GPU programming language

- A language to seemlessly take advange of the GPU -

Project Report

SW413F15

Aalborg University
Department of Computer Science

Here you can write something about which tools and software you have used for typesetting the document, running simulations and creating figures. If you do not know what to write, either leave this page blank or have a look at the colophon in some of your books.

AALBORG UNIVERSITY

STUDENT REPORT

Title:
GPU programming Language

Theme:
Programming languages and compilers

Project Period:
Spring Semester 2015

Project Group:
SW413F15

Participant(s):
Marc Tom Thorgersen
Søren Hvidberg Frandsen
Troels Beck Krøgh
Mathias Corlin Mikkelsen
Morten Pedersen
Mathias Sass Michno

Supervisor(s):
Thomas Kobber Panum

Copies: 9

Page Numbers: **??**

Date of Completion:
February 21, 2015

Abstract:

Here is the abstract

Contents

Todo list

Preface

This report is the product of a semesters work by 6 students. Everything written in this is written by the students, unless otherwise is obvious such as quotes.

Aalborg University, February 21, 2015

Mathias Corlin Mikkelsen

Mathias Sass Michno

Marc Tom Thorgersen

Søren Hvidberg Frandsen

Morten Pedersen

Troels Beck Krøgh

Introduction 1

New central processing units ("CPUs") are able to calculate at ever greater speeds. Moore's law predicts an exponential growth in transister count, thus faster computation, however experts tredict that this trend may end as soon as 2020 or 2022. [1]

This begs the question, if our processors are reaching their maximum potential what other computational devices can be used?

Graphics processing units ("GPUs") are extremely fast at calculating in parallel due to their internal architecture. (More on this in chapter) GPUs are starting to be used around the world for calculating very large numerical computations, since it is possible to parallelize tasks some computations. Most compilers, like GCC, targets only the CPU, and it can be difficult to transfer these computations to the GPU instead using languages like C, or C++. To use the GPU for general purpose computation (General-purpose computing on graphics processing units "GPGPU"), there exists the frameworks OpenCL and CUDA. CUDA is a virtual instruction set created by NVIDIA, and is therefore only usable by CUDA enabled NVIDIA GPUs. OpenCL is an ópen stardard and supported by more GPUs, most noticible Intel, AMD and NVIDIA GPUs. (More on OpenCl and CUDA in chapter)

Marks kapitel

Mortens kapitel

Therefore this project paper will research the possibilities of creating a programming language and a compiler, which can perform numerical computations on the GPU seamlessly to the programmer.

1.1 Graphics Processing Units

Graphic Processing Units (GPUs) were initially used to accelerate memory-intensive work for rendering graphics. They consist of a higher amount of cores, these cores while less effecient than the cores used in a CPU, are so plentiful, that with enough work, more **Instructions Per Cycle(IPC)** can be performed. These cores sacrifice the architectural components that enables the CPU run single instruction streams in exchange for more computing power, such as increasing the number of **Arithmetic Logic Units(ALUs)** fig. 1.1 on the following page shows a basic idea of the differnce between the GPU and CPU.

This concept of using many cores to process a high number of threads giving GPUs a high throughput, is the heart of this very architecture. In recent years GPUs have been used for more than just graphical purposes, such as GPU-accelerated computing, the use of both GPU and CPU to accelerate scientific, analytics and likewise applications. Applications which benefit from the highly parallel processing power of the GPU, while still using the CPU for sequential problem solving.

1.1.1 GPU architecture

For the purpose of this section it is worth noting that depending on the GPU manufactorer and the series of the GPU, the exact architecture may vary. This section will therefore describe the architecture

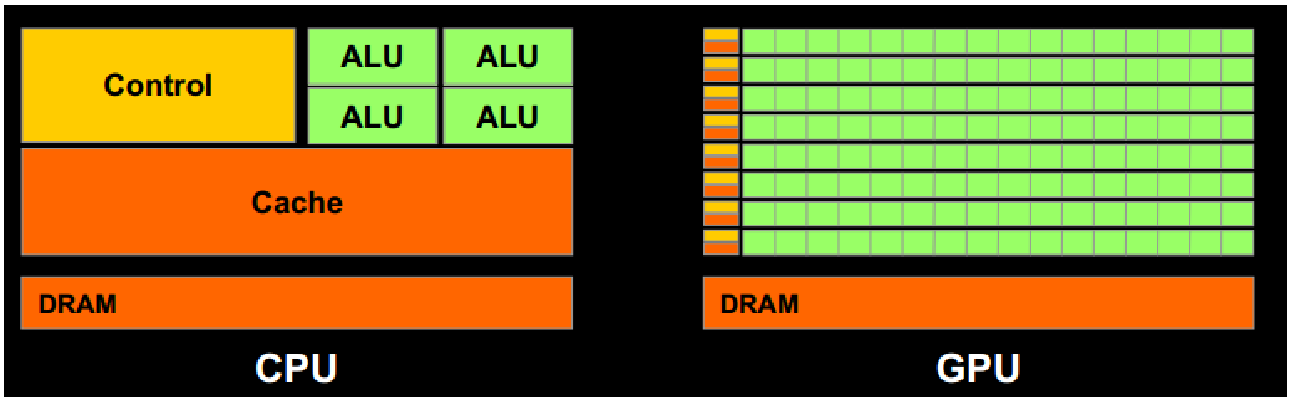


Figure 1.1. A basic representation of the Transistor allocation on the GPU compared to the CPU

in a conceptual manor, and not go into specifics. Also note that the main difference between earlier and later models of the same manufactorer, is primarily size, not the architectural concepts themselves. The GPU architecture chosen as the basis for the following section is Nvidia. They are the market share leader, discounting integrated graphics solutions. The compute device(The GPU) consists of Compute units(**Streaming Multiprocessors(SMs)**) which each consists of processing elements(**Streaming Processors(SPs)**), registers, execution cores for integer and floating point operations aswell as several caches; shared memory, texture memory and constant memory cache. For GPU-accelerated computing the GPU acts as a co-processor, the execution of the application using the GPU originates from the CPU, once compute intensive functions that are not sequential are met. The CPU calls the GPU driver and starts a data transfer with the GPU or a **kernel** launch on the GPU. A kernel is a function executed on the GPU, this is explained further on **??** on page ??. A kernel specifies the number of threads and data it requires. The GPU, trough its architecture uses **Single Instruction Multiple Data(SIMD)** to do its computations, this is a concept that is used for its parallel computing power. This concept is brought to life through exploiting SMs computing power, aswell as the number of SMs.

1.1.2 Memory Hierachy

The GPU memory is comprised of the following different memory components, how it is communicates is shown on **??** on page ??. Memory managment of the GPU is also explicit, this means that the data must first be moved from the host, to global memory, then to local memory where the SMs can start to use it. Once they are done it must be sent back the same way. The memory architecture is designed with regard to increased performence by reducing memory traffic. Although the hierachy and its components were originally designed to improve performance for graphical applications, it can also be used efficiently in some GPU computing applications, how the memory is used is further explained in **??** on page ??.

Registers:

For any given SM, thousands of registers are contained, this is the most plentiful type of memory used in the SM. Once a kernel is launched, registers are allocated to threads as per the kernels requirements. This also means that the registers shared across the SM, but only to the allocated to a thread the allocated register is no longer available to the entire SM, untill the thread is done and it can be reallocated. The registers can contain both integers and floating-points.

Local Memory:

Local memory is used for spilling registers aswell as holding lcoal variables. Each SM has its own allocated local memory available which is available to the entire SM, unlike registers it is not controlled by the kernel being executed.

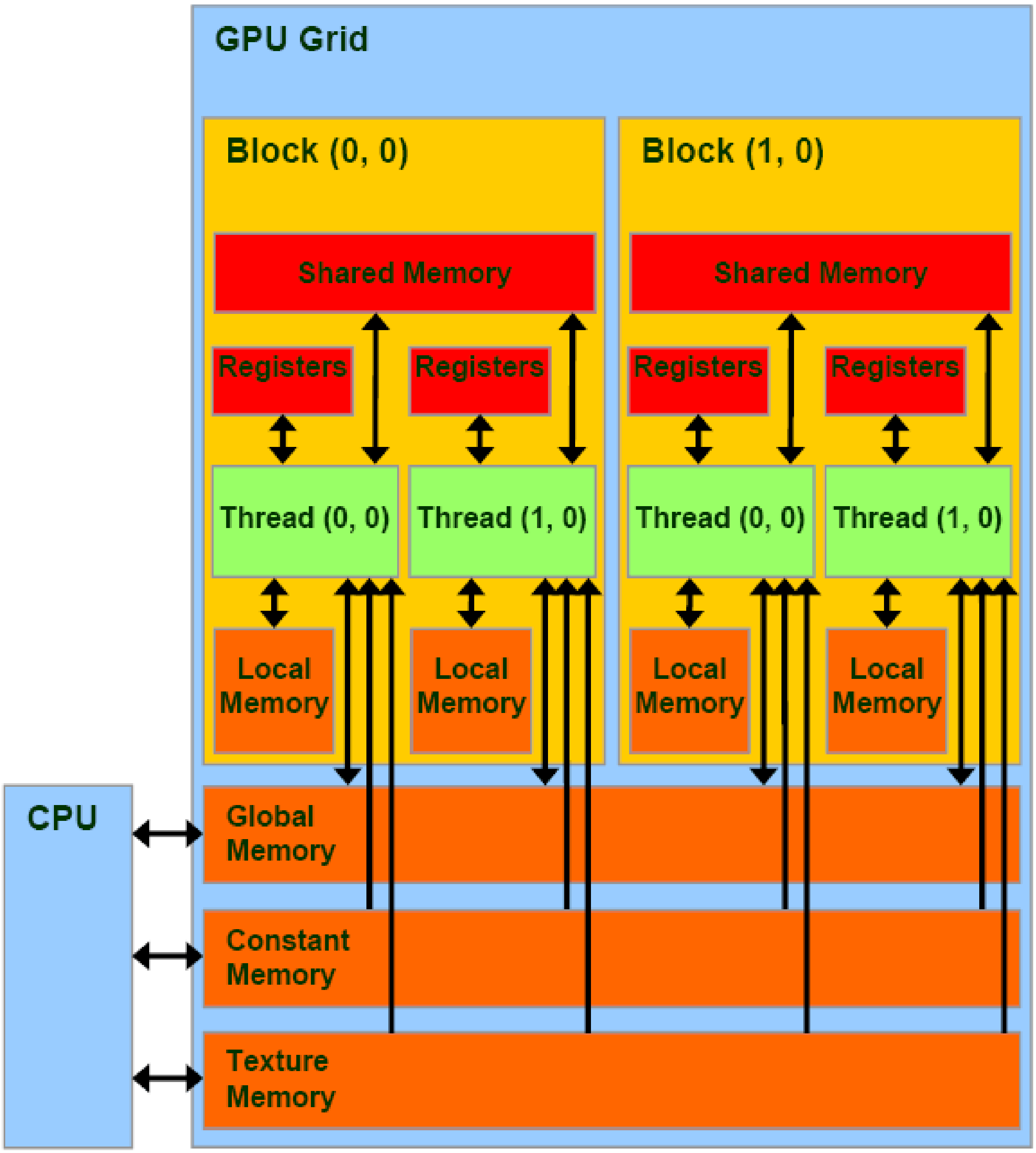


Figure 1.2. A basic representation of GPU memory hierachy, note a block corresponds to a SM, and a thread to a SP.

Global and Constant Memory:

The SMs can read and write to global memory, this is however significantly slower than using local memory. Constant memory is read only for SM however each SM contains a small optimized chache for the purpose of reading from this memory such that less reading is required. Both of these memory types are located on the DRAM and not in the SM.

Shared Memory:

Shared memory, as mentioned earlier, is part of the SM. It is a very fast on-chip memory that threads can use for data interchange withing thread blocks. Although it is fast, it being an on-chip per-SM ressource, using it can affect the number of blocks and threads can use for computations.

Texture Memory:

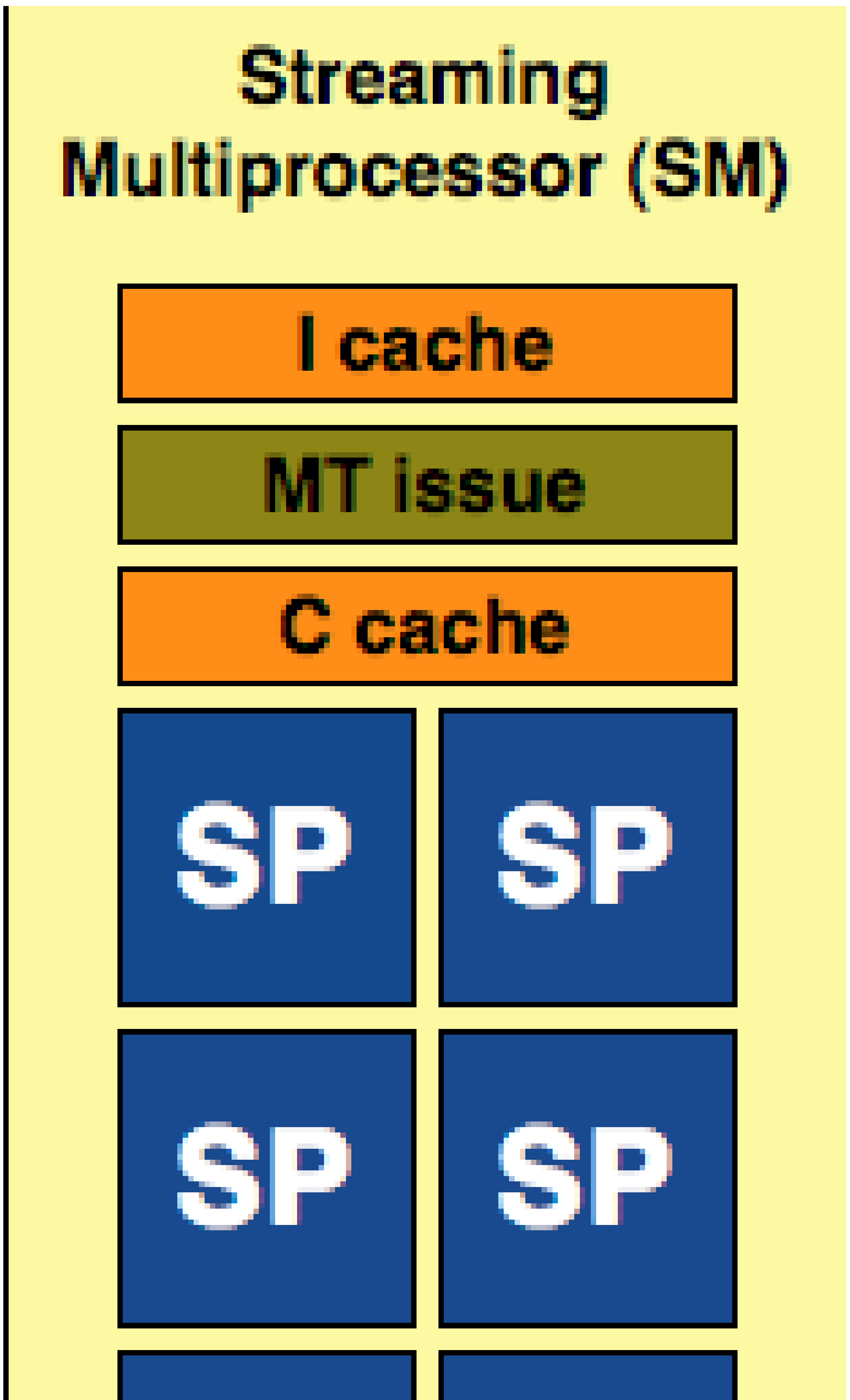
Texture memory is shared amongst a number of SMs, like constant memory this is read-only and is cached in its own texture chache on the SM

1.1.3 Streaming Multiprocessors

On the GPU the work is divided by the hardware. When a kernel is launched, the hardware dynamically assign work groups to SMs, doing so while balancing the load across the GPU. For each work group, a number of work items exists, these are the threads that execute the kernel. Work items running on a SM can communicate through local memory, all the different work groups can communicate through global memory. A collection of a given number of SMs are refered to as a **Thread Processing Cluster(TPC)**, this TPC shares a controller, aswell as a texture cache. All SMs consist of a number of ALUs, **multiply-add(MAD)** and special function units for doing computations, multithreaded instruction fetch and issue unit, instruction cache, read-only constant cache and a shared memory cache, fig. 1.3 on page 5 shows an example of this, note that the ALUs and MAD corresponds to the SPs. The SM schedules and executes threads in groups called warps. A warp waits untill all its threads have finished execution, before the next instruction is fetched.

1.1.4 Conclusion

The result of this architecture is increased parallel computing power through the sacrifice of sequential computing power. The GPU sacrifices the cache memory of the CPU, in exchange for more computing power. This results in a very high throughput, work done in an amount of time, given that there is enough work. It also results in higher latency, the time from which an instruction is initiated untill the effects have taken place. Is is the direct opposite of a CPU. In a CPU low latency is preferred, the latency is hidden through its ability to do work concurrently the low latency is achieved by the caches, something which the GPU sacrificed. Although with low latency, also follows low throughput, which is why the GPU is more optimized for heavy computational work that can be done in parallel. The structure of SMs, and the work distribution of the GPU, is what enables it to do use the SIMD model. The SM follows only one set of instructions but on multiple threads of data, the threads are managed and scheduled by the hardware itself. In conlusion, the GPU architecture allows us to efficiently compute any workload which can be done using parallel computing, and can tolerate the higher latency.


Streaming
Multiprocessor (SM)
I cache
MT issue
C cache
SP
SP
SP
SP

Bibliography

[1] Joel Hruska. Intel's former chief architect: Moore's law will be dead within a decade. http://www.extremetech.com/computing/165331-intels-former-chief-architect-moores-law-will-be-dead-within-a-decade, 2013. *Side 1*

Appendix A name A

Here is the first appendix