

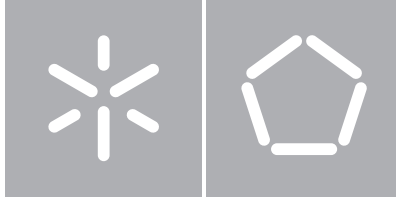


Universidade do Minho

Escola de Engenharia

Miguel Branco Palhas

**An Evaluation of the GAMA Framework for
Heterogeneous Platforms:
The Progressive Photon Mapping Algorithm**



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Miguel Branco Palhas

**An Evaluation of the GAMA Framework for
Heterogeneous Platforms:
The Progressive Photon Mapping Algorithm**

Dissertação de Mestrado

Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Alberto Proença

Professor Luis Paulo Santos

Abstract

High performance computing has suffered several changes in recent years, with the increasingly prominent evolution and usage of heterogeneous platforms: multiple devices with different architectures, characteristics, and programming models share application workload targetting an increased performance.

Several frameworks have been under development, to aid the programmer to efficiently explore these platforms, by dynamically scheduling the workload across the available resources and dealing with the inherent difficulties that come associated with the increased complexity of the system.

One of these is the GAMA framework, designed to deal with irregular applications and addressing task scheduling and resources management. GAMA aims to unify the multiple execution and memory models of each device into a single, hardware agnostic model. It handles the workload distribution across multiple computational resources, and attempts to balance them based on a scheduling policy as well as runtime information. So far, this framework has only been tested with benchmark kernels, for comparison with similar alternatives using standard algorithms, but it still lacks a deeper study, using a more realistic case study, to better understand its potential, and limitations.

The subject of this dissertation is to provide a more in-depth evaluation of GAMA, using the progressive photon mapping irregular algorithm as a case study. The goal is to validate its effectiveness with a robust irregular application, where resource management problems are more easily seen, and identify possible points of improvement for this project.

Resumo

Uma avaliação da framework GAMA para Plataformas Heterogéneas: O algoritmo de Progressive Photon Mapping

A área de computação de alto desempenho foi alvo de uma grande evolução nos últimos anos, com a cada vez maior utilização de plataformas heterogéneas, nas quais são utilizados vários dispositivos que diferem não só na arquitectura, mas também nas características, e no modelo de programação, trabalho em colaboração partilhando o conjunto de tarefas de uma aplicação com o objectivo final de melhorar o desempenho global.

Várias *frameworks* têm sido desenvolvidas com o intuito de facilitar a programação direccionada a estas plataformas, através de escalonamento dinâmico da carga de trabalho pelos dispositivos disponíveis, e lidando com os pormenores e dificuldades inerentes da utilização destes sistemas de maior complexidade.

Uma destas frameworks é o GAMA, desenhado para lidar com aplicações irregulares, nas quais o escalonamento de tarefas e a gestão de recursos é uma tarefa mais complicada. O GAMA propõe um modelo unificado de programação e de memória, agnóstico ao modelo usado internamente por cada dispositivo. A framework gere a carga de trabalho das várias unidades de computação, e tenta balanceá-la com base numa política de escalonamento e em informação obtida automaticamente em tempo de execução. Até agora, esta framework apenas foi testada para um pequeno conjunto de *kernels* típicos de *benchmarking*, para comparação com alternativas idênticas através de resultados baseados em algoritmos standardizados para o efeito. Mas necessita ainda de um estudo mais intensivo, usando um caso de estudo mais realista e robusto, para melhor entender as suas potencialidades, e as suas limitações.

Esta dissertação propõe-se a disponibilizar uma avaliação mais intensiva do GAMA, usando o algoritmo irregular de *Progressive Photon Mapping* como caso de estudo. O principal objectivo é validar a eficácia da framework para uma aplicação robusta, onde os problemas de gestão de recursos são mais evidentes, e identificar possíveis possibilidades para este projecto.

Contents

1	Introduction	4
1.1	Contextualization	4
1.2	Motivation & Goals	4
1.3	Case Study	5
1.4	Document structure	5
2	Technological Background	5
2.1	Parallel Computing	5
2.2	The Graphics Processing Unit (GPU) as a Computing Accelerator	6
2.2.1	Fermi Architecture	6
2.2.2	Kepler Architecture	7
2.3	Heterogeneous Platforms	8
2.3.1	A note on debugging	9
2.3.2	Scheduling	9
2.4	Software	9
2.4.1	OpenMP	10
2.4.2	CUDA	10
2.4.3	MPI	10
2.4.4	OpenACC	10
3	The GAMA Framework	11
3.1	Memory Model	11
3.1.1	Memory Consistency	11
3.1.2	Software Cache	11
3.2	Programming and Execution Model	12
3.3	Initial GAMA Hands-On	13
3.3.1	The Polu Case-Study	13
4	Progressive Photon Mapping as a Case Study	14
4.1	The Rendering Equation	14
4.2	Ray Tracing History	14
4.3	Photon Mapping	15
4.4	Extensions to Photon Mapping	15
5	Conclusions and Future Work	16

Glossary

AVX Advanced Vector Extensions

BRDF Bidirectional Reflectance Distribution Function)

CPU Central Processing Unit

CU Computing Unit

CUDA Compute Unified Device Architecture. A parallel computing platform for GPUs

DMA Direct Memory Access

DSP Digital Signal Processor

GAMA GPU And Multi-core Aware

GPU Graphics Processing Unit

GPGPU General Purpose GPU

HetPlat **H**eterogenous **P**latform

ISA Instruction Set Architecture

MIC Many Integrated Core

MPI Message Passing Interface

OpenACC Open Accelerator API

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PCIe Peripheral Component Interconnect Express

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Threads

SM Streaming Multiprocessor

SMX Kepler Streaming Multiprocessor. A redesign of the original SM

TBB Threading Building Blocks: C++ template library for task parallelism

List of Figures

1	Overview of the Fermi architecture	7
2	Overview of the Kepler architecture	8
3	GAMA Memory Model	11
4	Illustration of a Specular to Diffuse to Specular path (SDS)	15
5	Subsurface Scattering	16

1 Introduction

1.1 Contextualization

Heterogeneous high performance computing platforms are becoming popular, taking advantage of accelerator devices to provide higher performance at lower costs. These accelerators have characteristics that makes them suitable to perform different tasks, and as such are useful as co-processors that complement the work of conventional CPUs.

Accelerators first started to appear with GPUs, which gradually evolved from specific hardware dedicated to graphics rendering, to fully featured general programming devices, capable of massive data parallelism and performance, at lower power consumptions. As of 2012, over 50 of the ¹TOP500's list are powered by GPUs, which indicates an exponential growth in usage when compared to previous years. This increased usage is motivated by the effectiveness of these devices for general-purpose computing.

Other types of accelerators since emerged, like the recent Intel Many Integrated Core (MIC) architecture, and while all of them differ from the traditional CPU architecture, they also differ between themselves, providing different hardware specifications, along with different memory and programming models.

Development of applications targeting these devices tends to be harder, or at least different from conventional programming. One has to take into account the differences of the underlying architecture, as well as the programming model being used, in order to produce code that is not only correct in terms of the specification of that model, but also efficient. Efficient code targeted at one device might not be (and as a general rule, is not) adequate to a different device. As a result, developers of efficient code have to take into account the characteristics of each different device they are using within their applications. In addition, the parallel nature of these accelerators introduces yet another difficulty layer for developers.

Each accelerator can also be programmed in a variety of ways, ranging from low level programming models such as CUDA to higher level libraries like OpenMP or OpenACC. Each of these provides a different method of writing parallel programs, and has a different level of abstraction about the underlying architecture.

These different types of devices are most commonly used (in the context of general-computing) as accelerators, in a system where at least one CPU manages the main execution flow, and delegates specific tasks to the remaining computing resources. A system that uses different computational units is commonly referred to as a heterogeneous platform, here referred to as a **Heterogenous Platform** (HetPlat).

1.2 Motivation & Goals

As the complexity of applications increases, so does the difficulty of efficiently managing the different available resources. In a HetPlat, resource management is a key problem that depends on several different factors, and can be a difficult one to solve, especially for more complex applications. This is an even greater problem for irregular applications, where memory access patterns and task execution times are not always predictable.

GAMA is a framework that aims to provide the tools for developers to create dynamic applications, capable of efficiently running on these high performance computing platforms [1].

The GAMA framework is currently still in development, and supports only x86-64 CPUs and CUDA-capable GPUs. Although it has been deeply tested for a wide variety of kernels to validate the correctness and efficiency of its memory and execution model, it currently lacks a more intensive assessment, with a more robust and realistic application.

¹A list of the most powerful supercomputers in the world, updated twice a year (<http://www.top500.org/>)

Small kernels have a wide range of applications, are deeply studied and optimized, and are a good source for an initial analysis on the performance results of the execution model. However, when considering a real and more resource intensive application, where possibly multiple tasks must share the available resources, other problems may arise. Therefore, an evaluation of the framework with a realistic application, as opposed to the previously used synthetic benchmarks, is needed to understand its effectiveness.

The main goal of this work is to perform a quantitative and qualitative analysis of the GAMA framework, applied to a large scale irregular algorithm, in order to validate its effectiveness, and identify possible soft-spots, especially when compared to other similar frameworks. The work will be focused on fully understanding GAMA, how to better take advantage of it, and how to further improve it.

This will be accomplished by using GAMA to implement a computational intensive irregular application. The result will be used to evaluate the overall efficiency of the GAMA framework, applied to a large scale application, instead of only a single computational kernel.

The work will also be useful to understand how to better take advantage of GAMA, and where it should be improved. An analysis of the execution results of the application should provide insight about the way GAMA is handling the existing jobs and data sets, and from there identify possible bottlenecks or situations where an improvement could be possible. The comparative analysis will also be extremely useful in this process, as it will make it possible to know whether or not different implementations perform better than GAMA, and why.

1.3 Case Study

The problem chosen as a case-study for this work is the progressive photon mapping algorithm, first proposed in [2], which is a global illumination method, and an evolution of classical ray-tracing and path-tracing. This family of methods has a wide variety of applications, the most common one being the rendering of photo realistic scenes, by computing the global illumination to solve the rendering equation.

Rendering methods such as photon mapping are a common example of resource demanding, irregular applications, due to the high amount of data required to accurately describe a 3D scene, and to realistically simulate all of its lighting effects. Therefore, it should serve as a suitable case study in the context of this dissertation.

The details of this algorithm will be explained in more depth in Section 4

1.4 Document structure

Section 2 describes the literature search made during the initial work stages of this dissertation, as well as some technologies that will be useful (or potentially useful) in later work. Section 3 shows the initial analysis of the GAMA framework, and presents some concepts and considerations, based on sample code study and hands-on experiments with a couple of small test cases. Section 4 describes in greater detail the progressive photon mapping algorithm, which will be used as the main case study for the rest of this dissertation, to evaluate the advantages and disadvantages, as well as potential of GAMA. Finally, Section 5 presents some initial conclusions and explains the line of work to be followed for the remaining dissertation work.

2 Technological Background

2.1 Parallel Computing

Traditional computer programs are written in a sequential manner. It is natural to think of an algorithm as a sequence of steps that can be serially performed to achieve a final result. This has actually been

the most common programming paradigm since the early days of computing, and it synergizes well with single processor machines. Optimizations related to instruction-level parallelism, such as out-of-order execution, pipelining, branch prediction or superscalarity, were mostly handled by the compiler or the hardware itself, and transparent to the programmer. Vector processing was also a key performance factor, allowing the same instruction to be applied to a vector of data simultaneously, rather than one at a time (commonly referred to as Single Instruction, Multiple Data (SIMD)).

But in the beginning of the XXI century, the development of computational chips shifted from a single faster core perspective, to a multi core one. The evolution of single-core processors was already reaching its peak, and was slowing down due to the increasing difficulty in reducing transistor size or increasing clock frequencies, while introducing or aggravating other problems, such as heat dissipation, which becomes harder with the increased complexity of a chip. The solution was to move a multi-core perspective, coupling more cores in the same chip, to share the workload and allow overall computational capabilities to keep evolving.

This has allowed hardware development to keep in conformance with Moore's Law² [3]. And while it was a necessary step from a hardware's perspective, this has important implications in software development. In order for an application to take advantage of multi-core technology, it needs to be broken into smaller tasks, that can be independently executed, or with some level of synchronization and communication between them. Writing parallel algorithms requires an adaptation to this new paradigm, as a sequential line of execution does not provide efficient results in platforms that support parallel execution of several threads or processes.

2.2 The GPU as a Computing Accelerator

With the increasing demand for highly data-parallel algorithms, and the growing amount of data to process, hardware development started shifting towards the goal of solving that problem. Initially, that started with the increased support for vector instructions in common CPUs, and the SIMD model. This allowed a single instruction to operate on a set of elements at once, effectively achieving a kind of parallelism which was extremely useful with highly data-parallel applications. Modern Intel processors, starting with the Sandy Bridge family, already support Advanced Vector Extensions (AVX), an extension to the *x86* instruction set allowing SIMD instructions capable of handling 256 bits registers. This extension also introduces three-operand SIMD instructions that allow more general $c = a + b$ operations to be handled with a single instruction, in addition to the previous instructions which only allowed two operands ($a = a + b$).

A similar and more flexible model is the usage of a single instruction across multiple threads running concurrently, called Single Instruction, Multiple Threads (SIMT). This is the programming model behind GPUs, which gradually started to gain more attention for their general purpose computing capabilities. Although the hardware of a GPUs is still tightly coupled with graphics processing and rendering, there have also been several advances in their usage as General Purpose GPUs (GPGPUs).

2.2.1 Fermi Architecture

The Fermi architecture was an important milestone of GPUs technology, as it was one of the first generations target directly towards GPGPU and high performance computing, rather than just graphics rendering. The first Fermi devices were released in 2010, and were the first NVidia products to include support for double precision floating point number, which was an important feature for high performance computing to provide more accurate results. Fermi devices also included a GDDR5 memory controller with support for Direct Memory Access (DMA) through the PCIe bus, and up to 16 Streaming Multiprocessor (SM), for a total of up to 512 CUDA Cores.

²A prediction by Gordon Moore, stating that approximately every two years, the number of transistors in a computer chip would double

Each SM can access a total of 64KB of private memory, which can be configured to be split between shared memory and L1 cache, although configuration was restricted to either 48KB + 16KB or 16KB + 48KB. This provided some flexibility to the devices, since some problems would benefit more from the extra shared memory and not so much from cache, and vice-versa.

Additionally, all SMs share a common 768KB L2 cache, and a final level of global memory, external to the chip, that can go up to 6GB of GDDR5, with a bandwidth of 192.6 GB/s in the Tesla C2070 [4].

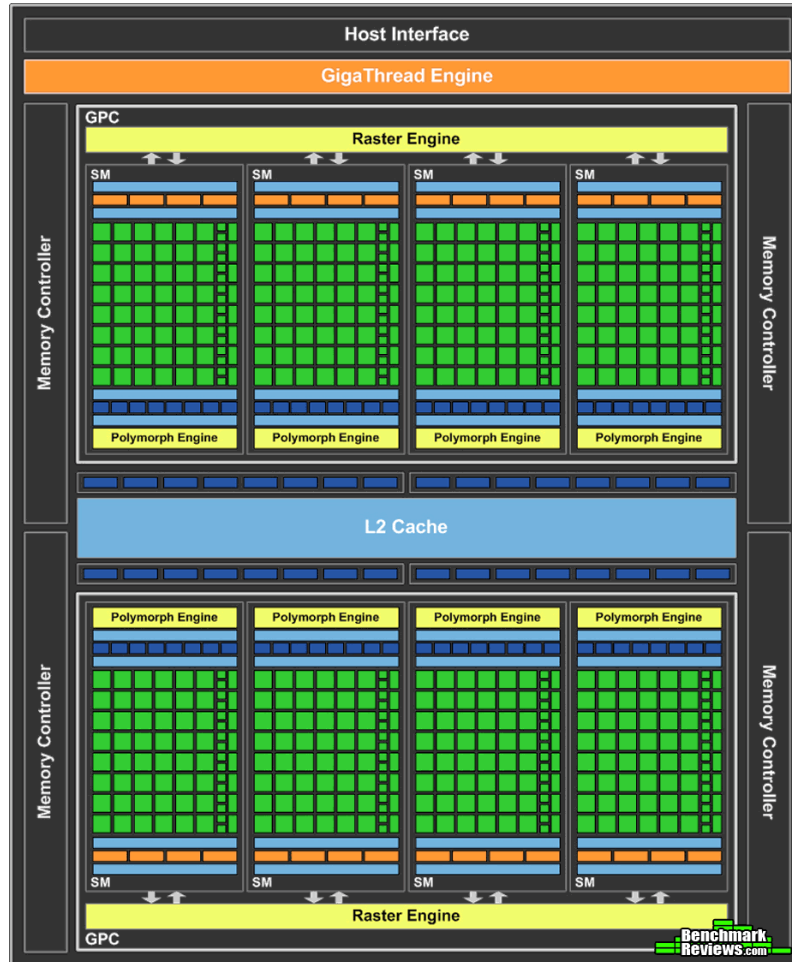


Figure 1: Overview of the Fermi architecture

This architecture is backed by a hardware-based thread scheduler, located within each SM, that attempt to feed the execution unit with threads grouped in blocks of 32, or *warps*. All threads within a warp are considered to be independent from each other, so there is no need for dependency checking. Any thread that is waiting for memory accesses to be finished will remain unscheduled until the required data is available. Since the scheduling is made directly via hardware, the switch between threads is nearly free, at least when compared with thread scheduling on a CPU. As a result, this strategy works better when the total amount of threads competing for resources is much higher than the amount of execution units, allowing for the latency of memory accesses to be hidden away by instantly scheduling a different *warp*, effectively hiding memory latency while still keeping execution units busy. This is very different from CPU scheduling policies, where switching between threads requires a context switch, which takes considerably longer, making that approach not as feasible as for a GPUs.

2.2.2 Kepler Architecture

The follow-up generation to Fermi is in many ways similar to its predecessor. One of the most notorious change is the increase in the total amount of available CUDA cores, capable of going up to 2880 in high-

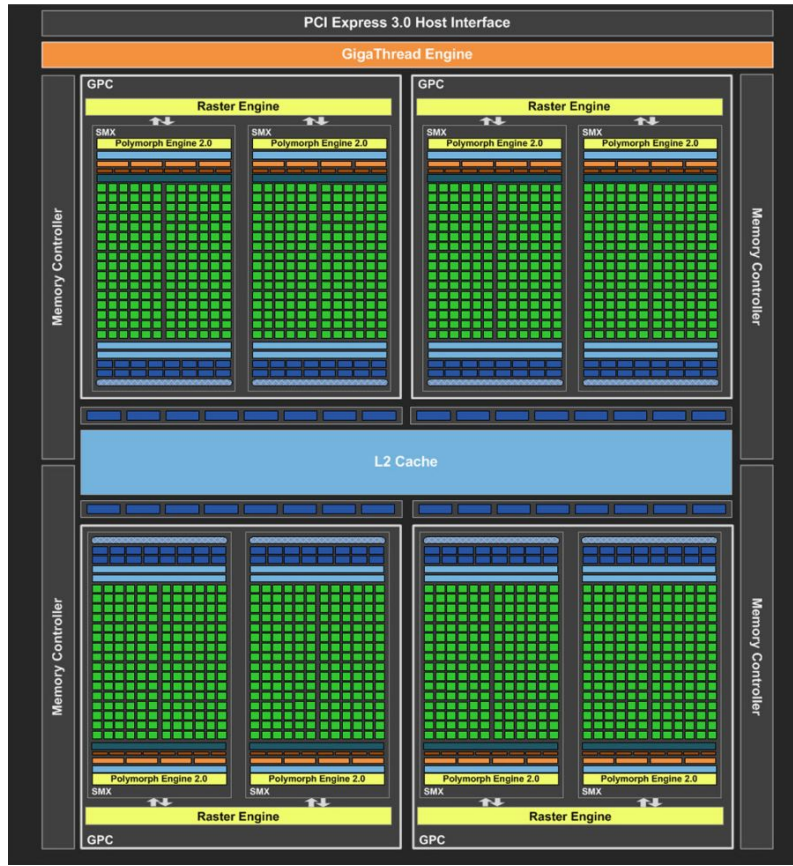


Figure 2: Overview of the Kepler architecture

end devices, due to the redesign of the Streaming Multiprocessor, now called SMX, each one with 192 CUDA Cores, although working at lower frequencies than before, due to the removal of shader frequency, a compromise to make room for the extra CUDA Cores. The entire chip now works based on the core frequency. Overall, individual core efficiency is lowered, but the global system becomes more efficient.

The memory hierarchy was also upgraded over the previous generations, with the maximum amount of registers per thread going from 63 to 255, the addition of an extra 48KB L1 read-only cache, while still maintaining the original L1 cache (which now also allows a configuration of 32KB shared + 32KB cache), and L2 cache going up to a maximum of 1.5MB with increased bandwidth. Memory reads have been upgraded to allow for 256-bit blocks to be read, as opposed to 128-bit in Fermi [5].

The programming model has been extended with the addition of dynamic parallelism, allowing CUDA thread to spawn new threads, a feature not possible with previous versions. This is an important feature for irregular algorithms, such as photon mapping. It is also possible to invoke multiple kernels for a single GPU, transparently dividing them between the available SMXs.

2.3 Heterogeneous Platforms

Initial programming methodologies for HetPlats consists mostly on programming the main application structure to a regular CPU core, and offload some of the heavily data-parallel work to an accelerator, usually a GPU. This often is coupled with manual testing to assert whether or not executing the task on the accelerator device, along with the required memory transactions, are actually beneficial to the overall performance. With the increased acceptance of accelerators as co-processors, development efforts have been made towards making this development process much easier, and thus produce more, and better applications.

To address this problem, several frameworks have been proposed in the last years. These frameworks are usually targeted specifically at HetPlats, and are designed with a focus on their specific scheduling issues, which are considered a key issue. These frameworks include StartPU [6], Harmony [7], and GAMA.

These frameworks aim to provide a bridge for programmers to develop applications suited to HetPlat, by addressing problems such as memory management or the scheduling of multiple tasks issued for execution. Most of them however, provide mechanisms suited mostly for regular applications. When dealing with irregular applications, problems like scheduling become even more difficult, in particular because execution times and memory usage patterns are less predictable. This directly affects the decisions of the scheduler, and as such must be taken into account by the performance model employed by the framework. Irregular applications are one of the targets of the GAMA framework, which will be object of study throughout this dissertation, through the implementation of an irregular algorithm as a case study.

2.3.1 A note on debugging

It is hard enough to debug multi-threaded CPU applications, and the addition of an extra device, with a different architecture, and possibly different mechanisms to handle debugging, introduces yet another layer of complexity to the development process. Conceptually, it is much harder for the programmer to keep track of the debugging process, since there is not as much control over what is happening. For instance, unless threads are fully synchronous, a breakpoint inserted at any given point of a parallel region might result in different threads stopping at arbitrarily different regions, affecting the global state of the application.

2.3.2 Scheduling

Perhaps the biggest issue about the usage of accelerators is the ability to fully take advantage of the resources. It is extremely important to not overload one processor with too much data while others remain idle, or to offload work to a different device when the execution time together with the cost of data transfers will result in no benefit, or possibly in even worse results. In irregular algorithms this is an even more difficult problem, due to the increased unpredictability. For those reasons, in order for any HetPlat framework to properly manage the executions of multiple tasks, the scheduling policy is one of the most important aspects. In GAMA, the scheduling policy is also backed up by the global memory system, which handles memory utilization and data transfers [8, 9, 10].

2.4 Software

Several technologies already exist to assist in the production of parallel code. These softwares range from low level drivers, which are sometimes required to access hardware-level features (e.g. CUDA or OpenCL, which acts as a bridge between the programmer and the GPU driver) to fully featured libraries that aim to provide higher level directives to developers, providing more flexibility to work on the algorithm itself rather than on hardware specific details and optimizations, which are sometimes hidden away.

However, most of these libraries are limited to a specific programming model, such as shared memory systems like multi-core CPUs. This subsection presents an overview on some of the existing software development tools to program in parallel environments. While some examples presented here may not be directly used throughout the dissertation, they were useful to analyze different software approaches, and understand how to explore the capabilities and potential of GAMA.

2.4.1 OpenMP

OpenMP can be described as a set of compiler directives and routines applicable to Fortran and C/C++ code in order to express shared memory parallelism [11]. It provides a high-level API with a wide range of processors supporting it. Using the defined directives, the programmer is able to describe how subsection of an algorithm should be parallelized without introducing too much additional complexity within the code. The programmer is only left with the task of ensuring there are no data races or dependencies within a parallel task. More advanced options like specifying task granularity, private variables, or explicit barriers is also possible, enabling more control over the parallel region of the code.

This has been one of the standards for parallelization on 86 CPUs, including the recent Intel MIC architecture.

2.4.2 CUDA

CUDA is the platform created by NVidia to be used along with their GPUs. The complete CUDA toolkit provides a driver, a compiler and a set of tools and libraries to assist in the development of highly parallel GPU code in either C/C++ or Fortran. Wrappers for other languages are also available, as a result of third party work.

CUDA programmers are usually required to pay attention to architectural details of their code, in order to best take advantage of the platform. Unlike OpenMP for instance, where the parallelism is mostly abstracted away from the developer, in CUDA one has to consider the correct use of the available resources, and how to structure the algorithm to fit them. As a result, some higher level wrappers are starting to emerge (e.g. OpenACC). Despite the added complexity, CUDA has been extensively used in scientific fields such as computational biology, cryptography and many others.

2.4.3 MPI

The Message Passing Interface (MPI) is a specification for a language-independent communication protocol, used in the parallelization of programs with a distributed memory paradigm. This standard defines the rules of a library that enables message passing capabilities, allowing programs to spawn several processes, and communicate between each other by sending and receiving messages. The processes created can be physically located on different machines (hosts), with the communication being made across the available network connections. It is also possible to use MPI to communicate with accelerator devices that support it, such as the Intel Xeon Phi. The main goals of MPI are towards high performance, scalability and portability, and is one of the most dominant parallelization models in current high performance computing [12].

2.4.4 OpenACC

OpenACC aims to provide a standard for programming parallel devices, and simplify the programming of heterogeneous systems. The programming model introduced by this standard is similar in many ways to OpenMP, especially in the usage of compiler directives to specify the parallelism rules. Unlike OpenMP however, code produced by OpenACC may also be targeted at different devices, such as GPUs, and more recently the new Intel Xeon Phi [13].

3 The GAMA Framework

The GPU And Multi-core Aware (GAMA) is a framework to aid computational scientists in the development or porting of data-parallel applications to heterogeneous computing platforms. Currently HetPlat support includes only systems composed of traditional CPUs cores and one or several CUDA-capable GPU devices.

GAMA provides an abstraction of the hardware platform, attempting to free the programmer from the workload scheduling and data movement issues across the different resources. In GAMA, an application is composed of a collection of jobs, each defined as a set of tasks applied to a different input data set. Every job shares a global address space, instead of directly using the private memory of each device.

3.1 Memory Model

GAMA uses an unified programming model that assumes a hierarchy composed of multiple devices (both CPUs and GPUs), where each device has access to a private address space (shared by all computational units, or cores, within that device), and a distributed memory system between devices. The framework assumes that multiple computational units of an individual device can cooperate and communicate through a shared memory space, and that the underlying programming and execution model of that device provides synchronization mechanisms (barriers, atomics and memory fences). To abstract the distributed memory model that is used between devices, GAMA introduces a global address space. Figure 3 illustrates how GAMA understands the memory hierarchy of a HetPlat.

3.1.1 Memory Consistency

Communication between the different memory spaces of each device is expensive due to the need of synchronization and communication between the host CPU and the devices. Due to this, a relaxed consistency model is used, which enables the system to optimize data movements between devices, offering the developer a single synchronization primitive to enforce memory consistency.

3.1.2 Software Cache

Some applications require safe exclusive access to specific partitions of a data set. To address this issue, a software cache among devices was implemented. This ensures that the required data is as close to the device as possible, taking advantage of the local memory of each device. It also provides a safeguard mechanism in combination with the global memory system, to ensure each device has a copy of a specific partition, when requested by the developer. Additionally, the cache local copies on the device shared memory space use semantically correct synchronization primitives within the device.

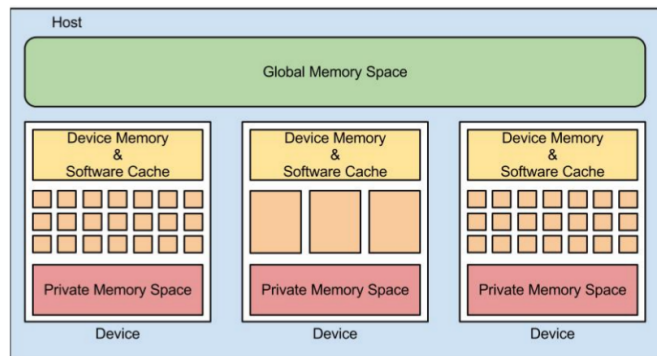


Figure 3: GAMA Memory Model

3.2 Programming and Execution Model

To better understand the programming and execution model employed in GAMA, some key concepts are introduced in this subsection:

Computing Unit (CU)

In GAMA, a Computing Unit is an individual execution unit, capable of executing a general-purpose application. In the context of a CPU, a Computing Unit represents a single core, while on a GPU, in the current implementation represents a single Streaming Multiprocessor (SM). Thus the terms CU and core may be used with the same meaning.

Device or Worker

Represents a collection of Computing Units that share some level of memory (e.g. the CPU cores on the same machine, or the SMs of a single GPU).

Host

The group of all devices within a single computational node.

Domain

A global view of a particular data structure that enables developers to access any memory location using the global address space, and hiding the complexity of the underlying memory system. At the application level, the user is able to define filters of partial views of a single domain, allowing the system to identify the required communication primitives and enforce the global address space, the memory consistency model, and cache and synchronization mechanisms.

Job

A tuple associating data domains with the corresponding computations related to it (the computational kernel), and a specialized dicing function that defines the best strategy for job granularity, recursively splitting the job into smaller tasks across the data domains. This dicing function is somewhat analogous to the ability of defining task granularity with tools such as OpenMP, but it can employ more flexible solutions, to account for the irregularity of the algorithms.

Kernel

The computation associated with a job. In a best-case scenario, a computational kernel can be mapped directly to a given device simply with the help of the toolkit supporting that device. In most cases however, the kernel needs to be tailored to a specific device's programming model. This is achievable by extending the job description with the addition of the specialized kernel for a specific device. This feature also enhances the programming model by enabling developers to tailor specific computational kernels for each platform, taking advantage of architecture-specific features.

The organization of the execution model between Computational Units, Devices and Hosts ensures that a consistent model can be assumed implicitly, where CUs within the same device share a common address space, allowing the usage of device-specific synchronization mechanisms to manage the coordination of concurrent executions within that device.

An application in GAMA is a collection of jobs submitted to a run-time system for scheduling among the available computational resources. Dependencies among jobs can be specified with explicit synchronization barriers. The main goal of the runtime scheduler is to reduce the overall execution time of any given application. The scheduler uses information provided by each job in order to determine the best scheduling policy, based on current runtime states, as well as execution history. If the granularity of a job is too coarse to enable a balanced scheduling policy, GAMA will recursively employ the dicing function of a job to adjust it to the capabilities of the device.

3.3 Initial GAMA Hands-On

While studying the details of the framework, as well as the theoretical foundations of the progressive photon mapping case study, some tests were already done with GAMA. Initially some of the source code of the included samples, such as the SAXPY and Barnes-Hut algorithms were studied.

Later, an implementation of a first order finite volume method was implemented using GAMA, using the previously implemented versions of that same algorithm as basis of comparison. This versions included a sequential implementation, and two parallelized implementations, one with OpenMP, and another with CUDA. The details of the algorithm are described in more detail in Section 3.3.1

3.3.1 The Polu Case-Study

The application, here called `polu`, was already the subject of a parallelization study in [14], which described the incremental work where the application was improved from a sequential implementation, first through a process of analysis and sequential optimization, and then subject to parallelization using two techniques, a shared memory CPU implementation with OpenMP, and a GPU implementation with CUDA. A third implementation stage focused on the development of a distributed memory implementation using MPI, but without providing any positive performance results.

The `polu` application, computes the spread of a material (e.g. a pollutant) in a bi-dimensional surface through the course of time. This surface is discretely represented as a mesh, composed mainly of edges and cells. The input data set contains information about the mesh description, the velocity vector for each cell and an initial pollution distribution. Both inputs and outputs are compatible with the `gmsh` application, for data visualization.

The Algorithm

The algorithm used by this application is a first order finite volume method. This means that each mesh element only communicates directly with its first level neighbours in the mesh, which makes this a typical case of a stencil computation. In terms of performance, being a stencil algorithm implies that the operational intensity will most likely remain constant with larger problem sizes [15, 16]. Despite this, the algorithm is still very irregular in terms of memory access patterns, because meshes generated by `gmsh` suffered from deep locality issues, turning memory accesses ordered by cells or edges close to random.

The execution of the algorithm consists of looping through two main kernels, advancing in time until an input-defined limit is reached. These two kernels are:

`compute_flux`

In this step, a flux is calculated for each edge, based on the current pollution concentration of each of the adjacent cells. A constant known as Dirichlet condition is used for the boundary edges of the mesh, replacing the value of the missing cell. This flux value represents the amount of pollution that travels across that edge in that time step.

`update`

With the previously calculated fluxes, all cell values are updated, with each cell receiving contributions from all the adjacent edges. After this, one time step has passed.

Implementation

In order to run the algorithm using the framework, both kernels had to be re-implemented using GAMA jobs. Additionally, data structures had to be re-written to make use of the facilities provided by GAMA to allow memory to be automatically handled by the global address space. This presents an obviously large amount of development work, since mostly everything had to be re-written according to GAMA rules. However, it has to be taken into account the fact that this additional work also had to be performed in

the previous implementations studied in [14], since most of the original code was not suitable for efficient parallelization.

From this, one initial consideration can already be made about the framework, in the sense that the effort required to parallelize it might be too high if a given application is already written with some concerns regarding parallelization (although without GAMA). Since specific data structures and job definitions need to be used, this may hamper the adoption of GAMA by already implemented solutions, unless the performance advantages are significant enough to justify the additional effort.

Study limitations

Unfortunately, there are several restrictions to the input generation for this algorithm. In particular, the utility required to generate a mesh with arbitrary resolution has an estimated complexity of $O(N^3)$ which prevented large enough test cases to be generated. The largest available input contained only around 400,000 cells, and represented a total memory footprint of just over 40MB, which is extremely small, and does not allow a good enough analysis on resource usage. With such a low resource occupancy, the scheduling policy employed by GAMA will most likely assign all the workload to a single device, as the cost of data transfers, and the low execution time for each kernel for such a small data set would not justify otherwise. Additionally, this being a typical stencil, each iteration requires a barrier, allowing no execution of two simultaneous iterations, which would be an additional way of improving parallelism.

Knowing this, any result obtained by profiling the `polu` application under these conditions would not provide a correct insight about the algorithm, or about the framework, and as such, these results are not presented here. The `polu` test case still served as an initial basis to gain some insight into GAMA, and to better prepare the implementation of the progressive photon mapping case study.

4 Progressive Photon Mapping as a Case Study

4.1 The Rendering Equation

The realistic simulation of illumination of an environment is a complex problem. In theory, a simulation is truly realistic when it completely simulates, or closely approximates, the rendering equation. This equation, first proposed in 1986 [17], is based on the laws of conservation of energy, and describes the total amount of emitted light from any given point, based on incoming light and reflection distribution. The equation is presented in Section 4.1.

$$L_s(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + \int_{\Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\omega_i \quad (1)$$

In short, the equation defines the surface radiance $L_s(x, \vec{\omega}_r)$, leaving the point x in the direction $\vec{\omega}_r$. This is given by $L_e(x, \vec{\omega}_r)$, which represents light self-emitted by a surface, and $L_i(x, \vec{\omega}_i)$, which is the radiance along a given incidence direction. f_r is the Bidirectional Reflectance Distribution Function (BRDF) and Ω represents the semi-sphere of incoming directions centered in x .

4.2 Ray Tracing History

Prior to photon mapping, typical approaches to approximate the rendering equation would use combinations of more than one method, such as Ray Tracing, Radiosity or Metropolis Light Transport [18, 19]. Each method attempts to simulate the travel of light particles across the scene, and model the various interactions with the environment, but with different approaches, advantages and limitations.

Ray Tracing methods work by simulating light particles traveling from the eye into the scene, being reflected and refracted until they reach a light source.

Radiosity follows an opposite approach, and simulates the path light takes from the light source, until it reaches the eye. It only deals with diffuse interactions, however, and is commonly used in combination with other techniques.

4.3 Photon Mapping

Photon mapping is based on another method to approximate the rendering equation, first proposed in 1996 [20], and works as a two-pass algorithm. In a first pass, photons are traced from the light sources into the scene and stored in a photon map as they interact with the surfaces. This creates a structure that roughly represents the light distribution within the scene. The second pass is rendering, in which the photon map is used to estimate the illumination in the scene, using common Ray Tracing techniques (for example, Monte Carlo ray tracing). The photon map is used to aid the computation of the total radiance. It is useful not only to increase performance, mostly by allowing a reduction in the number of samples to cast while still providing accurate results, but also to allow the modeling of some light effects that are not preset, or are inefficient to process in other rendering methods.

One case in particular is when light is being transported along a specular-to-diffuse-to-specular path (SDS path) illustrated in Figure 4 before reaching the eye. This is what is commonly known as a caustic, such as, for example, the shimmering light seen on the bottom of a pool, or any light source enclosed in glass. This scenario is very common since most artificial light sources are enclosed in glass, but is particularly hard to simulate, particularly when the light source is small, making the sampling probability very low when using Monte Carlo methods.

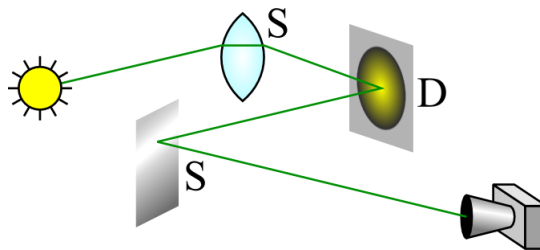


Figure 4: Illustration of a Specular to Diffuse to Specular path (SDS)

Another example of a typically hard to simulate effect is Subsurface Scattering, which is observed when light enters the surface of a translucent object, is scattered when interacting with the material, and finally exits the surface at a different point. Generally, light will be reflected several times within the surface before backing out at an angle different from the one it would have taken had it been reflected by the surface. This is visible in materials such as marble, or skin, and can be seen in Figure 5

Both of these can be simulated well by Photon Mapping algorithms, although a high amount of caustics will hinder performance considerably.

4.4 Extensions to Photon Mapping

The algorithm initially proposed [20] has been the basis for several improved versions proposed throughout the years.

In 2005, a proposed version introduced the concept of Reverse photon mapping [21], in which the two steps were reversed. The ray tracing step was the first to be computed, and would build an accelerating structure (such as a *kd-tree*), used in the later photon tracing steps to find the nearest hit points that a photon contributes to. This approach would improve performance when large numbers of rays are used.



Figure 5: Subsurface Scattering

In 2000, Suykens and Willems [22], and later in Hachisuka, Ogaki and Jensen 2008 [2] attempted an algorithm that would asymptotically converge to a solution, by gradually reducing the search radius of the radiance estimate based on the number of photons already found.

A new formulation of photon mapping was proposed in 2011 [23], which used a probabilistic formulation that allows for a memoryless algorithm, without requiring statistics to be maintained, and removes dependencies between each individual step, allowing them to be independently calculated, and thus being able to run in parallel.

5 Conclusions and Future Work

This report presented the initial literature search and contextualization of the dissertation, and presented an overview of the case study and the GAMA framework for irregular applications on HetPlats, that will be used throughout the work. The main goal is to evaluate GAMA by providing a study using a computational intensive algorithm. With that in mind, the progressive photon mapping algorithm was selected, providing both the irregular characteristics, parallelism possibilities and irregularities that are desired to properly validate GAMA.

The next stage will involve the implementation of the case study using GAMA to handle work distribution between the available CPUs and GPUs. Based on the performance results of the implemented algorithm, it will be possible to better understand how the existing jobs and data structures are being managed, and where improvements should be made. It will also be interesting to see how the implemented algorithm behaves, performance-wise, against similar implementations without the framework, and evaluate whether the effort of using GAMA paid off. The development effort taken into the implementation will also be taken into account, since it is also important for frameworks to work as a good *middleware* tool to assist the programmer and reduce development effort, without compromising application performance.

This dissertation work also aims to analyze the feasibility and limitations of extending GAMA to provide support for additional accelerator architectures other than GPUs.

References

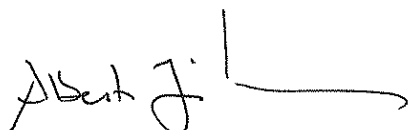
- [1] João Barbosa. *GAMA framework: Hardware Aware Scheduling in Heterogeneous Environments*. Tech. rep. Computer Science Dept., University of Texas at Austin, Sept. 2012.
- [2] T. Hachisuka, S. Ogaki, and H.W. Jensen. “Progressive photon mapping”. In: *ACM Transactions on Graphics (TOG)*. Vol. 27. 5. ACM. 2008, p. 130.
- [3] R.R. Schaller. “Moore’s law: past, present and future”. In: *Spectrum, IEEE* 34.6 (1997), pp. 52–59.
- [4] NVIDIA. *NVIDIA’s Next Generation CUDA Compute Architectur: Fermi*. Tech. rep. 2009.
- [5] NVIDIA. *NVIDIA’s Next Generation CUDA Compute Architectur: Kepler GK110*. Tech. rep. 2012.
- [6] C. Augonnet et al. “StarPU: A unified platform for task scheduling on heterogeneous multicore architectures”. In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198.
- [7] G.F. Damos and S. Yalamanchili. “Harmony: an execution model and runtime for heterogeneous many core systems”. In: *Proceedings of the 17th International Symposium on High performance distributed computing*. ACM. 2008, pp. 197–200.
- [8] A. Mariano. “Scheduling (ir)regular applications on heterogeneous platforms”. Master Thesis. Sept. 2012.
- [9] Artur Mariano et al. “A (ir)regularity-aware task scheduler for heterogeneous platforms”. In: *Proceedings of the 2nd International Conference on High Performance Computing*. Kiev, October 2012, pp. 45–56.
- [10] Ricardo Daniel Queirós Alves. *Distributed Shared Memory on Heterogeneous CPUs+GPUs Platforms*. Master Thesis. 2012.
- [11] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.
- [12] S. Sur, M.J. Koop, and D.K. Panda. “High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 105.
- [13] HPC Wire. *OpenACC Group Reports Expanding Support for Accelerator Programming Standard*. June 2012. URL: http://www.hpcwire.com/hpcwire/2012-06-20/openacc_group_reports_expanding_support_for_accelerator_programming_standard.html.
- [14] Miguel Palhas and Pedro Costa. *A Finite Volume Case Study From An Industrial Application*. 2012.
- [15] S. Williams, A. Waterman, and D. Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.
- [16] S. Williams et al. “The Roofline Model”. In: *Performance Tuning of Scientific Applications*. CRC (2010).
- [17] J.T. Kajiya. “The rendering equation”. In: *ACM SIGGRAPH Computer Graphics* 20.4 (1986), pp. 143–150.
- [18] J.R. Wallace, M.F. Cohen, and D.P. Greenberg. *A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods*. Vol. 21. 4. ACM, 1987.
- [19] E. Veach and L.J. Guibas. “Metropolis light transport”. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 1997, pp. 65–76.
- [20] H.W. Jensen. “Global illumination using photon maps”. In: *Rendering Techniques* 96 (1996), pp. 21–30.
- [21] V. Havran, R. Herzog, and H.P. Seidel. “Fast final gathering via reverse photon mapping”. In: *Computer Graphics Forum*. Vol. 24. 3. Wiley Online Library. 2005, pp. 323–332.

- [22] F. Suykens and Y.D. Willems. “Adaptive filtering for progressive monte carlo image rendering”. In: *WSCG 2000 Conference Proceedings*. 2000.
- [23] C. Knaus and M. Zwicker. “Progressive photon mapping: A probabilistic approach”. In: *ACM Transactions on Graphics (TOG)* 30.3 (2011), p. 25.

Parecer de Conformidade

Declaro que o Relatório de Pré-Dissertação de Miguel Branco Palhas foi revisto pelo orientador e co-orientador desta dissertação e que se encontra em conformidade com o Plano de Trabalhos já submetido.

Universidade do Minho, 7 de fevereiro de 2013.

A handwritten signature in black ink, appearing to read 'Albert J.' followed by a long horizontal stroke.

(Orientador)

Alberto José Proença