basic example of dynamic phenomena that evolves through time in non-trivial, non-deterministic ways. Thus, while real models are more complex, their cores are always built around a similar loop of non-local operations. The Python code in Listing 4.4 is tested for 16 iterations, but the IR graph in Figure 4.4 has been shorten to 2 iterations. Here the consecutive Focal operations prevent fusion, leading to as many clusters as iterations in the loop. On the other hand, the succession of Focal operations make this the most computationally intense script of the four. This is amplified by the fact that the script takes no input, but starts with a randomly generated raster. A real-world example of dynamic model that builds on these same principles would be a simulation of flood inundation [2].

Listing 4.4: Game of Life

```
from map import * ## Map Algebra Compiler package

ds = [1024,1024] # raster size
bs = [256,256]   # block size
gs = [16,16]     # group size
N  = 16          # iterations

def life(dem):
  S = [[1,1,1],
       [1,0,1],
       [1,1,1]]
  return convolve(dem,S)

state = rand(seed=N,dtype=U8,ds,bs,gs)
state = state > 128 # uint8 --> bool

for i in range(N):
  nbh = life(state)
  state = (nbh == 3) + (nbh == 2) * state

write(state,'life.tif')
```

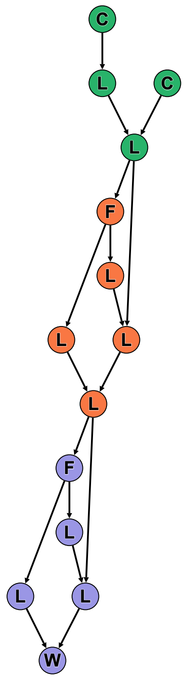


Figure 4.4: Game of Life graph IR, only displaying two iterations.

Table 4.2 lists the individual and collective speed-ups achieved by the parallel and locality optimizations. These speed-ups are measured with respect to the sequential interpreter approach in Figure 3.1a. The parallelization is applied at block level with multiple CPU threads and at cell level with a single GPU. The locality is attained at block level via the scheduling of active jobs and at cell level with fusion. This experiment is meant to demonstrate how locality outweighs parallelism in typical map algebra workloads. The Par-GPU, Par-Fus and Par-Sch columns show this, with fusion and the scheduler bringing larger speed-ups than GPUs. This result validates the compiler approach, given that interpreters cannot optimize for locality. A more detailed analysis can be found in paper 3.

Optimiza-tion / Script	Interp	Par	Par GPU	Par Fus	Par Sch	Par Fus Sch	Par GPU Fus	Par GPU Sch	All Opt.
W. Sum.	1	1.84	2.09	7.42	5.24	7.42	7.74	7.42	7.74
Statistics	1	2.00	2.67	8.00	4.27	8.00	16.00	14.22	18.29
Hillshade	1	1.92	2.26	18.19	8.21	26.17	45.57	29.70	69.56
Life (x16)	1	2.24	3.01	6.24	8.45	15.27	11.57	38.27	81.58
Avg. speed-up	-	2.00	2.48	9.06	6.27	12.41	15.98	18.61	29.94

***Interp**reted* ***Par**allel multi-threaded* ***GPU**-accelerated* ***Fus**ion* ***Sch**eduler* ***All Opt.** = Par,GPU,Fus,Sch*

CPU = Kaveri AMD A10-7850K w/ 13 GB memory *GPU = Spectre (integrated) 3 GB* *Disk = 256 GB SSD Raid0*

Table 4.2: Speed-ups of the parallel and locality optimizations, separate and combined. The reference map algebra interpreter uses no optimizations (left), while the compiler approach employs them all (right).

Figure 4.3 shows the performance of five machines of increasing computational power but fixed memory bandwidth. The numbers are given as the speed-up with respect to the Ivy-Ivy machine, which employs a quad-core CPU but no GPU. All the optimizations from Table 4.2 are now active, thus these results refer to the compiler approach. The machines are labeled as *Host,Device* pairs following the OpenCL model, where a host manages the devices. From left to right, the figure shows how higher compute power brings higher speed-ups to the compute bound scripts. However, memory bound script benefit from little to nothing (i.e. weighted sum) from the increased compute power. More information about the machines and dataset can be found in paper 3.

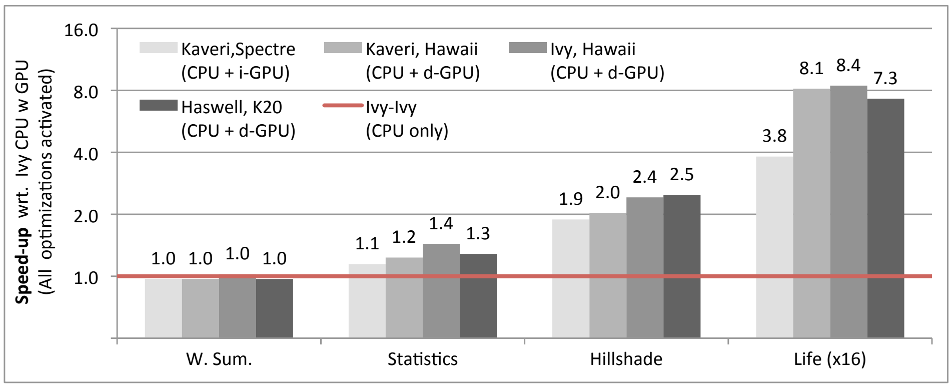


Table 4.3: Workloads performance under different machines of increasing computational power but fixed memory bandiwdth.

4.2 Viewshed Analysis

This section presents a viewshed algorithm that showcases the role of the specialization optimizations. A *viewshed* is the area of a terrain visible to an observer of certain position and altitude above the ground. To grasp the idea, Figure 4.5 depicts the one-dimensional equivalent of a viewshed, where an observer is capable of seeing those points which can be traced by a line-of-sight (LOS). A normal viewshed works very much the same way, just over a two-dimensional terrain typically encoded as a DEM. Figure 2.6 shows an example of normal viewshed in an urban environment. Viewsheds are a versatile tool, often used in optimization problems related to visibility and coverage. Examples are finding the min-cost max-coverage arrangement of radio-transmission towers [31], optimizing the placement of wind turbines for higher power generation and lower visual impact [47], or calculating the shortest flying route for a 3D mapping UAV [18].

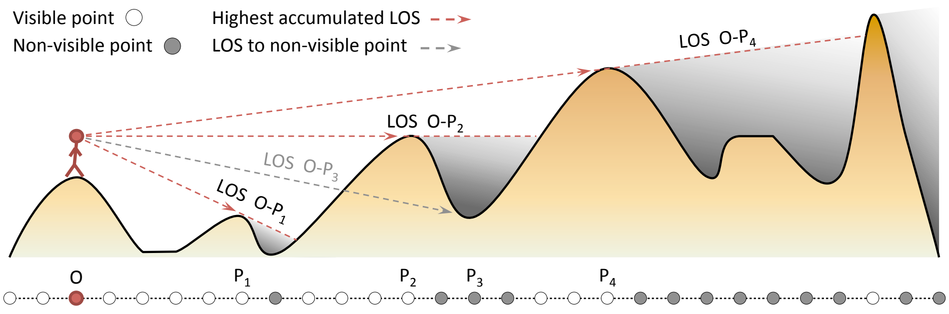


Figure 4.5: Illustrative one-dimensional viewshed

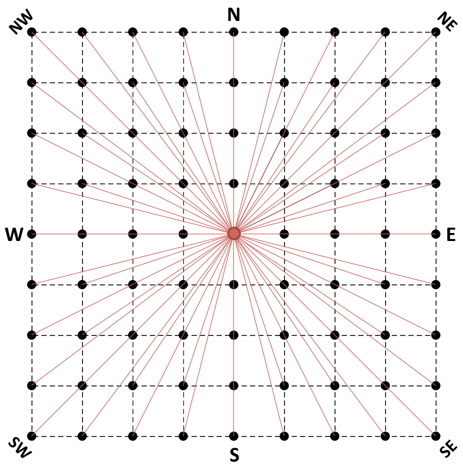
Several viewshed algorithms exist with varying tradeoffs in performance and precision (see Paper 1). The simplest, most precise, but least performant algorithm is *R*3 [30], presented in Listing 4.5. R3 consists of three nested loops and little arithmetic, taking just about 30 lines of pseudo C/C++ code. In brief, the observer exhaustively projects a LOS for every target cell of unknown visibility. A cell is denoted visible when no elevation in the path of the LOS obstructs said LOS. This condition is verified by walking the path, interpolating the elevation, and comparing to the LOS altitude. Although simple, the code entails a complexity of $O(N)$ per cell for an $N \times N$ sized raster. Worse yet, it traverses memory in a non-contiguous fashion and cannot benefit from cache memories. Together, these inefficiencies make the algorithm impractical for large datasets.

The experiment undertaken here employs a faster but less precise viewshed of $O(1)$ complexity, called *Xdraw* [29]. Xdraw is better understood by observing the one-dimensional case first, and then extrapolating to two dimensions. In Figure 4.5, a cell is known visible if its LOS presents a greater slope than the previous highest LOS. Thereby, the point P_3 is tagged non-visible because the slope of LOS OP_3 is lower that the slope of LOS OP_2. Here the visibility can be tested in a single comparison because all the cells lie in the same geometrical plane. However, in the two-dimensional case most LOSs lie in different planes, as can be seen in Figure 4.6a. Xdraw interpolates the LOS slopes to avoid walking each LOS like R3 did, thus compromising accuracy for speed. As a result, Xdraw can build the visibility map in a single incremental pass by computing one cell at a time as it moves from the observer toward the borders.

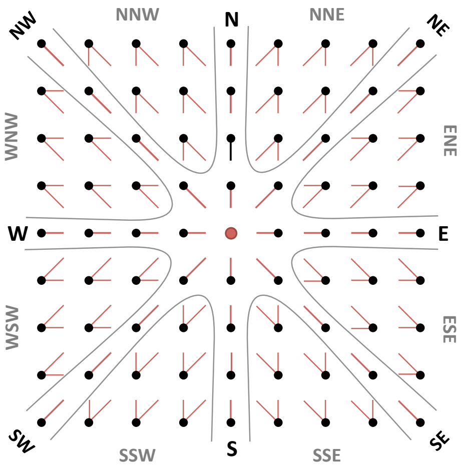
Listing 4.5: R3 algorithm in pseudo C/C++ code

```
dem = read('dem.tif') // digital elevation model
// 'ox', 'oy', 'oh' are inputs arguments to the program
observer = {ox,oy}; // observer X and Y coordinates
obs_elev = dem.at(observer); // elevation at observer cell
altitude = oh + obs_elev; // total observer altitude

for (int x = 0; x < dem.width; x ++) {
  for (int y = 0; y < dem.height; y ++) {
    target = {x,y};
    abs_dif = abs(dif(obsever,target));
    distance = hypot(abs_dif);
    los_steps = max(abs_dif);
    los_slope = (dem.at(target) - altitude) / distance;
    visible = true;

    for (int z = 0; z < los_steps; z ++) {
      coord = coord_in_los(observer,target,z);
      dist = hypot(abs(dif(observer,coord)));
      los_elev = dist * los_slope;
      coord_elev = dem.at(coord); // interpolates
      if (coord_elev >= los_elev) {
        visible = false;
        break;
      }
    }
    view[target] = visible;
  }
}

write(view,'viewshed.tif')
```

Xdraw is an interesting test case because of its predisposition for specialization. In Xdraw, the target cells follow different equations depending on their relative position to the observer. This is reproduced by Listing 4.6, where the calculation of the previous LOS slope is heavily branched. For instance, while a cell in the north direction will only access the neighbor in its south, a cell in the northeast-east sector will have to interpolate its two preceding neighbors (Fig. 4.6b). These discrepancies lead to an irregular



(a) R3 traces the exact LOS from the observer toward all target cells in the raster.



(b) Xdraw approximates the LOSs by interpolating the steepest accumulated LOS.

Figure 4.6: Lines-of-sight (LOSs) from a central observer toward all cells in two viewshed algorithms. R3 traverses each LOS for better accuracy (a), while Xdraw interpolates the LOS slopes to conclude in one pass (b).

and divergent code that goes against the SIMD parallel model, preventing the use of vector instructions on CPUs and severely impacting the performance on GPUs. However, the obstacle can be avoided with a specialized code path for each compass direction and sector, so that the code becomes branchless and can be efficiently executed by a SIMD processor.

Listing 4.6: Generic slope calculation, with a branch per direction

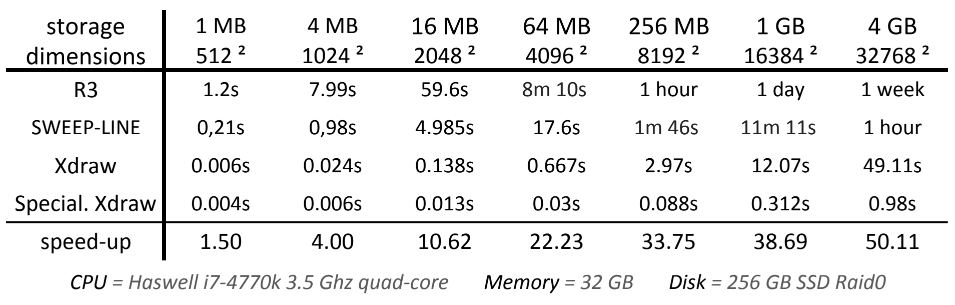
```
...

def prior_slope(slope,obs):
  ox, oy = obs
  x = index(slope,D1)
  y = index(slope,D2)
  dx = abs(x - ox) # for the interpolation in X
  dy = abs(y - oy) # for the interpolation in Y
  w1 = dy / dx * (dx - 1) - (dy - 1)
  w2 = dy / dx * (dx - 1) - (dy - 1)

  if (x == ox) and (y <  oy): # North vertical
    return slope([0,+1])
  if (x == ox) and (y >  oy): # South vertical
    return slope([0,-1])
  if (x >  ox) and (y == oy): # East horizonal
    return slope([-1,0])
  if (x <  ox) and (y == oy): # West horizonal
    return slope([+1,0])

  if (x < oy) and (y < oy) and (dx == dy): # NW diagonal
    return slope([+1,+1])
  if (x > ox) and (y < oy) and (dx == dy): # NE diagonal
    return slope([-1,+1])
  if (x > ox) and (y > oy) and (dx == dy): # SE diagonal
    return slope([-1,-1])
  if (x < ox) and (y > oy) and (dx == dy): # SW diagonal
    return slope([+1,-1])

  if (x < ox) and (y < oy) and (dx > dy): # WNW sector
    return w1*slope([+1,0]) + (1-w1)*slope([+1,+1])
  if (x < ox) and (y < oy) and (dx < dy): # NNW sector
    return w2*slope([0,+1]) + (1-w2)*slope([+1,+1])
  if (x > ox) and (y < oy) and (dx < dy): # NNE sector
    return w2*slope([0,+1]) + (1-w2)*slope([-1,+1])
  if (x > ox) and (y < oy) and (dx > dy): # ENE sector
    return w1*slope([-1,0]) + (1-w1)*slope([-1,+1])
  if (x > ox) and (y > oy) and (dx > dy): # ESE sector
    return w1*slope([-1,0]) + (1-w1)*slope([-1,-1])
  if (x > ox) and (y > oy) and (dx < dy): # SSE sector
    return w2*slope([0,-1]) + (1-w2)*slope([-1,-1])
  if (x < ox) and (y > oy) and (dx < dy): # SSW sector
    return w2*slope([0,-1]) + (1-w2)*slope([+1,-1])
  if (x < ox) and (y > oy) and (dx > dy): # WSW sector
    return w1*slope([+1,0]) + (1-w1)*slope([+1,-1])

...
```

While performance calls for specialization, writing multiple code versions is obviously unproductive. The compiler approach is architected so that this complexity is moved from the front-end into the back-end. Multiple codes are still written, but this is done by the compiler developers, not by the modelers. The specialization is attained with the creation of new IR nodes, Global classes, task logics and skeleton versions. In particular, Xdraw is specialized by the Radial class previously introduced in subsection 2.2.3. Thus, a single Radial operation can substitute lines 15 to 19 in Listing 4.7 and Listing 4.6 all together. This not only drastically simplifies the Python script, but also enables the generation of better code.

Table 4.4 shows the speed-ups obtained by the specialized code for increasing raster sizes. For a small raster of 512 by 512 cells and 1 Mb of storage the specialized Xdraw is only 50% faster than the generic version (Listing 4.7). However, as the size increases the speed-up peaks at about 50x for a raster of a billion cells and 4 GB of space. Additionally, the table also includes the R3 and *SWEEP-LINE* viewshed algorithms for comparison. SWEEP-LINE is an *O(logN)* algorithm [37] with balanced accuracy and cost between R3 and Xdraw. The tested SWEEP-LINE implementation belongs to the GRASS GIS package [38], while we implemented the rest of viewsheds algorithms as described by the Listings. Finally, the last column shows how only the specialized Xdraw maintains competitive times (i.e. in the scale of seconds) as the data sizes surpass the gigabyte mark.

Listing 4.7: Generic Xdraw

```
from map import * # "Parallel Map Algebra" package

def prior_slope(slope,obs):
    ... # defined in Listing 4.6

def epsilon(floating):
  return floating * 10**-5

def xdraw(dem,obs,oh):
  altit = oh + dem[obs]    # total observer altitude
  shift = dem - height     # shifts so dem[obs]==0
  dist  = distance(dem,obs) # distances to observer
  slope = shift / dist     # slopes of all LOSs
        ____________________
  width, height = slope.datasize()  # scan of max slopes
  steps = max(width,height)         # via loop of Focals
  for s in range(steps):
    prior = prior_slope(slope,obs)  # generic but
    slope = max2(slope,prior)       # inefficient
        ____________________
  los_elev = slope * distance
  visible = los_elev - dem < epsilon(dem)
  return con(visible, dem, zeros_like(dem))

dem = read('dem.tif')         # reads elevation model
ox, oy, oh = 50, 70, 5        # observer x, y, z
view = xdraw(dem,[ox,oy],oh)  # computes viewshed
write(view,'view.tif')        # writes visibility map
```



storage dimensions	1 MB 512^2	4 MB 1024^2	16 MB 2048^2	64 MB 4096^2	256 MB 8192^2	1 GB 16384^2	4 GB 32768^2
R3	1.2s	7.99s	59.6s	8m 10s	1 hour	1 day	1 week
SWEEP-LINE	0,21s	0,98s	4.985s	17.6s	1m 46s	11m 11s	1 hour
Xdraw	0.006s	0.024s	0.138s	0.667s	2.97s	12.07s	49.11s
Special. Xdraw	0.004s	0.006s	0.013s	0.03s	0.088s	0.312s	0.98s
speed-up	1.50	4.00	10.62	22.23	33.75	38.69	50.11

CPU = Haswell i7-4770k 3.5 Ghz quad-core *Memory* = 32 GB *Disk* = 256 GB SSD Raid0

Table 4.4: Execution time of different viewsheds for increasing raster sizes, and speed-up of the specialized over the generic Xdraw version.

4.3 Urban Development

This section covers a dynamic cellular automata model for the analysis and forecasting of urban development. Urban development has become an active field of study as 50% of the world population now lives in urban areas. This proportion, which was about 30% during 1950, is growing quickly and could reach 66% by 2050 [63]. It is also estimated that cities already consume 75% of the world's natural resources in just about 5% of its surface. Such high concentrations of human activity create opportunities for economic, cultural and social development, but at the same time it raises risks of poverty, environmental degradation and social inequality [61]. Today, modeling and simulation enable us to identify, study and predict these urban challenges. Urban cellular automata models are one attempt to do that.

A cellular automaton (CA) is classically defined as a lattice of cells whose state evolves according to some fixed rule. Our interpretation is more general: we employ multiple lattices (i.e. rasters) and multiple rules (i.e. map algebra operations) that are not fixed (i.e. change with control flow) and can be probabilistic (i.e. random numbers). Such relaxed formulation provides a useful framework for the quantitative study of complex spatio-temporal phenomena. This fact is well documented in the literature, with numerous raster CA models simulating lava flow [24], wildfires [87], landslides [22], and soil erosion [80] among many others. These works are motivated by the ability of CAs to reproduce the self-organizing[3] nature of spatial phenomena, while still being simple enough to fit in a short list of code. This is also the reason behind the popularity of urban CAs, which gather hundreds of citations in the literature [71].

In this experiment we test a model originally proposed by Wu [94] and later parallelized by Guan [36]. The consensus in urban planning is that cities develop as a combination of global and local factors. Examples of local factors are the terrain elevation, slope, soil type and access by vehicles, while important global factors are the distances to city centers, road networks and public infrastructure. Wu's model combines both types of factors for more realistic results, while still being easy to grasp. The equations are also probabilistic and will draw different outputs depending on the random number generator. As a result, it becomes necessary to couple the model with some type of sensibility analysis to assess its uncertainty. Regarding the calibration and validation parts, these were tackled by the original authors and are omitted here.

Listing 4.8 presents Wu's urban CA model. Lines 3 to 11 define several parameters and coefficients that control the simulation. Lines 13 to 19 read the multiple input rasters employed by the model. Lines 23 to 28 reproduce

[3]Self-organization is a property of many natural and artificial systems whereby macroscopic behaviors emerge from the local interactions of microscopic parts.

Wu's equations for one year of urban simulation. The equations work as follows. Line 23 computes a linear regression with the rasters and coefficients defined above. Line 24 transform the regression result into a probability for urbanization in the range $[0, 1]$. Line 25 zeroes the probability of those cells that are excluded, already urban or isolated. Line 26 translates the probability to a power distribution according to the dispersion parameter. Line 27 applies the annual limit, so that only the desired number of cells becomes urban on average. Finally, line 28 turns into urban those cells whose probability surpasses a random number between $[0, 1]$. More information is given in paper 4 and in the literature [94, 36, 53].

Listing 4.8: Wu's urban cellular automata model

```
 1  from map import * ## Map Algebra Compiler package
 2
 3  a  = 6.4640   # Constant coefficient
 4  b1 = 43.5404  # Elevation coefficient
 5  b2 = 1.9150   # Slope coefficient
 6  b3 = 41.3441  # Distance to city centers coefficients
 7  b4 = 12.5878  # Distance to transportations coefficient
 8  b5 = [0,0,-9.865,-8.746,-9.268,-8.032,-9.169,-8.942,-9.45]
 9  # {water,urban,barren,forest,shrub,woody,herb,crop,wetlad}
10  d  = 5        # dispersion parameter
11  q  = 16000    # max cells to become urban per year
12
13  x1 = read('dem')      # elevation layer
14  x2 = read('slope')    # slope layer
15  x3 = read('center')   # distance to centers layer
16  x4 = read('transp')   # distance to transportations layer
17  x5 = read('landuse')  # land use layer
18  e  = read('excl')     # exclusion layer (e.g. water bodies)
19  s  = read('urban')    # initial state: urban / not-urban
20  N  = 50               # years of simulation i.e. time steps
21
22  for i in range(0,N) :
23      z  = a + b1*x1 + b2*x2 + b3*x3 + b4*x4 + pick(x5,b5)
24      pg = exp(z) / (1 + exp(z))
25      pc = pg * !e * !s * focalSum(s) / (3*3-1)
26      pd = pc * exp(-d * (1 - pc / max(pc)))
27      ps = q * pd / sum(pd)
28      s  = s || ps > rand()
29
30  write(s,'output')
```

The model was run with the statewide dataset of California provided by the original authors [36, 53]. Figure 4.7 shows a small portion corresponding to southern Marin County, 10 km north from San Francisco. The dataset is composed of seven layers of $30m^2$ resolution and 23851×40460 cells each. They store the elevation, slope, land use, excluded sites, distances to city centers and distances to road networks. The seventh layer consists of the present urban areas and is not included in Figure 4.7. The full dataset occupies above 20 GB, which is larger than the 16 GB of memory on the test machine. Note that although the full dataset cannot fit into memory, the spatial decomposition guarantees that the individual blocks, groups and cell will fit at the multiple levels of the memory hierarchy.

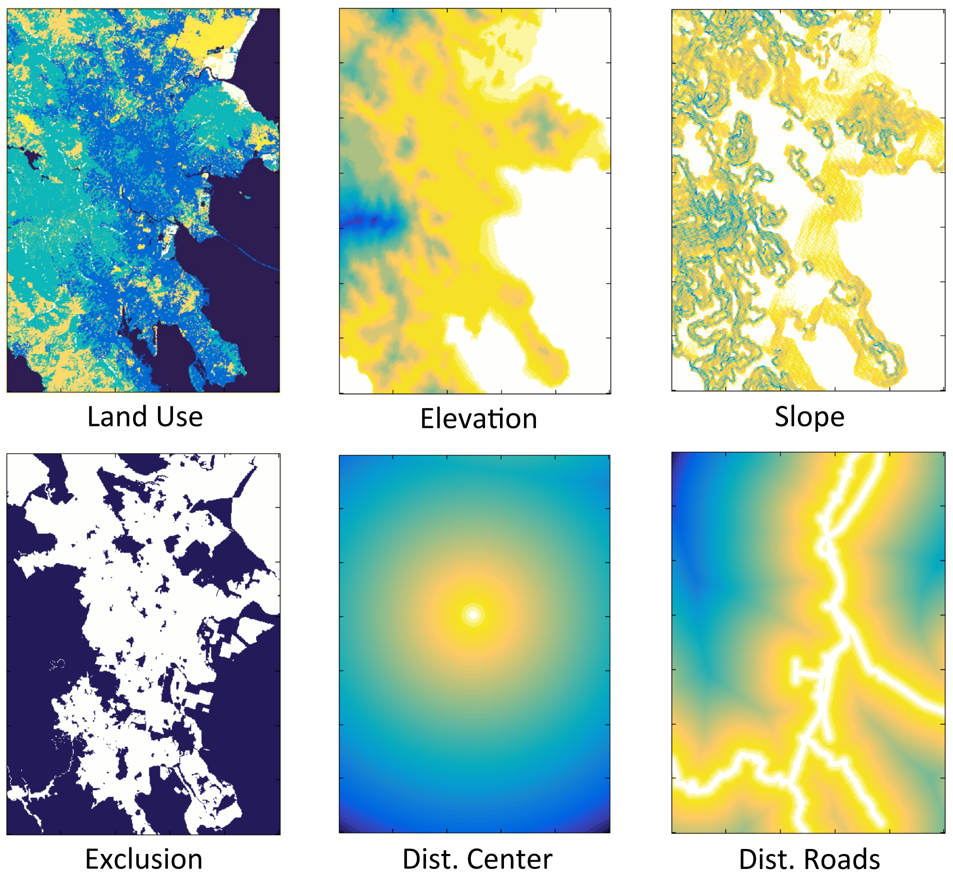
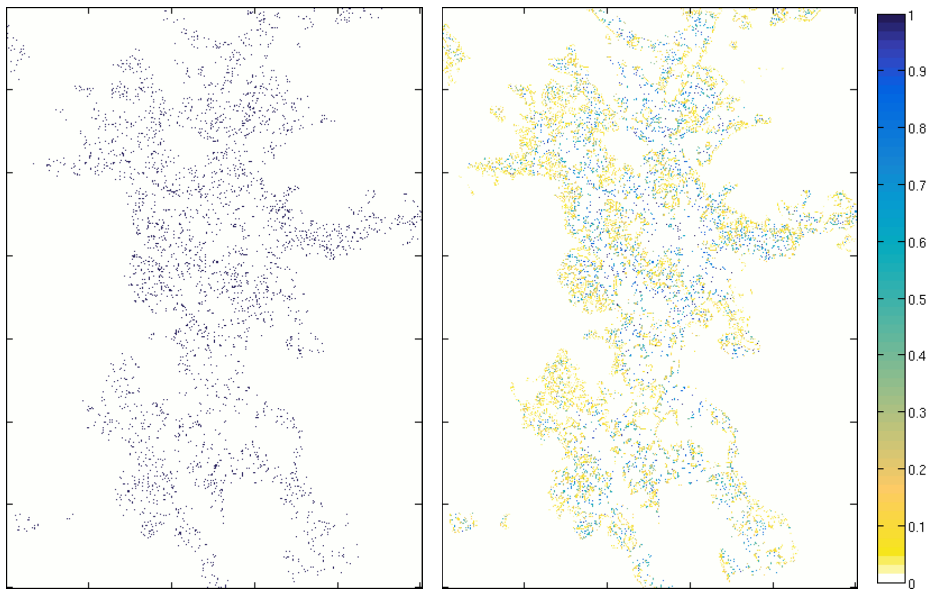


Figure 4.7: California dataset input layers for Marin County

Figure 4.8 shows the outcomes of running Listing 4.8 for the area shown in Figure 4.7. Note that the model needs to run for the whole state of California even though only the County of Marin is shown here. Figure 4.8a represents a single 50-years simulation starting from the present urbanized level. It shows those cells that, at some point during the 50 iterations, develop into new urban areas. Such direct use of the model already gives useful insights on the spatial phenomena, but it is not to be trusted. Urban development is a complex and somewhat spontaneous event, which means perfect predictions are not possible. Figure 4.8b couples the model with the Monte Carlo method to expose the uncertainties in the model. This is basically achieved by running multiple simulations with different random seeds and averaging the output. Running 100 iterations returns the *probability* of urbanization and, while many cells still match side a), their development is not certain anymore as extensive areas of low to medium probability surround them now.



(a) One single execution (b) 100 Monte Carlo iterations

Figure 4.8: Monte Carlo method provides a simple sensibility analysis to assess the uncertainty of the model.

Back to the performance metrics, recall that this thesis does not seek to justify the listed spatial models, but it strives to solve the PPP tradeoff via the compiler approach with a focus on computer performance. In that sense, the selected urban model is clearly not comprehensive enough to steer urban planning. Nevertheless, its current form is sufficient to evaluate the performance of the compiler optimizations. Table 4.5 shows the execution times and speed-ups for this CA as the optimization are activated. It starts with no optimizations, in what would be equivalent to a sequential map algebra interpreter. Then it activates the parallel, locality and sparsity optimizations one after another. The specialization optimizations are always active because they are built into the map algebra patterns. Further details are found in the legend at the bottom of the Table.

Following is a summary of the results in Table 4.5. First, the interpreter is the reference version with the highest loading, computing and storing costs. This column is obviously the slowest version as it suffers from the memory bottleneck. Second, the parallel version overlaps I/O and computation by employing multiple worker threads. This does not reduce the number of loaded blocks or computed jobs, but lessens their total cost. Third, the GPU version relieves compute load by offloading the kernel codes to the accelerator. This decreases the compute cost considerably thanks to the high parallel throughput of GPUs. Fourth, fusion avoids intermediate memory movements at cell level by generating larger kernels. This reduces the number of blocks and jobs, improving the performance at all levels. Fifth, the scheduler prevents further memory movements at block level in collaboration with the cache. This brings more moderate improvements compared to fusion, and only on the I/O part. Sixth, the sparsity optimizations avoid the computation of all those cells falling outside the state of California. This nearly halves the number of blocks and jobs to be processed and roughly doubles the performance. Finally, activating the -O2 optimization level in the C++ compiler brings the last speed-ups. This last column reveals that the logic executed by the workers is also a bottleneck at runtime. Further details can be found in paper 4.

Optimizations	Interpreter (0)	Parallel (1)	GPU (2)	Fusion (3)	Scheduler (4)	Sparse (5)	C++ -O2 (6)	
Execution time	5,183.31	2,274.62	1,700.11	419.68	172.65	95.58	65.68	seconds
Speed-Up	Relative Absolute	2.28 2.28	1.34 3.05	4.05 12.35	2.43 30.02	1.81 54.23	1.46 78.92	times
loading	26.8% 1,387.6	9.5% 214.95 ↓	12.7% 215.06 ↔	23.3% 97.66 ⇣	18.5% 31.97 ↓	13.1% 12.49 ⇣	26.9% 17.69 ↑	
computing	33.8% 1,749.9	18.3% 415.80 ↓	2.7% 45.56 ⬇	3.3% 14.02 ↓	7.00% 12.09 ↔	5.5% 5.27 ⇣	16.9% 11.08 ↑	seconds
storing	25.9% 1,339.9	58.2% 1,323.1 ↔	69.0% 1,173.4 ↔	56.5% 237.04 ⬇	39.5% 68.21 ↓	20.7% 19.82 ⇣	38.6% 25.32 ↑	
# loaded	5,930,120	5,930,120 ↔	5,930,120 ↔	2,583,105 ↓	≈ 761,000 ↓	≈ 342,000 ⇣	≈ 342,000 ↔	
# computed	3,515,600	3,515,600 ↔	3,515,600 ↔	564,000 ⬇	564,000 ↔	308,919 ⇣	308,919 ↔	units
# stored	3,139,600	3,139,600 ↔	3,139,600 ↔	752,000 ↓	≈ 563,000 ⇣	≈ 226,000 ⇣	≈ 226,000 ↔	

__Interpreter__ = sequential w/ no optimizations, *__Parallel__ = multi-threaded,* *__GPU__ = OpenCL GPU,* *__Fusion__ = clusters nodes,* *__Scheduler__ = cache & scheduler,* *__Sparse__ = exploits data sparsity,* *__C++ -O2__ = compiler -O2*

__Execution time__ = wall-clock time, *__Relative Speed-Up__ = execution time from previous optimization / execution time with current optimization,* *__Absolute Speed-Up__ = interpreter execution time / execution time with current optimization*

__loading__ = execution time spent loading data, *__computing__ = execution time spent computing results,* *__storing__ = execution time spent storing data* ⬇ ↓ ⇣ *= considerable, moderate, slight __decrease__*

__# loaded__ = total number of loaded blocks, *__# computed__ = total number of computed jobs,* *__# stored__ = total number of stored blocks* ↔ *= __constant__* ↑ *= slight __increase__*

Table 4.5: Execution times and speed-ups of the urban CA as the optimization are increasingly activated.

Chapter 5

Conclusions

A good dissertation is a done dissertation. A great dissertation is a published dissertation. A perfect dissertation is neither.

~ **wise supervisor to frustrated doktorand**

In this last chapter we recap the problem, why it matters and how we addressed it. We also discuss how the research goals where met, and whether new questions arose. Naturally, the time spent on this thesis was limited and many interesting possibilities had to be ignored. Therefore we also mention the limitations of the approach and some possible further work.

5.1 Summary and Discussion

This thesis has focused on the modeling and simulation of spatial phenomena via digital and computer models. The selected digital models are raster data, for example a digital elevation model of the Earth. The computer models are expressed with map algebra, a scripting language for raster spatial analysis. The topic matters because, in the face of a growing world, modeling and simulation provides valuable tools to study, plan and optimize many spatial problems with far-reaching societal impact. An example is to employ topographical and elevation models to determine the most optimal layout of a new highway.

The problem addressed by our research is the need for productivity, performance and portability. Traditional map algebras are not able to cope with the increasing volumes of spatial data and with the parallel evolution of computer architectures without compromising one of the three qualities. However, the three are necessary for the gradual development of computer models (Figure 1.7): without productivity, the development (e.g. design, implementation, testing) becomes lengthy and expensive; without performance, the assessment (e.g. uncertainty, calibration, validation) takes too long to execute; and without portability, the operation (e.g. scaling, deployment) would not be possible across different machines.

The proposed solution is a compiler approach to map algebra that breaks the PPP tradeoff. Thereby, productivity, performance and portability can be addressed independently of each other. Productivity derives from a simple Python interface adapted to the modeler's ability and domain. Performance comes from the compiler optimizations and code generation targeting modern architectures. Portability is achieved with an IR which splits interface from execution, while preserving the semantics.

With said hypotheses we built a prototype of a map algebra compiler [5] to experiment with. We first tested simple map algebra workloads that resemble building blocks of typical spatial models. This confirmed our suspicion that memory, and not computation, is the performance bottleneck in most workloads. We then tested more complex scripts dealing with spatial phenomena of direct interest to modelers. They served to evaluate the multiple performance optimizations incorporated in the compiler.

The experiments confirmed the performance quality of the solution from a quantitative point of view. We also included the Python scripts to show that, qualitatively, the productivity of map algebra remained intact. The portability aspect was scarcely evaluated, since only multi-core CPUs and many-core GPUs where employed. In retrospective, the objectives where met because our research goals were narrowed toward performance. However, many interesting issues were left untouched and new questions also came up during the research. Surely, the problem at hand requires more ideas, more experiments, more analyses and further developments.

5.2 Limitations and Further Work

First, it is important to note that our approach is confined to map algebra and raster data. Vector data is not covered and, although not a fatal limitation, many users would request such feature. An initial step in this direction could be the addition of vector-to-raster conversion operations. While simple, this would raise precision problems due to the way rasterization works[1]. The right approach is to treat vectors as primitive data and enhance the compiler and its IR to that end. This, however, requires such endeavor that would become not just another thesis, but a whole research venture.

Second, while the prototype served our research well, this work would still benefit from many additions: from more scripts for different spatial

[1]Rasterization is the process by which geometrical shapes (i.e. vector data) are converted into pixels (i.e. raster cells) that form a bitmap image (i.e. raster data).

phenomena, to new programming constructs for productivity, to better optimizations for performance, to more supported architectures for portability.

Although our tested scripts were selected to cover typical workloads, they are not complete. Real-life computer models are considerably more complicated and have dozens of parameters that need tuning. Furthermore, the execution of dynamic models is affected by the input data, therefore those models should also be evaluated with atypical data.

Productivity is hard to assess since all programmers have their own workflow and preferences. As a result, evaluating the productivity requires a large community with people of diverse backgrounds. Moreover, we have not covered topics like debugging and testing, very necessary to efficiently find bugs, solve errors and resume the development.

Performance can still be improved with more compiler optimizations, better code generation and more scalable runtime. The fusion and scheduling optimizations are complex problems that would benefit from better search algorithms. LLVM [48] would be a better fit than OpenCL, because it opens new optimization possibilities for the kernels. The runtime is a complex concurrent system that could probably be optimized with wait/lock-free algorithms [41].

Portability is not hard to improve, but it is the most time consuming of the qualities in this heterogeneous era where computer architectures continue to evolve and differentiate. Besides CPUs and GPUs, it would be interesting to test distributed systems such as clusters and supercomputers.

Regardless of the limitations and much future possibilities, this work has already delivered what it promised. We found and solved an important problem in a field of increasing relevance and large social impact. Our compiler approach to map algebra establishes a basis for future successful raster spatial modeling.

Bibliography

[1] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM.

[2] Bates, P. D. and De Roo, A. (2000). A simple raster-based model for flood inundation simulation. *Journal of hydrology*, 236(1-2):54–77.

[3] Bohr, M. (2007). A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13.

[4] Campbell, J. and Shin, M. (2015). *Geographic information system basics.* The Saylor Foundation. Accessed: 2018-12-09.

[5] Carabaño, J. (2018). Code repository of a prototype implementation of the map algebra compiler. https://www.github.com/jcaraban/map. Accessed: 2019-25-01.

[6] Carabaño, J., Sarjakoski, T., and Westerholm, J. (2015). Efficient implementation of a fast viewshed algorithm on simd architectures. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Disturbed, and Network-Based Processing*, pages 199–202. IEEE.

[7] Carabaño, J. and Westerholm, J. (2017). From python scripting to parallel spatial modeling: Cellular automata simulations of land use, hydrology and pest dynamics. In *Proceedings of the 25th Euromicro International Conference on Parallel, Disturbed, and Network-Based Processing*, pages 511–518. IEEE.

[8] Carabaño, J. and Westerholm, J. (2019). A compiler and runtime approach to parallel spatial modeling. Technical Report 1200, TUCS. ISSN 1239-1891, No 1203.

[9] Carabaño, J., Westerholm, J., and Sarjakoski, T. (2018). A compiler approach to map algebra: automatic parallelization, locality optimization,

and gpu acceleration of raster spatial analysis. *GeoInformatica*, 22(2):211–235.

[10] Centre for Remote Imaging, Sensing and Processing CRISP (1997). Principles of remote sensing. https://crisp.nus.edu.sg/~research/tutorial/rsmain.htm. Accessed: 2018-12-08.

[11] Cheng, G., Liu, L., Jing, N., Chen, L., and Xiong, W. (2012). General-purpose optimization methods for parallelization of digital terrain analysis based on cellular automata. *Computers & Geosciences*, 45:57–67.

[12] Cheramie, K. D. (2011). The scale of nature: Modeling the mississippi river. *Places Journal*. Available at https://doi.org/10.22269/110321. Accessed: 2018-12-08.

[13] Chomsky, N. and Lightfoot, D. W. (2002). *Syntactic structures*. Walter de Gruyter.

[14] Cole, M. I. (1989). *Algorithmic skeletons: structured management of parallel computation*. Pitman London.

[15] Cooper, K. and Torczon, L. (2011). *Engineering a compiler*. Elsevier.

[16] Copeland, J. (2006). Colossus and the rise of the modern computer. In Copeland, J., editor, *Colossus: The Secrets of Bletchley Park's Codebreaking Computers*. Oxford University Press, Oxford.

[17] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured programming*. Academic Press Ltd.

[18] De Filippis, L., Guglieri, G., and Quagliotti, F. (2012). Path planning strategies for uavs in 3d environments. *Journal of Intelligent & Robotic Systems*, 65(1-4):247–264.

[19] Dean, J., Patterson, D., and Young, C. (2018). A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29.

[20] Department of Defense, Department of the Army, Office of the Chief Signal Officer (1927). Great Mississippi Flood of 1927. https://archive.org/details/mississippi_flood_1927. Accessed: 2018-12-08.

[21] Di Gregorio, S., Filippone, G., Spataro, W., and Trunfio, G. A. (2013). Accelerating wildfire susceptibility mapping through gpgpu. *Journal of Parallel and Distributed Computing*, 73(8):1183–1194.

[22] Di Gregorio, S., Kongo, R., Siciliano, C., Sorriso-Valvo, M., and Spataro, W. (1999). Mount ontake landslide simulation by the cellular automata model sciddica-3. *Physics and Chemistry of the Earth, Part A: Solid Earth and Geodesy*, 24(2):131–137.

[23] Downey, A. B. (2017). *Modeling and Simulation in Python.* Green Tea Press. Available at http://greenteapress.com/wp/modsimpy.

[24] D'Ambrosio, D., Filippone, G., Rongo, R., Spataro, W., and Trunfio, G. A. (2012). Cellular automata and gpgpu: an application to lava flow modeling. *International Journal of Grid and High Performance Computing (IJGHPC)*, 4(3):30–47.

[25] Faggin, F., Hoff, M. E., Mazor, S., and Shima, M. (1996). The history of the 4004. *IEEE Micro*, 16(6):10–20.

[26] Farr, T. G., Rosen, P. A., Caro, E., Crippen, R., Duren, R., Hensley, S., Kobrick, M., Paller, M., Rodriguez, E., Roth, L., et al. (2007). The shuttle radar topography mission. *Reviews of geophysics*, 45(2).

[27] Financial Times (2015). Intel chief raises doubts over moore's law. https://www.ft.com/content/36b722bc-2b49-11e5-8613-e7aedbb7bdb7. Accessed: 2018-12-08.

[28] Flachs, B., Gschwind, M., Yamazaki, T., Hopkins, M., Hofstee, H. P., and Watanabe, Y. (2006). Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26:10–24.

[29] Franklin, W. and Ray, C. (1994). Higher isn't necessarily better: Visibility algorithms and experiments. *Advances in GIS research: sixth international symposium on spatial data handling*, 2:1–22.

[30] Franklin, W., Ray, C., and Mehta, S. (1994). Geometric algorithms for siting of air defense missile batteries. Technical Report 2756, US Army Research Office Scientific Services Program.

[31] Franklin, W. R. (2002). Siting observers on terrain. In *Advances in Spatial Data Handling*, pages 109–120. Springer Berlin Heidelberg.

[32] Fuglsang, M., Hansen, H. S., and Münier, B. (2011). Accessibility analysis and modelling in public transport networks–a raster based approach. In *International Conference on Computational Science and Its Applications*, pages 207–224. Springer.

[33] González-Vélez, H. and Leyton, M. (2010). A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160.

[34] Goodchild, M. F. (1992). Geographical information science. *International journal of geographical information systems*, 6(1):31–45.

[35] Guan, Q. and Clarke, K. C. (2010). A general-purpose parallel raster processing programming library test application using a geographic cellular automata model. *International Journal of Geographical Information Science*, 24(5):695–722.

[36] Guan, Q., Shi, X., Huang, M., and Lai, C. (2016). A hybrid parallel cellular automata model for urban growth simulation over gpu/cpu heterogeneous architectures. *International Journal of Geographical Information Science*, 30(3):494–514.

[37] Haverkort, H., Toma, L., and Zhuang, Y. (2009a). Computing visibility on terrains in external memory. *Journal of Experimental Algorithmics (JEA)*, 13:5.

[38] Haverkort, H., Toma, L., and Zhuang, Y. (2009b). r.viewshed module from grass gis 7.0. http://grass.osgeo.org/grass70/manuals/r.viewshed.html. Accessed: 2019-25-01.

[39] Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

[40] Hennessy, J. L. and Patterson, D. A. (2019). A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60.

[41] Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149.

[42] Horn, B. K. (1981). Hill shading and the reflectance map. *Proceedings of the IEEE*, 69(1):14–47.

[43] Horowitz, M., Alon, E., Patil, D., Naffziger, S., Kumar, R., and Bernstein, K. (2005). Scaling, power, and the future of cmos. In *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, pages 7–pp. IEEE.

[44] Hutchinson, D., Lanthier, M., Maheshwari, A., Nussbaum, D., Roytenberg, D., and Sack, J.-R. (1996). Parallel neighbourhood modelling. In *Proceedings of the fourth ACM workshop on Advances in geographic information systems - GIS '96*, pages 25–34. ACM Press.

[45] International Organization for Standardization ISO (2016). ISO/IEC 25022:2016, Systems and software Quality Requirements and Evaluation (SQuaRE). https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed: 2018-12-09.

[46] Khronos Group (2008). Open computing language. https://www.khronos.org/opencl. Accessed: 2018-12-09.

[47] Ladenburg, J., Termansen, M., and Hasler, B. (2013). Assessing acceptability of two onshore wind power development schemes: A test of viewshed effects and the cumulative effects of wind turbines. *Energy*, 54:45–54.

[48] Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA.

[49] Lillesand, T., Kiefer, R. W., and Chipman, J. (2014). *Remote sensing and image interpretation*. John Wiley & Sons.

[50] Longley, P. A., Goodchild, M. F., Maguire, D. J., and Rhind, D. W. (2005). *Geographic information systems and science*. John Wiley & Sons.

[51] Malczewski, J. (2004). Gis-based land-use suitability analysis: a critical overview. *Progress in planning*, 62(1):3–65.

[52] McLeod, K. S. (2000). Our sense of snow: the myth of john snow in medical geography. *Social science & medicine*, 50(7-8):923–935.

[53] Miao, J., Guan, Q., and Hu, S. (2017). prpl+ pgtiol: The marriage of a parallel processing library and a parallel i/o library for big raster data. *Environmental modelling & software*, 96:347–360.

[54] Mitchell, A. (2005). *The ESRI Guide to GIS Analysis: Volume 2: Spatial Measurements & Statistics*, volume 2. ESRI press Redlands, CA.

[55] Moore, G. E. (2006). Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35.

[56] Moore, I. D., Gessler, P., Nielsen, G., and Peterson, G. (1993). Soil attribute prediction using terrain analysis. *Soil Science Society of America Journal*, 57(2):443–452.

[57] Morton, G. M. (1966). A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM.

[58] Munoz, S. E., Giosan, L., Therrell, M. D., Remo, J. W. F., Shen, Z., Sullivan, R. M., Wiman, C., O'Donnell, M., and Donnelly, J. P. (2018). Climatic control of mississippi river flood hazard amplified by river engineering. *Nature*, 556:95 EP –.

[59] NASA Earth Observatory (2000). Moderate Solution Imaging Spectroradiometer MODIS. https://www.nasa.gov/multimedia/imagegallery/image_feature_300.html. Accessed: 2018-12-08.

[60] National Aeronautics and Space Administration NASA (2018). Images of Change. https://climate.nasa.gov/images-of-change. Accessed: 2019-25-01.

[61] Netzband, M., Stefanov, W. L., and Redman, C. (2007). *Applied remote sensing for urban planning, governance and sustainability.* Springer Science & Business Media.

[62] Nvidia (2007). Cuda programming guide. https://docs.nvidia.com/cuda. Accessed: 2018-12-09.

[63] Organization, W. H. and UN-Habitat (2016). Global report on urban health. Technical report, World Health Organization.

[64] O'sullivan, D. and Unwin, D. (2014). *Geographic information analysis.* John Wiley & Sons.

[65] Planchon, O. and Darboux, F. (2002). A fast, simple and versatile algorithm to fill the depressions of digital elevation models. *Catena*, 46(2-3):159–176.

[66] Pullar, D. (2001). Mapscript: A map algebra programming language incorporating neighborhood analysis. *GeoInformatica*, 5(2):145–163.

[67] Pullar, D. (2003). Simulation modelling applied to runoff modelling using mapscript. *Transactions in GIS*, 7(2):267–283.

[68] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2005). Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, page 268. ACM.

[69] Qin, C.-Z., Zhan, L.-J., Zhu, A.-X., and Zhou, C.-H. (2014). A strategy for raster-based geocomputation under different parallel computing platforms. *International Journal of Geographical Information Science*, 28(11):2127–2144.

[70] Rupp, K. (2018). Years of microprocessor trend data. https://github.com/karlrupp/microprocessor-trend-data. Accessed: 2018-12-08.

[71] Santé, I., García, A. M., Miranda, D., and Crecente, R. (2010). Cellular automata models for the simulation of real-world urban processes: A review and analysis. *Landscape and Urban Planning*, 96(2):108–122.

[72] Sebesta, R. W. (2015). *Concepts of programming languages.* Pearson.

[73] Shapiro, M. and Westervelt, J. (1994). r. mapcalc: An algebra for gis and image processing. Technical report, Construction Engineering Research Lab (ARMY) Champaign IL.

[74] Shook, E., Hodgson, M. E., Wang, S., Behzad, B., Soltani, K., Hiscox, A., and Ajayakumar, J. (2016). Parallel cartographic modeling: a methodology for parallelizing spatial data processing. *International Journal of Geographical Information Science*, 30(12):2355–2376.

[75] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354.

[76] Steinbach, M. and Hemmerling, R. (2012). Accelerating batch processing of spatial raster analysis using gpu. *Computers & Geosciences*, 45:212 – 220.

[77] Steuwer, M., Kegel, P., and Gorlatch, S. (2011). Skelcl-a portable skeleton library for high-level gpu programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1176–1182. IEEE.

[78] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210.

[79] Takeyama, M. and Couclelis, H. (1997). Map dynamics: integrating cellular automata and gis through geo-algebra. *International Journal of Geographical Information Science*, 11(1):73–91.

[80] Ting, M., Cheng-Hu, Z., and Qiang-Guo, C. (2009). Modeling of hillslope runoff and soil erosion at rainfall events using cellular automata approach. *Pedosphere*, 19(6):711–718.

[81] Tomlin, C. and Berry, J. (1979). Mathematical structure for cartographic modeling in environmental analysis. In *Proceedings of the American Congress on Surveying and Mapping annual meeting.*

[82] Tomlin, C. D. (1980). *The map analysis package.* Yale University School of Forestry.

[83] Tomlin, C. D. (1994). Map algebra: one perspective. *Landscape and Urban Planning*, 30(1-2):3–12.

[84] Tomlin, C. D. (2013). *GIS and cartographic modeling*, volume 380. Esri Press Redlands, CA.

[85] Tomlin, C. D. (2017). Cartographic modeling. *International Encyclopedia of Geography*, pages 1–6.

[86] Tracz, W. J. (1979). Computer programming and the human thought process. *Software: Practice and Experience*, 9(2):127–137.

[87] Trunfio, G. A., D'Ambrosio, D., Rongo, R., Spataro, W., and Di Gregorio, S. (2011). A new algorithm for simulating wildfire spread through cellular automata. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 22(1):6.

[88] Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36.

[89] Wall, D. W. (1991). Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188, New York, NY, USA. ACM.

[90] Wesselung, C. G., KARSSENBERG, D.-J., Burrough, P. A., and van Deursen, W. P. (1996). Integrating dynamic environmental models in gis: the development of a dynamic modelling language. *Transactions in GIS*, 1(1):40–48.

[91] WikiChip (2017). Intel's coffee lake microarchitecture. https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake. Accessed: 2018-12-09.

[92] WikiChip (2018). Nvidias's tegra xavier system-on-chip. https://en.wikichip.org/wiki/nvidia/tegra/xavier. Accessed: 2018-12-09.

[93] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

[94] Wu, F. (2002). Calibration of stochastic cellular automata: the application to rural-urban land conversions. *International Journal of Geographical Information Science*, 16(8):795–818.

[95] Wu, Y., Ge, Y., Yan, W., and Li, X. (2007). Improving the performance of spatial raster analysis in gis using gpu. In *Geoinformatics 2007: Geospatial Information Technology and Applications*, volume 6754, page 67540P. International Society for Optics and Photonics.

[96] Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24.

[97] Yuan, M. (1994). Wildfire conceptual modeling for building gis space-time models. In *proceedings of GIS/LIS*, volume 94, pages 860–869.

ISBN 978-952-12-3853-6
ISSN 1239-1883

Jesús Carabaño Bravo A Compiler Approach to Map Algebra for Raster Spatial Modeling

Jesús Carabaño Bravo A Compiler Approach to Map Algebra for Raster Spatial Modeling

Jesús Carabaño Bravo A Compiler Approach to Map Algebra for Raster Spatial Modeling