sid.inpe.br/mtc-m21c/2021/02.17.20.54-TDI

PERFORMANCE OPTIMIZATION OF THE MGB HYDROLOGICAL MODEL FOR MULTI-CORE AND GPU ARCHITECTURES

Henrique Rennó de Azeredo Freitas

Doctoral thesis of the Graduate Program in Applied Computing, advised by Ph.D. Celso Luiz Mendes, approved on January 25th, 2021.

URL of the original document:
<http://urlib.net/8JMKD3MGP3W34R/447AB2L>

INPE
São José dos Campos
2021

PUBLISHED BY:

Instituto Nacional de Pesquisas Espaciais - INPE
Coordenação de Ensino, Pesquisa e Extensão (COEPE)
Divisão de Biblioteca (DIBIB)
CEP 12.227-010
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/7348
E-mail: pubtc@inpe.br

sid.inpe.br/mtc-m21c/2021/02.17.20.54-TDI

PERFORMANCE OPTIMIZATION OF THE MGB HYDROLOGICAL MODEL FOR MULTI-CORE AND GPU ARCHITECTURES

Henrique Rennó de Azeredo Freitas

Doctoral thesis of the Graduate Program in Applied Computing, advised by Ph.D. Celso Luiz Mendes, approved on January 25th, 2021.

URL of the original document:
<http://urlib.net/8JMKD3MGP3W34R/447AB2L>

INPE
São José dos Campos
2021

Cataloging in Publication Data

Freitas, Henrique Rennó de Azeredo.

F884p Performance optimization of the MGB hydrological model for multi-core and GPU architectures / Henrique Rennó de Azeredo Freitas. – São José dos Campos : INPE, 2021.

xxiv + 79 p. ; (sid.inpe.br/mtc-m21c/2021/02.17.20.54-TDI)

Thesis (Doctorate in Applied Computing) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2021.

Guiding : Dr. Celso Luiz Mendes.

1. Computer systems performance. 2. CPU/GPU. 3. Roofline model. 4. Hydrology models. 5. Parameterization. I.Title.

CDU 004.78

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÕES

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

DEFESA FINAL DE TESE DE HENRIQUE RENNÓ DE AZEREDO FREITAS

No dia 25 de janeiro de 2021, às 14h, por videoconferência, o(a) aluno(a) mencionado(a) acima defendeu seu trabalho final (apresentação oral seguida de arguição) perante uma Banca Examinadora, cujos membros estão listados abaixo. O(A) aluno(a) foi APROVADO(A) pela Banca Examinadora por unanimidade, em cumprimento ao requisito exigido para obtenção do Título de Doutor em Computação Aplicada. O trabalho precisa da incorporação das correções sugeridas pela Banca Examinadora e revisão final pelo(s) orientador(es).

Novo Título: "Performance optimization of the MGB hydrological model for multi-core and GPU architectures"

Eu, Haroldo Fraga de Campos Velho, como Presidente da Banca Examinadora, assino esta ATA em nome de todos os membros.

Membros da Banca

Dr. Haroldo Fraga de Campos Velho, Presidente INPE.
Dr. Celso Luiz Mendes, Orientador(a) INPE
Dr. Stephan Stephany, Membro da Banca, INPE
Dr. João Ricardo de Freitas Oliveira, Membro da Banca, INPE
Dr. Walter Collischonn, Convidado(a) UFRGS
Dr. Daniel Andrés Rodriguez, Convidado(a) UFRJ

Referência: Processo nº 01340.000470/2021-18 SEI nº 6405980

"Reserve your right to think, for even to think wrongly is better than not to think at all."

Hypatia of Alexandria

To my parents Lenita and Ramon

ACKNOWLEDGEMENTS

I am deeply thankful to my parents who have always supported me.

I am grateful to my advisor for the opportunity to develop this work.

I thank to Instituto de Pesquisas Hidráulicas - Universidade Federal do Rio Grande do Sul (IPH-UFRGS, Brazil) for the availability of the MGB hydrological model and datasets, and also to Aleksandar Ilic from Instituto de Engenharia de Sistemas e Computadores - Investigação e Desenvolvimento (INESC-ID, Portugal) and Sergio Rosim from Instituto Nacional de Pesquisas Espaciais (INPE, Brazil) for the availability of High Performance Computing systems.

I also thank to Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) for the financial support.

ABSTRACT

Large-scale hydrological models are extensively used for the understanding of watershed processes with applications in water resources, climate change, land use, and forecast systems. The quality of the hydrological results mainly depends on calibrating the optimal sets of watershed parameters, a time-consuming task that requires repeated hydrological model simulations. The ever-growing availability of hydrometeorological data from extensive regions also contributes to the increase in the demand for more computational resources. The performance of optimization methods in hydrological applications has been continuously addressed. However, improving the performance of an application on a modern computer requires a detailed investigation about the interaction between the application and the underlying system, to find the techniques that provide the best performance improvements. This thesis aims at performance optimizations on the well-established MGB hydrological model (simulation) and the MOCOM-UA method (calibration) for real-world input datasets, the Purus (Brazil) and Niger (Africa) watersheds. The optimization strategies investigated in this thesis target state-of-the-art CPU and GPU systems by exploiting techniques that include AVX-512 vectorization, and multi-core (CPU) and many-core (GPU) parallelisms, to increase the usefulness of both simulation and calibration using the MGB model. Significant speedups of up to 20× were achieved on CPU with the proposed optimizations, while the roofline analysis confirmed that the CPU and GPU optimizations more effectively exploited the hardware resources, and improved the overall performance of the MGB model. An additional scalability analysis using a miniapp of the MGB model indicated that speedups up to 24× (CPU) and 65× (GPU) can be achieved for larger problem sizes. Moreover, the accuracy of the simulated results between the nonoptimized and optimized implementations was quantitatively evaluated, reaching maximum relative errors of approximately 6 % for discharges and objective functions. The investigated techniques applied on the MGB model are also valid for other scientific applications where a few key parts dominate the execution time when processing a large amount of data. Carefully employing these techniques to optimize such parts may significantly enhance the overall application performance on current CPUs and GPUs.

Keywords: computer systems performance, CPU/GPU, roofline model, hydrology models, parameterization

OTIMIZAÇÃO DE DESEMPENHO DO MODELO HIDROLÓGICO MGB PARA ARQUITETURAS MULTI-CORE E GPU

RESUMO

Modelos hidrológicos de bacias de grande escala são amplamente utilizados para a compreensão dos processos de bacias hidrográficas com aplicações em recursos hídricos, mudanças climáticas, uso da terra, e sistemas de previsão. A qualidade dos resultados hidrológicos depende principalmente em calibrar os conjuntos ótimos de parâmetros da bacia, uma tarefa demorada que exige repetidas simulações do modelo hidrológico. A crescente disponibilidade de dados hidrometeorológicos provenientes de regiões extensas também contribui para o aumento na demanda por mais recursos computacionais. O desempenho de métodos de otimização em aplicações hidrológicas tem sido continuamente abordado. Entretanto, melhorar o desempenho de uma aplicação em um computador moderno exige uma investigação detalhada sobre a interação entre a aplicação e o sistema, a fim de encontrar as técnicas que fornecem os melhores desempenhos. Esta tese busca otimizações de desempenho nos já bem estabelecidos modelo hidrológico MGB (simulação) e método MOCOM-UA (calibração) para conjuntos de dados de entrada reais, as bacias do Purus (Brasil) e Niger (África). As estratégias de otimização investigadas nesta tese visam sistemas computacionais CPU+GPU atuais explorando técnicas que incluem vetorização AVX-512, e paralelismos multi-core (CPU) e many-core (GPU) para aumentar a utilidade de ambas simulação e calibração utilizando o modelo MGB. Speedups significativos de até 20× foram obtidos em CPU com as otimizações propostas, enquanto que a análise roofline confirmou que as otimizações em CPU e GPU exploraram mais efetivamente os recursos de hardware, e melhoraram o desempenho geral do modelo MGB. Uma análise adicional de escalabilidade utilizando um miniapp do modelo MGB indicou que speedups até 24× (CPU) e 65× (GPU) podem ser obtidos para tamanhos de problema maiores. Além disso, a acurácia dos resultados simulados entre as implementações não-otimizada e otimizada foi quantitativamente avaliada, atingindo erros relativos máximos de aproximadamente 6 % para vazões e funções objetivo. As técnicas investigadas aplicadas no modelo MGB também são válidas para outras aplicações científicas onde algumas poucas partes cruciais dominam o tempo de execução ao processar uma grande quantidade de dados. Empregando cuidadosamente essas técnicas para otimizar tais partes pode melhorar significativamente o desempenho geral da aplicação em CPUs e GPUs atuais.

Palavras-chave: desempenho de sistemas computacionais, CPU/GPU, modelo roofline, modelos hidrológicos, parametrização

LIST OF FIGURES

Page

LIST OF TABLES

Page

LIST OF ABBREVIATIONS

1D	–	One Dimensional
2D	–	Two Dimensional
AMALGAM	–	A Multi-Algorithm Genetically Adaptive Multi-objective method
AI	–	Arithmetic Intensity
ANA	–	Agência Nacional de Águas
API	–	Application Programming Interface
AVX	–	Advanced Vector Extensions
AVX2	–	Advanced Vector Extensions 2
AVX-512	–	512-bit Advanced Vector Extensions
CAL	–	Calibration
CARM	–	Cache-Aware Roofline model
CentOS	–	Community Enterprise Operating System
CON	–	Routine continuity
CPTEC	–	Centro de Previsão de Tempo e Estudos Climáticos
CPU	–	Central Processing Unit
CUDA	–	Compute Unified Device Architecture
DIS	–	Routine discharge
DP	–	Double Precision
DYRIM	–	Digital Yellow River Integrated Model
DRAM	–	Dynamic Random Access Memory
EGA	–	Epidemic Genetic Algorithm
ERR	–	Systematic Error
FLOP	–	Floating-point Operation
FMA	–	Fused Multiply-Add
FPGA	–	Field Programmable Gate Array
FSDHM	–	Fully Sequential Dependent Hydrological Model
GIS	–	Geographic Information System
GNU	–	GNU's Not Unix
GPU	–	Graphics Processing Unit
HPC	–	High Performance Computing
HRU	–	Hydrological Response Unit
ID	–	Identification
I/O	–	Input/Output
INPE	–	Instituto Nacional de Pesquisas Espaciais
IPH	–	Instituto de Pesquisas Hidráulicas
L1	–	L1 cache
L2	–	L2 cache
L3	–	L3 cache
L1d	–	L1 data cache
L1i	–	L1 instruction cache

LLC	–	Last Level Cache
MAMEO	–	Multi-core Aware Multi-objective Evolutionary Optimization
MGB	–	Modelo de Grandes Bacias
MOCOM-UA	–	Multi-Objective Complex Evolution - University of Arizona
MPI	–	Message Passing Interface
NC	–	Number of Catchments
NCORES	–	Number of CPU Cores
NRMAX	–	Number of worst-ranked samples
NSE	–	Nash-Sutcliffe model Efficiency coefficient
NT	–	Number of Time steps
NTH	–	Number of Threads
O()	–	Big O Notation
OpenACC	–	Open Accelerators
OpenCL	–	Open Computing Language
OpenMP	–	Open Multi-Processing
ORM	–	Original Roofline Model
PAPI	–	Performance API
PGI	–	Portland Group Incorporated
RMAX	–	Maximum Pareto ranking of samples
SCE-UA	–	Shuffled Complex Evolution - University of Arizona
SIM	–	Simulation
SIMD	–	Single Instruction Multiple Data
SIMT	–	Single Instruction Multiple Threads
SM	–	Streaming Multiprocessor
SP	–	Single Precision
SRTM	–	Shuttle Radar Topography Mission
SSE	–	Streaming SIMD Extensions
STE	–	Routine timestep
SUFI2	–	Sequential Uncertainty Fitting 2
SWAT	–	Soil and Water Assessment Tool
SWAT-CUP	–	SWAT Calibration and Uncertainty Procedures
TIN	–	Triangulated Irregular Network
tRIBS	–	TIN-based Real-time Integrated Basin Simulator
UFRGS	–	Universidade Federal do Rio Grande do Sul

LIST OF SYMBOLS

α	–	Constant for numerical stability
b	–	Shape measure from relation between storage and saturation
b_g	–	Number of bits transferred per clock cycle
B	–	Memory bandwidth
B_s	–	Bandwidth of shared memory
B_{L2}	–	Bandwidth of L2 cache memory
B_g	–	Bandwidth of global memory
c	–	Number of catchments
C	–	Number of processor cores
CB	–	Delay correction for propagation of groundwater flows
CI	–	Delay correction for propagation of subsurface flows
CS	–	Delay correction for propagation of surface flows
Δt	–	Minimum time step
Δx	–	Length of river stretch
f	–	Fraction of operations for sequential execution
F	–	Processor performance
F_c	–	Clock frequency of processor core
F_g	–	Clock frequency of memory
g	–	Acceleration of gravity
h	–	water height
I	–	Arithmetic Intensity
K_{grn}	–	Maximum percolation on saturated soil for groundwater flows (mm/day)
K_{sub}	–	Maximum percolation on saturated soil for subsurface flows (mm/day)
m	–	Constant factor from input data
n	–	Manning coefficient
P	–	Performance
ψ	–	Speedup from Amdahl's law
q	–	Discharge
Q_{obs}	–	Observed discharge
$\overline{Q}_{obs}$	–	Average observed discharge
Q_{sim}	–	Simulated discharge
r	–	Additional samples for simplex operation
r_g	–	Data rate of memory
s	–	Number of samples or parameter sets
s_c	–	Centroid of selected samples for simplex operation
s_{con}	–	Sample from contraction operation
s_{ref}	–	Sample from reflection operation
s_w	–	Worst-ranked sample
V	–	Water volume in table
w_g	–	Bus width of memory

W – Water volume in catchment
Wm_d – Maximum water storage capacity for forest on deep soil (mm)
Wm_s – Maximum water storage capacity for forest on shallow soil (mm)
y – Water level
z – Elevation of river bottom

CONTENTS

Page

1 INTRODUCTION

High Performance Computing (HPC) has become an essential part in the development of scientific knowledge. Most of the recent advances and discoveries in several fields of science such as climate modeling, protein folding, drug discovery, energy research, data analysis, artificial intelligence, and universe evolution (PACHECO, 2011) were possible only through the use of the ever-growing computational resources available in modern computer systems.

Nowadays, a large number of commercial and research centers employ computers comprised of different types of technologies and functionalities specifically developed for parallel processing, namely, multi-core processor or Central Processing Unit (CPU) with vector extensions, many-core Graphics Processing Unit (GPU), Field Programmable Gate Array (FPGA), and many other hardware architectures. The fastest and most powerful existing HPC systems (MEUER et al., 2021) are constantly used worldwide to tackle problems in the aforementioned fields due to the complex mathematical models that are associated with those problems, which highly depend on solutions that demand significant hardware resources. In general, high-performance computer systems allow researchers to address important scientific questions that pose big computational challenges.

Hardware architectures with parallel resources are ubiquitous in the scientific community. Modern CPUs commonly consist of multiple independent cores (some may feature dozens) (RALSTON, 2008) with additional support to vector extensions, which also increases the level of parallelism. The current computing landscape also includes general-purpose GPUs, which are accelerators that implement the SIMT (Single Instruction Multiple Threads) architecture with a large number of parallel threads, either hundreds or thousands, for massively parallel workloads.

Both CPUs and GPUs are designed considering a memory hierarchy, providing different levels of memory that differ in speed and size. Moreover, there are heterogeneous hardware architectures where the CPU and the GPU are interconnected, sharing and processing data, and working as one computing device. Therefore, the particular features of each hardware architecture can be exploited for performance improvements in computationally intensive applications (HENNESSY; PATTERSON, 2007).

The parallel hardware resources of modern CPUs can be accessed and used via a programming interface, or API (Application Programming Interface), available for

the development of parallel software on shared and distributed memory systems. In particular, for shared-memory systems the Open Multi-Processing (OpenMP) standard is able to produce scalable and portable parallel codes (DAGUM; MENON, 1998), whereas for distributed-memory systems the Message Passing Interface (MPI) is widely accepted in the parallel community (GROPP et al., 2014). Both interfaces are constantly employed together in a large number of applications, defining a hybrid approach that combines MPI and OpenMP. Moreover, high-performance applications can also be efficiently implemented with the Compute Unified Device Architecture (CUDA) developed for NVIDIA GPUs (NVIDIA, 2012).

Furthermore, building efficient and portable applications requires additional performance analysis tools to guide developers on how to fully exploit the hardware potentials. The Cache-Aware Roofline Model (CARM) (MARQUES et al., 2020; LOPES, 2016; ILIC et al., 2013) is a bound and bottleneck-based tool that identifies the factors limiting the performance (flops/s) of applications, visually representing the compute and memory behavior of an application/system pair, thus providing guidelines for optimizations according to the application's arithmetic intensity (flops/byte).

Given a specific computer/system, the CARM is based on defining two Cartesian axes, the abscissa axis with the application's arithmetic intensity and the ordinate axis with the computing performance. Each application is characterized by its arithmetic intensity and computing performance, where either can change according to the degree of the optimizations. The system's computing performance for any arithmetic intensity value is bound by either the memory performance (a slanted line given by the memory bandwith) or the computing performance (a horizontal line). Therefore, a 2D region is defined under these lines that resemble a roofline and each vertical line, which corresponds to a particular arithmetic intensity, is limited by the roofline to the attainable computing performance of the application.

Applications that involve modeling the complex dynamics of hydrological processes require significant computational resources and are extremely important in environmental issues, being the subject of several recent researches that focus on how natural resources change in our planet. Large-scale hydrological models are extensively used for the understanding of watershed processes, which are closely connected to applications in water resources, climate change, land use, and forecast systems (PAIVA et al., 2011).

Those models simulate complex nonlinear hydrological processes and usually contain a set of nonphysical parameters, i.e. not measurable from actual data

measured/acquired in field, which are embedded in the model to function as conceptual representations that express watershed characteristics, thus allowing the model simulated responses to satisfactorily adjust to the real-world observed data and simulate the behavior of the watershed (EFSTRATIADIS; KOUTSOYIANNIS, 2010), for instance, the watershed discharge. The calibration process is a time-consuming task that requires repeated model simulations, and consists in estimating the optimal sets of model parameters that can simulate the watershed behavior, given actual initial and boundary conditions.

The estimation of the optimal sets of model parameters is solved as an inverse problem, which is formulated as an optimization problem iteratively solved to minimize specific objective functions. In this thesis, a stochastic algorithm generates candidate solutions that are successively evaluated by means of a multi-objective function, which measures, according to a norm, the difference between simulated and measured data. Therefore, once the sets of model parameters for a given watershed are estimated, the parameters can be generically used in subsequent simulations, providing hydrological responses of higher quality and accuracy.

The present work targets performance optimizations on the MGB hydrological model (COLLISCHONN, 2001), used for simulations, and the MOCOM-UA multi-objective calibration method, based on a double-layer approach to execute the MGB model at each iteration. Two watersheds were chosen for the simulation test cases, corresponding to the Purus (Brazil) and the Niger (Africa) rivers, but only the former watershed was employed for calibration.

The MGB model implements a numerical method, referred to as inertial model, that solves the Saint-Venant equations to estimate water height and discharge (stream flow) along the longitudinal extension of the river in the watershed (FAN et al., 2014), which is useful to numerous hydrological applications (FAGUNDES et al., 2020; FLEISCHMANN et al., 2019; GORGOGLIONE et al., 2019; FLEISCHMANN et al., 2017; PAIVA et al., 2011). In the MGB model, the watershed parameters are used to set the initial and boundary conditions for simulations, and also to compute hydrometeorological data such as radiation, evapotranspiration, and water balance in soil. Therefore, those parameters are not directly used to compute the water flows with the numerical method, but instead to update the variables used as input to the inertial model.

The computational time requirements of the calibration phase were reduced by the optimizations proposed herein that exploit the ever-growing parallel resources of computer systems. The MGB model was thoroughly optimized for CPU on

shared-memory systems with vectorization and multi-core parallelism, as well as for GPU with many-core parallelism implemented as a separate version, whereas the MOCOM-UA calibration method was optimized for CPU with OpenMP.

Fully exploiting the hardware resources not only decrease the overall execution times of the MGB model and improve the performance of the calibration procedure, but also provide more understanding about the MGB model's computational requirements. Moreover, the main hydrological results of the optimized implementations, both simulation and calibration, are compared with the original results for a quantitative analysis of the numerical solution of the optimized version.

The roofline characterization of the MGB model identifies how the memory and compute hardware resources of the employed architectures are more effectively exploited with the proposed optimizations, and allows to evaluate the achieved levels of optimization, revealing that careful and effective use of the capabilities of each system may achieve performance improvements. The mixing of explicit vectorization and directive-based parallelization, proposed as a set of advanced optimizations by exploiting most of the capacity of vector resources and shared-memory parallelism, offers more opportunities for improving the performance of hydrological models with similar characteristics.

A scalability analysis using a miniapp, a reliable proxy of the MGB model developed to process larger problem sizes, shows the performance gains achievable if large datasets of either high-resolution data or more extensive watershed regions are available. Miniapps are considered as testbeds that should accurately model the performance bottleneck of the full application (STONE et al., 2012), and also be able to scale the problem size while matching the performance. The prediction of how the full application is affected by prototype changes, and the evaluation of new algorithms, data structures, and programming models are major issues investigated using miniapps in different real-world applications (MURAI et al., 2017; LIN et al., 2015).

1.1 Motivation

As previously mentioned, many scientific applications are complex and demand ever-growing computational resources, which can be supplied with the design of highly parallel computer systems. Such systems are able to handle bigger amounts of data in shorter execution times, thus providing more processing power to solve larger instances of problems.

Exploiting the features available in the current parallel hardware architectures of CPUs and GPUs helps scientists and researchers to produce more reliable results from more precise and detailed datasets. In addition, getting most of the performance out of these architectures depends on how the application makes use of the hardware resources, so improvements on performance analysis techniques covering each architecture are essential to increase the knowledge about the processing times, speedups, scalability behavior, and upper-bound performance limits of the applications on such architectures.

In particular, the optimization of the MGB model has a major impact in the hydrology community. Research centers such as IPH-UFRGS and CPTEC-INPE in Brazil are constantly employing the MGB model in several environmental applications, producing important scientific contributions, significant advances towards mitigating climate change, and also positive effects on increasing knowledge in many fields related to hydrological modeling.

1.2 Objectives

This thesis aims at performance optimization and analysis of a hydrological application comprised of the MGB hydrological model, employed in simulations and in calibration with parallel optimization strategies using either CPU or CPU+GPU architectures. The performance analysis uses basic measures such as runtimes and speedups, and also the CPU and GPU roofline characterizations, which are useful to evaluate and compare the achieved levels of optimization for each architecture. In this thesis, the following optimization strategies were investigated for the implementations:

- CPU vectorization using Intel AVX-512 vector instructions (SIMD paradigm) with Intel Intrinsics;
- Multi-core CPU thread parallelism with OpenMP;
- Many-core GPU data parallelism with CUDA (SIMT paradigm).

1.3 Contributions

The contributions of this thesis are:

- Explicitly vectorized and directive-based, highly optimized MGB model for multi-core CPUs with vector extensions;

- Optimized data-parallel version for many-core GPU accelerators;
- Optimized implementation of the MOCOM-UA calibration method for multi-core CPUs;
- Additional CPU and GPU miniapps implemented as reliable proxies of the MGB model, extending executions to larger problems for detailed scalability analysis;
- CPU and GPU roofline model characterizations as a mean to verify the evolution of the MGB model's performance improvements.

To the author's knowledge, the optimizations proposed herein are the first that employ explicit vector instructions with Intel Intrinsics for hydrological models. Also, this is the first time that the roofline model has been employed to analyze performance improvements in a hydrological model.

1.4 Thesis organization

This document is organized as follows. Chapter 2 contains a detailed description of the current technology of parallel hardware architectures available on HPC systems and the programming interfaces used to exploit those hardware resources. This chapter also illustrates the use of parallelism in other hydrological models. Chapter 3 presents the MGB hydrological model (simulation), including equations, model structure, and code analysis, as well as the MOCOM-UA method (calibration). Chapter 4 describes the optimizations performed on the MGB model and the MOCOM-UA calibration method. In Chapter 5, the computer systems and datasets employed are described, the performance improvements obtained with the optimizations are discussed, and the scalability analysis of the MGB model is evaluated. Chapter 6 provides CPU and GPU roofline analysis of the MGB model and quantitative accuracy analysis of the hydrological results. Chapter 7 presents the conclusion together with suggestions for future work.

2 BACKGROUND ON HIGH PERFORMANCE COMPUTING AND HYDROLOGICAL APPLICATIONS

This chapter describes the hardware and software technologies used for optimizations in this thesis. The hardware characteristics of multi-core CPUs and many-core GPUs are provided, also including fundamental knowledge of software programming interfaces such as Intrinsics (vectorization), OpenMP (multi-core parallelism), and CUDA (many-core parallelism). The roofline model is introduced, and related works that explored parallelization of hydrological applications are summarized.

2.1 Multi-core CPUs

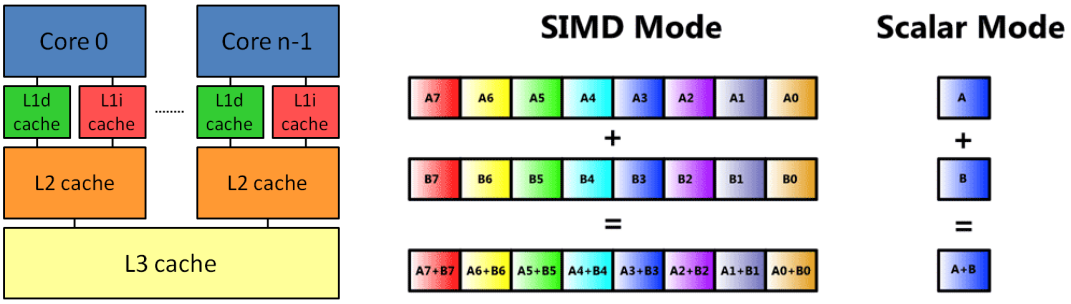
The development and availability of parallel hardware resources has become a standard since HPC systems started to provide high-performance solutions to big computational problems. Historically, the need for performance forced CPU designers to increase the clock speed of the processors. However, there is a limit to the rate of instructions that can be executed inside a CPU due to power consumption and heat emission, which is hard to dissipate. To solve this problem, processors were designed with separate compute units referred to as cores, so performance gains could be achieved with the processing power of multiple independent cores.

Besides the design of multiple cores in a single processor, the set of memory units or caches also changed in order to improve the access from each core to the data stored in memory. Cores have access to a hierarchical memory set comprised of faster but smaller private caches, and slower but larger shared caches. Most CPU architectures include two levels of private memory (L1 and L2 caches), and one level of shared memory (L3 cache). Usually, the L1 cache is divided into L1 data cache (L1d) and L1 instruction cache (L1i). Figure 2.2(a) exhibits the scheme of a processor or CPU with n cores numbered from 0 to $n-1$ that access private and shared caches.

2.1.1 Vectorization

Most modern CPUs support vector extensions that provide data-parallel strategies such as Single Instruction Multiple Data (SIMD), as shown in Figure 2.2(b). Vector processing requires data to be contiguously stored into CPU registers of fixed size (in bits), so that vector instructions can be concurrently executed on all registers. Particularly, current Intel CPUs support different types of vector extensions, namely, SSE (128 bits), AVX/AVX2 (256 bits), and AVX-512 (512 bits), which are accessible via assembly instructions or via the Intel Intrinsics library (INTEL, 2019).

Figure 2.1 - Hardware features of multi-core CPUs.



(a) CPU scheme. (b) SIMD operation.

SOURCE: (a) Author, (b) Intel (2019).

Code 2.1 is an example in the C programming language that illustrates an explicit SIMD code for the array sum of Figure 2.2(b), computed with vector instructions of the Intel Intrinsics library, which is available in C/C++. In this case, AVX-512 vector instructions are used for processing double-precision arrays (64 bits), so the loop iterates the arrays with stride equal to 8, i.e. 8 elements are loaded from memory, simultaneously added, and then stored into memory.

Code 2.1 - Stride-8 loop for array sum with Intel Intrinsics in C.

```
1  #include <immintrin.h>
2
3  void array_sum(double *A, double *B, double *C, int *N) {
4      __m512d vecA, vecB, vecC;
5      int i;
6
7      for(i = 0;i < *N;i+=8) {
8          vecA = _mm512_load_pd(A+i);
9          vecB = _mm512_load_pd(B+i);
10         vecC = _mm512_add_pd(vecA, vecB);
11         _mm512_store_pd(C+i, vecC);
12     }
13 }
```

For more complex applications, hand-tuned vectorized codes execute faster than the auto-vectorized equivalents because compilers are not able to fully identify all the instructions that can be executed in parallel. However, explicit vectorized codes employ low-level instructions that requires careful and detailed code analysis, and more development time. Mitra et al. (2013) consistently observed speedups of up to

5.32× with SSE vector instructions (128 bits) using Intel Intrinsics for single-precision floating-point data in several applications from different benchmarks.

2.1.2 Multi-core CPU parallelism

In addition to vector instructions, the wide availability of multiple cores in modern CPUs enables thread-parallel strategies to be further explored in shared-memory environments. In this type of optimization, the workload is processed in parallel after being distributed among threads, which are associated to the multiple CPU cores.

Thread-level parallelism on shared-memory environments is usually designed with the OpenMP standard (DAGUM; MENON, 1998) available in C/C++ and Fortran. OpenMP provides compiler directives and a library of functions that distribute independent workloads to the processor cores. The main types of parallel processing with OpenMP are data and task parallelism, where the former is for arrays and matrices that are divided up in parts among the cores, whereas the latter allows functions to be simultaneously called by different cores.

OpenMP threads can view variables as either private or shared, depending on how data must be accessed. Specific directives are used for thread synchronization to ensure correctness of results, avoiding data race conditions, and different schemes of workload distribution can be specified with particular clauses for more efficient load balancing to improve the use of the available CPU hardware resources. OpenMP executes a **fork/join** type of parallelism that dynamically activate/deactivate parallel threads during the execution of a program. The master thread spawns multiple threads as requested (`fork`), and when the parallel work finishes all threads are terminated (`join`). Code 2.2 illustrates the array sum optimized with OpenMP in Fortran.

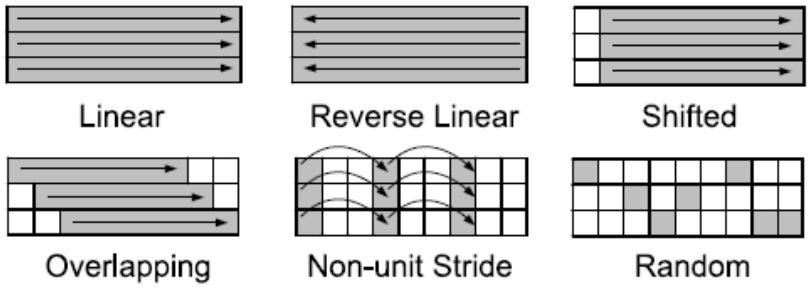
Code 2.2 - Shared-memory parallel array sum with OpenMP in Fortran.

```
1  subroutine array_sum(A, B, C, N)
2      use omp_lib
3      integer i, N, A(N), B(N), C(N)
4
5      !$omp parallel do private(i) shared(A,B,C,N) num_threads(8)
6      do i=1,N
7          C(i) = A(i)+B(i)
8      end do
9      !$omp end parallel do
10 end subroutine
```

Optimizations that exploit CPU parallel resources such as vector instructions and multiple cores generally result in improvements on the performance of applications. However, performance gains depend on the complexity of the application, particularly on the pattern of memory accesses. For example, accessing data that is not contiguously stored in memory usually decreases performance because the frequency of accessing data located in lower levels of memory, i.e. L3 cache and DRAM, is higher. Programmers must write codes to more efficiently store data in memory in order to improve as much as possible the memory access patterns.

Figure 2.2 shows different memory access patterns that commonly occur in applications (JANG et al., 2011). Each type of memory access incurs more or less latency for either reading or writing data. Accessing contiguous data in memory without skipping memory positions, and following the standard direction of data storage (**Linear**) is the optimal memory access pattern, whereas randomly accessing memory positions without a clear access pattern (**Random**) decreases performance, for not benefiting from the sequential data storage in memory.

Figure 2.2 - Different memory access patterns commonly found in applications.



SOURCE: Jang et al. (2011).

The overall performance of applications increases with the use of parallel hardware resources not only because workloads can be divided and distributed among multiple parallel compute units, but also because executing more computations in parallel effectively hides the latency of accesses to memory. As hardware architectures consist of separate functional units for memory and compute instructions, parallel programs are able to execute larger amounts of computation while memory is accessed for reading/writing data. Therefore, exploiting the available parallel resources of hardware architectures enhances the memory efficiency of applications.

In a parallel program, increasing the number of compute units decreases the execution time. However, communication time increases and, at some point, it becomes higher

than the computation time, so that the execution time does not decrease. The use of more compute units usually speeds up the execution of the program, but only if the amount of data can be divided with enough computations to hide the latency of memory accesses and of communication. These situations lead to the concepts of *granularity* and *scalability*, i.e. the ratio between computation time and communication/synchronization time (QUINN, 2004), and the performance behavior with increasing number of compute units (CHAPMAN et al., 2008), respectively.

For shared-memory programs parallelized with OpenMP, each thread must be allocated a particular set of resources, including which memory regions can be accessed. Whenever more than one thread accesses the same memory location, mainly for executing write instructions of shared variables, synchronization directives are necessary to ensure that only one thread executes the instruction at a time, avoiding a data race condition in order to guarantee the correctness of the parallel program. Consequently, synchronization reduces granularity and causes performance degradation, and this impact is more pronounced if the number of threads increases.

However, even though increasing the number of threads in a parallel program also reduces granularity, performance still improves if all threads are assigned enough data for computations, hiding synchronization effects and the latency of memory accesses. As an example, applications with larger proportions of compute instructions per bytes accessed from memory, and also larger percentages of code that can be parallelized, are likely to scale more efficiently, thus resulting in higher performance and parallel speedup (HILL; MARTY, 2008).

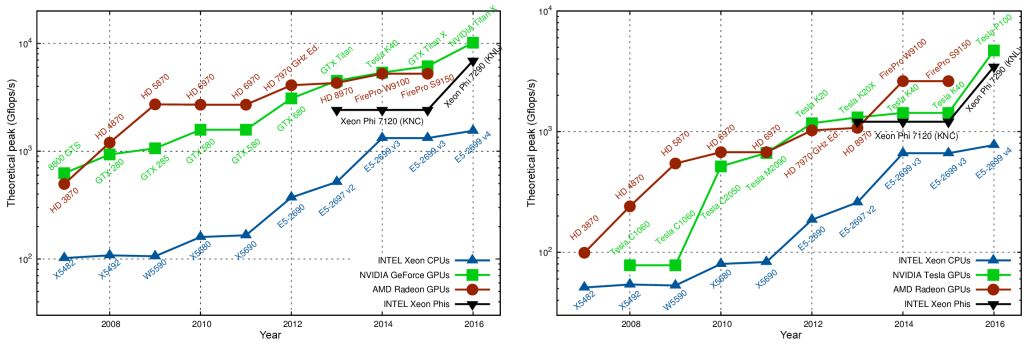
At this point, it is important to add that the performance of a parallel program can be theoretically predicted with measures that indicate how much performance improvement is possible to be achieved by increasing the number of threads. The parallel speedup is the ratio between the sequential and parallel execution times, i.e. how many times the parallel execution is faster than the sequential execution. The Amdahl's law (QUINN, 2004) provides an upper bound on the parallel speedup achievable with p processors, depending on the fraction f of operations that must be sequentially executed, where $0 \leq f \leq 1$. The maximum speedup ψ is given by equation (2.1), which does not consider the time lost with overheads of communication and synchronization.

$$\psi \leq \frac{1}{f+(1-f)/p} \tag{2.1}$$

2.2 Many-core GPUs

In addition to the use of multi-core CPUs, the current computing landscape also includes GPUs as general-purpose processors for data-parallel codes. GPU architectures are classified as many-core for consisting of a massive amount of arithmetic compute units (either hundreds or thousands), thus providing more processing power for numerical intensive integer and floating-point workloads. GPU performance is much higher when compared to CPU, as shown in Figure 2.3.

Figure 2.3 - Theoretical peak performance between different CPUs and GPUs for data in single-precision (left) and double-precision (right).



(a) Performance in single-precision. (b) Performance in double-precision.

SOURCE: Adapted from Rupp (2016).

This difference in performance motivates software developers to exploit the potential of GPUs to reduce the execution time of applications. The porting of codes to GPUs is usually done with frameworks such as CUDA (NVIDIA, 2019a), OpenCL (TOMPSON; SCHLACHTER, 2012), and OpenACC (WIENKE et al., 2012), each with its advantages and disadvantages. For example, OpenACC is based on compiler directives similar to OpenMP, which provides codes that are easily portable to different GPU architectures. However, in most cases, the highest performance is achieved with the CUDA framework, although it requires more effort and time of development.

Besides the advantages in the optimization of numerical applications, GPUs are not suited to tasks that involve either too much transfer of data between host (CPU) and device (GPU) memories or communication with the operating system and control of I/O devices. These latter tasks should be assigned to CPUs, which have control logic units with more complex instructions (KIRK; HWU, 2017).

GPUs are designed with a set of high bandwidth memories that increases the performance even more. Similarly to CPU architectures, GPUs also consist of hierarchical memory sets, although there are more specific types of memory rather than the traditional scheme of DRAM and caches of CPUs. For instance, each NVIDIA GPU architecture has its own characteristics, but usually the L1 cache or *local* memory is the private memory of each thread, whereas the *shared* memory is the private memory of each block shared by all threads in the block. The L2 cache is shared by all blocks as the Last Level Cache (LLC). *Texture* and *constant* memories are read-only memories, the former for graphic purposes, and the latter for storing constants and parameters (LOPES, 2016). The device or *global* memory is the larger but slower memory accessible by all threads.

The compute performance and memory bandwidth related to the GPU hardware are essential to the GPU roofline analysis used in this work. The peak performance of GPUs is computed from the total number of cores C and the clock frequency F_c set to the GPU cores, where the actual value is multiplied by two because each core has Fused Multiply-Add (FMA) instructions that execute two floating-point operations (multiplication and addition) in one cycle. Thus, the maximum performance P achievable by a GPU, in flops/s, is given by equation (2.2).

$$P = 2 \times F_c \times C \tag{2.2}$$

$$B_g = r_g \times \frac{w_g}{8} \times F_g \tag{2.3}$$

$$B_s = \frac{b_s}{8} \times F_c \tag{2.4}$$

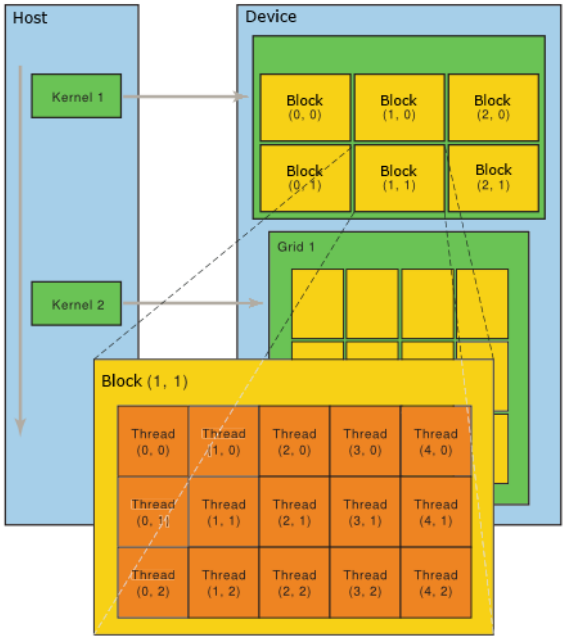
Analogously, the peak bandwidth B_g of the *global* memory, in bytes/s, is computed from the memory's data rate r_g (transfers per cycle), bus width w_g (in bits), and clock frequency F_g, as in equation (2.3). In addition, the peak bandwidth B_s of the *shared* memory, given by equation (2.4), is computed from the total number of bits transferred per clock cycle b_s and the clock frequency of the cores F_c. However, the bandwidth B_{L2} of the L2 cache is not simple to be quantified because its value depends on how the GPU architecture works, requiring benchmarks specifically developed to exploit all in-depth details of the architecture (LOPES, 2016).

2.2.1 Many-core GPU parallelism

The many-core GPU parallelism employed in this thesis uses the CUDA framework. CUDA offers an interface for sharing data between host and device memories, as well as internal structures for identifying each parallel thread from the organization scheme

defined for threads inside the GPU. In particular, NVIDIA GPUs are comprised of multiple Streaming Multiprocessors (SMs), and each SM contains a specified number of CUDA cores or floating-point units. All threads are arranged in sets of 32 threads called warps in multi-dimensional blocks inside the SMs, which are grouped in a multi-dimensional grid (NVIDIA, 2012). This scheme is illustrated in Figure 2.4.

Figure 2.4 - Organization of blocks and threads as viewed by CUDA inside NVIDIA GPUs.



SOURCE: NVIDIA (2019b).

CUDA was originally supported by programming languages such as C/C++ in 2007, and extensions were created for Fortran in 2009. The functions called from the host to be executed on the device are referred to as CUDA kernels. Each thread in a kernel is initially identified from the block index in the grid, and then from the thread index inside the block. After threads are assigned unique identification numbers (thread IDs), each thread can process independent workloads accessing data from memory indexed by the thread ID.

Transfer of data between host and device memories for reading and writing data is executed either before or after calls to kernels. In addition, calls to CUDA kernels use a syntax that includes both the number of blocks and threads per block to be allocated in the device, where the number of blocks is usually computed as the ratio between the problem size and the number of threads per block.

Code 2.3 exhibits the CUDA kernel for the parallel array sum. Firstly, the thread ID is stored into variable **i** used to check all valid array positions, which are accessed by one single thread and processed in parallel. The kernel is called from the host as `call array_sum<<<cudaBlocks,cudaThreadsPerBlock>>>()`, where **cudaBlocks** and **cudaThreadsPerBlock** specify blocks and threads per block, respectively.

Code 2.3 - Parallel array sum in CUDA Fortran.

```
1 subroutine array_sum(A, B, C, N)
2   i = blockDim%x*(blockIdx%x-1)+threadIdx%x
3   if(i <= N)then
4     C(i) = A(i)+B(i)
5   end if
6 end subroutine
```

2.3 Roofline model

The performance analysis of applications usually considers measures from two basic hardware components of computer systems: processor and memory. Applications that require intensive use of the compute units of processors are limited by the compute performance of the hardware architecture, whereas others that demand more memory accesses are bounded by the bandwidth of the available memory hierarchy. Therefore, there is a close relationship between the potential compute and memory capabilities available in computer systems and the behavior of the applications executed on such systems.

The roofline model (WILLIAMS et al., 2009; WILLIAMS, 2008) is a valuable tool that provides insights about the behavior of applications on computer systems with diverse features and capabilities of processor and memory. The development and optimization of applications can be guided by the model for better utilization of the hardware resources, aiming at performance improvements.

More specifically, the roofline model uses the arithmetic intensity I, in flops/byte, of an algorithm to identify whether its performance on a particular system is bounded by either compute or memory limits. On a computer system with peak floating-point processor performance F (flops/s) and peak memory bandwidth B (bytes/s), the maximum attainable performance P for I is $P = \min\{B \times I, F\}$.

However, the arithmetic intensity used in the Original Roofline Model (ORM) is computed only from off-chip memory transfers between the Last Level Cache

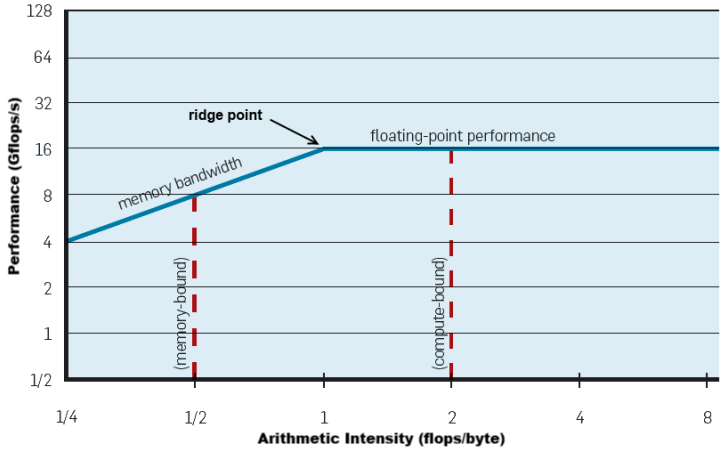
(LLC) and the DRAM, which is not sufficient to fully describe the performance of applications on modern hardware architectures. A more precise roofline model, referred to as Cache-Aware Roofline Model (CARM) (MARQUES et al., 2020; ILIC et al., 2013), also accounts for the on-chip memory traffic with data transfers between all cache levels, providing a more accurate behavior of the applications on computer systems designed with current hardware technology.

As previously described, the roofline model consists of a 2D plot, with axes typically on logarithmic scale, that relates arithmetic intensity and performance, which are placed on the x and y axes, respectively. Compute and memory roofs indicate the fixed upper-bound limits of performance that can be reached on the underlying hardware depending on the arithmetic intensity of the application. Standard compute roofs are for scalar, vectorized, and FMA computations, each in single-precision and double-precision. Memory roofs commonly specify the performance achieved with the bandwidth of each type of memory from the memory hierarchy of the hardware architecture, including caches and DRAM. These latter roofs are more precisely found through the use of micro-benchmarks that exploit each level of memory with particular low-level memory instructions, which decrease the latency of memory measures.

Figure 2.5 exhibits a simplified roofline model with compute and memory roofs that are the upper-bound limits of performance. As an example, two applications of different arithmetic intensity are illustrated (red dashed lines), one compute-bound, and other memory-bound. The point where the roofs meet is known as the ridge point, i.e. the minimum arithmetic intensity required for attaining maximum performance. The ridge point of computer systems shifts from right to left with increasing memory bandwidths. The closer the ridge point is to the y-axis, the larger is the number of applications that can reach higher performance.

Besides the performance analysis of applications with the roofline model from CPU hardware architectures, some works recently explored the roofline model from GPU devices, particularly from NVIDIA GPUs (YANG et al., 2019; LOPES, 2016; JIA et al., 2012; KIM et al., 2011). CPU and GPU roofline models are similar models because both include compute and memory roofs, although the roofs for GPUs are usually higher, as GPUs provide more performance than CPUs due to the massive amount of compute units and the high bandwidth memories, when compared to the current performance achieved by the compute units and the memory hierarchy associated to multi-core CPUs.

Figure 2.5 - Roofline model with compute and memory roofs as the upper-bound limits of performance for applications with different arithmetic intensity.



SOURCE: Adapted from Williams et al. (2009).

Previous works that use the GPU ORM for performance analysis (YANG et al., 2019; JIA et al., 2012; KIM et al., 2011) do not fully characterize the behavior of applications on GPUs, as the GPU ORM considers data transfers only between LLC and global memory. The GPU CARM approach (LOPES, 2016) fully explores the GPU hardware architecture, thus providing a more in-depth characterization of applications.

This thesis employs the GPU CARM for the roofline model characterization from NVIDIA GPUs. In this case, flops and bytes are collected with the CUDA component of the Performance API (PAPI) (DONGARRA et al., 2001), a widely used interface that can access hardware performance counters from a large number of CPU and GPU architectures. More specifically, the performance counters selected to collect flops include double-precision, single-precision, and special (mathematical functions) floating-point operations, whereas the performance counters selected to compute transferred bytes include load and store transactions per memory request, which differ in number depending on the GPU architecture. Therefore, the arithmetic intensity and performance for the roofline characterization are computed from these particular GPU hardware performance counters.

2.4 Parallelization of hydrological applications

Several factors affect the efficiency of large-scale hydrological models, which usually demand large amounts of computations for more accurate simulations of water flows. The main factors include the continuous advances in GIS software and technologies, the increase in the availability of high-resolution spatial and temporal

hydrometeorological data, such as in-situ measurements and remote sensing data, the great extent of the regions processed in the simulations, and the number of model parameters. In calibration, manual trial-and-error selection of the optimal sets of watershed parameters is still a predominant procedure despite being a time-consuming task, even for experienced model users. Computer-based automatic methods provide an alternative solution, and have recently been considered as an effective and efficient option for calibration using hydrological models (VRUGT et al., 2003).

Different hydrological models were modified with either CPU or GPU optimizations aiming at performance improvements. A simplified version of the LISFLOOD-FP model (BATES et al., 2010), named LISMIN model, reached theoretical speedups with OpenMP and MPI (NEAL et al., 2010), but without processing 1D channel flow and dry checking functions, and more recently was optimized to compute 2D water flows and depths on GPU using the CUDA framework (SARATES, 2015).

OpenMP was also applied on a grid-based Fully Sequential Dependent Hydrological Model (FSDHM) to process simulation units based on a layered approach, computing overland and channel flows in parallel (LIU et al., 2014). MPI was used in the SWAT model (ARNOLD et al., 1998) with a scheme devised to decrease execution times by reducing the overheads of communication (WU et al., 2013), and in the tRIBS model (VIVONI et al., 2011) to decompose watersheds, formed by triangulated irregular networks (TINs), as directed graphs from drainage network channels, where independent sub-basins were assigned to MPI processes exchanging data across boundaries.

The performance of calibration methods in hydrological applications is a key issue that has been continuously addressed by employing different optimization and parallel computing techniques, mainly designed and applied to large-scale hydrological models with the purpose of reducing the runtimes for calibrating watershed parameters. The computational efficiency of the single-objective SCE-UA method (DUAN et al., 1993), processing up to millions of objective function evaluations, was improved by using parallel implementations of the Xinanjiang model (ZHAO, 1992) designed for CPU (OpenMP) and for GPU (CUDA) (KAN et al., 2018). Furthermore, Zhang et al. (2016) proposed a double-layer parallel approach for a genetic algorithm-based calibration method, using the DYRIM model (WANG et al., 2007), with parallelism in the spatial decomposition of the watershed, divided into independent units (lower-layer), and also in simultaneous model simulations with different combinations of parameters (upper-layer).

A large number of calibration methods using the SWAT model were also optimized. MPI and Python were employed to parallelize the multi-objective AMALGAM calibration method with concurrent evaluations of simulations (ZHANG et al., 2013). The Python multiprocessing module was used to develop a parallel multi-objective calibration tool (MAMEO) for multi-core processors (ZHANG et al., 2012). A framework based on the optimization program SUFI2 for the SWAT-CUP calibration procedure automatically and transparently submit parallel jobs on multi-core CPUs (ROUHOLAHNEJAD et al., 2012).

Although those other hydrological models employed traditional parallelization techniques, no effort has been previously employed on vectorization. Similarly, most of those models do not contain some of the advanced modeling capabilities available in the MGB model, which is described in the next chapter. Thus, exploring advanced vectorization/parallelization techniques for the MGB model that can benefit from the hardware features of modern processors is highly desirable, and was the main focus of this thesis.

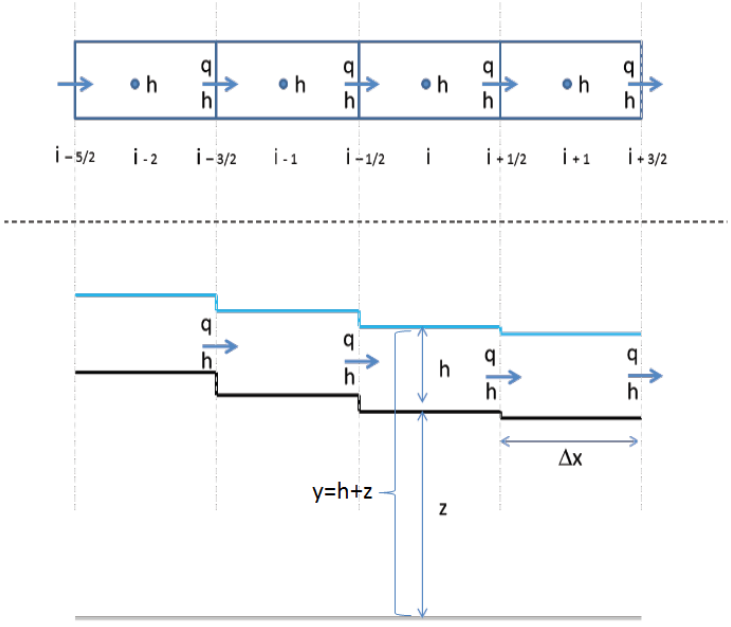
3 MGB HYDROLOGICAL MODEL AND MOCOM-UA CALIBRATION METHOD

This chapter introduces the fundamental background for the hydrological application considered in this thesis. The MGB hydrological model is described with details of equations, units of spatial discretization, scheme of numerical solution, and code structure. The MOCOM-UA calibration method is also presented, including main features and steps executed by the method to optimize, according to the objective functions, the sets of parameters used in the MGB model simulations. The implementations consist of 52 Fortran source files, with a total of 3 k lines of code.

3.1 MGB hydrological model

The MGB ("Modelo de Grandes Bacias") hydrological model (COLLISCHONN, 2001) is developed at IPH-UFRGS ("Instituto de Pesquisas Hidráulicas"-"Universidade Federal do Rio Grande do Sul") in Brazil, focusing on improving the knowledge of hydrological processes in large-scale watersheds, mainly in the South America region. Simulations of the MGB model generate 1D propagation of water flows in rivers as illustrated in Figure 3.1, which exhibits the elevation z, water height h, water level y, discharge q, and length Δx of each segment of the river stretch discretization from the numerical scheme detailed in Subsection 3.1.1.

Figure 3.1 - River stretch discretization.

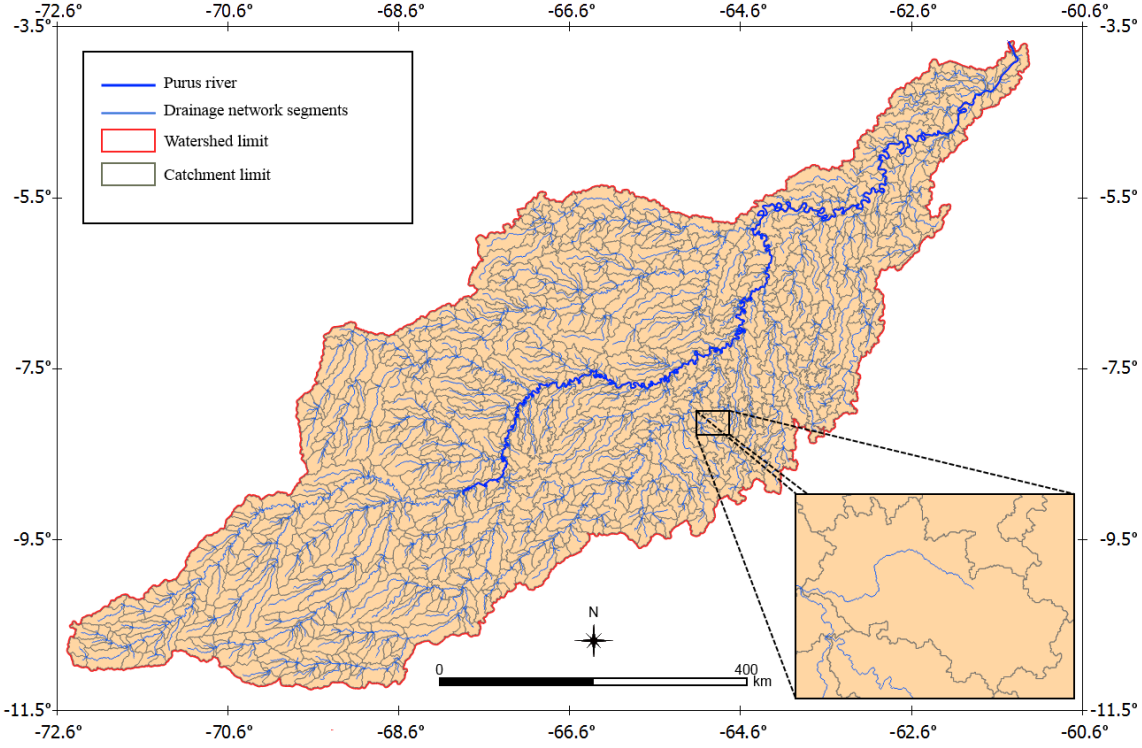


SOURCE: Adapted from Fan et al. (2014).

The model simulates the hydrological cycle computing soil water and energy budgets, evapotranspiration, interception, surface, subsurface, and groundwater flows, in either daily or hourly time steps, for the forecast of river discharge, analysis of extreme events (floods and droughts), estimation of the effects of soil and vegetation cover on climate, water quality and sediment transport (PAIVA et al., 2011). The spatial discretization of the MGB model is comprised of three hydrological units: catchments, sub-basins, and hydrological response units (HRUs). Catchments and sub-basins consist in surface regions that contribute water to drainage network segments and to outlet points, respectively, whereas HRUs are regions that usually combine land use, land cover, soil and slope maps based on user-defined thresholds (KALCIC et al., 2015).

Figure 3.2 illustrates the catchments (smallest spatial units) delineated for a drainage network of the Purus watershed. The drainage network was defined from a threshold of accumulated flows equal to 10 000 m^2 and generated with the TerraHidro computational platform for distributed hydrological modeling (JARDIM, 2017; ROSIM, 2008).

Figure 3.2 - Watershed of Purus river and catchments of each drainage network segment.



SOURCE: Author.

The MGB model has been effectively employed in a diverse and large number of applications, such as sediment modeling for estimation of soil erosion, sediment transport and deposition (FAGUNDES et al., 2020), modeling of reservoirs as an internal boundary condition for simulation of hydrodynamic processes and their interaction with upstream and downstream floodplains (FLEISCHMANN et al., 2019), analysis of water conflicts in transboundary watersheds to improve the sustainability of water allocation (GORGOGLIONE et al., 2019), representation of the interactions between hydrology and hydrodynamics, mainly infiltration from floodplains into soil, to improve model estimates (FLEISCHMANN et al., 2017), and hydrological and hydrodynamic modeling to simulate fluvial processes of wave delay and attenuation, backwater effects, flood inundation and its effects on flood waves (PAIVA et al., 2011).

3.1.1 Water flow equations and numerical scheme

The spatial and temporal distribution of the water flows in a watershed, usually measured by hydrological variables such as water height and discharge, is computed in the MGB model through an implementation of the inertial simplification (FAN et al., 2014) of the Saint-Venant equations (CUNGE et al., 1980), a set of partial differential equations also known as shallow water equations. These equations are referred to as continuity (3.1) and momentum (3.2) equations, where h is water height, q is discharge, $y = h + z$ is water level given by the sum of water height h and elevation z, g is the acceleration of gravity, n is the Manning coefficient, t is time, and x is longitudinal distance.

$$\frac{\partial h}{\partial t} + \frac{\partial q}{\partial x} = 0 \tag{3.1}$$

$$\frac{\partial q}{\partial t} + gh\frac{\partial y}{\partial x} + g\frac{|q|qn^2}{h^{7/3}} = 0 \tag{3.2}$$

Equations (3.1) and (3.2) are numerically solved with forward-in-time and centered-in-space finite difference approximations that integrate through time from initial conditions computed from soil moisture, water volumes, and water flows in reservoirs, and also boundary conditions from accumulated flows (Muskingum-Cunge) and reference discharges. The numerical solution is a particular scheme of wave propagation that is commonly employed in rainfall-runoff hydrological models. The explicit numerical scheme is shown in equations (3.3)-(3.7), where c is the number of catchments that form the watershed, and i and k are spatial and temporal indexes, respectively.

The time steps computed for each catchment i from equation (3.3) require $0<\alpha<1$ in order to avoid numerical instability (BATES et al., 2010), where $\alpha=0.7$ in the MGB model (FAN et al., 2014). In addition, the minimum time step Δt corresponds to the current maximum water height h_i^k among all catchments. The water height $h_{i+1/2}^k$ at position $i+1/2$ (border) is computed from the difference between the maximum values of water level y and elevation z at positions i and $i+1$, as in equation (3.4).

The discharge $q_{i+1/2}^{k+1}$ of the outflow of each catchment, computed for the next time step $k+1$, is found through equation (3.5), and depends on the previous discharge $q_{i+1/2}^k$, water height $h_{i+1/2}^k$, and water levels y_{i+1}^k and y_i^k. Finally, the water height h_i^{k+1} and level y_i^{k+1}, computed for the time step $k+1$ at position i (center), are obtained from the updated discharges $q_{i+1/2}^{k+1}$ and $q_{i-1/2}^{k+1}$, as in equation (3.6), and from the water height h_i^{k+1}, as in equation (3.7), respectively.

$$\Delta t = min(\alpha \frac{\Delta x}{\sqrt{g h_i^k}}), i = 1, 2, \ldots, c \tag{3.3}$$

$$h_{i+1/2}^k = max(y_i^k, y_{i+1}^k) - max(z_i, z_{i+1}) \tag{3.4}$$

$$q_{i+1/2}^{k+1} = \frac{q_{i+1/2}^k - g\Delta t\left(h_{i+1/2}^k\right)\left(y_{i+1}^k - y_i^k\right)/\Delta x}{1+g\Delta t\left|q_{i+1/2}^k\right| n^2 \Big/ \left(h_{i+1/2}^k\right)^{7/3}} \tag{3.5}$$

$$h_i^{k+1} = h_i^k - \frac{\Delta t}{\Delta x}\left(q_{i+1/2}^{k+1} - q_{i-1/2}^{k+1}\right) \tag{3.6}$$

$$y_i^{k+1} = z_i + h_i^{k+1} \tag{3.7}$$

The numerical scheme comprised of equations (3.3)-(3.7) is computed m times for each of the c catchments, where m is a constant factor that depends on the input data, so that the computational complexity of this numerical scheme is $O(m.c)$. Therefore, the amount of computations executed by the MGB model to find the numerical solution at each time step is mostly determined by the number of catchments c, as this value is the only one that can theoretically increase without limits.

3.1.2 MGB code structure

The numerical scheme of the MGB model is referred to as inertial model, which is divided into three routines inside the model, namely, `timestep` (STE), `discharge` (DIS), and `continuity` (CON). These routines are successively called for each minimum time step Δt that satisfies the condition for the inertial model's stability, being accumulated until reaching the time step specified for the MGB model.

The computations of each routine are executed for all iterations of the MGB model, a fixed number of steps defined in the input data. The routines of the inertial model iterate over each catchment computing the minimum time step in the STE routine, from equation (3.3), water height and discharge in the DIS routine, from equations (3.4)-(3.5), and water height and level in the CON routine, from equations (3.6)-(3.7).

Besides solving the Saint-Venant equations, the MGB model also computes hydrometeorogical data such as radiation, evapotranspiration (Penman-Monteith equations) (SHUTTLEWORTH, 1993), and vertical water balance in soil (COLLISCHONN, 2001) for surface, subsurface, and groundwater flows. However, the inertial model is the main part executed in the MGB model, and the time length of each execution varies with the minimum stable time step Δt that is computed from the input data.

Code 3.1 provides a simplified scheme of the MGB model's code structure. This code includes the calls to the routines that are used to load the rainfall and climate data (`load_rainfall_climate_data`), to compute the hydrometeorogical data (`calc_radiation`, `calc_evapotranspiration`, and `calc_water_balance`), and to execute the inertial model for computing water flows with the numerical scheme previously described (`inertial_model`).

The STE routine computes time steps for all catchments from equation (3.3), executing a reduction operation to find the minimum time step **Dt** used in the DIS and CON routines. The DIS routine, which accounts for the largest percentage of execution time, updates the river discharge from equations (3.4)-(3.5) for the CON routine, which uses equations (3.6)-(3.7) to update the water height and level, as well as the water volume and its corresponding cross-sectional area. Next, the inertial model repeats by executing again the STE routine with the updated water height from the CON routine, and this process continues as long as the minimum time step **Dt** accumulates to the time step **MGBDt** of the MGB model.

3.1.3 Profiling of serial execution

Based on serial executions of the original MGB code, compiled with default optimization flags, the three routines of the inertial model, i.e. STE, DIS, and CON, were identified as the most time-consuming routines from reports obtained with the GNU's `gprof` profiling tool. Those routines accounted for the highest percentage of total execution time from either thousands or millions of calls for different simulations, taking on average 92.96 % of execution time, and thus constituting the main execution bottleneck of the MGB model.

Code 3.1 - Structure of the MGB model.

```
1  subroutine inertial_model()
2    ineTotDt = 0.0 ! initialize time step of the inertial model
3    ! repeat inertial model until the time step of the MGB model is reached
4    do while(ineTotDt < MGBDt)
5      call STE()
6      call DIS()
7      call CON()
8      ineTotDt = ineTotDt+Dt ! accumulates time step of the inertial model
9    end do
10 end subroutine
11
12 program MGB_model()
13   read input
14   step = 0 ! initialize iterations of the MGB model
15   do while(step < MGBTimeSteps) ! process iterations of the MGB model
16     call load_rainfall_climate_data()
17     call calc_radiation()
18     call calc_evapotranspiration()
19     call calc_water_balance()
20     call inertial_model() ! call inertial model to solve the numerical scheme
21     step = step+1
22   end do
23   write output
24 end program
```

Table 3.1 provides the profiling details, including the average (mean value) runtime and percentage of execution time of each routine relative to the execution time of the MGB model, and also the number of calls. The values reported were obtained from simulations using different datasets.

Table 3.1 - Profiling information from serial executions of the MGB model.

Name	Runtime (s)	% of time	Calls
STE	3.36	2.71	634K/1.2M
DIS	67.78	54.69	634K/1.2M
CON	44.07	35.56	634K/1.2M
Other	8.73	7.04	-
MGB	123.94	100.00	-

3.2 MOCOM-UA calibration method

Calibration methods can be either local (iterating from an initial solution to the optimal one) or global (generating sets of possible solutions), and typically employ objective functions, computed from model simulations, as measures to be minimized for the optimization of the sets of parameters. Some methods focus on only one objective function, such as the search-based and evolutionary Shuffled Complex Evolution (SCE-UA) method (DUAN et al., 1993), commonly employed using hydrological models. However, single-objective methods do not provide solutions that can truly model the behavior of the observed data in hydrological simulations, particularly in the extreme situations of wet and dry periods (TIAN et al., 2019).

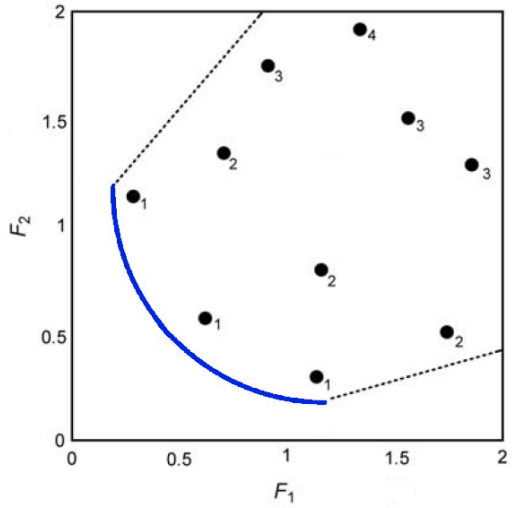
In contrast, multi-objective methods were developed and employed in hydrological modeling to evaluate and capture various properties of the observed data, with the critical advantage of providing hydrological models high-quality parameters that are estimated to produce more consistent and accurate simulated results (VRUGT et al., 2003), even in extreme events of floods and droughts. More specifically, the Multi-Objective Complex Evolution (MOCOM-UA) method (YAPO et al., 1998) has been successfully applied to a variety of environmental applications (FAGUNDES et al., 2019; GERITANA et al., 2014; ISLAM; DÉRY, 2017; TATSUMI, 2016; TESEMMA et al., 2015), being predominantly used for general-purpose global multi-objective optimization. In addition, an epidemic genetic algorithm (EGA) (ARAÚJO et al., 2013) was proposed as a variant of the MOCOM-UA method, to be employed in calibration using the IPH2 hydrological model developed for small basins (TUCCI, 2005).

The MOCOM-UA method formulates the estimation of the sets of model parameters as an inverse problem that is implicitly solved, i.e. successively generating candidate solutions that try to minimize multiple objective functions, which express the difference between simulated and measured data. The calibration procedure is based on determinism to converge to the local/global optimal solutions (simplex search), probability to better explore the search space *escaping* from local minima (random search), evolution to improve the solutions (population evolution), and nonuniqueness of solutions to provide multiple optimal solutions (Pareto front). However, the optimal solutions are never assured to be found, only suboptimal solutions, as it is not possible to ensure that a given solution is optimal, specially in multi-dimensional problems. The MOCOM-UA method is widely adopted as an effective and efficient global multi-objective method, being selected in this thesis for the optimization of the sets of watershed parameters used in the MGB model.

Similarly to other optimization methods, the MOCOM-UA method generates a population with a fixed number of candidate solutions, or independent samples, in a multi-dimensional search space where each dimension represents a particular model parameter. Each solution or sample corresponds to a point in the multi-dimensional space with coordinates given by the parameter values, which are randomly computed from restricted ranges of the domain for the initial iteration. Each combination of parameters is used in simulations, and the objective function values, computed from the observed and simulated data, are compared to rank all samples according to the Pareto ranking. In this ranking, a sample with all objective function values greater than the corresponding ones from another sample is considered inferior, and identified as *dominated*, otherwise it is *nondominated*.

Firstly, all nondominated samples are ranked 1, then the remaining dominated samples are checked, and the new nondominated ones are ranked 2, and so on. In the end, the worst-ranked samples are ranked RMAX (maximum rank). The optimal set of samples (sets of parameters) is located in the region of the objective space called Pareto front, as exhibited in Figure 3.3, which consists of all feasible solutions that are equally acceptable as optimal solutions, although each solution defines a set of parameters with unique objective function values. In the Pareto ranking, lower-ranked samples are considered superior and located closer to the Pareto front.

Figure 3.3 - Objective space for two objective functions F_1 and F_2 with the Pareto front (blue) and ranked samples (numbered dots).



SOURCE: Adapted from Vrugt et al. (2003).

All worst-ranked samples are improved with specific operations, namely, reflection and contraction (simplex search), using other samples randomly selected (random search) in relation to a probability distribution obtained from the ranks of each sample, where samples with smaller (larger) rank values have higher (lower) probability to be selected. The simplex operations applied on each worst-ranked sample s_w require r more samples, where r is the number of parameters, thus forming a simplex of $r+1$ samples. In this process, a new sample is generated from the sample s_w and the centroid s_c of the other r samples, either from the reflection or the contraction operation.

Firstly, the reflection operation is executed, and if the new sample s_{ref} is considered nondominated relative to the r selected samples, it replaces s_w, otherwise the new sample s_{con} obtained from the contraction operation replaces it. The reflection and contraction operations for s_{ref} and s_{con} are given by equations (3.8) and (3.9), respectively. Finally, new samples replace old ones in the population (population evolution), the Pareto ranking is recalculated, and this procedure repeats until all samples have rank 1. The flowchart of the MOCOM-UA calibration method is exhibited in Figure 3.4.

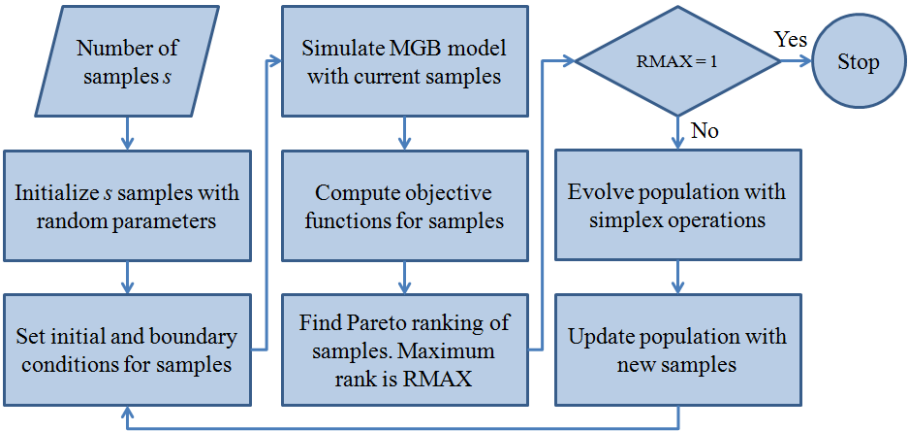
$$s_{ref} = 2s_c - s_w \tag{3.8}$$

$$s_{con} = 0.5(s_c + s_w) \tag{3.9}$$

It is worth noting that each possible feasible solution represents particular characteristics of the watershed due to trade-offs between the objective functions, as improving one causes the deterioration of at least another one. More specifically, the calibration in this thesis particularly employs three objective functions (SORRIBAS et al., 2013), shown in equations (3.10)-(3.12), where NT is the number of time steps.

Equation (3.10) is the Nash-Sutcliffe model efficiency coefficient (NSE) (NASH; SUTCLIFFE, 1970), computed from values of observed discharge Q_{obs}, average observed discharge $\overline{Q}_{obs}$, and simulated discharge Q_{sim}, whereas equation (3.11) considers the NSE computed from the natural logarithm of the discharge values (NSE_{log}). Both NSE and NSE_{log} are within the range $(-\infty,1]$, where 1 means the best fit. Equation (3.12) is a bias, or systematic error, between the simulated and observed discharges (ERR), given as a percentage.

Figure 3.4 - Flowchart of the MOCOM-UA calibration method.



SOURCE: Author.

$$\mathrm{NSE} = 1 - \frac{\sum_{t=1}^{\mathrm{NT}} \left(Q_{obs}^{t} - Q_{sim}^{t}\right)^2}{\sum_{t=1}^{\mathrm{NT}} \left(Q_{obs}^{t} - \overline{Q}_{obs}\right)^2} \tag{3.10}$$

$$\mathrm{NSE}_{log} = 1 - \frac{\sum_{t=1}^{\mathrm{NT}} \left(\log\left(Q_{obs}^{t}\right) - \log\left(Q_{sim}^{t}\right)\right)^2}{\sum_{t=1}^{\mathrm{NT}} \left(\log\left(Q_{obs}^{t}\right) - \log\left(\overline{Q}_{obs}\right)\right)^2} \tag{3.11}$$

$$\mathrm{ERR} = \left(\frac{\sum_{t=1}^{\mathrm{NT}} Q_{sim}^{t}}{\sum_{t=1}^{\mathrm{NT}} Q_{obs}^{t}} - 1\right) \times 100 \tag{3.12}$$

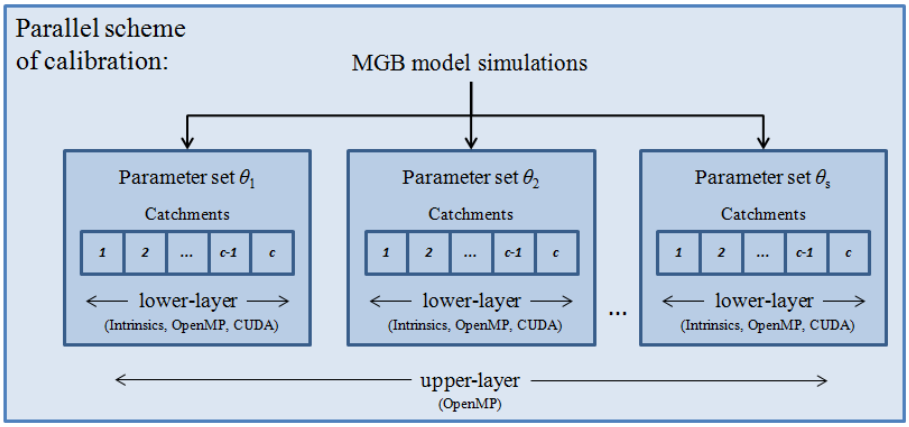
As already pointed out, the calibration procedure in hydrological applications is a time-consuming task that requires computational resources for extensive periods of time, as it demands hundreds or thousands of executions of the hydrological model in order to obtain suboptimal sets of model parameters. It is common to observe calibration executions that take weeks or even months to finish on single current-generation CPUs.

4 PERFORMANCE OPTIMIZATION TECHNIQUES

This thesis employs the MOCOM-UA method as the calibration procedure that finds the sets of watershed parameters using the MGB hydrological model for simulations. The optimizations on the calibration and simulation parts follow the double-layer approach, where the MGB model's simulations are simultaneously executed on different CPU cores (upper-layer), and each simulation is parallelized (lower-layer). The types of optimization selected to boost the performance of the MOCOM-UA method and of the MGB model consist in data-parallel and thread-parallel strategies: vectorization, multi-core CPU and many-core GPU parallelisms.

The upper-layer part employs multi-core CPU parallelism (OpenMP) for processing MGB simulations with different samples (parameter sets) θ_i ($i = 1, 2, \ldots, s$) in parallel, where s is the number of samples. The lower-layer part implements at least one of three types of optimization (Intrinsics, OpenMP, CUDA) for the parallel computations of all catchments that form the watershed. The MGB model's profiling, as shown in Subsection 3.1.3, revealed that the routines of the inertial model constitute the main execution bottleneck, accounting for the highest percentages of total execution time, so the focus of the optimizations was on those routines. Figure 4.1 illustrates the parallelization scheme.

Figure 4.1 - Parallel scheme of calibration procedure with simultaneous MGB model simulations (upper-layer) and parallel processing of catchments (lower-layer).



SOURCE: Author.

4.1 Simulation: MGB model

As already described, the routines of the inertial model implement an explicit numerical scheme devised to process all catchments of the watershed that form a

domain that can be divided into independent subdomains, which in turn are updated using only previous values of water flows. This scheme is different from implicit numerical schemes that would require a system of equations to be solved at each time step (HOFFMAN, 2001). Each routine of the inertial model includes a single loop that iterates over the catchments with successive and independent memory accesses, defining codes highly suitable for different parallelization strategies.

4.1.1 Optimizations for CPU

The multi-core CPU optimizations proposed herein employ explicit vectorization, via interoperability between the Intel Intrinsics library available in C/C++ (INTEL, 2019) and the Fortran MGB model's code, and OpenMP parallelization. The use of OpenMP provides portable optimized codes, although the vectorization requires specific CPU hardware resources (AVX-512 vector extensions). These optimizations are illustrated in detail for the STE routine, which is the first routine called in the inertial model, and that processes each catchment to compute the minimum time step Δt, as in equation (3.3). The OpenMP threads are created outside the routine in a parallel region declared for simultaneous independent calls to the inertial model.

Code 4.1 exhibits the STE routine optimized with vectorization and thread parallelism. In this case, the input variables used for the computations are the height h of the water flow (**Hflow**), the length Δx of the river stretch (**Lriver**), the constant α (**alpha_**), the constant of gravity g (**g_**), the time step value (**Dt**), the minimum time step of each thread (**minLocalDt**[]), the thread identification (**th**), the number of threads (NTH), and the number of catchments (NC). The constant α is multiplied by the length of the river stretch Δx (**Dx**), and then divided by the square root of the constant g multiplied by the height of the water flow h (**h**).

The variables declared as `__m512d` type are vector variables that store multiple data into the CPU registers for the data-parallel vector processing. Firstly, the local minimum time step **minLocalDt** of each thread **th**, and the corresponding vector values **minDt**, are initialized with the time step of the MGB model (largest time step possible), passed as the argument **Dt**. The constants α and g are also initialized.

After the initialization, the loop of catchments is processed by the OpenMP threads, so that each thread cooperates by executing the computations from a number of iterations multiple of 8*NTH, whereas the remaining iterations are sequentially processed outside the routine. The stride value of 8 means that 512-bit CPU registers and vector instructions (**load**, **mul**, **div**, **sqrt**, **min**) are used to simultaneously

process eight 64-bit double-precision data, computing time steps for catchments in parallel. Moreover, the OpenMP clause for `static` scheduling uniformly distributes the workload, which is well balanced in the loop iterations that execute similar instructions.

Code 4.1 - STE routine optimized for CPU with Intrinsics and OpenMP.

```
1  void STE_CPU(Hflow, Lriver, alpha_, g_, Dt, minLocalDt[], th, NTH, NC) {
2    __m512d minDt, alpha, g, h, Dx, num, denom, DtCatchment;
3    int i, NCVec;
4    minLocalDt[th] = Dt; // initialize minimum time step of each thread
5    minDt = _mm512_set1_pd(Dt); // initialize minimum time step
6    alpha = _mm512_set1_pd(alpha_); // initialize constant alpha
7    g = _mm512_set1_pd(g_); // initialize constant g
8    NCVec = NC-NC%(8*NTH);
9    #pragma omp for schedule(static) // process loop iterations in parallel
10   for (i = 0;i < NCVec;i+=8) {
11     // load data: water height and river length
12     h = _mm512_load_pd(Hflow+i);
13     Dx = _mm512_load_pd(Lriver+i);
14     // compute time steps
15     num = _mm512_mul_pd(alpha, Dx);
16     denom = _mm512_mul_pd(g, h);
17     denom = _mm512_sqrt_pd(denom);
18     DtCatchment = _mm512_div_pd(num, denom);
19     // compute minimum time steps of catchments
20     minDt = _mm512_min_pd(minDt, DtCatchment);
21   }
22   // compute local minimum time step for each thread
23   minLocalDt[th] = _mm512_reduce_min_pd(minDt);
24   // wait for all threads and compute global minimum time step
25   #pragma omp barrier
26   if (th == 0) for(i = 0;i < NTH;i++) Dt = min(Dt, minLocalDt[i]);
27 }
```

Whenever each thread **th** finishes processing the loop iterations, its minimum time step is locally computed with a minimum reduction vector operation (**reduce**), and separately stored into **minLocalDt[th]**. Finally, all threads wait at a synchronization barrier to ensure that all local minimum time steps are promptly available, so that the master thread (**th** = 0) can update the global minimum time step **Dt**. The STE

routine is the only routine that requires a synchronization mechanism for ensuring thread safety and correct behavior with multiple threads.

The DIS and CON routines follow similar optimizations, although more extensive and complex, so the full codes are not shown here, only a simplified version. The DIS routine executes the computations from equations (3.4)-(3.5) to update the discharge q of each catchment, as illustrated in Code 4.2. All catchments are processed using as input variables the elevation z of the river bottom (**Z**), the water height h (**Hflow**) used to compute the water level y, the length Δx (**Lriver**) and the width of the river stretch, the Manning coefficient n (**Mann**), the current discharge q (**Qflow**), the minimum time step Δt (**Dt**) computed in the STE routine, the constant of gravity g (**g**), and the reference to the downstream catchment (**Down**), which is the single catchment where the water flows from a particular catchment.

Code 4.2 - DIS routine optimized for CPU with Intrinsics and OpenMP.

```
1  void DIS_CPU(Z, Hflow, Lriver, Mann, Qflow, Dt, g, Down, NTH, NC) {
2    __m512d Z1, Y1, Z2, Y2, Hfl, Lrv, Dwn, Dy, Qfl, n, Num;
3    int i, NCVec;
4    NCVec = NC-NC%(8*NTH);
5    #pragma omp for schedule(static) // process loop iterations in parallel
6    for(i = 0;i < NCVec;i+=8) {
7      (Z1,Hfl,Lrv,Dwn) = load(Z+i,Hflow+i,Lriver+i,Down+i);
8      msk = mask(Dwn != -1); // downstream flow is present
9      Y1 = add(Z1,Hfl,msk);
10     (Z2,Y2) = add(Z1,Y1,Hfl,msk);
11     Dy = sub(max(Y1,Y2),max(Z1,Z2)); // equation (3.4)
12     (Qfl,n) = load(Qflow+i,Mann+i);
13     Num = sub(Qfl,div(mul(g,Dt,Dy,Y1,Y2),Lrv));
14     Qfl = div(Num,add(1,div(mul(g,Dt,Qfl,n),Dy))); // equation (3.5)
15     store(Qflow+i,Qfl); // store updated discharge Qflow
16   }
17 }
```

The DIS routine also executes various arithmetic vector instructions, but it additionally includes logical vector instructions, referred to as mask vector instructions, that function as logical conditions to concurrently check whether the downstream water flows are present or not for each catchment (line 8). These instructions provide vector masks that indicate which catchments require the discharge q to be updated, i.e. only where the downstream water flow is present.

Code 4.3 - CON routine optimized for CPU with Intrinsics and OpenMP.

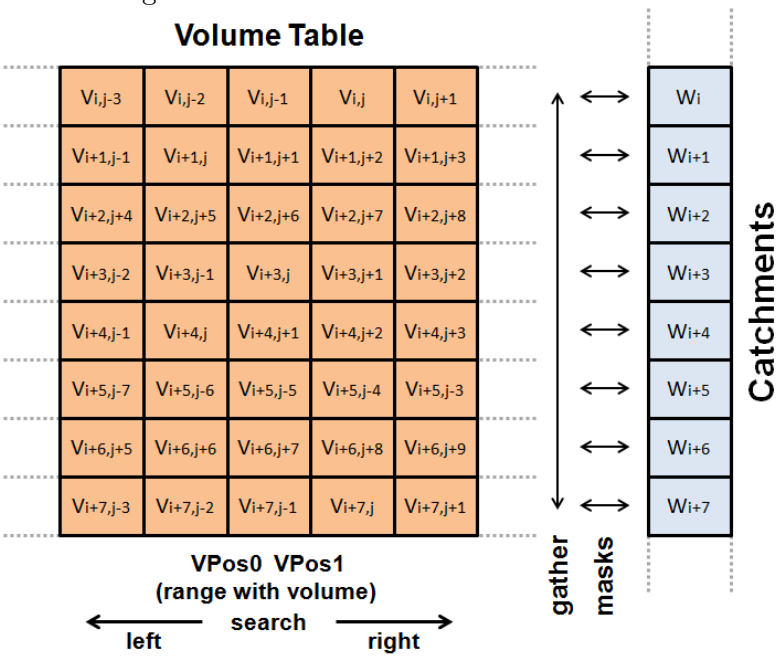
```
1  void CON_CPU(Dt, Qflow, QUp, Evap, Rain, Lriver, MGBDt, Tab, VTab, ATab,
       ZTab, Volume, Area, Hflow, Yflow, NTH, NC) {
2    __m512d SumQ, Qfl, Vol, Lrv, Evp, Rn, Ar, VPos0, VPos1, f, Hfl, Yfl;
3    __m512d VTb0, ATb0, ZTb0, VTb1, ATb1, ZTb1;
4    int i, NCVec;
5    NCVec = NC-NC%(8*NTH);
6    #pragma omp for schedule(static) // process loop iterations in parallel
7    for(i = 0;i < NCVec;i+=8) {
8      for(j = 0;j < 8;j++) QSum[j] = acc(QUp+64*i+8*j,NUp); // upstream sums
9      SumQ = load(QSum);
10     (Qfl,Vol,Lrv) = load(Qflow+i,Volume+i,Lriver+i);
11     (Evp,Rn,Ar) = load(Evap+i,Rain+i,Area+i);
12     Vol = calcVol(SumQ,Qfl,Vol,Evp,Rn,Ar,Dt,MGBDt);
13     VPos0 = load(Tab+i); // initial range position
14     do { // table search
15       VPos1 = inc(VPos0); // final range position
16       (VTb0,ATb0,ZTb0) = gather(VPos0,VTab,ATab,ZTab);
17       (VTb1,ATb1,ZTb1) = gather(VPos1,VTab,ATab,ZTab);
18       VgeMsk = mask(Vol >= VTb0);
19       VltMsk = mask(Vol < VTb1);
20       f = calcFrac(Vol,VTb0,VTb1); // volume fraction
21       Ar = calcArea(ATb0,ATb1,f,VgeMsk,VltMsk);
22       Yfl = calcY(ZTb0,ZTb1,f,VgeMsk,VltMsk);
23       nBitsRange = sumBits(VgeMsk,VltMsk);
24       if(nBitsRange == 16) break; // 8 left/right positions
25       VltMsk = mask(Vol < VTb0);
26       VPos0 = sub(VPos0,VltMsk); // decrement positions
27       VgtMsk = mask(Vol > VTb1);
28       VPos0 = add(VPos0,VgtMsk); // increment positions
29     }
30     while(true);
31     Hfl = sub(Hfl,mul(div(Dt,Lrv),Qfl)); // equation (3.6)
32     Yfl = add(ZTb0,Hfl); // equation (3.7)
33     store(Tab+i,VPos0); // update positions for table search
34     store(Area+i,Ar,Yflow+i,Yfl,Hflow+i,Hfl,Volume+i,Vol);
35   }
36 }
```

The CON routine uses the minimum time step Δt (**Dt**) from the STE routine, and the discharge q (**Qflow**) from the DIS routine. Catchments are also independently processed, using input variables such as the discharges from the upstream catchments (**QUp**), evapotranspiration (**Evap**), rainfall (**Rain**), the length Δx of the river stretch (**Lriver**), the MGB model's time step (**MGBDt**), as well as the range positions (**Tab**) of precomputed water volumes (**VTab**), cross-sectional areas (**ATab**), and elevations (**ZTab**) that are stored in nondecreasing order as fixed 2D tables for each catchment. This routine updates the variables of water volume (**Volume**), the cross-sectional area (**Area**), water height h (**Hflow**), and water level y (**Yflow**) as shown in Code 4.3, which are used in the next iteration of the inertial model.

More specifically, the sum of the upstream discharges of each catchment (line 8) is initially computed with a sum reduction vector operation by collecting and adding up the discharges from the upstream catchments. The upstream discharge values were adapted in the optimized routine to be contiguously arranged in memory, thus allowing the vector instructions to execute more efficient memory accesses. The upstream discharges are used to compute the new water volumes of the catchments (line 12), and each catchment searches its own table of water volumes for the new value. The table search retrieves precomputed water volumes, cross-sectional areas, and elevations from the ranges that include the new water volume, where gather vector instructions collect these values from the memory positions that correspond to the selected ranges (lines 16,17). The new water volume is used to compute a volume fraction for the interpolation of the cross-sectional area and water level (lines 21,22). The boundaries of the selected ranges are checked with vector masks (lines 25,27), and the table search shifts either left or right to locate the ranges that contain the new water volumes. The search stops when all ranges are found (line 24), and the current range positions are updated (lines 26,28).

Figure 4.2 illustrates the relation between the catchments and the water volume table. Table rows store the precomputed water volumes V for each catchment $1 \leq i \leq c$, and the eight positions stored into the vector variables **VPos0** and **VPos1** specify the ranges that contain the current water volumes: $V_{i,j-1} \leq W_i < V_{i,j}$, $V_{i+1,j+1} \leq W_{i+1} < V_{i+1,j+2}$, $V_{i+2,j+6} \leq W_{i+2} < V_{i+2,j+7}$, and so on. Groups of eight values of the water volumes V, located in the range [**VPos0**,**VPos1**], are collected from memory positions using a gather vector instruction (**gather**) and compared using a logical vector instruction (**masks**) with the volumes W of the catchments. Whenever the water volumes W are updated, the current positions **VPos0** independently shift for each catchment, moving either left or right (table columns) to the new range.

Figure 4.2 - Table search of the CON routine.



SOURCE: Freitas et al. (2020a).

4.1.2 Optimizations for GPU

Besides the CPU optimizations, the STE, DIS, and CON routines were also executed on GPU as CUDA kernels with the CUDA Fortran programming interface (NVIDIA, 2020a), supported by the PGI `pgf90` compiler (PGI, 2020). The CUDA framework was chosen because we previously applied directive-based GPU programming standards (OpenACC) on the MGB model (FREITAS; MENDES, 2019), but that particular optimization decreased the MGB model's performance. The low performance was mainly caused by the overheads associated to the OpenACC directives, introduced to the runtimes from millions of routine calls. Those overheads were not amortized by the thousands of loop iterations (number of catchments), i.e. the computations were not GPU intensive.

Current NVIDIA GPU architectures provide thousands of parallel threads, which is enough to cover all loop iterations in the routines of the inertial model for the available datasets, so that the CUDA kernels assign each parallel thread to one single catchment, increasing the degree of parallelism. In the CUDA implementation, the variables of the inertial model were globally declared as *device* variables in a Fortran module, and data was transferred between host (CPU) and device (GPU) memories by directly copying values through the assignment operator.

Code 4.4 illustrates the Fortran module with the CUDA version of the STE routine, which required an explicit minimum reduction operation to compute the minimum stable time step Δt of the inertial model, executed in one single CUDA block for thread synchronization. CUDA does not include the minimum reduction functionality as a predefined operation, differently from Intel Intrinsics and OpenMP that provide a library function and a directive clause, respectively.

Code 4.4 - STE routine optimized for GPU with CUDA Fortran.

```
1  module kernel_ste_cuda
2    contains
3    attributes(global) subroutine ste_cuda
4      use vars_cuda ! global module of variables
5      integer i ! local variables
6      i = blockDim%x*(blockIdx%x-1)+threadIdx%x ! thread ID
7      ! process loop iterations in parallel
8      if (i <= NC) then
9        localDt(i) = alpha*Lriver(i)/(g*Hflow(i))**0.5 ! compute time steps
10     end if
11   end subroutine
12
13   attributes(global) subroutine dt_cuda
14     use vars_cuda ! global module of variables
15     integer i, pow, pow2NC ! local variables
16     i = blockDim%x*(blockIdx%x-1)+threadIdx%x ! thread ID
17     ! compute minimum time step with explicit minimum reduction in parallel
18     pow = pow2NC/2
19     do while (pow > 0)
20       if (i <= pow) then
21         ! update first half of the array with minimum local dt
22         localDt(i) = min(localDt(i), localDt(i+pow))
23       end if
24       call syncthreads() ! synchronize threads
25       pow = pow/2 ! access only half of the array in the next iteration
26     end do
27     ! update global minimum time step computed from threads
28     if (i == 1) Dt = localDt(1)
29   end subroutine
30 end module
```

Firstly, each thread is identified from internal CUDA structures and stored into the variable **i** (line 6), where **blockDim%x**, **blockIdx%x**, and **threadIdx%x** are the dimension (size) of the CUDA blocks, the index of the CUDA block, and the index of the thread inside the CUDA block, in this order, for the dimension **x** of the CUDA grid. The variable **i** is used as an index to the array position of each catchment, which must be checked if it is a valid index, i.e. not larger than the number of catchments NC (line 8).

All valid threads compute the time steps of the catchments in parallel in the **ste_cuda** routine, independently storing the values into the **localDt** array (line 9). Next, the **dt_cuda** routine executes the minimum reduction operation in one single CUDA block, processing only half of the array (positions **pow**, line 18), thus simultaneously computing and updating the minimum time step between positions **i** (first half) and **i+pow** (second half) (line 22). The size **pow2NC** is equal to the smaller power of two that is either greater than or equal to NC.

Therefore, the minimum time step values are kept only in the first half of the array, all threads are synchronized with the CUDA function **syncthreads()** (line 24), and this procedure repeats by halving **pow** at each iteration of the reduction operation (line 25), so that in the end the global minimum time step is stored into the first position of the **localDt** array. Finally, the value **localDt(1)** is used to update the global minimum time step **Dt** (line 28) for the DIS and CON routines, which also implement equivalent CUDA versions for execution on GPU.

4.2 Calibration: MOCOM-UA method

The calibration procedure based on the MOCOM-UA method is implemented using the MGB model to find the optimal sets of parameters for a particular watershed. The MGB model's parameters selected for calibration are shown in Table 4.1, which includes soil parameters that largely affect the hydrological responses in wet and dry periods (COLLISCHONN, 2001). Those parameters are defined in the MGB model to set the initial and boundary conditions for simulations, and also to compute the hydrometeorological data, i.e. radiation, evapotranspiration, and water balance in soil. The ranges in Table 4.1 specify multipliers for the parameters relative to the initial value.

In addition to the optimizations on the MGB model (simulation), which include vectorization with Intel Intrinsics, multi-core parallelism with OpenMP, and many-core parallelism with CUDA, like discussed in the previous section, the calibration

Table 4.1 - MGB model's watershed parameters selected for calibration.

Parameter	Description	Range	Initial value
Wm_s	Maximum water storage capacity for forest on shallow soil (mm)	[0.1,1.1]	700.0
Wm_d	Maximum water storage capacity for forest on deep soil (mm)	[0.6,1.1]	900.0
K_{grn}	Maximum percolation on saturated soil for groundwater flows (mm/day)	[0.3,1.8]	0.5
K_{sub}	Maximum percolation on saturated soil for subsurface flows (mm/day)	[0.3,1.8]	5.0
b	Shape measure from relation between storage and saturation	[0.3,1.8]	0.1
CS	Delay correction for propagation of surface flows	[0.1,2.0]	10.0
CI	Delay correction for propagation of subsurface flows	[1.0,4.0]	50.0
CB	Delay correction for propagation of groundwater flows	[0.7,6.7]	1500.0

was also optimized for performance improvements. From the flowchart in Figure 3.4, the steps of simulating the MGB model with the current set of samples and of evolving the population of samples with simplex operations require most of the computational resources in the calibration procedure, as both steps execute complete simulations of the MGB model.

The former step executes the MGB model to compute the simulated discharge values that are used to evaluate the objective functions for each set of parameters in order to update the Pareto ranking of all samples. The latter step executes simulations for all worst-ranked samples modified with simplex operations, i.e. the samples located far from the Pareto front. As the samples are independent from each other, it is possible to process all of them in parallel for both steps.

However, for the parallel executions of the MGB model to properly work, each sample requires separate executions of four routines that are successively called in the calibration procedure to: (a) set the values of the parameters, (b) set the initial and boundary conditions, (c) simulate the MGB model, and (d) compute the objective functions. Each sample is assigned a new Pareto rank only after these routines have been completely executed for all samples. Code 4.5 illustrates the calibration implementation parallelized with OpenMP for simultaneous MGB model's executions, where s is the only input argument that is used to define the number of samples (sets of parameters) to be optimized in the calibration procedure.

Code 4.5 - Calibration procedure of the MOCOM-UA method.

```
1  subroutine MOCOM_calibration(s)
2    call initial_random_parameters(samples_population(s))
3    !$omp parallel do num_threads(NCORES) schedule(static,1)
4    do i=1,s ! th = thread ID, set with get_thread_num(th)
5      call set_parameters(samples_population(i),th)
6      call set_conditions(i,th)
7      call MGB_model(i,th) ! execute on either CPU or GPU
8      call objective_functions(i,th)
9    end do
10   !$omp end parallel do
11   ! loop for parameter optimization
12   do while (RMAX > 1) ! initially, RMAX = s
13     call pareto_ranking(samples_population)
14     call samples_probability(samples_population)
15     call simplex_ops(worst_samples(NRMAX,2)) ! reflection=1/contraction=2
16     !$omp parallel do num_threads(NCORES) schedule(static,1)
17     do i=1,NRMAX ! th = thread ID, set with get_thread_num(th)
18       call set_parameters(worst_samples(i,1),th) ! reflection
19       call set_conditions(i,th)
20       call MGB_model(i,th) ! execute on either CPU or GPU
21       call objective_functions(i,th)
22       if (worst_samples(i,1) is nondominated) then
23         call replace_sample(samples_population, worst_samples(i,1))
24       else
25         call set_parameters(worst_samples(i,2),th) ! contraction
26         call set_conditions(i,th)
27         call MGB_model(i,th) ! execute on either CPU or GPU
28         call objective_functions(i,th)
29         call replace_sample(samples_population, worst_samples(i,2))
30       end if
31     end do
32     !$omp end parallel do
33   end do
34 end subroutine
```

The **initial_random_parameters** routine allocates the s samples in the **samples_population** array (line 2), storing the initial random parameter values of each sample. After the initialization, the samples are independently processed

in a loop with the aforementioned routines, so that all loop iterations process the samples in parallel (upper-layer), simultaneously executing the **set_parameters**, **set_conditions**, **MGB_model**, and **objective_functions** routines required for the initial Pareto ranking of all samples. This loop (line 4) is parallelized with an OpenMP directive that assigns each loop iteration (sample) to one single thread, where the number of threads is set to NCORES (number of CPU cores). The clause **schedule(static,1)** cyclically assigns one loop iteration at a time to the threads.

Following the initialization loop, the calibration procedure starts and repeats while there are samples to be improved, i.e. there is at least one sample with Pareto rank greater than one (RMAX >1, line 12). Firstly, the **pareto_ranking** routine (line 13) assigns ranks to the samples computed from the objective function values, and each rank gives a probability value that is assigned to the sample in the **samples_probability** routine (line 14). The NRMAX worst-ranked samples are selected, and the reflection and contraction simplex operations are executed in the **simplex_ops** routine (line 15) in order to generate two new samples for each worst-ranked sample, so that one of these new samples updates the current population of samples by replacing the corresponding worst-ranked one.

The loop that updates the population of samples (line 17) is also parallelized with OpenMP. The reflection sample is attempted first (line 18), and if it is accepted, i.e. if it is considered a nondominated sample (line 22), the new reflection sample replaces the current worst-ranked sample (line 23), otherwise the contraction sample replaces it (line 29). After all worst-ranked samples are processed, the condition of the calibration loop is retested in order to check whether the calibration procedure continues or not.

In the calibration, the MGB model executes on either CPU or GPU (lower-layer), whereas all the other routines execute only on CPU. In addition, it is worth mentioning that the random probability value, which is generated to select samples for the simplex operations, determines how fast the worst-ranked samples move towards the Pareto front, so that the execution times of different calibration runs can randomly vary even with the same initial set of samples.

The insertion of parallelism on the calibration procedure for simultaneous executions of the aforementioned routines required a thorough adaptation of the MGB model and the MOCOM-UA method implementations to avoid data race conditions. All global variables used in the **set_parameters**, **set_conditions**, **MGB_model**, and **objective_functions** routines needed an extra dimension to be allocated as the

total number of parallel threads (NCORES) and independently indexed by each thread ID (variable **th** in Code 4.5). Moreover, all variables used in other routines such as radiation, evapotranspiration, and water balance in soil, had to be locally defined to be private to each thread. This adjustment was carefully made to hundreds of variables in order to ensure that all simultaneous executions of those routines correctly worked, not interfering with each other while the calibration was executed.

As an additional modification made to the calibration procedure, a different implementation considers the parallel processing of more samples from the population, not only the worst-ranked samples. This approach tries to further increase the performance of the current implementation, by also selecting the samples with Pareto ranks just below the worst rank RMAX, i.e. RMAX-1, RMAX-2, and so on. The purpose of this strategy is to process and improve a larger number of bad samples at the same iteration of the calibration procedure, while still maintaining the quality of the calibration results, so that the objective function values remain close to the values achieved by the original calibration.

This proposed approach has the advantage of more efficiently exploiting the hardware resources. By executing the calibration loops with the number of OpenMP threads equal to the number of available CPU cores, the degree of parallelism increases because more samples can be processed in parallel. Therefore, more parallel threads are simultaneously working, reducing the amount of core resources that remain idle.

5 COMPUTATIONAL PERFORMANCE RESULTS

This chapter presents the execution times (runtimes) and corresponding speedups (ratio between sequential and parallel runtimes) achieved by the optimized implementations, which are basic measures for performance analysis. The computational testbed and input datasets are also presented.

5.1 Computational testbed

The computational testbed employed to conduct the experiments consists of two multi-core CPU+GPU computer systems with Linux CentOS 7.5 that include Intel 19.0.5, NVIDIA 10.2, and PGI 19.10 software development tools. The hardware specification of each computer system is detailed in Table 5.1 that exhibits the processor and memory capabilities of the CPU and GPU architectures.

The floating-point double-precision CPU performance and the DRAM memory bandwidth were experimentally obtained with the Intel Advisor performance analysis tool (INTEL, 2020). Both CPU architectures are based on Intel's Skylake that supports AVX-512 vector instructions (512 bits). The bandwidth of L1 (*shared*) and L2 caches of GPUs are from load throughput found with specific micro-benchmarks (JIA et al., 2018).

Table 5.1 - Computer systems used as computational testbed.

Name		**system01**		**system02**	
		CPU	GPU	CPU	GPU
	Architecture	Intel Core i7-7820X	Pascal GP104	Intel Core i9-7900X	Volta Titan V
Processor	Cores	8	2560	10	5120
	Frequency (GHz)	3.60	1.77	3.30	1.46
	Performance (Gflops/s)	919.5	283.2[a]	1052.6	7475.2
DRAM	Size (GB)	32	8	64	12
	Bandwidth (GB/s)	40.3	320.3[b]	68.5	652.8[b]
L1	Size (kB)	32	48	32	48
	Bandwidth (GB/s)	3874.1	3555.0	5377.3	12080.0
L2	Size (MB)	1	2	1	4.6
	Bandwidth (GB/s)	1813.4	979.0	1473.6	2155.0
L3	Size (MB)	11	-	13.75	-
	Bandwidth (GB/s)	226.7	-	317.8	-

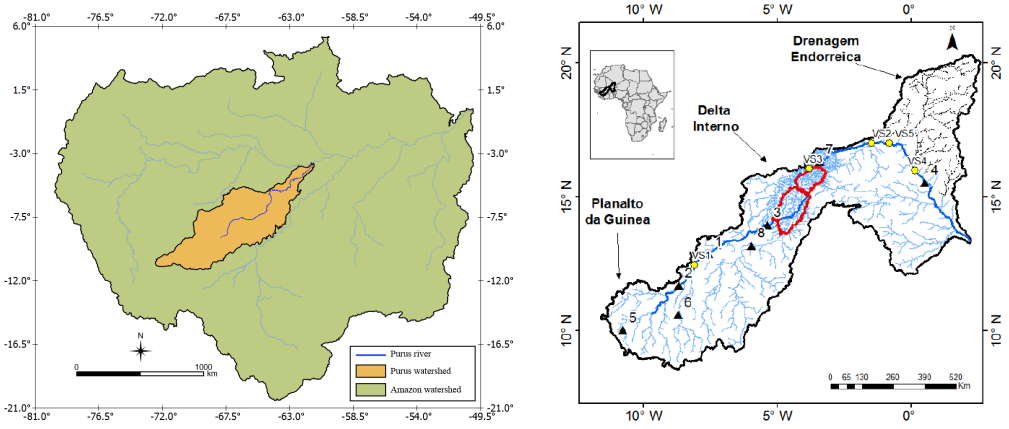
[a]1/32 of FMA single-precision performance. [b]From NVIDIA tool.

5.2 Input datasets

Different datasets were used as input to the MGB model providing hydrometeorogical data from two regions of the world. These regions are the locations of the watersheds of the Purus and Niger rivers. The Purus river is one of the main tributaries of the Amazon river in Brazil with drainage area of 370 000 km^2 and average discharge of 11 000 m^3/s (PAIVA et al., 2011). This region (Figure 5.2(a)) is mostly covered by tropical rainforests, which provide high annual rainfall rates, and the watershed topography ranges from 10 m, at the outlet, to 581 m, at the highest headwater, measured from the Shuttle Radar Topography Mission (SRTM) 30 m topographic data (FARR et al., 2007).

The Niger river (Figure 5.2(b)) covers the Guinea highlands and the Sahel desert, and is the largest river in West Africa with drainage area of 657 000 km^2 (FLEISCHMANN et al., 2017) and average discharge of 900 m^3/s (MAHÉ et al., 2009). The geographical region of this river presents semi-arid climate with large seasonal variation in rainfall and river flow (ZWARTS et al., 2005), where approximately 40 % of the water available from the headwaters are lost due to either evaporation or infiltration processes, and also because of the small number of tributaries along the extension of its watercourse. The construction of dams and irrigation systems provides solutions of hydroelectric structures to optimize the scarce water availability.

Figure 5.1 - Regions of the Purus (left) and Niger (right) rivers used as input datasets.



(a) Purus river and watershed inside the Amazon region.

(b) Niger river and watershed in West Africa.

SOURCE: (a) Author, (b) Fleischmann et al. (2017).

Both input datasets were provided by IPH-UFRGS as worst-case datasets for these regions, requiring high computational effort to be processed. Each input dataset contains discrete data for the hydrological units of catchments, sub-basins, and HRUs (spatial), for each time step (temporal). Two datasets were used for the Purus region, one for simulation and other for calibration, whereas the dataset for the Niger region is only for simulation. The total number of catchments, sub-basins, HRUs, and time steps are displayed in Table 5.2.

Table 5.2 - Input datasets used in the MGB model for simulation and calibration.

Dataset	Purus		Niger
	SIM[a]	CAL[b]	SIM[a]
Catchments	1984	2957	4307
Sub-basins	16	1	9
HRUs	9	9	11
Time steps	4747	5486	5800

[a]Simulation. [b]Calibration.

5.3 Performance on multi-core CPUs and many-core GPUs

This section presents the performance of the simulation (MGB model) and calibration (MOCOM-UA method) implementations. The results include both the CPU and GPU performance achieved on each computer system for the available input datasets.

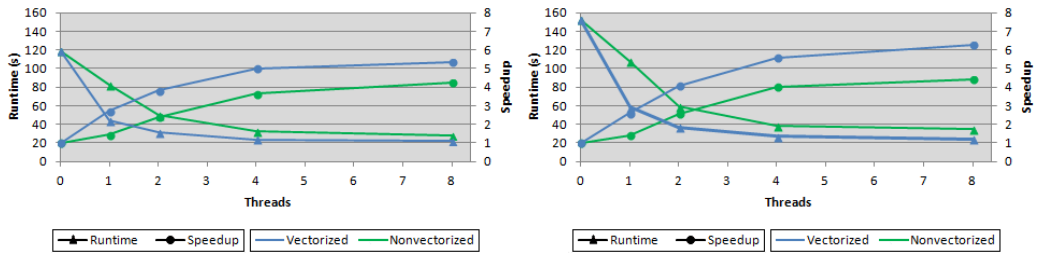
5.3.1 Simulation

The experiments conducted on multi-core CPUs considered executions of the original nonoptimized version of the MGB model, as well as of the optimized version that includes AVX-512 vector instructions and OpenMP directives. The number of OpenMP threads used in the experiments ranges from one to the maximum number of CPU cores available on each multi-core CPU, disregarding hyper-threading, as the results with hyper-threading showed only marginal improvements on the final performance of the optimized version.

For the performance analysis of the fully optimized MGB model, it is also important to evaluate the performance achieved without vectorization, i.e. using only thread parallelism with OpenMP, regarding portability issues. As the vectorized code requires CPUs with AVX-512 vector extensions, the nonvectorized code can serve as a basis to verify how much the performance increases with the additional use of vectorization.

As an example, Figure 5.2 illustrates the differences in runtime and speedup for the vectorized and nonvectorized implementations of the MGB model, plotted as a function of the number of OpenMP threads and executed with 1, 2, 4, and 8 threads on **system01** for the Purus and Niger datasets. In all cases, the vectorized MGB model was faster than the nonvectorized version, achieving speedup differences of up to 1.86 for 8 OpenMP threads (about 42 % higher). Moreover, one notices that the runtimes of the nonvectorized version executed with only one OpenMP thread were lower than the runtimes of the original nonoptimized version. This probably occurs because the OpenMP directives trigger additional compiler optimizations that more efficiently exploit the CPU resources.

Figure 5.2 - Runtimes (s) and speedups of the vectorized and nonvectorized implementations of the MGB model on **system01** for the simulation input datasets. "Threads = 0" corresponds to the original nonoptimized code.



(a) Purus dataset. (b) Niger dataset.

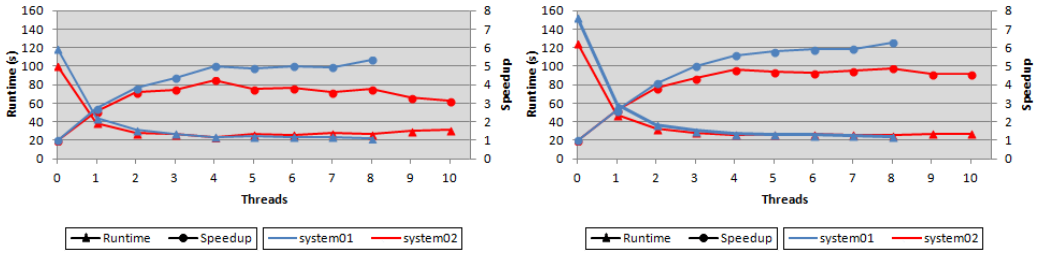
SOURCE: Author.

Figure 5.3 exhibits the runtimes and speedups achieved by the fully optimized MGB model on the multi-core CPUs of each computer system for the simulation input datasets. The single-threaded vectorized MGB model reached the highest speedup of 2.71× for the Purus dataset on **system01** (the inertial model reached the speedup of 3.03×), which indicates that the overall performance of the MGB model significantly improved with only vectorization. In addition, the lowest runtimes for the single-threaded version were achieved on **system02**, which consists of a more advanced CPU and higher bandwidth memory when compared to the hardware resources of **system01**.

The performance analysis reported from each system by the Intel VTune profiler (INTEL, 2020) indicated that the performance of the single-threaded vectorized MGB model was limited by how many floating-point instructions of the original code were vectorized, which did not exceed 65.3 %, resulting in the presence of scalar floating-point instructions in the compiled code. Besides this limitation, measurements of the amount of CPU cycles spent on long-latency division operations and on pipeline

slots stalled due to the demand of memory load/store instructions reached up to 27.9 % and 25.7 %, respectively, which also hindered performance.

Figure 5.3 - Runtimes (s) and speedups of the MGB model on multi-core CPUs for the simulation input datasets. "Threads = 0" corresponds to the original nonoptimized code.



(a) Purus dataset. (b) Niger dataset.

SOURCE: Author.

The combination of vectorization and multi-core parallelism resulted in larger reductions on the runtimes of the MGB model, increasing the corresponding speedups, although the runtimes reached a plateau showing only slight changes, independently of the number of threads. This occurred because the amount of workload processed by the routines of the inertial model, i.e. the number of catchments, is on the order of thousands of loop iterations, which is not enough to provide good scalability, as each thread is assigned a small amount of loop iterations to be processed.

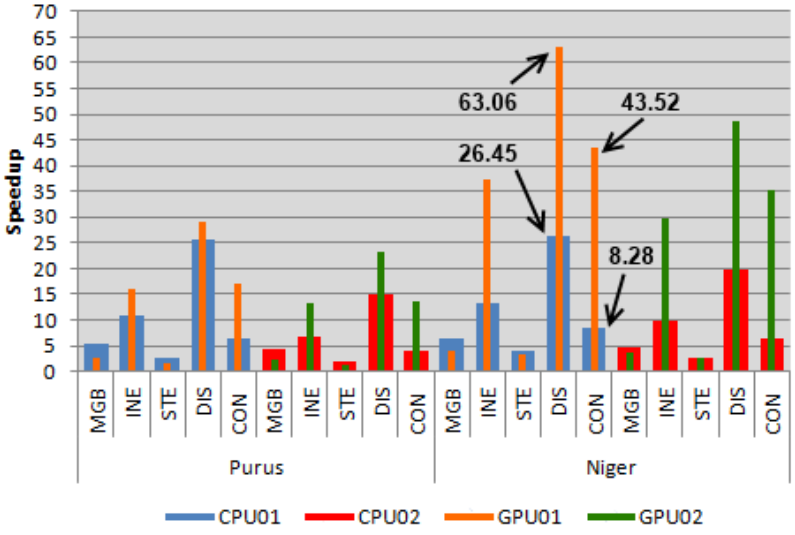
This effect can be noticed by comparing the speedups achieved by the Purus and Niger test cases. As the workload of the Niger dataset is approximately 2× larger than the workload of the Purus dataset, the available hardware resources are more efficiently exploited in the former case, thus resulting in higher speedups. In addition, even though the parallel runtimes across the systems were similar, the speedups on **system01** were higher because the execution of the nonoptimized MGB model on this system was slower, so that the effects of the optimizations were more pronounced than on **system02**.

From Amdahl's law, the highest speedups achievable by the MGB model on **system01** and **system02**, assuming full parallelization of the inertial model, are 5.75× and 5.88×, respectively. These theoretical speedups were computed from the average runtime percentage of the inertial model (Subsection 3.1.3) relative to the full execution of the original nonoptimized MGB model across the Purus and Niger datasets on each system, considering the total number of cores available.

The highest speedups achieved with the optimized MGB model on **system01** were 5.32× (Purus) and 6.27× (Niger), both with 8 threads, whereas on **system02** were 4.26× (Purus) and 4.89× (Niger), the former with 4 threads and the latter with 8 threads. This indicates that significant speedups were achieved with the optimizations proposed herein, and the case in which the speedup achieved for the Niger dataset on **system01** was higher than the theoretical peak probably occurred due to cache effects, as more L2 caches (faster accesses) were available with multi-core processing.

Besides the executions of the MGB model on multi-core CPUs, the optimized CUDA version implemented for many-core GPUs achieved higher speedups for the inertial model, computed from serial CPU runtimes and parallel GPU runtimes. As GPUs work with sets of 32 threads (warps), the GPU implementation executed the CUDA kernel of each routine using 256 threads per block (multiple of 32) that resulted in the highest speedups for the MGB model. Figure 5.4 depicts a comparison between the optimal CPU speedups and the GPU speedups achieved relative to the original CPU runtimes for each input dataset, showing that the inertial model's performance on GPU was higher than on CPU in all cases.

Figure 5.4 - Speedups of the MGB model, the inertial model (INE), and the STE, DIS, and CON routines on CPU and GPU for the Purus and Niger input datasets on **system01** and **system02**.



SOURCE: Author.

The runtimes of the STE, DIS, and CON routines of the inertial model were greatly reduced with the executions of the corresponding CUDA kernels on GPU. The average runtimes of the routines for the Purus and Niger datasets, considering both GPUs,

were 2.33 s and 1.27 s, respectively, which explains the noticeably high speedups achieved for the DIS and CON routines that execute most of the computations, as highlighted for the Niger dataset on **system01**. The performance of the STE and CON routines on CPU and GPU was hindered by the insufficient number of floating-point operations (STE) and by the table search that requires many memory movements (CON), not reaching significant speedups as the DIS routine, which more effectively exploited the CPU and GPU hardware resources.

However, although the runtimes of the routines were considerably small, the speedups of the MGB model on both the Pascal GPU of **system01** (GPU01) and the Volta GPU of **system02** (GPU02) were lower than the speedups achieved on the multi-core CPUs of each system (CPU01 and CPU02). This result was mainly caused by the overheads of data transfer between host (CPU) and device (GPU) memories, and of kernel launches.

By collecting performance data with the CUDA profiling tool, the total percentage of the MGB model's execution time on GPU that was spent on data transfers and kernel launches ranged from 68.90 % to 71.88 % for the Purus dataset and from 53.79 % to 54.57 % for the Niger dataset, as Table 5.3 shows. More specifically, the Purus dataset required more calls to the CUDA API function of kernel launches (4 272 300) when compared to the Niger dataset (2 791 500).

Table 5.3 - Percentages of CPU/GPU data transfers and GPU kernel launches.

	system01		system02	
Dataset	Purus	Niger	Purus	Niger
Data transfer	22.33 %	21.02 %	18.47 %	13.84 %
Kernel launches	46.57 %	33.55 %	53.41 %	39.95 %
Total	68.90 %	54.57 %	71.88 %	53.79 %

The MGB model's performance could increase with larger workloads, as the available input datasets are not GPU-friendly. These circumstances indicate that the development of more efficient CPU codes is more advantageous than the GPU solution. Even though the GPUs provided enough threads to assign each loop iteration to one single GPU thread, significantly reducing the inertial model's runtimes, the MGB model did not scale when executed on GPU because of the overheads that were added to the final runtimes, decreasing the overall MGB model's performance.

5.3.2 Calibration

In addition to the simulation results, the performance of the calibration procedure was also evaluated. As previously mentioned, the calibration was executed for a particular Purus input dataset that additionally employs a time series of observed discharges, which are essential data to compute the objective functions to calibrate the watershed parameters. The observed discharges are from the municipality of Canutama in the Amazonas state, Brazil, located at the outlet of the Purus watershed. All discharge values were automatically collected with a fluvial gauge controlled by the Brazilian National Water Agency, or ANA ("Agência Nacional de Águas"), for the period from January 2^{nd} 2000 to January 7^{th} 2015.

The calibration procedure was executed with the following optimizations: (a) either vectorization and multi-core CPU parallelism or many-core GPU parallelism, for the loops that process catchments (simulation), and (b) multi-core CPU parallelism, for the loops that process samples (calibration). Therefore, the analysis of the performance of the calibration procedure considers different types of executions depending on which optimizations were selected: (a) nonoptimized simulation and calibration (original code), where neither the simulation nor the calibration were optimized; (b) optimized simulation, where only the simulation was optimized; and (c) optimized simulation and calibration, where both the simulation and calibration were optimized.

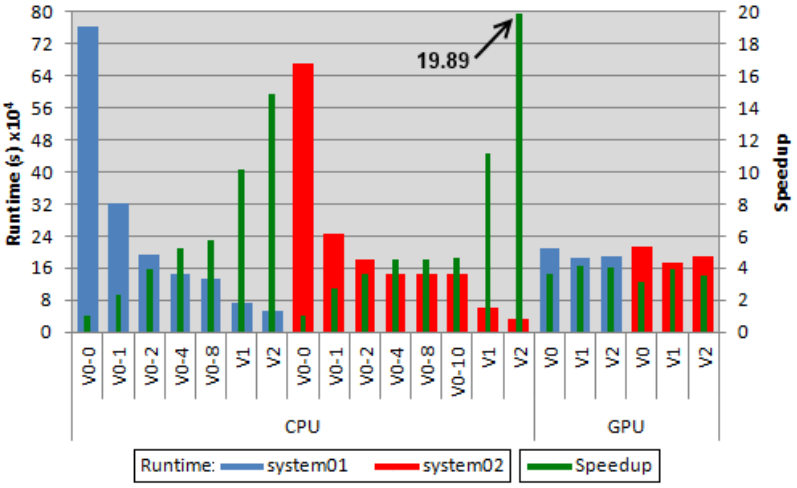
Besides setting the number of samples $s = 100$ as the only input argument, the calibration procedure was also configured to limit the number of generations in the evolution of samples to 400 generations. This value was obtained from fully executing the nonoptimized original calibration on **system01** (422) and **system02** (434), i.e. until all samples were considered nondominated.

Figure 5.5 exhibits the runtimes and speedups for all types of execution. Each type is identified by a code that indicates whether or not the simulation and calibration were optimized and the level of parallelism used. The types `V0-X` for CPU specify that the calibration was not optimized (`V0`), and `X` threads were used in the simulation. The value `X`=0 is for the original nonoptimized code and the values `X`=$\{1, 2, 4, 8, 10\}$ are for the simulation optimized with vectorization and `X` threads.

The types `V1` and `V2` are the optimized calibration that assigned samples to CPU threads selected from either the worst-ranked samples (`V1`) or also other bad samples with Pareto rank larger than one in the same generation (`V2`). In both

cases, simulations were executed with only vectorization, as adding nested parallel regions for simulation and calibration resulted in performance degradation due to overheads introduced for allocating and managing new parallel regions and sub-threads, whenever the STE, DIS, and CON routines were called in simulations.

Figure 5.5 - Runtimes (s) and speedups of the calibration procedure on CPU and GPU for the Purus input dataset.



SOURCE: Author.

The results for the type `V2` were obtained from calibration executions that processed bad samples of Pareto rank up to RMAX-5, which was the case with the highest speedups. Furthermore, the types `V0`, `V1`, and `V2` for GPU also assigned samples to CPU threads in the calibration, although simulations were executed on GPU in all cases.

It can be noticed that the calibration procedure is a really time-consuming task. The original nonoptimized calibration (`V0-0`) took more than seven and eight days to reach the limit of 400 generations, when executed uninterruptedly on **system02** and **system01**, respectively. This indicates that whenever the input dataset needs to be updated for a recalibration, computational resources are used for long periods of time, thus directly affecting the usefulness of the calibration procedure.

The calibration with optimizations only in the simulation (`V0-X`) showed considerable performance gains. The smallest runtime achieved was 132 073 s (1 d 12 h 41 min 13 s) on **system01** with 8 threads (`V0-8`), reaching the speedup of 5.78×. In addition, the use of 10 threads on **system02** (`V0-10`) had a small effect on decreasing the runtime, possibly because the workload is not large enough to exploit more CPU hardware resources.

Even though the performance results for the types `V0-X` reached significant speedups, the highest performance levels were achieved with the type `V2` on both systems, i.e. processing in parallel all bad samples of Pareto rank up to RMAX-5. This indicates that selecting and processing a larger number of samples for each generation in the evolution of samples, not only the worst-ranked samples (RMAX) as originally established in the MOCOM-UA method, resulted in more performance gains. In this case, the smallest runtime was 33 738 s (9 h 22 min 18 s) on **system02**, reaching the highest speedup of 19.89×.

The GPU results were similar for all types of optimization and did not exhibit performance gains when compared to the best CPU results. The GPU runtimes and speedups were comparable to the CPU results obtained from the type `V0-2`. This behavior was expected because the previous analysis about the performance of the simulations of the MGB model executed on GPU showed that the overheads of data transfer between host and device memories, and of kernel launches, degraded performance. This effect was more pronounced in the calibration procedure that required a large number of simulations, as indicated in Table 5.4.

Table 5.4 - Number of simulations executed on GPU in the calibration procedure.

Type	system01	system02
V0	4729	4600
V1	3847	3619
V2	5409	4122

5.4 Scalability with problem size

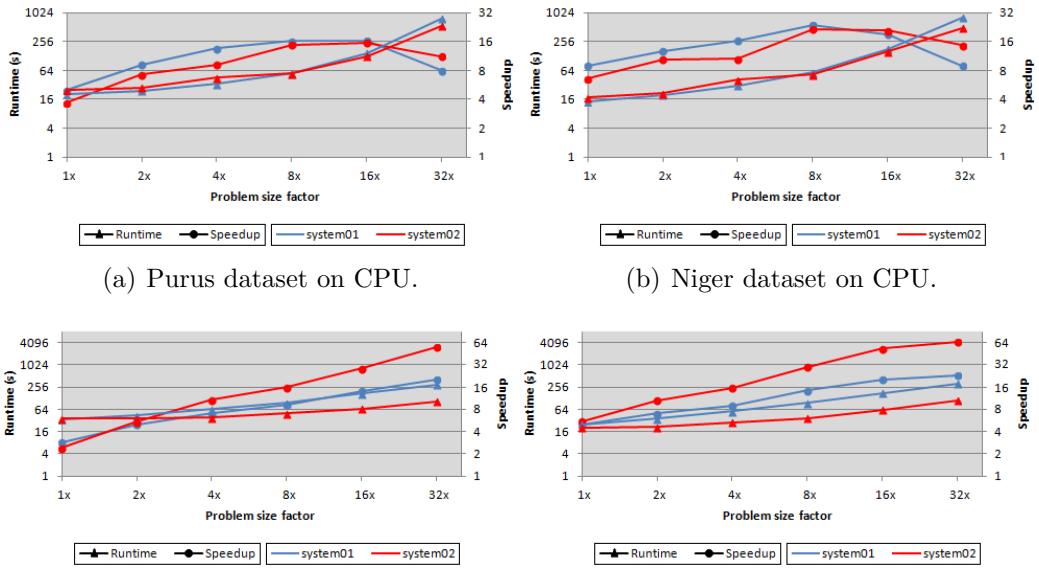
The available input datasets (Purus and Niger) used in this thesis for simulations of real-world hydrological scenarios using the MGB model did not provide workloads, i.e. the number of catchments (loop iterations), of sizes large enough to reach optimal scalability, even by exploiting most hardware resources of the multi-core CPUs and many-core GPUs. Additional insights on the limitations of the MGB model's parallel speedups were obtained from a scalability analysis conducted with synthetic hydrometeorological datasets, larger than those originally available, designed from replication of parts of the original data and used as input to a miniapp application. This miniapp defines a reliable proxy of the MGB model that reproduces the same computations and optimizations, while allowing the use of workloads of varying sizes.

The code structure of the MGB model's miniapp includes the key STE, DIS, and CON routines of the inertial model. However, the data that is explicitly computed at each time step in other routines of the MGB model such as radiation, evapotranspiration, and water balance, was previously stored into separate files with the original model to be read by the miniapp, and then used in the inertial model.

The miniapp avoids a full simulation of the MGB model, so that only the critical STE, DIS, and CON routines process varying amounts of workload (catchments), which was replicated from the original data in the loop of each routine as a multiple of the size of the original dataset. More flexible settings (larger datasets) allow to evaluate how the MGB model's performance scales with problem size.

Figure 5.6, in logarithmic scale, illustrates the runtimes and parallel speedups of the MGB model's miniapp, plotted as a function of the problem size, executed on CPU and GPU for the Purus and Niger input datasets on **system01** and **system02**. The problem sizes of the workloads range from 1× to 32× the original size of each input dataset. The CPU results are from simulations optimized with vectorization and multi-core parallelism using the maximum number of cores available on each system, i.e. 8 cores on **system01** and 10 cores on **system02**, whereas the GPU results are from simulations optimized with CUDA.

Figure 5.6 - Runtimes (s) and speedups of the MGB model's miniapp on CPU and GPU for the Purus and Niger input datasets on **system01** and **system02**.



(a) Purus dataset on CPU.

(b) Niger dataset on CPU.

(c) Purus dataset on GPU.

(d) Niger dataset on GPU.

SOURCE: Author.

The speedups on CPU increased up to the problem size of 8× for both input datasets, reaching the highest value of 23.56× for the Niger dataset on **system01**. Most speedups seemed not to improve with problem sizes either equal to or larger than 16×, as larger datasets increase the number of accesses to the slower DRAM memory, and the memory latency is not totally hidden even with the larger availability of faster L2 caches of the CPU cores.

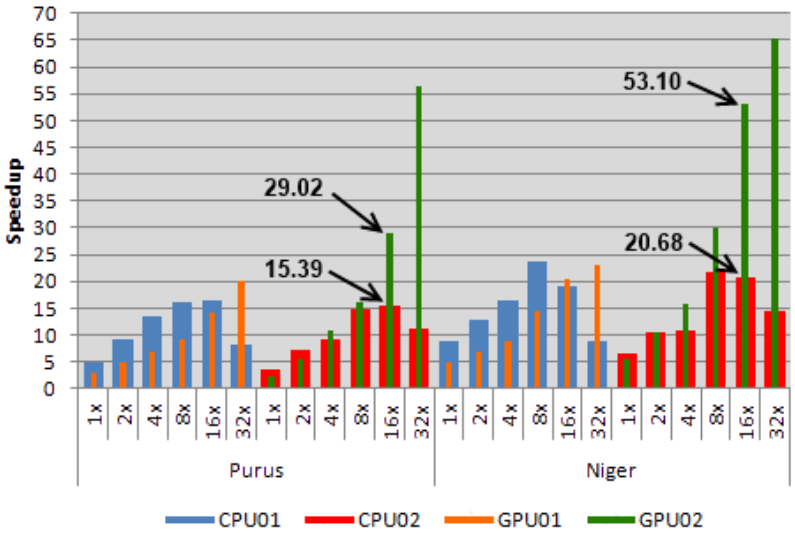
However, the lowest speedup of 8.07× for the Purus dataset on **system01** with the problem size of 32× was still a satisfactory result, achieved using 8 CPU threads. In addition, the higher speedups achieved by the MGB model's miniapp on **system02** with the problem size of 32× are explained by the fact that this system provides more CPU cores, and also larger L3 cache size and higher bandwidth memories.

From a similar analysis, the speedups on GPU showed a different behavior. The speedups of the MGB model's miniapp on both GPUs kept increasing with the problem size, although at different rates. The gap between the speedups from the Pascal GPU (**system01**) and the Volta GPU (**system02**) was largest for the Niger dataset with the problem size of 32×, where the speedup on the Pascal GPU was 22.90× and on the Volta GPU was 65.20× (2.85× higher). This was expected as both the performance (7475.2 Gflops/s) and DRAM bandwidth (652.8 GB/s) of the Volta GPU are higher than the performance (283.2 Gflops/s) and DRAM bandwidth (320.3 GB/s) of the Pascal GPU.

By comparing the CPU and GPU speedups of the MGB model's miniapp as in Figure 5.7, smaller problem sizes present higher speedups on CPU, mainly when compared to the Pascal GPU. As previously discussed, this occurs because the overheads of data transfers and kernel launches associated with the MGB model's execution on GPU for small input datasets degrade performance (speedup behavior on **system01**).

Furthermore, the maximum performance of both CPUs ($\geq$ 919.5 Gflops/s) is higher than the performance of the Pascal GPU, which has more impact in small dataset cases than the slower CPU DRAM memory bandwidth ($\leq$ 68.5 GB/s). However, for larger problem sizes, the performance on GPU was higher than on CPU (shown for the problem size of 16× on the Volta GPU). The scalability analysis of the MGB model's miniapp indicates that, if large datasets (substantial number of catchments) are available, the MGB model's performance will be higher on GPU than on CPU.

Figure 5.7 - Speedups of the MGB model's miniapp on CPU and GPU for the Purus and Niger input datasets on **system01** and **system02**.



SOURCE: Author.

6 OPTIMIZATION ANALYSIS

This chapter provides a performance analysis of the optimized implementations. This analysis was based on CPU and GPU roofline characterizations of the original and optimized MGB model, to confirm that the optimizations more efficiently exploited the available resources of the underlying systems. In addition, the numerical accuracy of the hydrological results computed from the optimized implementations was quantitatively evaluated, to ensure that the optimized versions of the MGB model produced valid numerical results.

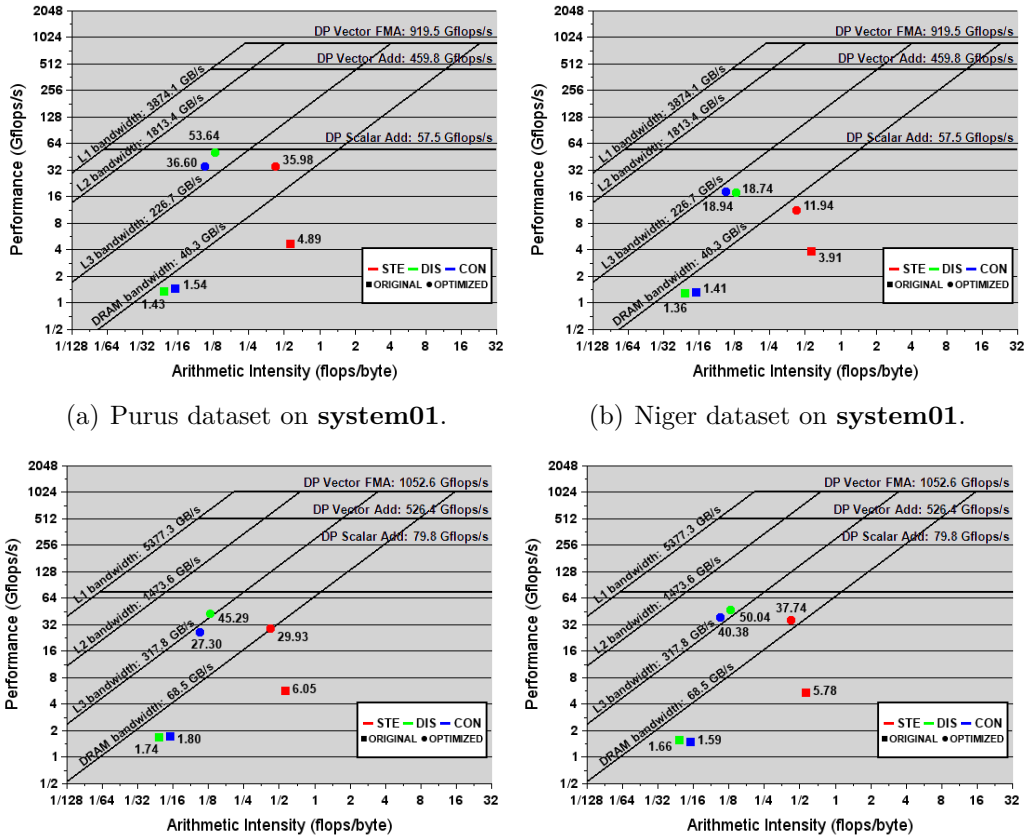
6.1 CPU and GPU roofline analysis

The roofline model provided a visual analysis of the performance behavior of the MGB model on the computer systems used in this thesis. The CPU roofline characterization obtained from executions of the MGB model includes both the original nonoptimized and optimized versions, where only the key routines, i.e. the routines of the inertial model, were considered for the analysis. The amounts of flops and bytes processed by each routine, which are required for the roofline characterization, were collected with the Intel Advisor performance analysis tool (INTEL, 2020) that is based on the Cache-Aware Roofline Model (CARM) (MARQUES et al., 2020; ILIC et al., 2013).

Figure 6.1 exhibits the CPU roofline characterization of the STE, DIS, and CON routines for each input dataset (Purus and Niger) on **system01** and **system02**. In all cases, the position of the optimized routine was moved upwards, indicating that the performance increased in relation to the nonoptimized version, thus confirming that the optimized version with vectorization and multi-core parallelism more effectively used the available hardware resources. The characterization of the optimized routines shown in these rooflines are for the performance achieved using the maximum number of cores on each system, i.e. 8 cores on **system01** and 10 cores on **system02**, which resulted in the highest performance in each case.

From each roofline characterization, it can be noticed that the nonoptimized routines were located under the DRAM memory roof. This indicates that the higher levels of memory (L1, L2, and L3 caches) were not frequently accessed, so that the routines required more accesses to the slower DRAM memory, which affected the overall performance. However, as all the original routines were also located under compute roofs, the compute optimizations (Intrinsics+OpenMP) were able to increase the performance, moving the routines closer to higher roofs.

Figure 6.1 - CPU roofline characterization of the STE, DIS, and CON routines for the Purus and Niger input datasets on **system01** and **system02**.



(a) Purus dataset on **system01**.

(b) Niger dataset on **system01**.

(c) Purus dataset on **system02**.

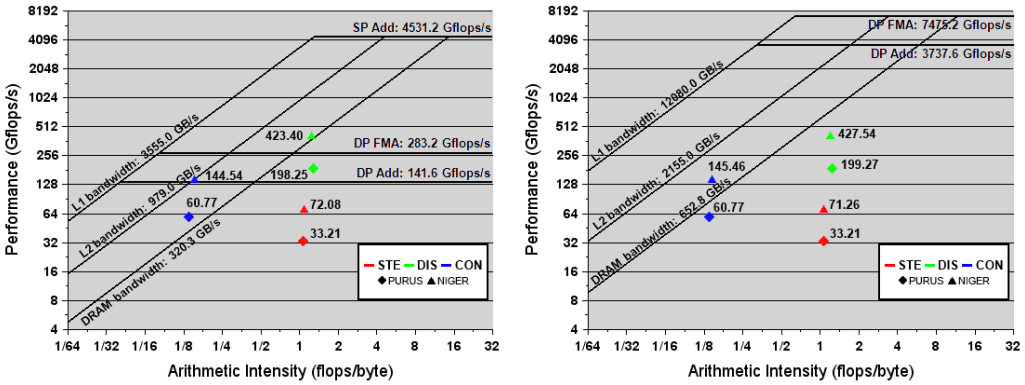
(d) Niger dataset on **system02**.

SOURCE: Author.

The addition of vectorization in the optimized codes decreased the amount of data that is loaded from memory, as more data is kept into the CPU registers, which are the fastest units of data storage in the CPU hardware. This optimization enabled to approximately double the arithmetic intensity of the DIS and CON routines.

Moreover, the additional use of thread-level parallelism moved the performance of those routines from under the DRAM roof closer to the cache roofs, in particular the L2 and L3 cache roofs, mainly due to the larger availability of L2 caches with the use of multiple cores. In contrast, the arithmetic intensity of the STE routine did not increase because the amount of bytes transferred through the memory hierarchy became larger in the optimized code, possibly due to the thread synchronization that is required to execute the minimum reduction operation. In summary, the arithmetic intensity changes between the original nonoptimized and optimized versions of each routine were: STE $= 0.58 \rightarrow 0.44$, DIS $= 0.05 \rightarrow 0.13$, and CON $= 0.06 \rightarrow 0.11$.

Figure 6.2 - GPU roofline characterization of the STE, DIS, and CON routines for the Purus and Niger input datasets on the Pascal and Volta GPUs.



(a) Purus and Niger datasets on Pascal GPU. (b) Purus and Niger datasets on Volta GPU.
SOURCE: Author.

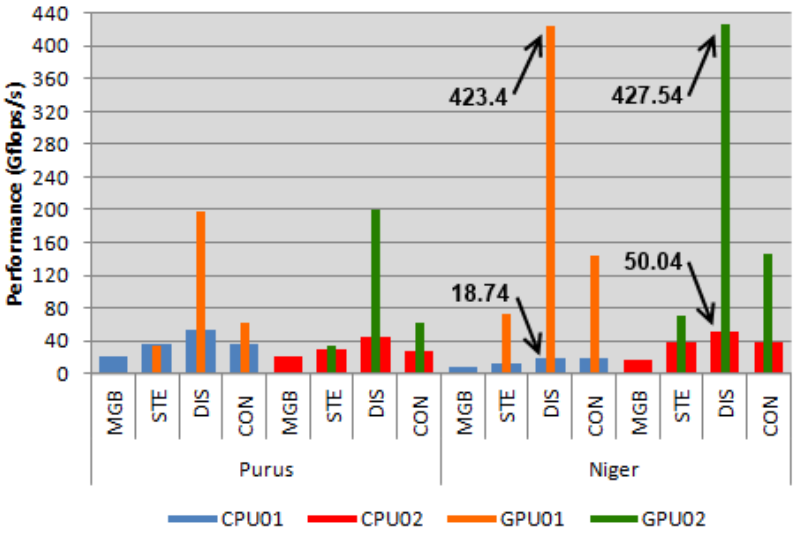
Besides the performance analysis with the CPU roofline model, the routines of the inertial model were also characterized with the GPU roofline models obtained from the NVIDIA GPUs used in this thesis (Pascal and Volta). Figure 6.2 exhibits the GPU roofline characterization of the optimized STE, DIS, and CON routines executed on GPUs obtained with the PAPI interface. For each routine, the arithmetic intensities on GPU are different from the corresponding ones on CPU due to the different hardware features between CPU and GPU architectures. As an example, the arithmetic intensity of each routine on GPU was: STE = 1.10, DIS = 1.31, and CON = 0.15.

The performance levels of the routines were nearly the same on both GPUs for each input dataset, showing that the performance improvements achieved with CUDA were independent of the processor and memory capabilities of the GPUs. Almost all cases from routine/dataset pairs remained under the DRAM roof (global memory) with the exception of the CON routine, as this routine more often executes the table search that keeps data in cache. Furthermore, the Niger dataset required more floating-point operations, resulting in higher performance when compared to the Purus dataset.

In summary, the performance results obtained from Intel Advisor (CPU) and from PAPI (GPU) are shown in Figure 6.3. The GPU performance of the MGB model was not computed because the performance counters were collected only for the routines executed on GPU as CUDA kernels, not the full MGB model. Figure 6.3 shows that the overall GPU performance was higher for the Niger dataset (more floating-point operations). The only case with similar CPU and GPU performance

levels was the STE routine for the Purus dataset, regardless of the computer system. The performance values highlighted in Figure 6.3 show the biggest performance differences between CPU and GPU that occurred with the DIS routine for the Niger dataset, as this routine executes the largest number of computations, and more effectively used the GPU computational resources, reaching more than 400 Gflops/s.

Figure 6.3 - Performance (Gflops/s) of the MGB model (CPU) and STE, DIS, and CON routines (CPU and GPU) for the Purus and Niger input datasets on **system01** and **system02**.



SOURCE: Author.

In addition, the CPU performance for the Purus dataset exhibited the same pattern on both systems, but the larger Niger dataset showed higher CPU performance on **system02**. This system provides larger L3 cache and more CPU cores that increases the availability of faster caches, reducing the number of DRAM memory accesses for larger workloads, which directly affected performance.

6.2 Accuracy of hydrological results

The development of optimized codes to boost the performance of scientific applications, which employ complex computational models, requires detailed code analysis to better understand how optimizations must be executed on the original nonoptimized implementation. More specifically, distinct low-level compiler instructions generated from nonoptimized and optimized versions of code can result in numerical variations in computations, introducing differences in final outputs of the application, and thus affecting its accuracy.

The use of explicit vectorization with Intel Intrinsics in the optimized CPU version of the MGB model revealed to be a time-consuming development task. The vectorized code demanded extensive tests to detect which vector instructions precisely simulated the original CPU MGB model's code (FREITAS et al., 2020b).

Small differences in computations throughout simulations, mainly due to how the Fortran language treats single-precision and double-precision constants, produced final outputs differing in the order of either hundreds or thousands between the nonoptimized and optimized implementations, as those differences were accumulated at each time step. Nevertheless, the optimized CPU version of the MGB model was carefully implemented to solve data type conversion issues, and final simulation outputs from both nonoptimized and optimized CPU versions matched exactly.

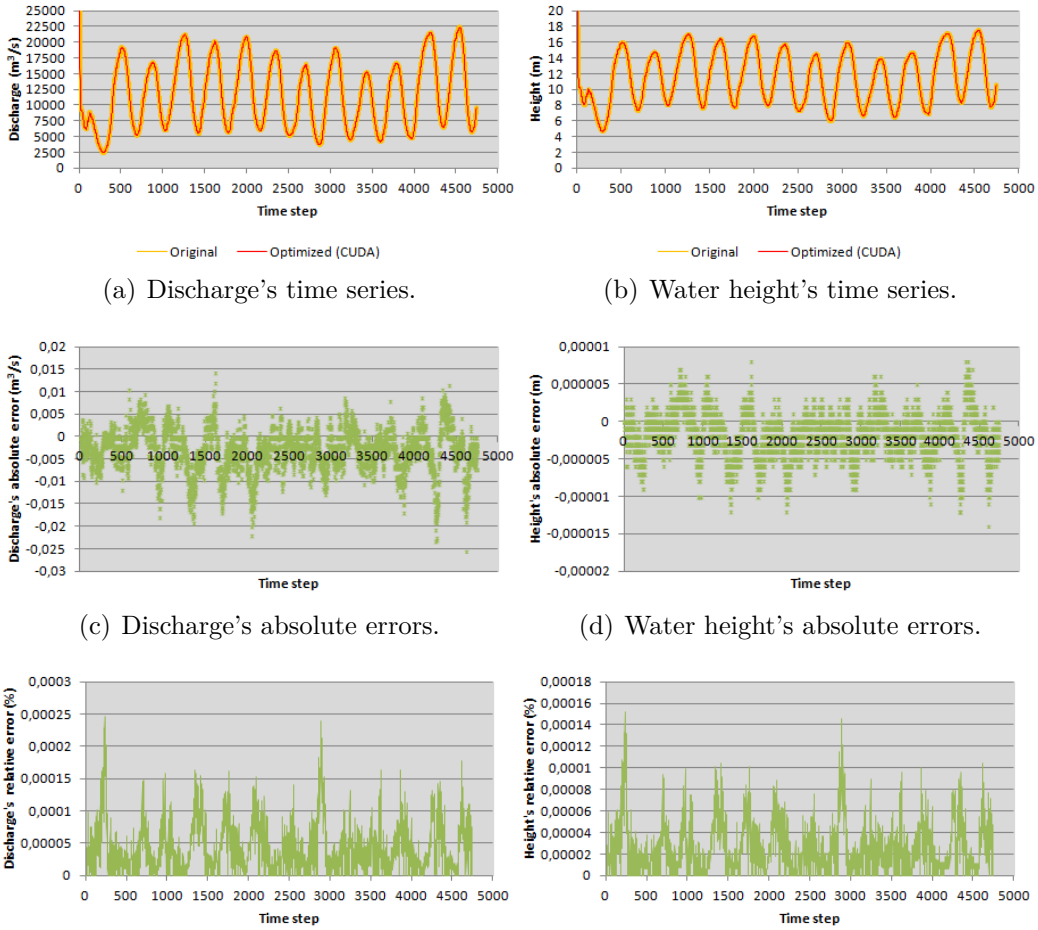
In contrast, the GPU MGB model's version optimized with CUDA did not produce the same simulation outputs when compared to the original nonoptimized CPU MGB model's code (reference). However, the results between the two versions showed only minor variations.

The implementations of the mathematical functions in the NVIDIA CUDA math library may be different from the ones available for CPU, and the corresponding results should not be expected to exactly match for a given input (NVIDIA, 2020b). Although various sets of compiler flags related to floating-point operations were tested with the PGI compiler on GPU, trying to simulate the Intel compiler's arithmetic behavior on CPU, all attempts were unsuccessful.

Figure 6.4 illustrates differences between main simulation outputs, i.e. discharge and water height, at the Purus watershed's outlet obtained from the original nonoptimized CPU and optimized GPU versions of the MGB model. The time series of simulated discharges and water heights were practically the same, showing that GPU simulation outputs almost exactly agreed with outputs produced on CPU. Mean and maximum relative errors for discharges were $4.18 \times 10^{-5}\,\%$ and $2.46 \times 10^{-4}\,\%$, respectively, whereas for water heights were $2.51 \times 10^{-5}\,\%$ and $1.53 \times 10^{-4}\,\%$.

However, for the Niger dataset, simulation outputs presented lower accuracy, as shown in Figure 6.5. Visually, the time series of simulated discharges and water heights were nearly identical, but the quantitative error analysis showed larger absolute and relative errors, mainly for discharge values, reaching mean and maximum relative errors of 0.59 % and 6.06 % for discharges, whereas 0.36 % and 3.60 % for water heights.

Figure 6.4 - Time series, absolute and relative errors of discharge and water height for the Purus dataset from simulations on GPU, considering CPU outputs as reference.



(a) Discharge's time series.

(b) Water height's time series.

(c) Discharge's absolute errors.

(d) Water height's absolute errors.

(e) Discharge's relative errors.

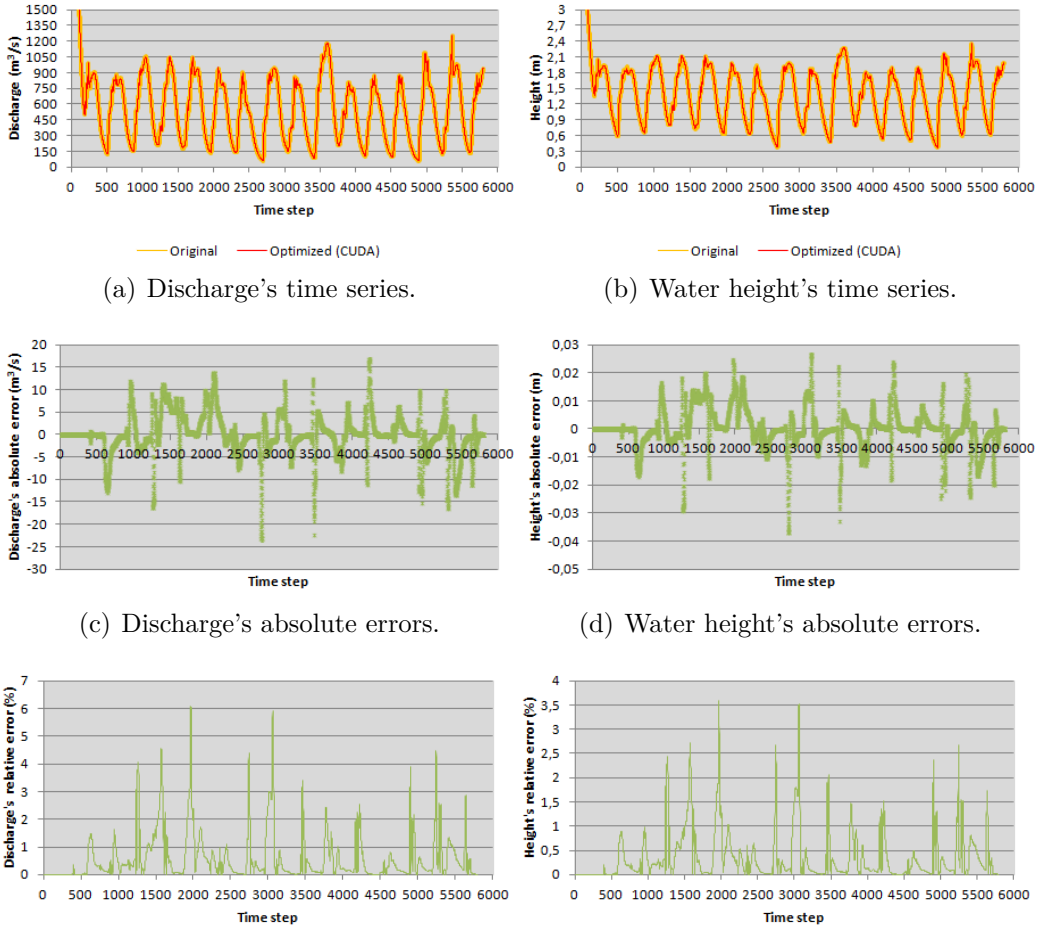
(f) Water height's relative errors.

SOURCE: Author.

Thus, the Niger dataset was more sensitive to differences from intermediate computations of the MGB model. In addition, it is worth mentioning that the first values of all time series are large because initial and boundary conditions are set to default values.

For the analysis of calibration results obtained from the optimized implementations, the accuracy was measured by comparing minimum values of the objective functions from each type of optimization with minimum values from the original nonoptimized implementation. As samples processed in the calibration procedure are selected based on a random probability value, calibration results of the optimized implementations are expected to show differences relative to original results (reference).

Figure 6.5 - Time series, absolute and relative errors of discharge and water height for the Niger dataset from simulations on GPU, considering CPU outputs as reference.



(a) Discharge's time series.

(b) Water height's time series.

(c) Discharge's absolute errors.

(d) Water height's absolute errors.

(e) Discharge's relative errors.

(f) Water height's relative errors.

SOURCE: Author.

Table 6.1 exhibits minimum values of the objective functions and relative errors for each type of optimization in comparison to results for the type `V0-0` (original nonoptimized implementation). All values were computed as the average between the results from the calibration procedure executed on **system01** and **system02**, except the results for the type `V0-10`, which were available only from **system02**.

The largest relative errors occurred for the type `V2` on CPU, reaching 5.97 %, 3.24 %, and 1.52 % for the coefficients NSE and NSE_{log}, and the systematic error ERR, respectively. Therefore, these small errors indicate that calibration results were acceptable and validated the optimized implementations.

Table 6.1 - Minimum values (MIN) and relative errors (RE) of the objective functions used in the calibration procedure for each type of optimization.

Type		Objective function	NSE		NSE_{log}		ERR	
			MIN	RE (%)	MIN	RE (%)	MIN	RE (%)
Type	CPU	V0-0	0.1866	-	0.1172	-	17.1676	-
		V0-1	0.1886	1.09	0.1172	1.11	17.1974	0.43
		V0-2	0.1884	0.99	0.1172	0.73	17.2409	0.29
		V0-4	0.1875	1.05	0.1180	0.51	17.1987	0.10
		V0-8	0.1864	0.13	0.1178	0.47	17.1507	0.15
		V0-10	0.1921	2.40	0.1177	0.77	17.3324	1.06
		V1	0.1895	1.56	0.1187	1.32	17.1129	0.40
		V2	0.1977	5.97	0.1210	3.24	17.4277	1.52
	GPU	V0	0.1896	1.64	0.1186	1.20	17.1359	0.18
		V1	0.1901	1.88	0.1194	1.87	17.0583	0.64
		V2	0.1976	5.93	0.1188	1.67	17.3118	0.84

7 CONCLUSION

This thesis presented an investigation of performance optimization techniques applied on the MGB hydrological model and the MOCOM-UA calibration method, widely used by the hydrology community in Brazil. Different CPU and GPU optimizations improved the performance of the calibration procedure, increasing the usefulness in hydrological applications by reducing execution times from more than one week to only a few hours.

Vectorization via Intel Intrinsics and thread parallelism with OpenMP on CPU, and also data parallelism with CUDA on GPU, were employed to execute optimized simulation and calibration codes based on a double-layer approach. Experiments were conducted on state-of-the-art CPU+GPU systems available at INESC-ID (Portugal) for real-world input datasets (Purus and Niger rivers).

The vectorization of the key routines of the MGB model achieved significant speedups, considering that approximately two-thirds of code was vectorized, according to the Intel VTune's analysis. Moreover, the CPU speedups of the fully optimized MGB model (vectorization and thread parallelism) were close to the theoretical peak from Amdahl's law, or even higher due to cache effects of multi-core processing.

This level of performance would not be reached by simple vectorization via compiler flags. Even with modern compilers, the effective use of vector capabilities in current processors requires careful code reorganization. This reorganization makes the application code suitable to the use of a library with intrinsic vector functions, which maximize the exploitation of those vector resources inside the processors.

The runtimes of all routines of the inertial model were reduced to less than 3 s with the CUDA optimizations for GPU, but the MGB model's performance was hindered by overheads of data transfer between CPU and GPU memories and of kernel launches (too many routine calls). The workloads of the available input datasets are not GPU-friendly, in the sense that there was not enough work to keep the GPU functional units busy for extended periods of time, and thus not providing good scalability when executed on GPU, so the CPU optimizations resulted in a more appropriate solution. In addition, the performance of the calibration procedure was also significantly improved, reaching speedups close to 20× on CPU with a proposed approach that proved to be an efficient solution, as the parallelization of stochastic optimization algorithms may be trivial, where many candidate solutions can be evaluated in an independent manner in parallel.

The scalability analysis with the MGB model's miniapp provided additional insights on how the MGB model's performance scaled with problem size, indicating that the workloads of the available datasets were not enough for good scalability. Speedups of approximately 24× and 65× were achieved on CPU and GPU, respectively, for problem sizes of at least 8× larger than the original workloads. This analysis shows that the proposed optimizations enable effective MGB model's use with real-world watersheds containing many more catchments than the ones originally employed in the experiments.

The roofline analysis is a novel technique in the context of performance analysis of hydrological models. The CPU roofline analysis presented in this thesis confirmed that the proposed optimizations more effectively exploited the available CPU hardware resources, moving routines from under the slower DRAM roof to under faster cache roofs. Furthermore, the GPU roofline characterization showed that executing the MGB model on NVIDIA GPUs (Pascal and Volta) resulted in different values of arithmetic intensity and performance when compared to CPU. In particular, one of the routines achieved performance almost 8× higher than on CPU, as the CUDA kernels were significantly faster than the optimized CPU codes.

A quantitative analysis of hydrological results showed that the time series of discharges and water heights presented small relative numerical errors, where the largest error reached 6.06 % in a particular position of the time series of discharges for one of the datasets, although in this case the mean relative error was only 0.59 %. The comparison between minimum objective functions from the nonoptimized and optimized calibration implementations also resulted in acceptable accuracy. The largest relative error of one objective function was 5.97 %, which validated the optimized implementations.

Overall, this investigation proved that modern, state-of-the-art computer systems can be effectively exploited for legacy applications originally written for sequential processing. By carefully deploying appropriate parallelization techniques, it is possible to adjust and adapt those applications to the advanced capabilities of currently available hardware. Despite being a labor-intensive effort in some cases, those optimizations provide, in return, significant performance benefits that may justify the investments applied for acquisition of a modern computing system.

7.1 Potential future work

As possible routes for future work, the roofline analysis can be extended to the MGB model's miniapp for deeper understanding of how optimizations exploit hardware resources for larger problem sizes. Furthermore, CPU and GPU roofline characterizations of the optimized implementations can be further analyzed, so new insights about the optimized MGB model's behavior could provide more opportunities for additional optimizations and, consequently, more gains in performance.

The OpenCL framework can also be used to provide more optimized implementations, so that new experiments could be conducted on CPU, GPU, and also emerging CPU+GPU architectures that combine CPU and GPU in a single chip, as these hybrid architectures cannot be explored by using the CUDA framework. More importantly, despite expecting differences in performance, employing OpenCL would make the codes portable to a wide range of devices, including multi-core CPUs, GPUs from distinct vendors, and also FPGAs.

The estimation of high-quality sets of model parameters can also be extended by processing multiple independent calibration procedures on different nodes of a supercomputer, each with its own initial sets of parameters. This could provide more possibilities of finding the optimal sets of parameters for the MGB model, covering a wider range in the multi-dimensional search space and potentially expanding the calibration results to multiple Pareto fronts, which can all be further compared and analyzed for the selection of the optimal sets of model parameters.

The CPU and GPU optimized implementations of the MGB model, as presented in this thesis, can be employed in another version of the model that simulates hydrological processes for continental-scale datasets. The IPH research group is currently working with datasets that cover the entire South America, which can be approximately 8× larger than the Purus and Niger datasets. The performance of this large application is expected to be largely improved, as shown by the scalability analysis that considered problem sizes of up to 32× the original size.

In addition, a detailed numerical analysis of the difference equations obtained from the numerical scheme of the inertial model can also be considered in order to expand the knowledge about its stability and convergence conditions. Such numerical analysis was intentionally left outside the scope of this thesis, as the thesis focused on improving the MGB model's performance under the current modeling scheme that is already in use by hydrologists.

REFERENCES

ARAÚJO, A. S.; VELHO, H. F. C.; GOMES, V. C. F. Calibrating an hydrological model by evolutionary strategy for multi-objective optimization. **Inverse Problems in Science and Engineering**, v. 21, n. 3, p. 438–450, Apr. 2013. 27

ARNOLD, J. G.; SRINIVASAN, R.; MUTTIAH, R. S.; WILLIAMS, J. R. Large area hydrologic modeling and assessment part I: model development. **Journal of the American Water Resources Association**, v. 34, n. 1, p. 73–89, Feb. 1998. ISSN 1752-1688. 18

BATES, P.; HORRIT, M. S.; ; FEWTRELL, T. J. A simple inertial formulation of the shallow water equations for efficient two-dimensional flood inundation modelling. **Journal of Hydrology**, v. 387, n. 1–2, p. 33–45, June 2010. 18, 24

CHAPMAN, B.; JOST, G.; PAS, R. van der. **Using OpenMP - portable shared memory parallel programming**. Cambridge, MA, USA: MIT Press, 2008. ISBN 978-0-262-53302-7. 11

COLLISCHONN, W. Thesis (PhD in Engineering), **Simulação hidrológica de grandes bacias**. Porto Alegre, RS: URI:10183/2500, Dec. 2001. 181 p. 3, 21, 25, 39

CUNGE, J. A.; HOLLY, F. M.; VERWEY, A. **Practical aspects of computational river hydraulics**. Boston: Pitman Advanced Publishing Program, 1980. ISBN 0273084429. 23

DAGUM, L.; MENON, R. OpenMP: an industry standard API for shared-memory programming. **IEEE Computational Science and Engineering**, v. 5, n. 1, p. 46–55, Jan. 1998. 2, 9

DONGARRA, J.; LONDON, K.; MOORE, S.; MUCCI, P.; TERPSTRA, D. Using PAPI for hardware performance monitoring on Linux systems. In: **CONFERENCE ON LINUX CLUSTERS: THE HPC REVOLUTION**. Urbana, IL, USA: Proceedings... Linux Clusters Institute, 2001. v. 5. 17

DUAN, Q. Y.; GUPTA, V. K.; SOROOSHIAN, S. Shuffled complex evolution approach for effective and efficient global minimization. **Journal of Optimization Theory and Applications**, v. 76, p. 501–521, Mar. 1993. 18, 27

EFSTRATIADIS, A.; KOUTSOYIANNIS, D. One decade of multi-objective calibration approaches in hydrological modelling: a review. **Hydrological Sciences Journal**, v. 55, n. 1, p. 58–78, Aug. 2010. 3

FAGUNDES, H. O.; FAN, F. M.; PAIVA, R. C. D. Automatic calibration of a large-scale sediment model using suspended sediment concentration, water quality, and remote sensing data. **Brazilian Journal of Water Resources**, v. 24, n. 26, p. 1–18, Mar. 2019. ISSN 2318-0331. 27

FAGUNDES, H. O.; PAIVA, R. C. D.; FAN, F. M.; BUARQUE, D. C.; FASSONI-ANDRADE, A. C. Sediment modeling of a large-scale basin supported by remote sensing and in-situ observations. **CATENA**, v. 190, p. 1–13, Mar. 2020. ISSN 0341-8162. 3, 23

FAN, F. M.; PONTES, P. R. M.; PAIVA, R. C. D.; COLLISCHONN, W. Avaliação de um método de propagação de cheias em rios com aproximação inercial das equações de Saint-Venant. **Revista Brasileira de Recursos Hídricos**, v. 19, n. 4, p. 137–147, Oct. 2014. 3, 21, 23, 24

FARR, T. G.; ROSEN, P. A.; CARO, E.; CRIPPEN, R.; DUREN, R.; HENSLEY, S.; KOBRICK, M.; PALLER, M.; RODRIGUEZ, E.; ROTH, L.; SEAL, D.; SHAFFER, S.; SHIMADA, J.; UMLAND, J.; WERNER, M.; OSKIN, M.; BURBANK, D.; ALSDORF, D. The Shuttle Radar Topography Mission. **Reviews of Geophysics**, v. 45, n. 2, p. 1–33, May 2007. 46

FLEISCHMANN, A.; COLLISCHONN, W.; PAIVA, R. C. D.; TUCCI, C. E. Modeling the role of reservoirs versus floodplains on large-scale river hydrodynamics. **Natural Hazards**, v. 99, p. 1075–1104, Sept. 2019. 3, 23

FLEISCHMANN, A.; SIQUEIRA, V.; PARIS, A.; COLLISCHONN, W.; PAIVA, R. C. D.; PONTES, P. R. M.; BIANCAMARA, S.; GOSSET, M.; CALMANT, S. Representando interações entre hidrologia e hidrodinâmica em modelos de grande escala: estudo de caso no rio Níger, África. In: **SIMPÓSIO BRASILEIRO DE RECURSOS HÍDRICOS (SBRH), 22.** Florianópolis, SC, Brasil: ANAIS... ASSOCIAÇÃO BRASILEIRA DE RECURSOS HÍDRICOS (ABRH), 2017. 3, 23, 46

FREITAS, H. R. A.; MENDES, C. L. Roofline analysis and performance optimization of the MGB hydrological model. In: **SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (WSCAD), 20.** Campo Grande, MS, Brasil: ANAIS... SOCIEDADE BRASILEIRA DE COMPUTAÇÃO (SBC), 2019. 37

FREITAS, H. R. A.; MENDES, C. L.; ILIC, A. Performance optimization and scalability analysis of the MGB hydrological model. In: **IEEE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE**

COMPUTING, DATA, AND ANALYTICS, 27. India: Proceedings... HiPC, 2020. p. 31–40. ISBN 978-0-7381-1035-6. 37

_____. Performance optimization of a watershed parameter calibration method using the MGB hydrological model for CPU/GPU systems. Dec. 2020. Submitted for publication. 63

GERITANA, A. C. V.; BOONE, A.; PEUGEOT, C. Evaluating LSM-based water budgets over a west african basin assisted with a river routing scheme. **Journal of Hydrometeorology**, v. 15, n. 6, p. 2331–2346, Dec. 2014. 27

GORGOGLIONE, A.; CRISCI, M.; KAYSER, R. H.; CHRETIES, C.; COLLISCHONN, W. A new scenario-based framework for conflict resolution in water allocation in transboundary watersheds. **Water**, v. 11, n. 6, p. 1–24, June 2019. 3, 23

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI: portable parallel programming with the Message-Passing Interface**. 3. ed. Cambridge, MA, USA: MIT Press, 2014. 2

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture: a quantitative approach**. 4. ed. San Francisco, CA, USA: Morgan Kaufmann, 2007. 1

HILL, M. D.; MARTY, M. R. Amdahl's law in the multicore era. **IEEE Computer**, v. 41, n. 7, p. 33–38, July 2008. 11

HOFFMAN, J. D. **Numerical methods for engineers and scientists**. 2. ed. New York, NY, USA: Marcel Dekker, 2001. ISBN 0-8247-0443-6. 32

ILIC, A.; PRATAS, F.; SOUSA, L. Cache-aware roofline model: upgrading the loft. **IEEE Computer Architecture Letters**, v. 13, n. 1, p. 21–24, Apr. 2013. 2, 16, 59

INTEL. **Intel 64 and IA-32 architectures optimization reference manual**. Santa Clara, CA, USA, Sept. 2019. Available from: <https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual>. 7, 8, 32

_____. **Analyze applications for improved performance**. 2020. Online. Available from: <https://software.intel.com/content/www/us/en/develop/tools/parallel-studio-xe/analyze.html>. 45, 48, 59

ISLAM, S. U.; DÉRY, S. J. Evaluating uncertainties in modelling the snow hydrology of the Fraser River Basin, British Columbia, Canada. **Hydrology and Earth System Sciences**, v. 21, p. 1827–1847, Mar. 2017. 27

JANG, B.; SCHAA, D.; MISTRY, P.; KAELI, D. Exploiting memory access patterns to improve memory performance in data-parallel architectures. **IEEE Transactions on Parallel and Distributed Systems**, v. 22, n. 1, p. 105–118, Jan. 2011. 10

JARDIM, A. C. Thesis (PhD in Applied Computing), **Direções de fluxo em modelos digitais de elevação: um método com foco na qualidade da estimativa e processamento de grande volume de dados**. São José dos Campos, SP: sid.inpe.br/mtc-m21b/2017/05.17.13.26-TDI, Apr. 2017. 109 p. 22

JIA, H.; ZHANG, Y.; LONG, G.; XU, J.; YAN, S.; LI, Y. GPURoofline: a model for guiding performance optimizations on GPUs. In: **INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 18.** St. Charles, IL, USA: Proceedings... Springer, 2012. p. 920–932. 16, 17

JIA, Z.; MAGGIONI, M.; STAIGER, B.; SCARPAZZA, D. P. **Dissecting the NVIDIA Volta GPU architecture via microbenchmarking**. Chicago, IL, USA, April 2018. 45

KALCIC, M. M.; CHAUBEY, I.; FRANKENBERGER, J. Defining Soil and Water Assessment Tool (SWAT) hydrologic response units (HRUs) by field boundaries. **International Journal of Agricultural and Biological Engineering**, v. 8, n. 3, p. 69–80, June 2015. 22

KAN, G.; HE, X.; DING, L.; LI, J.; HONG, Y.; ZUO, D.; REN, M.; LEI, T.; LIANG, K. Fast hydrological model calibration based on the heterogeneous parallel computing accelerated shuffled complex evolution method. **Engineering Optimization**, v. 50, n. 1, p. 106–119, Apr. 2018. ISSN 1029-0273. 18

KIM, K.; KIM, K.; PARK, Q. Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model. **Computer Physics Communications**, v. 182, n. 6, p. 1201–1207, June 2011. 16, 17

KIRK, D. B.; HWU, W. W. **Programming massively parallel processors: a hands-on approach**. San Francisco, CA, USA: Morgan Kaufmann, 2017. ISBN 978-0-12-811986-0. 12

LIN, P. T.; HEROUX, M. A.; BARRET, R. F.; WILLIAMS, A. Assessing a mini-application as a performance proxy for a finite element method engineering application. **Concurrency and Computation: Practice and Experience**, v. 27, n. 17, p. 5374–5389, July 2015. 4

LIU, J.; ZHU, A. X.; LIU, Y.; ZHU, T.; QIN, C. Z. A layered approach to parallel computing for spatially distributed hydrological modeling. **Environmental Modelling & Software**, v. 51, p. 221–227, Jan. 2014. ISSN 1364-8152. 18

LOPES, A. F. A. Thesis (Master in Electrical and Computer Engineering), **Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling**. Lisboa, Portugal: INESC-ID:11224, Nov. 2016. 63 p. 2, 13, 16, 17

MAHÉ, G.; BAMBA, F.; SOUMAGUEL, A.; ORANGE, D.; OLIVRY, J. C. Water losses in the inner delta of the River Niger: water balance and flooded area. **Hydrological Processes**, v. 23, n. 22, p. 3157–3160, Oct. 2009. 46

MARQUES, D.; ILIC, A.; MATVEEV, Z. A.; SOUSA, L. Application-driven cache-aware roofline model. **Future Generation Computer Systems**, v. 107, p. 257–273, June 2020. 2, 16, 59

MEUER, H. W.; STROHMAIER, E.; DONGARRA, J.; SIMON, H.; MEUER, M. **TOP500**. 2021. Available from: <https://www.top500.org/>. 1

MITRA, G.; JOHNSTON, B.; RENDELL, A. P.; MCCREATH, E.; ZHOU, J. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. In: **IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED PROCESSING, WORKSHOPS, AND PHD FORUM**. Cambridge, MA, USA: Proceedings... IEEE, 2013. p. 1107–1116. 8

MURAI, H.; NAKAO, M.; IWASHITA, H.; SATO, M. Preliminary performance evaluation of coarray-based implementation of fiber miniapp suite using XcalableMP PGAS Language. In: ASSOCIATION FOR COMPUTING MACHINERY (ACM). **ANNUAL PGAS APPLICATIONS WORKSHOP, 2.** Denver, CO, USA: Proceedings... PAW17, 2017. p. 1–7. 4

NASH, J. E.; SUTCLIFFE, J. V. River flow forecasting through conceptual models part I - a discussion of principles. **Journal of Hydrology**, v. 10, n. 3, p. 282–290, Apr. 1970. 29

NEAL, J. C.; FEWTRELL, T. J.; BATES, P. D.; WRIGHT, N. G. A comparison of three parallelisation methods for 2D flood inundation models. **Environmental Modelling & Software**, v. 25, n. 4, p. 398–411, Apr. 2010. ISSN 1364-8152. 18

NVIDIA. **CUDA API reference manual**. 4.2. ed. Santa Clara, CA, USA, Apr. 2012. 2, 14

_____. **CUDA Fortran programming guide and reference**. Santa Clara, CA, USA, 2019. 12

_____. **Parallel thread execution ISA**. Santa Clara, CA, USA, Nov. 2019. Available from: <https://docs.nvidia.com/cuda/pdf/ptx_isa_6.5.pdf>. 14

_____. **CUDA Fortran programming guide**. Santa Clara, CA, USA, Nov. 2020. Available from: <https://docs.nvidia.com/hpc-sdk/compilers/pdf/hpc2011cudaforug.pdf>. 37

_____. **Precision and performance: floating point and IEEE 754 compliance for NVIDIA GPUs**. Santa Clara, CA, USA, Oct. 2020. White Paper. Available from: <https://docs.nvidia.com/cuda/pdf/Floating_Point_on_NVIDIA_GPU.pdf>. 63

PACHECO, P. S. **An introduction to parallel programming**. Burlington, MA, USA: Morgan Kaufmann, 2011. 1

PAIVA, R. C. D.; COLLISCHONN, W.; TUCCI, C. E. M. Large scale hydrologic and hydrodynamic modeling using limited data and a GIS based approach. **Journal of Hydrology**, v. 406, n. 3–4, p. 170–181, Sept. 2011. 2, 3, 22, 23, 46

PGI. **CUDA Fortran programming guide and reference**. Santa Clara, CA, USA, 2020. Available from: <https://www.pgroup.com/doc/pgicudaforug.pdf>. 37

QUINN, M. J. **Parallel programming in C with MPI and OpenMP**. New York, NY, USA: McGrawHill, 2004. ISBN 007-282256-2. 11

RALSTON, C. E. Making sense of multi-core processor technology for SAS environment. In: JANSEN, L. (Ed.). **SAS global forum 2008 (systems architecture)**. San Antonio, TX, USA, 2008. 1

ROSIM, S. Thesis (PhD in Applied Computing), **Estrutura baseada em grafos para representação unificada de fluxos locais para modelagem hidrológica distribuída**. São José dos Campos, SP: INPE-15320-TDI/1363, May 2008. 107 p. 22

ROUHOLAHNEJAD, E.; ABBASPOUR, K. C.; VEJDANI, M.; SRINIVASAN, R.; SCHULIN, R.; LEHMANN, A. A parallelization framework for calibration of hydrological models. **Environmental Modelling & Software**, v. 31, p. 28–36, May 2012. 19

RUPP, K. **CPU, GPU and MIC hardware characteristics over time**. Aug. 2016. Online. Available from: <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>. 12

SARATES, A. S. Thesis (Master in Computer Science), **Optimizing two-dimensional shallow water based flood hydrological model with stream architectures**. Porto Alegre, RS: URI:10183/118254, 2015. 79 p. 18

SHUTTLEWORTH, W. J. Evaporation. In: MAIDMENT, D. R. (Ed.). **Handbook of hydrology**. New York, NY, USA: McGraw-Hill Education, 1993. chapter 4, p. 1–53. 25

SORRIBAS, M. V.; BRAVO, J. M.; COLLISCHONN, W. Avaliação preliminar do algoritmo de otimização multi-objetivo MOSCEM-UA aplicado na calibração do modelo hidrológico MGB-IPH. In: **BRAZILIAN SYMPOSIUM OF WATER RESOURCES (SBRH), 20.** Bento Gonçalves, RS: Proceedings... Brazilian Association of Water Resources (ABRH), 2013. 29

STONE, A.; DENNIS, J.; STROUT, M. Establishing a miniapp as a programmability proxy. **ACM SIGPLAN Notices**, v. 47, n. 8, p. 333–334, Feb. 2012. 4

TATSUMI, K. Effects of automatic multi-objective optimization of crop models on corn yield reproducibility in the U.S.A. **Ecological Modelling**, v. 322, p. 124–137, 2016. ISSN 0304-3800. 27

TESEMMA, Z. K.; WEI, Y.; PEEL, M. C.; WESTERN, A. W. The effect of year-to-year variability of leaf area index on Variable Infiltration Capacity model performance and simulation of runoff. **Advances in Water Resources**, v. 83, p. 310–322, July 2015. ISSN 0309-1708. 27

TIAN, F.; HU, H.; SUN, Y.; LI, H.; LU, H. Searching for an optimized single-objective function matching multiple objectives with automatic calibration of hydrological models. **Chinese Geographical Science**, v. 29, n. 6, p. 934–948, July 2019. ISSN 1002-0063. 27

TOMPSON, J.; SCHLACHTER, K. **Introduction to the OpenCL programming model**. 2012. 12

TUCCI, C. E. M. **Modelos hidrológicos**. 2. ed. Porto Alegre, RS, Brasil: Associação Brasileira de Recursos Hídricos (ABRH), 2005. 27

VIVONI, E. R.; MASCARO, G.; MNISZEWSKI, S.; FASEL, P.; SPRINGER, E. P.; IVANOV, V. Y.; BRAS, R. L. Real-world hydrologic assessment of a fully-distributed hydrological model in a parallel computing environment. **Journal of Hydrology**, v. 409, n. 1–2, p. 483–496, Oct. 2011. ISSN 0022-1694. 18

VRUGT, J. A.; GUPTA, H. V.; BASTIDAS, L. A.; BOUTEN, W.; SOROOSHIAN, S. Effective and efficient algorithm for multiobjective optimization of hydrologic models. **Water Resources Research**, v. 39, n. 8, p. 1214–1232, Aug. 2003. 18, 27, 28

WANG, G.; WU, B.; LI, T. Digital Yellow river model. **Journal of Hydro-environment Research**, v. 1, n. 1, p. 1–11, Sept. 2007. 18

WIENKE, S.; SPRINGER, P.; TERBOVEN, C.; MEY, D. an. OpenACC - first experiences with real-world applications. In: **INTERNATIONAL EUROPEAN CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING, 18.** Rhodes Island, Greece: Proceedings... EURO-PAR 2012. Springer Verlag Berlin/Heidelberg, 2012. (Lecture Notes in Computer Science, v. 7484), p. 859–870. 12

WILLIAMS, S.; WATERMAN, A.; PATTERSON, D. Roofline: an insightful visual performance model for multicore architectures. **Communications of the ACM**, v. 52, n. 4, p. 65–76, Apr. 2009. 15, 17

WILLIAMS, S. W. Thesis (PhD in Electrical Engineering and Computer Sciences), **Auto-tuning performance on multicore computers**. Berkeley, CA, USA: UCB/EECS-2008-164, Dec. 2008. 218 p. 15

WU, Y.; LI, T.; SUN, L.; CHEN, J. Parallelization of a hydrological model using the message passing interface. **Environmental Modelling & Software**, v. 43, p. 124–132, 2013. ISSN 1364-8152. 18

YANG, C.; KURTH, T.; WILLIAMS, S. Hierarchical roofline analysis for GPUs: accelerating performance optimization for the NERSC-9 perlmutter system. **Concurrency and Computation: Practice and Experience**, v. 32, n. 20, p. 1–12, Nov. 2019. 16, 17

YAPO, P. O.; GUPTA, H. V.; SOROOSHIAN, S. Multi-objective global optimization for hydrologic models. **Journal of Hydrology**, v. 204, n. 1–4, p. 83–97, Jan. 1998. 27

ZHANG, A.; LI, T.; SI, Y.; LIU, R.; SHI, H.; LI, X.; LI, J.; WU, X. Double-layer parallelization for hydrological model calibration on HPC systems. **Journal of Hydrology**, v. 535, p. 737–747, Apr. 2016. 18

ZHANG, X.; BEESON, P.; LINK, R.; MANOWITZ, D.; IZAURRALDE, R. C.; SADEGHI, A.; M.THOMSON, A.; SAHAJPAL, R.; SRINIVASAN, R.; ARNOLD, J. G. Efficient multi-objective calibration of a computationally intensive hydrologic model with parallel computing software in Python. **Environmental Modelling & Software**, v. 46, p. 208–218, Aug. 2013. 19

ZHANG, X.; IZAURRALDE, R. C.; ZONG, Z.; ZHAO, K.; THOMSON, A. M. Evaluating the efficiency of a multi-core aware multi-objective optimization tool for calibrating the SWAT model. **Transactions of the ASABE**, v. 55, n. 5, p. 1723–1731, 2012. ISSN 2151-0032. 19

ZHAO, R. J. The Xinanjiang model applied in China. **Journal of Hydrology**, v. 135, n. 1–4, p. 371–381, July 1992. 18

ZWARTS, L.; BEUKERING, P. van; BAKARY, K.; WYMENGA, E. **The Niger, a lifeline - effective water management in the upper Niger basin**. Veenwouden, NL: Altenburg & Wymenga, 2005. 46