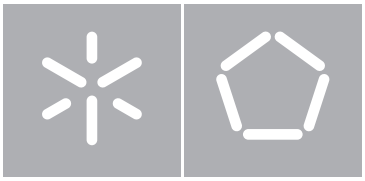
Universidade do Minho
Escola de Engenharia

Miguel Branco Palhas

An Evaluation of the GAMA/StarPU Frameworks for Heterogeneous Platforms: The Progressive Photon Mapping Algorithm

Setembro de 2013



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Miguel Branco Palhas

An Evaluation of the GAMA/StarPU Frameworks 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

Setembro de 2013

Agradecimentos

Ao meu orientador, prof. Alberto Proença, por uma excelente orientação, quer na disponibilidade completa durante todo o ano, quer no rigor exigido e análise ao trabalho efectuado. Ao meu co-orientador, prof. Luis Paulo Santos, pelos desafios lançados no íncio do trabalho, e pela disponibilidade que demonstrou sempre que solicitado.

Aos colegas do LabCG, que me acolheram durante este ano num ótimo ambiente de trabalho, que foi essencial para esta dissertação. Em especial ao Roberto Ribeiro, pela ajuda e discussões ao longo do ano, que foram uma grande contribuição para este trabalho.

Aos meus colegas e amigos André Pereira e Pedro Costa, por todas as discussões, ajuda mútua e companheirismo durante estes dois anos.

A todos os membros do CeSIUM, núcleo que me acolheu como uma segunda casa durante todo o meu percurso académico.

Ao grande grupo de amigos que formei nesta universidade, cuja lista é demasiado grande para enumerar aqui, que me acomparanham nestes que foram os melhores anos da minha vida, e me deram apoio durante os momentos de maior aoerto. Sem eles este trabalho não teria sido possível.

Aos colegas e amigos da GroupBuddies, em especial ao Roberto Machado, pela ajuda e compreensão demonstrada sempre que a minha ausência foi necessária.

Por fim, um especial agradecimento á minhão mãe e ao meu imão, pelo suporte e compreensão dada a minha acrescida ausência durante o ano.

Work funded by the Portuguese agency FCT, *Fundação para a Ciência e Tecnologia*, under the program UT Austin | Portugal.

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 targeting 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.

These frameworks include GAMA and StarPU. The first is being 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. StarPU has similar goals, but places a much greater emphasis in memory transfer latencies as the root cause of bottlenecks when offloading tasks to accelerators.

The subject of this dissertation is to provide a more in-depth evaluation of StarPU, using the progressive photon mapping irregular algorithm as a case study. The goal is to assert its effectiveness with a robust irregular application, and ultimately make a high-level comparison with the still under development GAMA, to understand where each of them has its weaknesses, and how GAMA's development could be further improved.

Resumo

Uma avaliação das frameworks GAMA/StarPU 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 caracteristicas, 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 dinamico da carga de trabalho pelos dispositivos disponíveis, e lidando com os pormenores e dificuldades inerentes da utilização destes sistemas de maior complexidade.

Nestas frameworks incluem-se o GAMA e o StarPU. A primeira está a ser desenvolvida de forma a tratar 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 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. O StarPU tem objectivos similares, mas coloca maior enfase na latência introduzida por transferências de memória, encarando-as como o principal limitador de

eficiência na utilização de aceleradores.

Esta dissertação propõe-se a disponibilizar uma avaliação intensiva do StarPU, 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, e culminar numa comparação com o GAMA, que ainda se encontra em desenvolvimento, de forma a identificar os pontos fracos de cada uma delas, e como poderá o GAMA evoluir da melhor forma.

Contents

Glossary

AVX Advanced Vector Extensions

BRDF Bidirectional Reflectance Distribution Function)

BVH Bounding Volume Hierarchy

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

HEFT Heterogeneous Earliest Finish Time

HetPlat **Het**erogenous **Plat**form

ISA Instruction Set Architecture

ILP Instruction Level Parallelism

MIC Many Integrated Core

MPI Message Passing Interface

NUMA Non-Uniform Memory Architecture

OpenACC Open Accelerator API

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PCIe Peripheral Component Interconnect Express

PM Photon Mapping

PPM Progressive Photon Mapping

PPMPA Progressive Photon Mapping with Probabilistic Approach

SPPM Stochastic Progressive Photon Mapping

SPPMPA Stochastic Progressive Photon Mapping with Probabilistic Approach

QBVH Quad Bounding Volume Hierarchy

QPI Quick Path Interconnect

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Threads

SM Streaming Multiprocessor

SMX Kepler Streaming Multiprocessor. A redesign of the original SM

List of Figures

Chapter 1

Introduction

1.1 Contextualization

Heterogeneous platforms are increasingly popular for high performance computing, with an increasing number of supercomputers taking advantage of accelerating devices in addition to the already powerful traditional CPUs, to provide higher performance at lower costs. These accelerators are not as general-purpose as a CPU, but have characteristics that make them more suitable to specific, usually highly parallel tasks, and as such are useful as co-processors that complement the work of conventional systems.

Moore's law [1, 2] predicted in 19755 that the performance of microprocessors would double every two year. That expectation has driven development of microprocessors until very recently. The number of transistors, and high clock frequencies of today's microprocessors is near the limit of power density, introducing problems such as heat dissipation and power consumption. Facing this limitations, research focus was driven towards multi-core solutions.

This marked the beginning of the multi-core era. While multi-core systems were already a reality, it was not until this point that they reached mainstream

production, and parallel paradigms began to emerge as more general-purpose solutions.

In addition to regular CPUs, other types of devices also emerged as good computational alternatives. In particular, the first GPUs supporting general purpose computing were introduced by NVidia around the year 2000.

These devices 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. They enable the acceleration of highly parallel tasks, being more efficient than CPUs in various scientific fields, but also more specialized. The usage of GPUs for general computing has been named General Purpose GPU (GPGPU), and has since become an industry standard. As of 2013, over 50 of the [1]TOP500's list were 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 recently 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, but also efficient. And efficiency for one device might have a different set of requirements or rules that are inadequate to a different device. As a result, developers code have to take into account the characteristics of each different device they are using within their applications, if they want to fully take advantage of them. Usually, the task of producing the most efficient code for a single platform is a time consuming task, and requires a deep knowledge of the architecture itself, In addition, the parallel

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

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 or OpenCL 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.

The complexity increases even further when it is considered that multiple accelerators might be used simultaneously. This aggravates the already existing problems concerning workload, scheduling, and communication.

Recent studies [3, 4] also show that overall speedups when using accelerators should not be expected to be as high as initially suggested. These studies show that, while the measured speedups of multiple applications ported to GPUs were real, the actual cause was not the better overall performance of the device, but actually the badly optimized original CPU code. Actually, when code is better designed, similar speedups can be obtained in traditional CPUs. This indicates that accelerators should not be regarded as the only source of high computational power, but rather as an additional resource, and the whole system should be used appropriately for better efficiency.

Today, accelerating 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 HetPlat. These types of systems are becoming particularly noticeable, as can be seen by the TOP500 ranking, where an increasing number of top-rated systems are heterogeneous.

Much like the phenomenon seem at the start of the multi-core era, a new paradigm shift must happen in order to correctly use a HetPlat. An even greater level of complexity is introduced, because one has to consider not only the multiple different architectures and programming models being used, but also the distribu-

tion of both work and data. A HetPlat is by definition a distributed system, since each device usually has its own memory hierarchy. As much as a given task may be fast on a given device, the data transfers required to offload such task may add an undesirable latency to the process, and can actually be one the performance bottlenecks of this strategy.

Even within a single device, memory hierarchy usage can have a big impact in performance. In a NUMA system, although each socket can access all memory nodes transparently, access times will be dependent on where the requested data is pinned. Performance problems arise from this if one considers the multiple sockets as one single multi-core CPU. Instead, the topology of the system can be considered when assigning tasks to each individual processing unit, and data transfered accordingly, to avoid expensive memory transactions.

Code efficiency is also becoming extremely volatile, as each new system that emerges usually requires architecture-specific optimizations, rendering previous code obsolete with respect to performance. There is an increasing need for a unified solution that allows developers to keep focuses on the algorithmic issues, and automatize these platform-specific issues, which present a barrier to the development of code targeting HetPlats.

Several frameworks have been developed since around 2008 to target these issues, and allow developers to abstract themselves from the underlying system. These frameworks usually manage the multiple resources of the system, treating both CPUs and co-processors as generic computational devices, able to execute tasks, and employ a scheduler to efficiently distribute workload. Memory management is also a key factor, with memory transfers playing a huge role in today's co-processor efficiently.

Some of these frameworks include GAMA [5], StarPU [6], MDR [7], Qilin [8], and a few others. These research focuses mostly on StarPU, although an overview and limited comparison with GAMA is also provided

These frameworks tend to encapsulate work by providing the concept of task,

which is usually not present in the underlying programming models of the programming languages used, and data dependencies, and employ a task scheduler to assign the existing workload to the available resources. The scheduler is considered one of the key features of these frameworks. It may takes into account multiple different factors in order to decide when and where to run the submitted tasks. These factors can range from the architectural details of the detected resources, to the measured performance of each task on each device, which can be done by building a history based on previous executions.

1.2 Motivation & Goals

HetPlats, and consequently frameworks targeting them, can still be seen as a recent computing environment, especially when considering the volatility and constant evolution of computing systems. As such, there is still much to develop when it comes to the efficient usage of a HetPlat.

GAMA is a recent framework that aims to provide the tools for developers to create dynamic applications, capable of efficiently running on these high performance computing platforms [5] This framework is currently still in development, and supports only x86-64 CPUs and CUDA-capable GPUs. It is somewhat inspired in a similar framework, StarPU, but with an emphasis on the scheduling of irregular algorithms, a class which presents extra problems when dealing with workload scheduling. Although GAMA 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. 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, a more realistic evaluation of the framework requires a more robust test case, as opposed to the currently used, more synthetic benchmarks.

The initial goal of this dissertation was to perform a quantitative and qualitative analysis of the GAMA framework, applied to a large scale algorithm, in order to validate its effectiveness, and identify possible soft-spots, especially when compared to other similar frameworks. This would be done through the implementation of a case study, particularly the Progressive Photon Mapping Algorithm.

However, GAMA presents itself as an unfinished product, still remaining in active development. This presented some concerns during the development of this work, as the current status of GAMA could be too volatile, difficulting the accuracy of any study made on it, at least until a more stable version was ready.

For this reasons, the focus switched to the StarPU, which is studied here through the implementation of a computationally intensive algorithm, providing the same analysis on the performance and usage of the framework, while serving as groundwork for future work to assert the effectiveness of GAMA once possible. Additionally, a comparative analysis is also made, in order to establish where each framework excels, and what features a future release of GAMA might require to be competitive against similar approaches with similar goals.

StarPU is an older project, and as a consequence, presents a more polished product, with the same overall goal of efficiently managing heterogeneous systems, but with a different philosophy and approach to the problem.

Overall, the work presented here consists on an analysis and comparison of GAMA and StarPU, with the later being used for the implementation of an algorithm as a case study. Framework-less implementations were also developed to establish the baseline for any profiling analysis.

1.3 Document Organization

Chapter 2 provides background information relevant to fully contextualize the reader about the technologies and issues being studied. Sections 3.1 and 3.2 introduces the two analyzed frameworks, and explains their purpose, features and the

methodologies behind them.

Chapter 4 presents the Progressive Photon Mapping algorithm, here being used as a case study. An initial background on ray tracing and its evolution is given, followed by the presentation of the algorithm itself, and the evolutions it has suffered.

Chapters 5 and 6 focus on the actual work developing the case study, particularly the implementation using StarPU, and their subsequent profiling. Initial scalability results and their analysis are presented, along with some considerations regarding the performance of StarPU, as well as the issues encountered along the way.

Finally, Chapter 7 presents the final conclusions of this work, and leaves suggestions of future work on topics that, although interesting, were not fully covered during this dissertation.

Chapter 2

Technological Background

2.1 Parallel Computing Background

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 Law. 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, usually with some form 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.

Writing parallel algorithms is generally not a trivial task compared their sequential counterpart. Several classes of problems are introduced to the programmer such as deadlocks, data races and memory consistency. Some of this problems may cause applications to behave unexpectedly under certain conditions. That, along with the fact that multiple execution lines are being processed in a sometimes very loose order, is also what makes the debugging of these applications much harder.

This is not helped by the fact that current development environments are still mostly unequipped to aid the programmer in such tasks. Support for debugging and profiling is still somewhat outdated in various cases, as should be expected from a paradigm that has not become mainstream until recent years.

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 is extremely useful with highly data-parallel applications.

This data-parallel model is also behind the architecture of GPUs, but at a much higher degree. While the concept is the same (applying the same instruction to a set of values, instead of a single value at a time), the architecture of a GPU, particularly a CUDA device, relies on the usage of several simple cores, grouped in multiple Streaming Multiprocessors, to achieve higher degrees of parallelism, and process massive amounts of data. This makes GPUs more suitable for tasks with a high degree of data-parallelism, and not the ideal candidate for more irregular problems, where its more difficult to find parallelization points in order to take advantage of the SIMD model.

Although the hardware of a GPU is still tightly coupled with graphics processing and rendering, there have also been several advances in its usage as a general computing device (GPGPU).

2.2.1 Fermi Architecture

The Fermi architecture was an important milestone of GPUs technology, as it was one of the first generations targeted directly towards GPGPU and high performance computing, rather than purely 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 many fields that require increased precision. 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.

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*. Since the scheduling is made directly via hardware, the switch between threads is nearly free, at least when compared with software 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

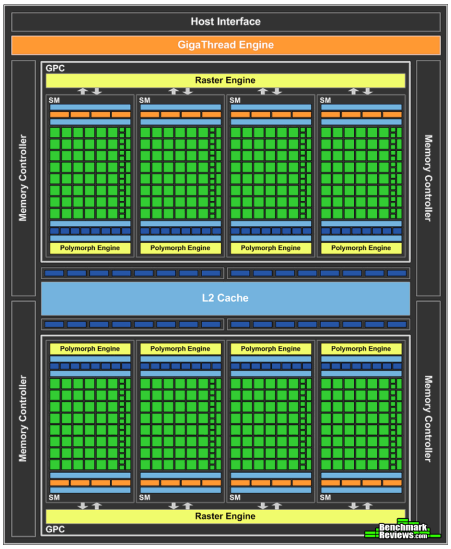


Figure 2.1: Overview of the Fermi Architecture

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 GPU.

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-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 programming model has been extended with the addition of dynamic parallelism, allowing CUDA thread to spawn new threads, a feature not possible with

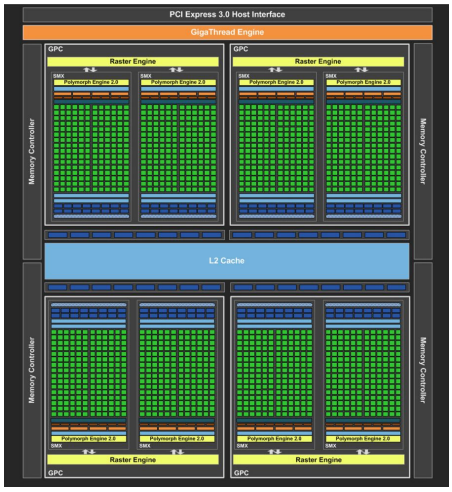


Figure 2.2: Overview of the Kepler Architecture

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.

This shows a clear evolution over the previous generation, and a response to the increasing demand for highly parallel computational provided by GPUs.

2.3 Heterogeneous Platforms

By combining multiple devices such as CPUs and GPUs, the obtained platform can obtain a much higher theoretical power. Its peak performance can be defined as the sum of its parts. But in practical terms, it becomes much harder, if not impossible at all, to achieve performances near this peak value.

On a homogeneous machine, the actual peak performance is limited by additional factors such as pipeline bubbles[1], or memory access times, which is aggravated in memory bound algorithms [9].

[1] a delay in the instruction pipeline, required to solve data, structure or control hazards, and limiting Instruction Level Parallelism (ILP)

When dealing with multiple devices, an additional layer of complexity is introduced, as these different devices need a mean of communication between each other. Usually, a HetPlat, such as the one represented in Figure 2.3 can be seen as a distributed memory system, since each device has its own memory hierarchy. Communication is thus needed for synchronization between tasks, and any required data transfers.

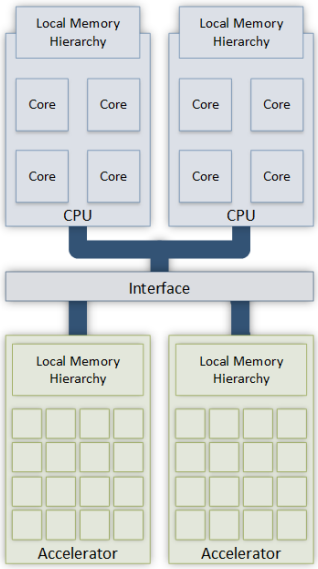


Figure 2.3: Example diagram of a HetPlat

In the case of GPUs, communication is done via a PCIe bus. Although this technology has evolved greatly over the last recent years, this still proves to be one a potential bottleneck for accelerating applications with a GPU.

Even disregarding PCIe devices, a single computational node can also be composed of multiple CPU sockets, connected to each other via a bus such as Quick Path Interconnect (QPI). This bus connects not only the multiple sockets but also the memory banks possibly associated with each one, as these types of machines are generally associated with a NUMA memory hierarchy, where each CPU is directly connected to a different memory bank. While access to all banks is shared between all CPUs, access costs are not uniform. They depend on the speed of the bus used, and also on the actual path a request has to make from the socket it

originated from until it reaches the desired memory bank.

While this is a completely transparent process to the programmer, it can be another source of poor performance, if most or all data might ends up pinned to a single memory bank, requiring all other sockets to share access to it. This is commonly disregarded by programmers, who end up treating these systems as if their main memory is unified. It is possible to control the affinity of both data and task in a NUMA system. For instance, the `hwloc`[10] library provides access to the topology of the system, as well as an API to control their bindings, effectively allowing control over each individual CPU and data bank.

However, libraries such as `hwloc` are very low-level ones, thus difficult to learn and use properly. For complex problems, worrying about low-level optimizations such as memory and core affinity can lead to over-optimization, making the application much more complex, and harder to debug or change. Instead, these libraries should be used as a control layer for other libraries or frameworks to work on top of, and allowing a more developer-friendly access to the desired features.

2.4 Heterogeneity and the Future

As stated before, current HetPlats are usually composed with CPUs and GPUs communicating through a PCIe slot. Newer technologies such as the Intel MIC also use this bus. Both of these accelerators employ their own memory hierarchy, different from the main one, and the latter one is actually *x86*-based, making it architecturally more similar to ordinary CPUs. However, as history has shown, this cannot be assumed as the global model for heterogeneous computation.

As an example, power efficiency has recently become a more prominent factor in technological evolution. Judging from that, it should be expected that short term evolution will yield more power efficient devices, rather that just providing a higher peak performance or increasing the number of cores. Power efficiency can also be regarded as a relevant factor by a scheduling framework, by making

decisions no only based on overall execution time of the application, but also based on its costs in terms of energy consumption. Other metrics can be though of, and their importance increased or decreased, depending on technological trends at each time.

All these factors emphasize the fact that optimizations based solely on architectural details (for example, software pre-fetching, core affinity and memory affinity) are not desirable to be coupled with an application if is desirable for it to perform well in the future, instead of becoming less performant. Such optimizations create tight dependencies between a program's performance, and the actual architecture(s) being used during development. So it becomes desirable that such issues be handled by a more generic tool, built in a modular way so that components can be plugged/unplugged or updated, and applications easily portable.

Chapter 3

Frameworks for Heterogeneous Platforms

The challenge of dynamically scheduling workload of an application across an entire heterogeneous system is an ambitious one, and has been focus of more attention as HetPlats emerge as efficient solutions for high performance computing. Among the several frameworks that have been proposed to target this and other issues are GAMA and StarPU, which are here presented in more detail. While both of them have common points in their philosophy (GAMA is to some degree inspired by StarPU), they use slightly different approaches to manage HetPlats.

3.1 GAMA

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.

One of the main focuses of GAMA is to optimize irregular applications, which are particularly harder to make estimations on. In an irregular application, input data-sets and their sizes cannot be used to make assumptions about how the task will perform under a similar, but not equal set of input. This does not hold true for regular applications, which is what makes their scheduling easier. As such, irregular applications can be more difficult to extract parallelism from, especially when workload is distributed, as is the case with HetPlats and their frameworks.

3.1.1 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 3.1 illustrates how GAMA understands the memory hierarchy of a HetPlat.

Memory Consistency

Communication between memory spaces of different devices 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.

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.

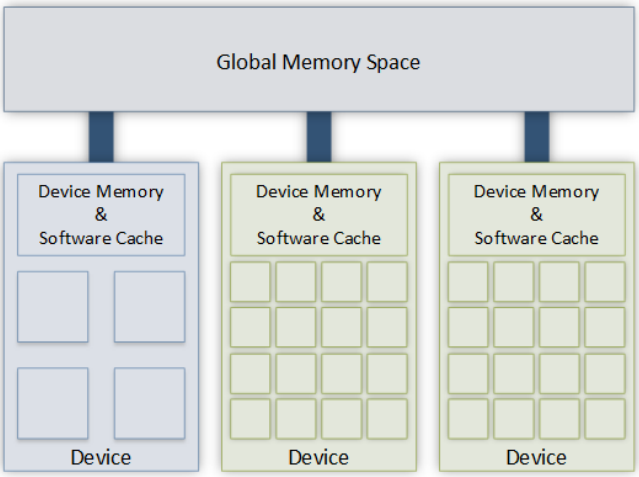


Figure 3.1: GAMA Memory Model, analogous to the one shown in Figure 2.3

3.1.2 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. Currently GAMA supports only a single-host, providing of taking advantage of multiple computational nodes. As such, the host represents the top-most hierarchy layer in GAMA's execution model

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.

Scheduling

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 devices.

Internally, GAMA uses a variant of the Heterogeneous Earliest Finish Time (HEFT) scheduling algorithm [11], which uses the computation and communication costs of each task, in order to assign every task to a device in such a way that minimizes the estimated finish time of the overall task pool. This variant of HEFT

instead attempts, every time it is applied to the task pool, to make a decision so that tasks on the multiple devices take the closest possible to execute [12].

3.1.3 The Polu Case-Study

While studying the details of the framework, some tests were 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 small, 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 3.1.3.

This was no more than a small hands-on study in order gain a better understanding of the usage of the framework, but not its performance, as the finite volume method employed is just as simple as the already implemented test samples, meaning that an interesting performance analysis would not be possible.

The application, here called `polu`, was already the subject of a parallelization study in [13], 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.

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.

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. Despite this, the algorithm is still very irregular in terms of memory access patterns, because the mesh input generator `gmsh`, suffers 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 [13],

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.

3.2 StarPU

StarPU [6] is a unified runtime system consisting on both software and a runtime API that aims to allow programmers of computational intensive applications to more easily exploit the power of available devices, supporting CPUs and GPUs.

Much like GAMA, this framework frees the programmer of the workload scheduling and data consistency inherent from a HetPlat. Task submissions are handled by the StarPU task scheduler, and data consistency is ensured via a data management library.

However, one of the main differences comes from the fact that StarPU attempts to increase performance by carefully considering and attempting to reduce memory transfer costs. This is one by using history information for each task and, accordingly to the scheduler's decision of where a task shall be executed, asynchronously prepare data dependencies, while the system is busy computing other tasks. The task scheduler itself can take this into account, and determine where a task should be executed by taking into account not only execution history, but also the estimation of data transfers latency.

3.2.1 Terminology

StarPU uses a well defined terminology to describe its libraries and API:

Data Handle
: References a memory block. The allocation of the required space, and the possibly required memory transfers to deliver information to each device can be completely handled by StarPU;

Codelet
: Describes a computational kernel that can be implemented in one or more architectures, such as CPUs, CUDA or OpenCL. It also stores information about thew amount and type of data buffers it should receive;

Task
Is defined as the association between a codelet and a set of data handles;

Partition
The subdivision of a data handle in smaller chunks, according to a partitioning function, which can be user defined;

Worker
A processing element, such as a CPU core, managed by StarPU to execute tasks;

Scheduler
The library in charge of assigning tasks to workers, based on a well defined scheduling policy.

3.2.2 Task Scheduling

The framework employs a task based programming model. Computational kernels must be encapsulated within a task. A given kernel can be implemented in multiple ways (i.e. for CPUs or for CUDA), and StarPU will handle the decision of where and when the task should be executed, based on a task scheduling policy.

Data manipulated by a task is automatically transferred as needed between the various processing devices, ensuring memory consistency and freeing the programmer from dealing directly with scheduling issues, data transfers and other requirements associated with it.

Previous work by the StarPU development team indicates that one of the most important issues with scheduling is about obtaining an accurate performance model for the execution time of a task [14, 15]. This is increasingly difficult when data transfers, which the team regards as a key issue, are taken into account, as shown in the latter paper. In it, a data-prefetching implementation for GPUs is present, and asynchronous data request capability is introduced as part of the

StarPU library, with the goal of preventing computational units from being stalled waiting for data.

3.2.3 Dependencies

StarPU automatically builds a dependency graph of all submitted tasks, and keeps them in a pool of *frozen tasks*, passing them onto the scheduler once all dependencies are met.

Dependencies can be implicitly given by the data manipulated by the task. Each task receives a set of buffers, each one corresponding to a piece of data managed by StarPU data management library, and will wait until all the buffers from which it must read are ready.

This includes the possible data transfers that are required to meet dependencies, in case different tasks that depend on the same data are scheduled to run on different computational nodes. StarPU will automatically make sure the required data transfers are made between each task execution to ensure data consistency.

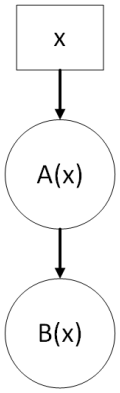
In addition to implicit data dependencies, other dependencies can be explicitly given in order to explicitly force the execution order of a given set of tasks.

Data Access Modes

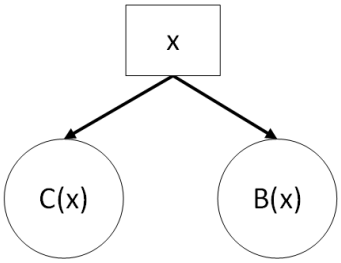
Each data dependency that is explicitly defined in a task can have a different access mode. Data can be used in read-only, write-only or read-write mode. This access mode does not serve the purpose of ensuring memory correctness. It is used to soften task dependencies by using a *Multiple Readers / Single Writer* model in dependency calculation.

This model describes a type of mutual exclusion pattern where a data block can be concurrently accessed by any number of reader, but must be exclusively accessed by a writer. StarPU uses this concept to further optimize data depen-

dency calculations. If multiple scheduled tasks depend on the same data handle, but only with reading access, then that dependency should not block the multiple tasks from running concurrently (see Figure 3.2. Temporary copies of the data can be created, possibly on different computational units, and later discarded immediately, since a read-only buffer is assumed to remain unchanged at the end of a task.



(a) Since A writes to x, B must wait for it to finish to solve the dependency



(b) Neither B nor C write to x, so they can be ran concurrently

Figure 3.2: Access mode illustration. Task A requires read/write access to x, while tasks B and C require read-only access

3.2.4 Virtual Shared Memory

The approach used by StarPU when it comes to memory management is simpler than the model employed by GAMA. The purpose is the same: to automatically manage memory allocations and transfers on all devices. This not only frees the programmer from the work of manually managing memory between tasks, but it also has the potential to lower the cost of such operations. StarPU manages memory by forcing the user to declare data handles to their data. These handles are used as arguments for tasks, allowing the scheduler to allocate and transfer all required data buffers to the correct computational unit prior to the task execution.

3.2.5 Multi-threading

In order to have an accurate view of the topology of the system, the framework can optionally use the `hwloc` library[1] [10] to detect the structure of the architecture, including all CPU sockets, NUMA nodes and cache hierarchy.

A tree is created representing the complete hierarchy of the system. Latest version of the framework also introduce support for parallel tasks, with the concept of combined workers. These workers exist in cases where the system has multiple computational units in the same node (such as CPU sockets where multiple cores share a cache hierarchy). In these situations, and if StarPU is using a parallel-task-aware scheduler (currently only `pheft` and `peager` exist), it is possible to specify the maximum degree of parallelism for a function. A combined worker can then be assigned to such tasks using, for example, OpenMP to take advantage of the multiple available cores.

3.2.6 API

Two API versions are made available. The high-level, `pragma`-based[2] API. This exposes StarPU's main functionality with a set of directives that can be added to the code in order to embed StarPU within it. It is particularly suited for less experienced developers, or developers who simply need to focus completely in the algorithmic issues with less or no knowledge about the underlying parallel hardware being used underneath.

The directives can be easily disabled, restoring the application to its original, StarPU-free nature, which also makes it ideal to add StarPU into already existing code, or applications that require a large degree of portability.

The low-level version is a more verbose one, which is actually used internally

[1] `hwloc`, or Portable Hardware Locality, is a package that provides access to an abstract topology of modern computational architectures.

[2] Directives inserted within the code, to instruct the compiler about how to process input. Can be used to extend the compiler and provide additional features, functionality or optimizations

by the high-level one, and provides a greater degree of control over what StarPU does.

This trade-off is not uncommon, with many existing libraries besides StarPU supporting both API levels. High level versions are designed to remove complexity and accelerate development time. They are often a subset of an underlying low level version, delivering only the more common features. More experienced developers should be able to achieve better results with a lower level API, with the cost of additional development time.

3.2.7 Performance Model

Most schedulers available with StarPU are history based. This relies on the programmer to configure a performance model for each defined codelet, in order to allow the framework to identify it, and use on-line calibration. Calibration results will be stored in a centralized directory, and inform StarPU about how each codelet behaves on each different device, with a certain input size.

Once a good calibration has been obtained (with a minimum number of executions of a task), StarPU can schedule tasks more efficiently, according to the policy defined by the scheduler. The list of the available scheduling policies is the following:

eager The default scheduler, using a single task queue, from which workers draw tasks. This method does not allow to take advantage of asynchronous data prefetching, since the assignment of a task is only done when a given worker requires so;

prio Similar to eager, but tasks can be sorted by a priority number;

random Tasks are distributed randomly, according to the estimated performance of each worker. Although extremely naive, this has the advantage of allowing data prefetching, and minimizing scheduler decision times;

ws (Work Stealing) Once a worker becomes idle, it steals a task from the most loaded worker

dm (Deque Model) Performs a HEFT-like strategy, similarly to GAMA (see Section 3.1.2). Each task is assigned to where the scheduler thinks its termination time will be minimized;

dmda (Deque Model Data Aware) Similar to **dm**, but also taking into account data transfer times;

dmdar (Deque Model Data Aware Ready) Similar to **dmda**, but tasks are sorted per-worker, based on the amount of already available data buffers for that worker. Devices where most or all data buffers a task depends on are more likely to receive that task, in order to attempt minimization of data transfers;

pheft (parallel HEFT) similar to **heft** (which is deprecated in StarPU, with **dmda** working in a very similar way), with support for parallel tasks

peager similar to **eager**, with support for parallel tasks

Additionally, schedulers are built as a pluggable system, allowing developers to write their own scheduling policies if desired, and easily use them within StarPU.

3.2.8 Task Granularity

The granularity of a task in GAMA can be automatically defined and readjusted with the use of a dicing function, that recursively adjusts it to find the best case scenario for each particular device. This feature is not available in StarPU, where granularity has to be defined manually by the programmer.

The API gives the ability to subdivide a data handle into smaller children, based on a partitioning function. This partitioning can be defined as a simple vector or matrix block partitioning, but more complex and custom methods can be defined.

After partitioning, each children can be used as a different data handle. This means that in order to operate in chunks of data at a time, one has to first partition data according to a user defined policy, and later submit multiple individual tasks, using each child data handle individually.

3.3 Comparison

Given that HetPlats can suffer major changes in the future, due to the constant technological evolution, a highly important feature of an application or framework is its modularity, so that individual features can be updated to meet the requirements of the constantly evolving computing platforms. StarPU seems to use this philosophy to some extent, with modules such as the scheduler itself being completely unpluggable and rewritable by a developer. It also provides the ability to assign user defined functions to make decisions within the framework, such as how to partition data.

Inversely, GAMA still does not provide a stable and organized API

3.3.1 Usability

From a developer's point of view, StarPU, being written in *C* provides an a clear but somewhat outdated API, in some aspects resembling `UNIX`-like libraries. More modern languages such as $C++$, in which GAMA is written provide less verbose and more structured code. The choice for the *C* language is possibly related to better compatibility and portability, but seems to somewhat limit the language, and even expose some unexpected behaviours of the framework (see Sections 3.3.3 and 3.3.3).

Even though GAMA does not yet provide a solid API to work from, but rather a more confusing architecture, it can still be considered harder to work on, although there is plenty of room for improvement, and once development advances, and

reaches a more stable position, it can have the conditions to be a much more usable framework than the low-level one provided by StarPU

3.3.2 Scheduling

Both StarPU and GAMA employ a variant of the HEFT algorithm for scheduling on heterogeneous systems, although StarPU gives much more emphasis to the latency caused by memory transactions, and can also support additional scheduling policies to be used.

3.3.3 Memory Management

GAMA has the ability of recursively changing the granularity of a task to adapt it to the characteristics of a device. This is presented as an important step by GAMA to automate decisions by the scheduler. Without this, granularity has to be manually defined, by subdividing the domain in arbitrarily sized chunks and process each one as an individual task. This is not only a cumbersome task for the developer, but also a possible weakness, as the ideal task granularity is not equal from system to system, or algorithm to algorithm, and may not be easy to determine without intensive testing and manual tunning.

This seems an extremely important feature in GAMA, at least from the usability point of view, as the task of finding the ideal granularity is thus automated by the framework.

Type Safety

Perhaps one of the most important limitations of StarPU's API is the fact that its task submission methods are not type-safe[3]. By definition, the C language (in which StarPU is implemented and exposed to the programmer) is by only type safe

[3]the extent to which a language discourages or prevents type errors

to a certain extent, since workarounds are often used in the language to achieve results similar to polymorphism or runtime casting.

This is the case with StarPU, resulting in an API that can be mistakenly used by the programmer. Each codelet defined in a program specifies how many data buffers its tasks will depend on, and their access modes. However, since data buffers are used only through their data handles, which are completely generic, no explicit type checking is made to ensure that the correct types of data handles is passed, and that they are received properly within the task. For example, a task may be expecting to receive buffers X and Y of completely different sizes and types. But on task submission, their order might be reversed, resulting in runtime errors which might be extremely difficult to trace.

The main result of this is a weak task submission API, since it can easily lead to runtime errors. More experienced developers might have enough understanding to easily identify these problems. But technical issues such as type safety should not have to be addressed by the developer, as they can pose serious problems to development time, but could be easily identified by a compiler.

Consistency

StarPU ensures consistency of all data assigned to it via data handles, but only within managed managed by its scheduler. The issue here is that, while the API allows the creation of a data handle associated with an already allocated data structure (usually pinned to main system memory), it is not ensured that tasks will write to that actual memory, and not a StarPU-managed copy. This has the side effect of consistency not being ensured outside of the context of a task. Writing to a buffer via conventional methods can thus be considered dangerous.

It is also impossible to change the size of a data buffer once it has been assigned to StarPU. When this is a requirement, is to destroy and redefine the buffer, which is not a particularly efficient method, and may introduce additional bottlenecks when used between task submissions.

On these subjects, GAMA seems to provide more powerful capabilities. All data that is to be managed by GAMA's unified memory system must be encapsulated in a wrapper that processes all accesses to it. While it is not documented how GAMA actually behaves when data is accessed outside of a job, this model should provide a more transparent solution for consistent memory access.

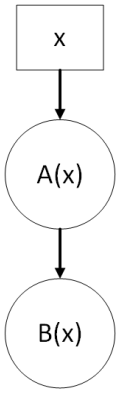
Data Access Modes

Another caveat of the usage of data handles is that it can lead to incoherent results, again due to possible and easily made developer mistakes. StarPU relies on the estimation of data transfer times to decide where to schedule a task to. An important factor of this comes from the access modes required for each data buffer in a task, as explained in Section 3.2.3.

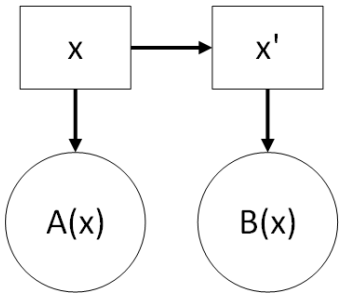
If a data buffer is declared as read-only within a task, it is assumed to be unchanged by that same task, this other tasks depending on it have that dependency already met. The caveat here is that a read-only buffer is only so for the framework, but not actually read-only for the programmer, or for the C compiler itself, and data can actually be overwritten for a read-only buffer. Although that is probably related to a development mistake, it can easily happen nonetheless

Adding to this is the fact that StarPU might create additional copies of data buffers to solve dependencies faster across multiple devices. Figure 3.3 exemplifies this problem. Tasks A and B have read-only access to the data buffer x, although as explained, both of them can write to it at will. If both tasks are scheduled to the same device (Figure 3.3a, they will be executed in order, and StarPU will reuse the initially existing memory, without the need to create temporary copies of the buffer. In this case, if task A writes to x, task B will later see these changes, since memory is shared.

In Figure 3.3b, the two tasks are scheduled to different devices. Since StarPU assumes x to be read-only, a copy of it, x' can immediately be created in the additional device. In this case, both tasks will run concurrently, with their own



(a) A and B are executed in order



(b) A and B are scheduled to different devices, creating a temporary copy of x, and running both tasks concurrently

Figure 3.3: Illustration of a possible mistake due to dependency management. Both A and B have read-only access to x

local copy of x. Task A will still write changes to this buffer, but task B will not see them.

Chapter 4

Progressive Photon Mapping Algorithm 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 [16], 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 Equation (4.1)

$$L_s(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + \int_{\Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i)(\vec{\omega}_i \cdot \vec{n}) d\omega_i \tag{4.1}$$

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

Ray Tracing is the designation of a family of algorithms that simulate ray interactions with an environment in order to determine the visibility of two points in space. Generally, this technique is used to compute light interactions within a three-dimensional scene, and obtain a two-dimensional image with a rendering of that scene. A high degree of visual realism can be achieved by these techniques, but also with greater computational costs.

Ray tracing algorithms are capable of simulating most optical effects, such as reflections, refractions, and scattering, although the actual set of simulated effects, and their overall quality differ between the multiple ray tracing algorithms that have already been developed.

Large efforts have been made recently to improve ray tracing technique, with several of them being performance-related, by using parallel architectures such as GPUs, which have always been associated with image processing. Some efforts are already showing advances in the area of real time ray tracing, which was unfeasible until very recently [17, 18]

4.2.1 Overview

The first use of Ray Tracing algorithms was with the introduction of the later called Ray Casting technique by Arthur Appel in [19]. The idea consisted of casting one ray from an origin point (usually called the eye, or camera) through each pixel of the image, and find the closest object in the scene that blocks the path of the ray. The material properties and light effects in the scene would then determine the final color of the pixel.

Later advances, such as [20], introduced the recursive ray tracing algorithm. In traditional ray casting, each ray would end after the first hit. This was a limiting factor that prevents that approach from dealing with effects such as reflections. Recursive ray tracing solves that, by recursively tracing more rays after each hit. When a ray hits a surface, it can generate new reflection or refraction, depending on the properties of the material hit. Shadow rays can also be cast in the direction of the light sources. If a shadow ray is blocked by an opaque object before reaching the light source, it means that surface is not illuminated by that light source, and thus is shadowed. This technique is illustrated in Figure 4.1

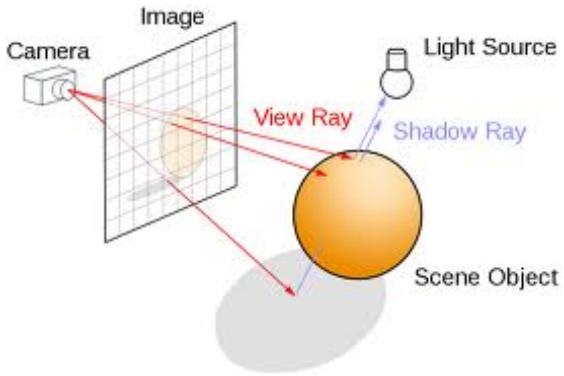


Figure 4.1: Ray Tracing

It is also not uncommon to approximate the rendering equation by using combinations of more than one method, such a Ray Tracing, Radiosity or Metropolis Light Transport [21, 22]. 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, so the combination of multiple method can combine the advantages of each one.

Traditional 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. Nowadays, most ray tracing rendering algorithms are in fact bidirectional, having photons shot from the light sources and their interactions computed in addition to regular eye paths.

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 a ray tracing method that intends to approximate the rendering equation, first proposed as a global illumination technique in 1996 [23], and works as a two-pass algorithm, working as an extension to ray tracing that allows the efficient computation of caustics[1] and indirect illumination of surfaces.

Unlike other algorithms such as Path Tracing or Metropolis Light Transport, this is a biased rendering method, meaning that a finite for a finite number of traced photons, the resulting estimation will always be different from the correct solution. However this can be worked around by increasing the size of the photon map structure used in the process, or by using variations of this technique, such as Stochastic Photon Mapping.

4.3.1 Algorithm

The original approach to photon mapping consists simply on a two step algorithm. One step to generate a structure with the illumination information for the scene, called the photon map, and a second step to trace rays from the camera to interact with the scene and the photon map, contributing to the final pixels of the generated image

First Step: Photon Map Construction

In this step, a large number of photons must be traced, starting from the existing light sources in the scene (see Figure 4.3). The tracing of a photon is just

[1] The light effects caused by light reaching a diffuse surface after being reflected or transmitted by a specular one

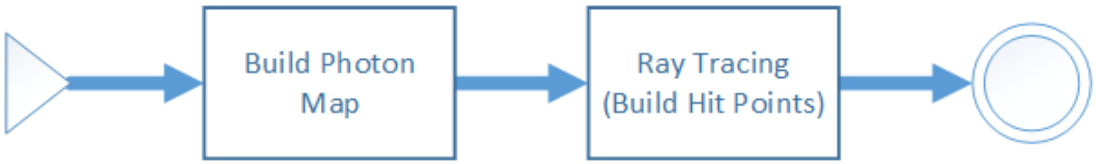


Figure 4.2: High Level Flowchart of Photon Mapping

like the tracing of a regular ray, interacting with the scene according to BRDF function, in a process similar to path tracing. Every hit by a photon is stored in a structure called a photon map. The original proposal [23] uses two different photon maps, the extra one being a higher density one used for the rendering of caustics. This is done by emitting paths towards specular objects in the scene, and storing them in the photon map as diffuse surfaces. The usage of this extra photon map is not, however, required for the implementation of the algorithm, and serves only as a way of providing additional quality in the rendering of caustics. As such, it was not considered during the implementation in this work.

The photon map structure generated serves as an approximation of the light within the entire scene. As another optional extension, shadow photons can also be cast during this step, which will reduce the amount of shadow rays necessary in the second step to correctly reproduce shadows.

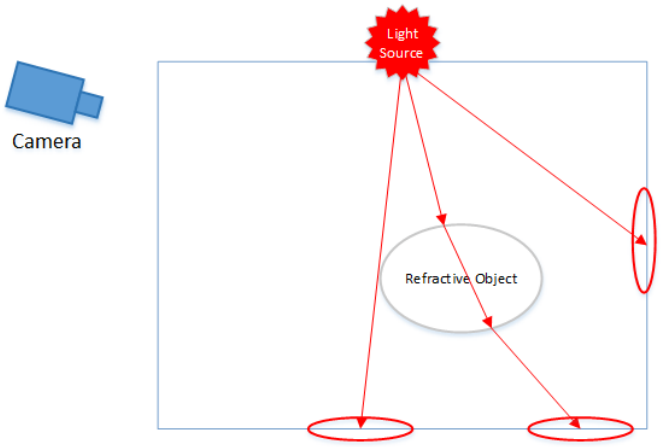


Figure 4.3: Overview of the first step of Photon Mapping

Second Step: Rendering

For the final image render, Monte Carlo ray tracing [24] can be used to traced rays from the camera position into the scene, as illustrated in Figure 4.4. During this step, the information in the photon map structure can be used to estimate the radiance of a surface, by using the N nearest photons to the hit point, that are within a sphere of radius r centered in the hit point x. The radius r is then used to estimate the surface area covered by the sphere, which is approximated to πr^2.

The photon map proves useful during this step, not only to increase performance, but also to allow the modeling of some light effects that are not present or are inefficient to process without such a structure.

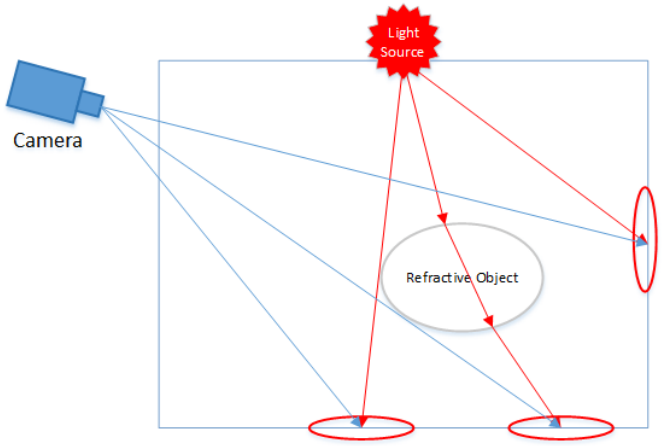


Figure 4.4: Overview of the second step of Photon Mapping

4.3.2 Applications

One particular case where photon mapping provides advantages over other methods is when light is being transported from a light source travels along a specular-to-diffuse path before reaching the eye (LSDE path), such as the one illustrated in Figure 4.5, before reaching the eye. This is what is commonly known as a caustic, such as, for example, the shimmering light seen of 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.

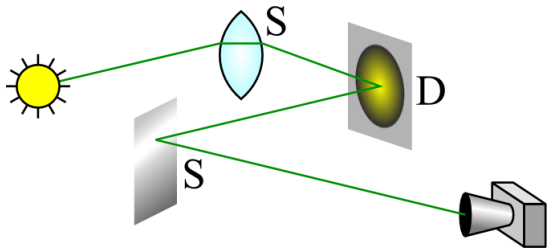


Figure 4.5: Illustration of a Specular to Diffuse to Specular path (SDS)

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 an angle different from the one it would have take had it been reflected by the surface. This is visible in materials such as marble, or skin, and can be seen in Figure 4.6

Figure 4.6: Example of a scene displaying Subsurface Scattering

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

4.4 Progressive Photon Mapping

The main problem with the original proposal for photon mapping is that the quality of the final result is limited by the size of the photon map, which in turn has its size limited by the amount of memory available.

Since effects such as caustics are simulated by directly using the information from the photon map, it is necessary to use a very large number of photons in order to avoid noise in the rendering. Thus, the overall accuracy is limited by the available memory. In other words, the accuracy of photon mapping is not only computationally bounded, but also memory bounded, while usual unbiased methods are only computationally bounded.

[25] proposes a progressive approach to photon mapping, which makes it possible to simulate global illumination, including the effects provided by traditional photon mapping, with arbitrary accuracy, and without being limited by memory. This is done by using a multi step algorithm instead of a two step one, where the first pass consists of a ray tracing step to capture a collection of hit points in the scene, and later multiple photon tracing steps are processed iteratively, with each new iteration improving the accuracy of the result in order to converge to an accurate solution, but without storing photon maps from previous iterations.

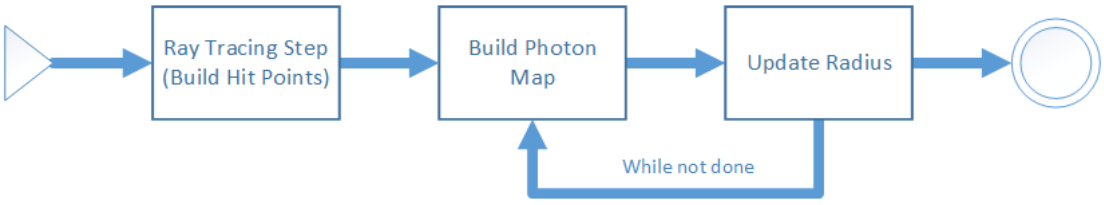


Figure 4.7: High Level Fluxogram of Progressive Photon Mapping

4.4.1 First Step: Ray Tracing

The first step is a standard ray tracing step, used to find all surfaces visible in the scene. Rays are traced through each pixel of the image plane, in order

to find all visible surfaces in the scene. For each ray, all hits with a non-specular component in the BRDF function are stored. This includes storing the hit location (x), the ray direction($\vec{\omega}$) and the pixel it originated from. Additionally, data for the progressive estimation is also included, such as a radius, intercepted flux, and number of photons within the defined radius.

4.4.2 Next Steps: Photon Tracing

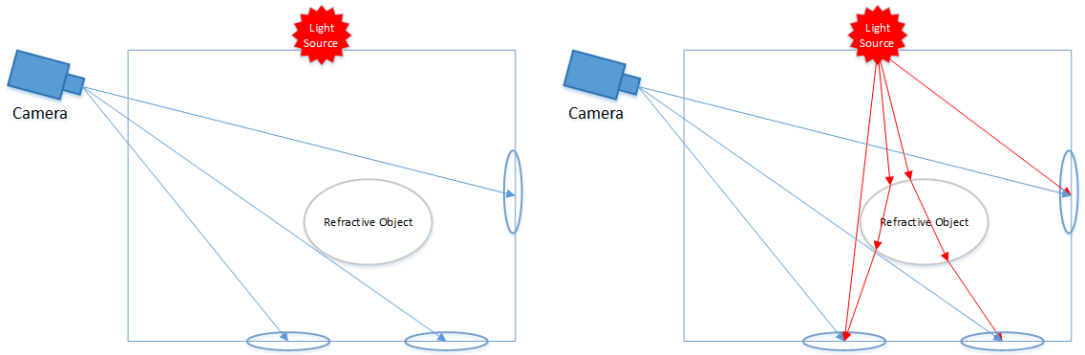
After the initial ray tracing step, an iterative process begins. Each iteration, a given number of photons is traced into the scene, building a photon map. At the end, all hit points stored from the initial step are processed, to find all the photons in the map that are within the radius of that hit point. These photons are used to refine the estimate of the illumination of the hit point.

Once this contribution is computed, the photon map is no longer needed, and can be discarded, before the next iteration repeats the process.

This provides two key advantages over the original, two-step approach. The total amount of photons traced is not limited by memory, but only by the amount of iterations that are computed. An arbitrary number of iterations can be used, without requiring any additional memory at all, and resulting in a better quality result. Also, after each photon pass an image can be rendered, and the progressive result can be shown while the image is progressively improved, instead of having to wait for the entire algorithm to finish.

4.4.3 Radiance Estimation

Traditional photon mapping estimates radiance by using the density of photons, given by Equation (4.2). This is based on locating the nearest N photons within a sphere of radius $R(x)$. The surface are is assumed to be flat, and the surface area is approximated to $\pi R(x)^2$. In progressive photon mapping, using this estimation may result in different iterations having different estimations for the same hit



(a) First step: Ray tracing (b) Next steps: Photon Tracing

Figure 4.8: Overview of Progressive Photon Mapping

point.

$$d(x) = \frac{N}{\pi R(x)^2} \tag{4.2}$$

To solve this, the estimations from each iteration can be averaged to obtain a more accurate estimate. However, the final result will not be more detailed than the result of each individual photon map, which is not desirable. Also, as the radius $R(x)$ is constant throughout the iterations, small details within that radius cannot be correctly solved, making the overall accuracy limited by the size of each individual photon map.

The solution to this, also proposed in [25] consists of combining the estimate from each photon map in such a way that the final estimation will converge to a correct solution. The key technique is to reduce the radius r at each hit point, for every new photon map computed.

Radius Reduction

Assuming each hit point has a radius $R(x)$, and that $N(x)$ photons have already been accumulated in it, after a new photon tracing step, resulting in $M(x)$ new photons within the radius $R(x)$, the new photon density $\hat{d}(x)$ can be given by

Equation (4.3)

$$\hat{d}(x) = \frac{N(x) + M(x)}{\pi R(x)^2} \tag{4.3}$$

The radius reduction step is about computing a new, smaller radius $\hat{R}(x)$ for each hit point, such that the amount of photons within the new radius $\hat{N}(x)$ is greater that the amount of photons that was present in the previous radius. This ensures that the final result is increasingly more accurate, and converges to a correct solution. The radius reduction is illustrated in Figure 4.9.

The proposed approach in [25] simplifies this by using a parameter α to control the fraction of photons to keep after an iteration. Therefore, $\hat{N}(x)$ can be given by Equation (4.4).

$$\hat{N}(x) = N(x) + \alpha M(x) \tag{4.4}$$

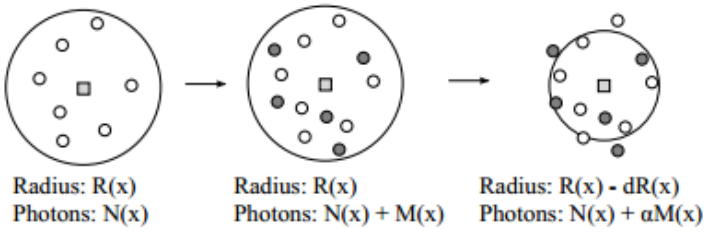


Figure 4.9: Radius Reduction after a Photon Tracing step

The final radius $\hat{R}(x)$ can be computed by combining Equations (4.2) to (4.4), and as shown by the original work on the subject, can be given, for each single hit point, by Equation (4.5).

$$\hat{R}(x) = R(x) - dR(x) = R(x)\sqrt{\frac{N(x) + \alpha M(x)}{N(x) + M(x)}} \tag{4.5}$$

4.5 Stochastic Progressive Photon Mapping

Progressive photon mapping still does not address the problem of computing the average radiance of an unknown region. While progressive photon mapping allows the computation of the estimated radiance on a given point x stored as a hit point, it does not allow the estimation of a different unknown point. This is a problem when trying to simulate distributed ray tracing effects, such as motion blur or depth-of-field.

The solution proposed in [26] presents a new formulation for the progressive radiance estimation, allowing the computation of the correct average radiance over a region.

In practice, the implementation of that formulation consists only in generate a new set of hit points after each photon pass (see Figure 4.10). The local statistics for each new hit point is taken directly from the previous hit point for that same pixel. Original progressive photon mapping generates a set of hit points, and then iteratively uses new photon maps to converge to a correct solution, based on those same hit point. This stochastic approach does not reuse the hit points, but only their local statistics. The results in the proposed work show that this solution provides better results for scenes with complex illumination, and including distributed ray tracing effects, such as motion blur, depth-of-field and glossy interactions.

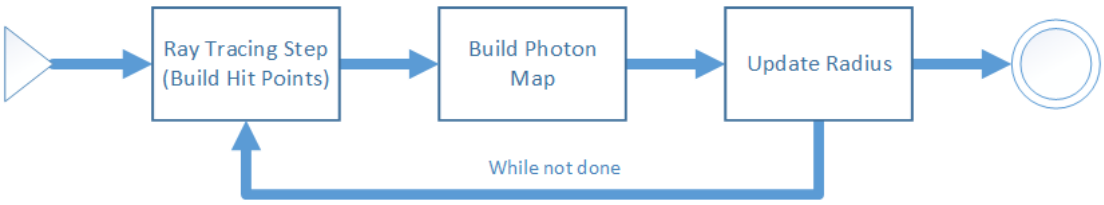


Figure 4.10: Fluxogram of Stochastic Progressive Photon Mapping

4.6 A Probabilistic Approach for Radius Estimation

Another evolution of photon mapping and progressive photon mapping comes from a probabilistic approach for the estimation of photon radius for each iteration, first presented in [27]. The proposed solution, much like original progressive photon mapping, is capable of of computing global illumination without bias, and with no theoretical limit in the amount of photons, allowing an arbitrary number of iterations to be computed.

The new formulation, called PPMPA, includes a probabilistic approach that does not require local photon statistics to be stored. It is shown in the original work that each different photon mapping step of the progressive photon mapping approach can be performed with complete independence from other steps, by using a probabilistic model to compute an estimation of the photon radius for each iteration, instead of gradually reducing it after each photon tracing step (see Figure 4.11.

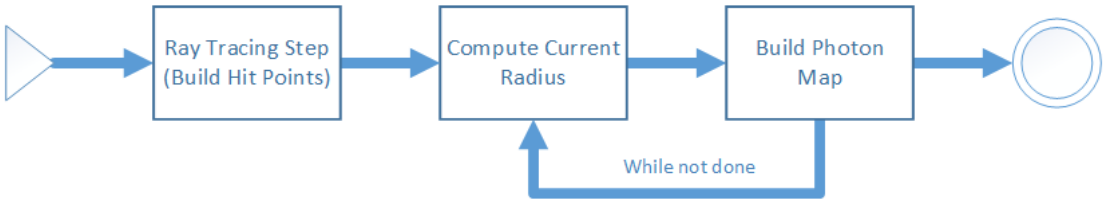


Figure 4.11: Fluxogram of PPMPA

In summary, the probabilistic analysis in the original work shows that for a photon mapping step i, the radius for for a hit point for that step, r_i, can be estimated by Equation (4.6)

$$r_i^2 = r_1^2 \left(\prod_{k=1}^{i-1} \frac{k+\alpha}{k}\right) \frac{1}{i} \tag{4.6}$$

The biggest benefit of this is that the radius computation kernel is not dependent on previous iterations, allowing for multiple photon mapping steps to be

concurrently computed, as shown in Figure 4.12.

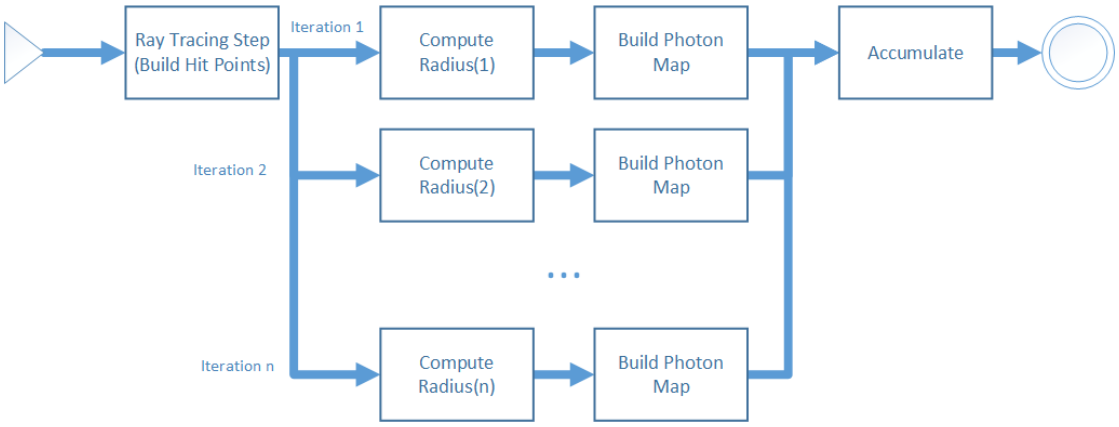


Figure 4.12: Fluxogram of PPMPA with concurrent iterations

The result is a memoryless algorithm that does not require the maintenance of intermediate statistics and allows the possibility of computing multiple iterations, or photon mapping steps, in parallel.

4.7 Summary

Several techniques were presented, from the basic Photon Mapping algorithm, following a progressive approach that enables arbitrary accuracy without being memory limited. Further improvements include a stochastic version that enables additional accuracy in the final result, and a new formulation of the radius estimation that removes dependencies between iterations, effectively allowing their concurrency.

The final result of this is the algorithm Stochastic Progressive Photon Mapping with a Probabilistic Approach, here shortly called SPPMPA, which is the case study employed during the rest of this work.

Chapter 5

Development of the Case Study

The case study was implemented based on an already existing implementation, available at the beginning of the project. This implementation provides the initial Progressive Photon Mapping method, described in Section 4.4, the stochastic extension presented in Section 4.5, and the probabilistic approach for radius reduction in Section 4.6. Support for both CPU and CUDA rendering is also included, although CUDA support was actually not fully completed until a later version. It was implemented on top of the LuxRender project, an open source, physically based and unbiased rendering engine. The source code of LuxRender provides an ideal basis of data structures to implement a rendering algorithm such as photon mapping, and this was exploited by the author of this implementation.

The implementation used here was also based on those same data structures, and other code from LuxRender. The algorithms themselves for the ray tracing and photon mapping steps, radiance estimation, and radius reduction were based not only on the theoretical research work already presented summarily in Chapter 4, but also on the already available implementation

This was helpful to speed up development time, by working with already existing code for the same algorithm, but also to serve as a validation tool, in order to assert whether the final solution, and the individual algorithms within it, produced

a correct result.

The final implementation developed for this work was based on the original one provided at the start of the project, although different and improved is some regards. Three main methods were made available, mostly for the sake of comparison of results and profiling. The first two are simple implementations using CPUs with OpenMP and GPUs with CUDA, respectively. Finally, the third method uses the StarPU framework to handle task management

It should be noted that only two of the presented algorithms were taken in consideration during this work, namely:

PPM (Progressive Photon Mapping)
: This corresponds to the original proposition for progressive photon mapping, described in Section 4.4.

SPPMPA
: (Stochastic Progressive Photon Mapping with Probabilistic Approach) This extends the initial PPM solution to include both the stochastic version, and the probabilistic approach for radius reduction.

There was no attempt to implement any of the intermediate versions (SPPM or PPMPA).

There were also attempts to port the original implementation to run on the MIC platform, whose details are also described in this section.

5.1 Data Structures

Most of the structured used throughout all the implementations were based on the ones used in the originally available implementation, which in turn was heavily supported on the source code of LuxRender. This resulted in the usage of several data structures from that same source code, particularly:

Basic Geometry

The basic geometric structures, such as vector, normals, 3D points and triangles;

Meshes

A collection of vertices, edges and faces that define the shape of a 3D object in a scene;

Scene

The full description of the 3D scene to render, including all meshes, materials and light descriptions associated with it. This scene is read initially by the LuxRender library, which handles the parsing of all data files associated with the scene (mesh descriptions, materials, light sources, textures, etc)

Bounding Volume Hierarchy

A tree structure used to index spatial objects, in this case the objects within the scene, in order to reduce the number of required ray intersection operations. The actual implementation used, a Quad Bounding Volume Hierarchy (QBVH), is an extension of a regular BVH, optimized for a low memory footprint, and SIMD computations [28, 29]. This structure is created at startup by LuxRender, and is used to spatially index all elements in the scene, allowing for faster computations of ray intersections;

In addition to these structures that LuxRender already provided, additional data structures were also required for the implementations described in this section:

Pointer-free Scene

The original scene structures available with LuxRender relied heavily on pointer based structures, which was not adequate for GPU computations. So a custom solution was required in order to store scene information in a compact manner, easily transferable and usable by a GPU

Hit Points

A data structure was required to store information about hit points position,

direction vector, the pixel that it originated from, and about accumulated photon radiance. In practice this was actually split into two different data structures, the first one storing only static information about the hit point, such as position and origin point, and the last one to store incident radiance. This separation allowed a more efficient memory usage, since the two different components are read and written to at different points during the algorithm.

Lookup Table

An acceleration structure used to index the hit points in order to quickly find all relevant hit points to update after a photon trace. This corresponds to all hit points within a given radius of the hit point of each photon, or in other words, all the hit points that photon will contribute to. The structure is implemented with an hash table that spatially indexes the hit points by dividing the 3D space into a grid, where each cell references all hit points that intersect it, based on the current radius.

Ray Buffer

In the original implementation, the total number of photons traced every photon pass was not directly processed at once. Instead, a ray buffer was used to process a specific, pre-configured number of photons at a time, independently from the total number of photons to process during that step. This was mostly to increase coherence on the CPU by processing a smaller batch of consecutive rays at a time.

5.2 Computational Tasks

Throughout all implementations, computations were divided across several kernels with more or less the same responsibilities. This was a concern from the start in order to ensure that all versions remained similar to each other, facilitating the comparison between them.

The approach used during development was mostly inspired by the original provided implementation. This section presents a list of all the computational

kernels that were already present in the original implementation, and that were reimplemented and reworked during the development of the case study:

1. **Initialize Seeds**

   This was not part of the algorithm itself, but was necessary to initialize a seed buffer used for the random number generation process required by the following tasks. The random generation is done using a Tausworthe Generator [30]. Throughout the algorithm, it was required to keep a buffer were each seed was stored. When a random number generation with a given index was required, the respective seed was rewritten to provide the next number in the random sequence. For this process, an initialization of this buffer was necessary, using each index in the buffer as the initial seed.

2. **Generate Eye Paths**

   In this step, eye paths were initialized, based on camera position

3. **Advance Eye Paths**

   Following the initialization of the eye paths, this task would compute their interactions with the scene, while building the set of hit points required by photon mapping. This is analogous to the Ray Tracing step in the photon mapping algorithms presented in Chapter 4.

4. **Update Radius**

   This is a small but necessary task that computes the hit point radius for the subsequent photon mapping step. When using the probabilistic approach, this is equivalent to the computation of Equation (4.6), presented in Section 4.6

5. **Rebuild Lookup Table**

   As explained in Section 5.1, a lookup table is used to index all hit points. This task would build that structure after all hit points were generated by the **Advance Eye Paths** task.

6. **Generate Photon Paths**

   Similarly to **Generate Eye Paths**, this was used to generate a set of photon paths, leaving the light sources into the scene.

7. Advance Photon Paths

During this step, the initialized photon paths were traced within the scene, updating the local accumulated flux of each hit point they interact with along the way.

8. Accumulate Flux

To finish the photon mapping stage, this additional step was separated from the previous one, to compute the final radiance of each hit point,

9. Update Frame Buffer

This maps all hit points, and their final radiance values to the corresponding pixels on the screen, creating a temporary buffer of the image generating during the current photon mapping step

10. Update Film

The temporary frame buffer was merged into a film structure, that represents the final image computed so far by all photon mapping steps. This directly represents the rendered 2D image, and was used to directly display the image if a live preview window was enabled, and to optionally save the current state of the render in an image file.

These tasks map almost directly into the theoretical algorithms for progressive photon mapping. Tasks **2** and **3** represent a Ray Tracing step, where the hit points are computed. This is the first step in the multi pass progressive photon mapping algorithm presented in Section 4.4, and also the first step of each iteration in the stochastic approach Section 4.5. Tasks **4**, **6**, **7** and **8** are the equivalent of a photon tracing step, which corresponds to a full iteration in the original progressive approach.

The remaining tasks are not specified by the photon mapping techniques, but are required for computational purposes. Task **5** represents an important step to make sure that access to each hit point is done efficiently when tracing the photons. By using a lookup table to index hit points, the average complexity in accessing a single hit point is lowered to $O(1)$.

5.3 Implementation

This section presents an overview of the differences and challenges between the multiple versions used.

While StarPU is the actual object of study in this project, initial development was focused on building a CPU only, implementation, similarly to the original version. Following that, an similar approach was done to build a CUDA version, while still avoiding the usage of StarPU or any HetPlat managing system.

The goal of this approach was to have a functional basis of comparison that shared as much of the functionality as possible with a future implementation that took advantage of HetPlats by using StarPU as the task manager.

During development, a large effort was put into trying to keep all computational code encapsulated in such a way that it would be reusable by future implementations. Not only does that help speeding up the development time of future versions, it also allows for more fair comparisons to be made between versions.

5.3.1 Original

The original implementation provides a few different versions of the algorithm, based on the multiple extensions on photon mapping described in Chapter 4. However, only PPM and SPPMPA are considered.

In this approach, tasks are encapsulated within a CPUWorker class and a CUDAWorker class, which implements the ray tracing and photon mapping steps using OpenMP and CUDA, respectively. While the initial PPM version, due to implicit dependency limitations (without the probabilistic approach for radius estimation, each photon mapping iteration is dependent on the previous one), can only run a single worker at a time, the later version (SPPMPA) allows the instantiation of multiple workers. In that case, each worker will share access to a centralized structure that keeps track of how many iterations have been finished

in total.

In practice, the SPPMPA version provided support for running one CPUWorker and two CUDAWorkers[1]. Since the CPUWorker uses OpenMP internally to take advantage of multi-threading, this approach effectively takes advantage of the full power of a multi-core machine with at most two CUDA devices.

When this implementation was first available, however, the CUDA implementation was not yet fully finished, with only the implementation of the photon tracing steps being run on a GPU. This means that the performance when using CUDA is limited by the necessary data transfers required during each iteration. Particularly, when using SPPMPA version, where, following the stochastic progressive photon mapping extension, new hit points are generated after each step, an even greater amount of communication is required, since the GPU has to wait for the new hit points before starting the computation of a new photon pass.

This versions was used only for validation of the later implementations, and an in-depth study of its performance was not considered interesting, as similar versions were to be produced, but with the advantage of being further structured and optimized, and not relying on a third-party code base.

5.3.2 CPU

The first implementation to be developed was one consisting only on the implementation of each task, using OpenMP for parallelism within each individual task. Of all tasks shown in Section 5.2, it should be noted that the following tasks were not parallelizable, and so are only implemented in sequential code:

- Update Radius
- Rebuild Lookup Table

[1]This was not a limitation of the implementation. Actually, extending it to support more than two CUDA devices would be completely straightforward. However, there was no interesting in doing so, as all test machines used throughout the project had at most two CUDA devices available.

- Update Frame Buffer
- Update Film

All other tasks (which in practice represent all the actual code for both the ray tracing and photon tracing steps) were fully parallelizable, as each ray can be independently intersected. The only exception to this is with the **Advance Photon Paths** task. Since a hit point might be simultaneously hit by multiple photons, a small critical section was required for every hit point update. That critical section, however, represents an extremely small portion of the entire task.

In this implementation, some of the original code from LuxRender was also employed. Particularly, the intersection code for a ray, which traverses the accelerating structure indexing the scene, a Quad Bounding Volume Hierarchy (QBVH), searching for the next ray hit. This intersection code is implemented using SSE intrinsic functions, meaning that SSE code was hard coded, instead of being compiler generated. This makes sure that the accelerating structure takes advantage of the optimizations for original BVH structures presented in [28].

Initially only the PPM version of the algorithm was developed. However, that decision was only to allow a gradual evolution, later adapting the code to implement SPPMPA instead. The reason for this is that evolving from basic PPM to one of its extensions is relatively straightforward. The tasks themselves remain almost identical, and most of the changes are related to when and how those tasks are actually called. Thus, the PPM version was not intended to be a finished application, but just a first step towards the final SPPMPA version.

Finally, even though the later algorithm theoretically allows multiple iterations to be run in parallel, this is not supported in the CPU-only implementation. This mimics the behaviour of the original implementation from Section 5.3.1, which was the intended result.

The main goal was to have an application as much similar as possible to the original CPU version, while still sharing task implementations with the future StarPU version.

5.3.3 CUDA

Following the CPU implementation, it was also desirable to produce a CUDA based implementation. Like the previous one, this served the purpose of producing an implementation similar to the original, but having task code shared with the future StarPU version, and minimize any details that would be different about the implementation.

The goal of this version is to port as much as possible of the CPU task code to CUDA. However, as explained previously (in Sections 5.2 and 5.3.2), some of the tasks are not parallelizable, and consequently, not adequate to massively parallel devices. This means that these tasks were kept running on the CPU. The obvious consequence of this is that a full iteration of the algorithm is not capable of running entirely on a GPU, requiring memory transfers in between to solve data dependencies.

The most problematic drawback of this decision is about the **Rebuild Lookup Table** task. Running this task on CPU will likely have a very noticeable impact on performance, since it requires to transfer the generated hit points from the GPU to the CPU, and later copy the generated hash table back to the GPU.

5.3.4 StarPU

For the final implementation, using the StarPU framework, most of the remaining work consisted on reusing the existing code for each computational tasks, and submit them as tasks to be scheduled by StarPU. The development process of the previous implementations (CPU and CUDA) resulted in a very modular solution, with very little coupling[2] between tasks implementation and the algorithm and scheduling code being used.

[2]the degree of dependency between the multiple modules of a system. Tight coupling tends to difficult refactoring of one module without requiring subsequent changes to dependant modules, difficulting the re-usability of code

Early Decisions

An early decision for this implementation was to consider only the low-level API, and not the `pragma`-based one (see Section 3.2.6). The reason for this is due to the high-level version being an earlier product, still being in earlier development stages, and not being fully capable of providing the full set of features of StarPU.

Additionally, the actual documentation for the framework is almost entirely focused on the low-level functions, making it easier getting up to speed and understand its usage.

An additional decision that was enforced by the existing implementations is related to the task scheduling policy used. Since the CPU implementation of all parallelizable tasks was based on OpenMP, it was intuitive to approach the problem by using the capabilities of parallel tasks and combined workers of StarPU (see Section 3.2.5). The only drawback that comes from this is that only the parallel-aware task schedulers (`pheft` and `peager`) are capable of parallelizing CPU tasks. This means that when using a non-parallel-aware scheduler, CPU tasks such as **Advance Photon Paths** will increase in cost.

Data Management

The first step for this implementation was to refactor data management, letting StarPU handle all necessary data for the algorithm. One exception was made to this, regarding the input information for the 3D scene. This information is stored in a somewhat complex structure, contrarily to all other dynamic data used throughout the photon mapping algorithm, which consists only on vectors whose size can be static and predetermined.

A small was necessary in the lookup table build process. This task previously generated a dynamically sized structure, since it is dependent on the number of photons that intersect the scene within the current radius of each hit point, which cannot be predetermined.

One solution for this would be to only register the lookup table in a StarPU data handle only after its generation is complete, and the size can be determined. This is not a desirable solution, as it would introduce a barrier on that point of the iteration, and forcing all future tasks to be submitted only once the lookup table build process is complete. This would prevent StarPU from having knowledge on future tasks, and impeding it from asynchronously prepare data buffers to solve dependencies, increasing the latency caused by the imposed barrier.

This seems to be a rather harsh limitation of StarPU, as irregular sized structures are commonly used. However, an alternative solution is possible for this specific problem. The cell size of the hash grid used is based on the photon radius for the current iteration, in such a way that a cell will never have a width, height or depth greater than the current radius. With that follows that any given hit point will always intersect at most 8 cells. Thus, it can be determined that the maximum hash table size can be set as $8 * \#hit\_points$, for any iteration.

Following this constraint, the hash table structure was refactored to be a fixed-size one, allowing StarPU to handle it without the need for barrier, and allowing future tasks to be submitted at will, using the lookup table data handle as a regular data buffer and dependency.

Task Submission

The other main change required is to wrap tasks around StarPU API calls. All data for each iteration is assigned to an individual data handle. No allocations are ever done manually during the main loop, making all memory managed by StarPU. While in the CPU and CUDA version, only a single iteration is considered at a time, so task invocation is made synchronously, here all tasks are submitted asynchronously, and dependencies are implicitly given by the data handles required by each task. These dependencies represented in Figure 5.1.

It can be seen from the graph that some task concurrency is possible, although limited. It is expected that the most time consuming tasks are **Advance Eye**

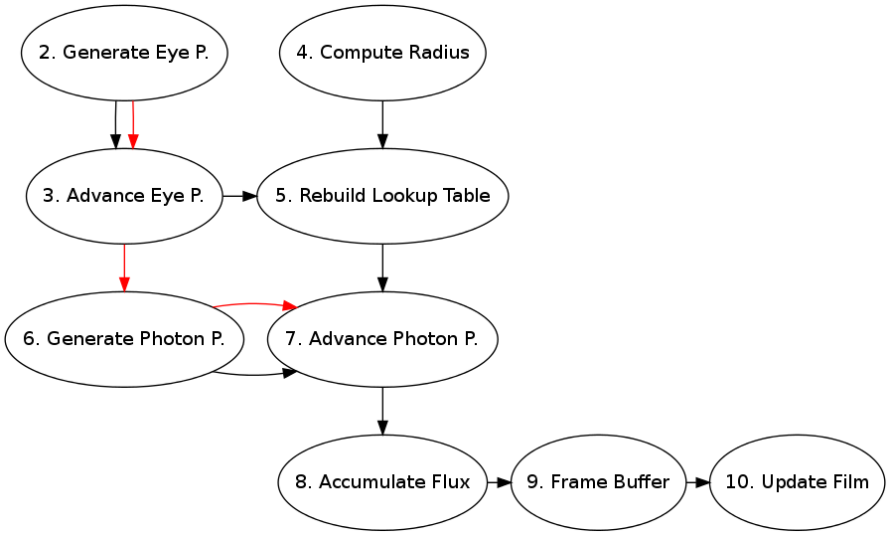


Figure 5.1: Dependency Graph of an iteration of SPPMPA. Red arrows represent dependencies related to the seed buffer, which are not imposed on the algorithm itself, but only a limitation of the implementation

Paths and **Advance Photon Paths**, as they are where scene intersections are computed. **Rebuild Lookup Table** can also be considered, especially when other tasks of the same iteration are run on a GPU. This will require synchronization memory transactions to solve the dependencies of this task. Prefetching cannot be employed here by StarPU, as these dependencies are not available until the immediately previous tasks are computed. This could provide a large bottleneck for the performance of each individual iteration.

One of the limiting factors of these dependencies is related to the number generation method employed, which relies on a read-write seed buffer, with each new value requested updating the corresponding seed. As can be seen in the graph, only one relevant dependency is actually created by this, between **Advance Eye Paths** and **Generate Photon Paths**. These tasks should be considered independent, but share a dependency on the seed buffer, and thus cannot execute concurrently as it would be expected. Other dependencies are created from this limitation (shown in red in the graph), but they do not represent a problem, since their elimination would not introduce any new concurrency possibilities.

A solution for this would be to change the random number generation method, to one that would not require intermediate seed memory to be passed between each task. However, that was not attempted, as it would require additional effort in changing the algorithm, as well as the previous implementations (if coherence between them was to be kept), and choosing an adequate and efficient new method. The final solution actually came as a consequence of enabling concurrency between iterations, explained in the next section

Enabling Concurrent Iterations

Since the high amount of dependencies between tasks does not allow a good degree of concurrency within a single iteration, efforts were focused in allowing the execution of multiple iterations concurrently. This is one of the main reasons SPPMPA was the only one focused on, as other versions without the probabilistic radius estimation would introduce dependencies between each iteration, preventing their parallelization.

The initial approach to port the implementation to StarPU relied on data handles being declared at the start of the rendering process, and release at the end.

As shown in Listing 1, data handles are created and kept during the whole rendering process. In practice, this means that each iteration will depend on the same data buffers as the previous one, even though they are to be completely rewritten, and their previous values ignored. This is not desirable, as it prevents concurrency between iterations, just like it happens due to the seed buffer dependency previously explained.

An alternative solution is to declare the handles in-loop, as shown in Listing 2

With this method, each iteration declares its own copy of the required data. StarPU provides the `starpu_data_unregister_submit` API call, which instructs the library that the given data buffer can be discarded as soon as existing tasks

```
1  void render() {
2    starpu_data_register(seeds, ...)
3    starpu_data_register(eye_paths, ...)
4    starpu_data_register(hit_points, ...)
5
6    starpu_insert_task(codelets::init_seeds, seeds);
7
8    int iteration = 0;
9    while(iteration < config.max_iters) {
10     starpu_insert_task(codelets::generate_eye_paths, eye_paths, seeds);
11     starpu_insert_task(codelets::advance_eye_paths,  eye_paths, seeds, hit_points);
12
13     ... // all other tasks for this iteration
14
15     iteration++;
16   }
17
18   starpu_data_unregister(seeds);
19   starpu_data_unregister(eye_paths);
20   starpu_data_unregister(hit_points);
21 }
```

Listing 1: The beginning of the main rendering loop, with global data handles

```
1  void render() {
2    starpu_data_register(seeds, ...)
3
4    starpu_insert_task(codelets::init_seeds, seeds);
5
6    int iteration = 0;
7    while(iteration < config.max_iters) {
8      starpu_data_register(eye_paths, ...)
9      starpu_insert_task(codelets::generate_eye_paths, eye_paths, seeds);
10     starpu_data_register(hit_points, ...)
11     starpu_insert_task(codelets::advance_eye_paths,  eye_paths, seeds, hit_point
12     starpu_data_unregister_submit(eye_paths);
13
14     ... // all other tasks for this iteration
15
16     starpu_data_unregister_submit(hit_points);
17
18     iteration++;
19   }
20
21   starpu_data_unregister(seeds);
22 }
```

Listing 2: The beginning of the main rendering loop, now with in-loop data handles

depending on it are finished. Since data is local to each iteration, this can actually be considered a more intuitive way to approach the problem.

However, an additional problem arose from this approach. Due to the asynchronous nature of the tasks, the program actually submits every single iteration to the scheduler, meaning that multiple copies of the `eye_paths` and `hit_points` buffers will be immediately requested. For a large enough number of iterations, this resulted in memory problems, and eventually program failures. Since StarPU does not provide any method to control this, the limitation had to be imposed manually, by inserting a barrier every few iterations (with the actual number being a configurable value).

The fact that the number of concurrent iterations is now configurable also allowed to tackle the data dependency provided by the seed buffer. Instead of a single buffer being shared by the entire algorithm, the implementation was adapted to instead use as many buffers as there are concurrent iterations, as shown in .

Chapter 6

Profiling Results

Initial analysis was focused on studying the scalability of both CPU and CUDA versions, when not employing StarPU. This provided a baseline to determine the overhead of using the framework. Different schedulers were attempted, particularly **peager** and **pheft** due to their awareness of combined workers, which allows OpenMP parallelization within CPU tasks. **dm** and **dmda** were also attempted, to analyze the impact of the consideration of memory transfers in the performance model of a scheduler.

6.1 Testing Environment

All tests were performed within the SeARCH[1] cluster, particularly using the most recent generation of hardware, in the node 711, which is fully described in Table 6.1.

All tests were compiled with GCC 4.6.2 (latest major version with full CUDA support), the boost library 1.49.0, and version 5.0 of the official CUDA compiler. Regarding StarPU, the latest version available, 1.1.0rc2, was used, as well as the

[1] http://search.di.uminho.pt

CPU model:	Intel Xeon E5-2670
# CPUs:	2
# Cores p/CPU:	8
# Threads p/Core:	2
Clock frequency:	2.66 GHz
L1 cache:	32 KB + 32 KB
L2 cache:	256 KB
L3 cache(shared):	20 MB
RAM:	64 GB
CUDA Device 0:	Kepler K20m
CUDA Device 1:	Tesla C20909

Table 6.1: Hardware description of the SeARCH computational node 711

hwloc 1.7 library for hardware topology, which StarPU internally uses.

6.2 Testing Methodology

For each measurement, only the time spent in the main rendering function was considered, discarding any input and output time spent by the program. A minimum of 10 executions were made for each measurement, for which the 3 best executions within at most 5% of each other were considered. When comparing results, the average time for each iteration of the main loop of SPPMPA was the base value to use (with each test running at least 20 iterations).

Whenever CUDA was employed, the CUDA Occupancy Calculator[2], as well as manual tuning, were used to find the correct block size used for each computational kernel.

[2]A spreadsheet by NVidia that helps estimating the ideal block size for a given kernel

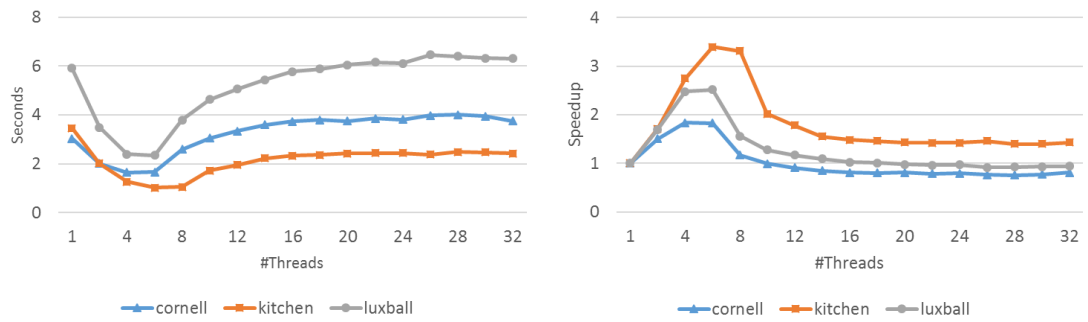
6.2.1 Input Scene

For simplicity, input reading was left to the LuxRender library, relying on the existing structures and parser to read all the information regarding a scene to render. This limits all testing to the available scenes shipped with LuxRender as samples. From these scenes, only three scenes were selected, namely, **kitchen**, **cornell** and **luxball**. However, the limited amount of scenes to choose from may also be a limiting factor of the analysis.

6.3 Without StarPU

6.3.1 CPU

For the CPU implementation, earlier analysis of the original implementation relatively bad scalability. The actual results are not presented here due to different algorithms being employed (profiling of original version was done only on PPM version and not SPPMPA), and no assumptions could be made about code quality, as explained in Section 5.3.1. Instead, a scalability analysis was made on the actually implemented CPU version, shown in Figure 6.1.



(a) Avg iteration time

(b) Speedup vs sequential version

Figure 6.1: CPU implementation

It can be seen that good scalability is achieved only until around 6 threads are

reached, which is more or less the point at which a single CPU socket is filled, at which point performance starts to degrade. For **luxball** and **cornell** scenes, performance with 16 threads (same amount as physical CPU cores in the test machine) actually barely differs from the sequential approach.

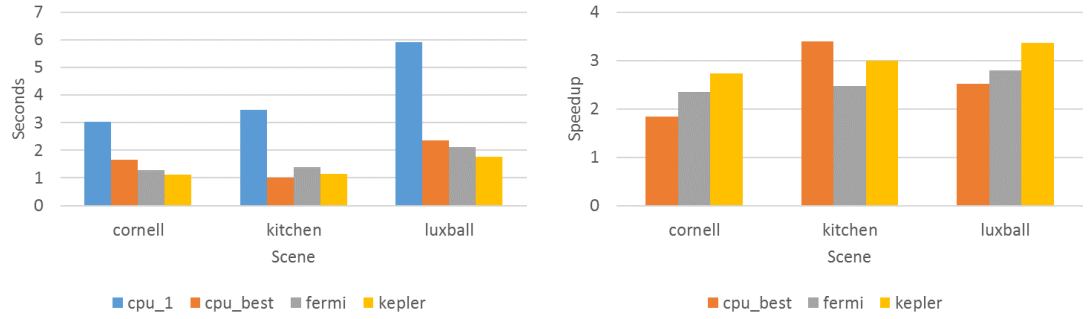
The end conclusion is that scalability is only good when a single socket is used. The algorithm is clearly memory bounded, especially considering the fact that the test machine uses a NUMA architecture. This pins all memory allocated by the master thread (such as the input scene, which is heavily used throughout the algorithm) to one of the sockets, leaving the other with slower access times to such memory.

Memory affinity tools such as `hwloc` could prove useful here, for example, by creating multiple copies of the input scene, and pinning each one to each NUMA node. Each socket would then benefit from faster accesses to its own memory node.

CUDA

When analyzing the GPU implementation, tests were made on both available GPUs, and compared against the base sequential CPU implementation. The best execution time on CPU was also included for comparison.

It should be taken into account that, as explained before, this is not a full-GPU implementation, requiring in-loop memory transfers, and some CPU computation to generate the lookup table. Since this was not implemented on GPU, it should add a considerable overhead to the execution. Still, Figure 6.2 shows the implementation is able to outperform the CPU in two of the three cases, achieving a speedup of around 3 when compared to the sequential approach.



(a) Avg iteration time

(b) Speedup vs sequential version

Figure 6.2: CUDA implementation

6.4 With StarPU

In order to accurately profile the performance obtained by using the framework

6.4.1 Scheduler Impact

Figure 6.3 shows the overhead of using the framework for task scheduling instead of directly invoking tasks. This was simulated by using each different scheduler, with only CPU tasks, and comparing against the best CPU times obtained without the framework. No major speedups are expected here, since the framework should itself create a small overhead (although the scheduler is actually one of the fastest ones available in terms of decision-making time), but it should be interesting to see if delegating the task of choosing OpenMP thread pool size to StarPU, rather than manually tuning it, directly impacts performance.

This was mostly a concern due to the scalability problems observed in Section 6.3. If StarPU, using one of the less smart schedulers, would opt to eagerly use all available CPUs, performance would degrade as seen in Figure 6.1. This was indeed the observed result with the **peager** scheduler. **dm** and **dmda** also degrade performance down to around 4x (which corresponds c«roughly to worst-case scenarios observed on CPU implementation), but that is to be expected since

these schedulers do not support parallel tasks. Since no accelerators are being used here, this results in all tasks being ran sequentially.

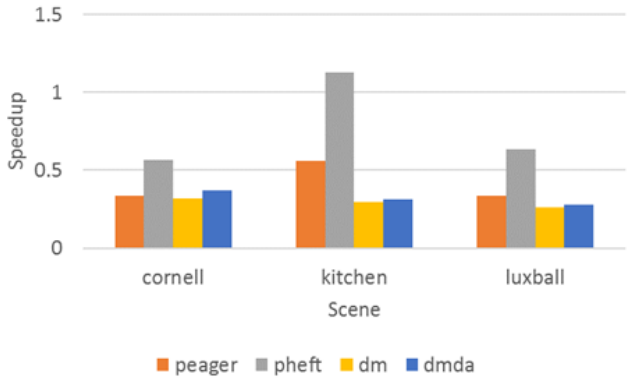


Figure 6.3: StarPU implementation, CPU only

6.4.2 Performance with accelerators

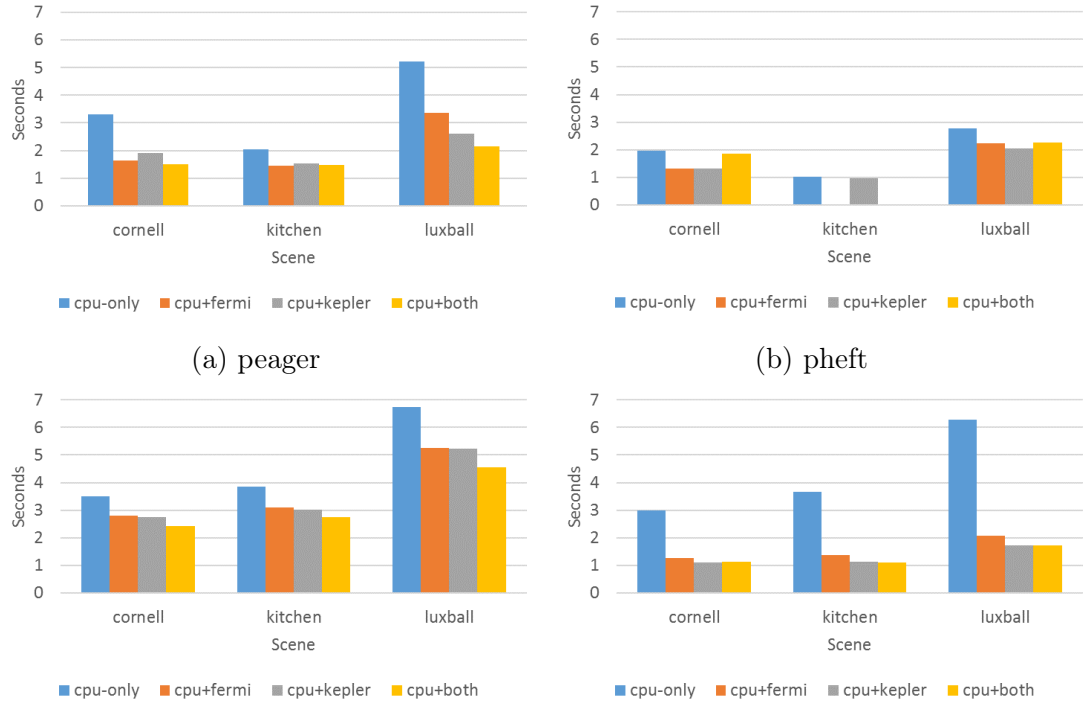
To test how CUDA devices influenced the algorithm, tests were made for both individual CUDA devices, as well as a final with both GPUs available for StarPU to use. When using accelerators, all schedulers are able to speedup the implementation when compared to the best CPU times (which are achieved with around 6 threads). However, the gain difference between each scheduler is noticeable. **dmda**, which performs poorly on CPU due to sequentializing tasks, sees the biggest improvements. This is expected since its essentially a comparison between sequential CPU tasks with massively parallel CUDA tasks. The fact to note here is that **dm** has much worse evolution under the same conditions.

This enforces the fact that memory transfers are extremely important to take into account by the scheduler, as this is the only difference between the two.

As for **peager** and **pheft**, their gains are not as large, since CPU code was already parallelized anyway, but it is relevant to note that the smarter **pheft** seems to be outperformed once the whole set of devices is used.

Unfortunately, when using **pheft** with the Fermi device, memory errors would

constantly be raised, so it was not possible to finish those tests successfully. This is most likely due to problems with this particular scheduler, which is still under development by the StarPU team, and thus cannot be assumed to be fully functional.



(a) peager

(b) pheft

(c) dm

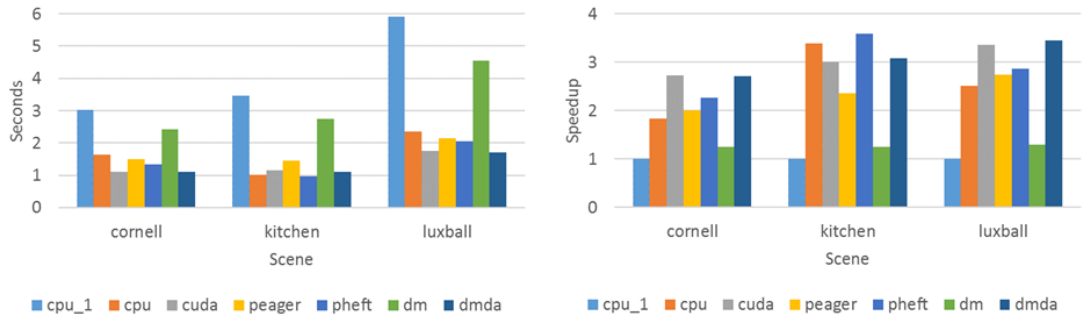
(d) dmda

Figure 6.4: Speedup of the different schedulers by successfully adding new devices

One of the most important aspects to analyze was how the chosen scheduler impacts performance.

6.4.3 Overall Performance Comparison

The best case scenario for each possible approach is shown in Figure 6.5. This serves to show the impact of StarPU with each different scheduler against framework-less solutions.



(a) Avg. iteration time

(b) Speedup

Figure 6.5: Best cases for each different implementation and scheduler

	Sequential	CPU	CUDA	**peager**	**pheft**	**dm**	**dmda**
cornell	3.03	1.65	1.11	1.51	1.33	2.42	1.12
kitchen	3.46	1.02	1.15	1.46	0.96	2.75	1.12
luxball	5.91	2.35	1.76	2.15	2.06	4.55	1.71

Table 6.2: Avg Iteration time for all versions

	CPU	CUDA	**peager**	**pheft**	**dm**	**dmda**
cornell	1.83	2.73	2.01	2.27	1.25	2.71
kitchen	3.39	2.30	2.36	3.60	1.26	3.09
luxball	2.52	3.36	2.75	2.87	1.30	3.45

Table 6.3: Avg speedup for all versions

6.4.4 Concurrent Iterations

With the employed approach, task-level parallelism is limited. The **dmda** does not support combined workers, greatly lowering efficiency of CPU tasks. **pheft** does support this, but its not a data aware scheduler, meaning that data transfers are not considered when assigning tasks. As a result, performance with StarPU is limited when using processing a single iteration.

By using concurrent iterations, however, extra speedups can be achieved, as shown in Figure 6.6.

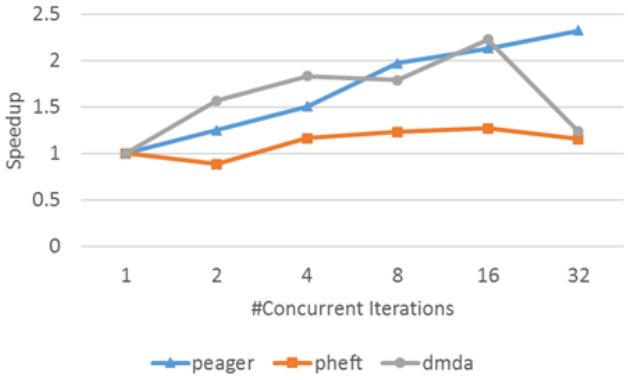


Figure 6.6: Speedup with concurrent iterations

Chapter 7

Conclusions

Frameworks targeting HetPlats are still an emerging solution for parallel computing. StarPU provides a solid API, backed by a relatively large user-base. A few problems were encountered in it. Design issues in the API allow common an easy to make developer errors to cause unexpected behaviour.

These should pose an even greater challenge considering that the scientific community, composed not only of actual developers, but also of scientists who are not as well learned in programming practices, places great interest in tools such as StarPU, to speed up their high scale computational applications. Using StarPU still requires some amount of knowledge regarding parallel computing. Questions such as "Is it worth the effort to implement a given task using an accelerator?" or "What scheduling policy best suits a given algorithm?" must be answered during implementation. From that comes that developers or scientists without a great understanding of such issues won't be able to take the best of the framework.

In terms of performance gains, it is promising that they can be obtained with little amount of work, starting from an already existing implementation and moving to a StarPU one, requiring only to wrap task invocations within StarPU API calls. However, a few caveats can come from this, and existing code can sometimes prove difficul to port. The proposed implementation of Progressive Photon

Mapping, here used as a case study for StarPU, provided some iteresting conclusions to be made regarding the framework, although the measured performance gains were not as good as initially expected. Here, a few implementation problems might be the root cause, as well as an initially not so correct understanding of the framework.

GAMA is still an unfinished product, without a clear and consistent API or documentation. However, it does provide an interesting set of ideas that could be useful to at least attempt to tackle these issues. The fact that it is implemented in $C++$ rather than in C as StarPU is a large benefit, since the language provides capabilities that solve the problems mentioned about StarPU's API in Section 3.3.

Results prove that scheduler choice is a very important factor in the performance of an algorithm. As such, the modularity aspect of StarPU is particularly helpful in tuning this option. GAMA currently relies solely on a **heft** like policy, but it could possibly benefit from a similar approach, in order to provide a more flexible solution.

Chapter 8

Future Work Suggestions

While this dissertation focused mostly on the implementation of a case study in StarPU, a similar effort should be made to produce a similar implementation with GAMA, to actually compare the two in terms of performance. Without such implementation, only a more shallow comparison could be made, regarding mostly the features, usability, and a few problems with each solution. It would also be interesting to test the usability of the `pragma`-based API of StarPU

Other possible points of improvement on top of this work are more related to the produced implementation. The first point is related to the random number generation, which could be further improved by using a different random number generator, that would not require an intermediate buffer, thus eliminating one dependency between tasks.

In addition, a new approach to task parallelization could be attempted, which did not depend on OpenMP, and as such would allow an efficient usage of other StarPU schedulers without support for combined workers, which are still under development by the framework's team.

Bibliography

[1] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965.

[2] Gordon E Moore. "Progress in digital integrated electronics". In: *Electron Devices Meeting, 1975 International*. Vol. 21. IEEE. 1975, pp. 11–13.

[3] V.W. Lee et al. "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU". In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 451–460.

[4] Rajesh Bordawekar, Uday Bondhugula, and Ravi Rao. "Believe it or not!: mult-core CPUs can match GPU performance for a FLOP-intensive application!" In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM. 2010, pp. 537–538.

[5] João Barbosa. *GAMA framework: Hardware Aware Scheduling in Heterogeneous Environments*. Tech. rep. Computer Science Dept., University of Texas at Austin, Sept. 2012.

[6] Cédric 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] Michael D Linderman et al. "Merge: a programming model for heterogeneous multi-core systems". In: *ACM SIGOPS operating systems review*. Vol. 42. 2. ACM. 2008, pp. 287–296.

[8] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping". In: *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE. 2009, pp. 45–55.

[9] 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.

[10] François Broquedis et al. "hwloc: A generic framework for managing hardware affinities in HPC applications". In: *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE. 2010, pp. 180–186.

[11] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *Parallel and Distributed Systems, IEEE Transactions on* 13.3 (2002), pp. 260–274.

[12] A. Mariano. "Scheduling (ir)regular applications on heterogeneous platforms". MA thesis. Sept. 2012.

[13] Miguel Palhas and Pedro Costa. *A Finite Volume Case Study From An Industrial Application*. 2012.

[14] Cedric Augonnet et al. "Data-aware task scheduling on multi-accelerator based platforms". In: *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. IEEE. 2010, pp. 291–298.

[15] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. "Automatic calibration of performance models on heterogeneous multicore architectures". In: *Euro-Par 2009–Parallel Processing Workshops*. Springer. 2010, pp. 56–65.

[16] J.T. Kajiya. "The rendering equation". In: *ACM SIGGRAPH Computer Graphics* 20.4 (1986), pp. 143–150.

[17] Steven G Parker et al. "Optix: a general purpose ray tracing engine". In: *ACM Transactions on Graphics (TOG)* 29.4 (2010), p. 66.

[18] Niklas Huss. "Real Time Ray Tracing". PhD thesis. LNU, 2004.

[19] Arthur Appel. "Some techniques for shading machine renderings of solids". In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM. 1968, pp. 37–45.

[20] Turner Whitted. "An improved illumination model for shaded display". In: *ACM SIGGRAPH 2005 Courses*. ACM. 2005, p. 4.

[21] 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.

[22] 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.

[23] H.W. Jensen. "Global illumination using photon maps". In: *Rendering Techniques* 96 (1996), pp. 21–30.

[24] Henrik Wann Jensen et al. "Monte Carlo ray tracing". In: *ACM SIGGRAPH*. 2003.

[25] T. Hachisuka, S. Ogaki, and H.W. Jensen. "Progressive photon mapping". In: *ACM Transactions on Graphics (TOG)*. Vol. 27. 5. ACM. 2008, p. 130.

[26] T. Hachisuka and H.W. Jensen. "Stochastic Progressive photon mapping". In: ().

[27] C. Knaus and M. Zwicker. "Progressive photon mapping: A probabilistic approach". In: *ACM Transactions on Graphics (TOG)* 30.3 (2011), p. 25.

[28] Holger Dammertz, Johannes Hanika, and Alexander Keller. "Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays". In: *Computer Graphics Forum*. Vol. 27. 4. Wiley Online Library. 2008, pp. 1225–1233.

[29] Martin Stich, Heiko Friedrich, and Andreas Dietrich. "Spatial Splits in Bounding Volume Hierarchies". In: *Proc. High-Performance Graphics 2009*. 2009.

[30] Robert C Tausworthe. "Random numbers generated by linear recurrence modulo two". In: *Mathematics of Computation* 19.90 (1965), pp. 201–209.