Automatic Code Optimizations on GPU Architectures

Johann Wagner
Otto-von-Guericke University
Magdeburg, Germany
johann.wagner@st.ovgu.de

***Abstract*—**

***Index Terms*—GPU Compiler Optimizations**

I. Introduction

General Purpose Computation on Graphics Processing Unit (GPGPU) is a frequently used term in data processing. Due to this, complex calculations can be accelerated by using GPU instead of CPU.

Todays compilers for CPU architectures are optimized a lot. There are many generally known optimizations like loop unrolling and constant precalcuation. The aim is to avoid problems of CPUs, such as costly instructions like jump- and division-operations [**?**].

Loop unrolling tries to get rid of jump-operations, that accelerates the assembler code by unrolling a loop to procedural code without jump-operations [**?**]. Constant precalcuation tries to evaluate a formula as best as possible without known variables [**?**].

Compilers for GPGPU applications, which are used to run general purpose applications on a GPU, have to solve different problems. One of the main problems is the memory access time, which increases, if multiple threads want to access the same part of memory. In this paper, we want to collect information about different optimizations for compilers to reduce time, that is spent on waiting for memory [**?**].

It is hard to write performant code, which uses the GPU to accelerate computation. NVIDIA provides CUDA, which is a programming model for different programming languages, which makes it easier to utilize the GPU for certain application. Nevertheless, CUDA introduces six different memory types and multi-threading, which is not the daily business of a programmer. Furthermore, it is important and necessary to optimize code as best as possible. [2]

II. Background

A. Threads and Thread Blocks

1) Threads: Threads are subroutines, so a small set of instructions, which can be managed and executed independently from other threads [**?**]. Threads typically have an independent stack for local variables and share heap with all other threads. Due to access from multiple independent threads, heap access must be managed.

2) Thread Blocks: Thread Blocks is a construct, which was introduced by CUDA to organize threads in groups. Threads, that are grouped in a thread block, can access a part of the storage simultaneously.

3) Hardware Implementation: NVIDIA GTX 1080 has 20 multiprocessors with 128 threads each. This means that the threads of 20 thread blocks can be executed simultaneously. If there are more than 20 thread blocks, they are executed one after the other [6]. That means, that 2560 threads can be executed in parallel.

B. General Purpose Computation on GPUs

Big Computations needs a lot of calculation time. Those computations can be mostly splitted into multiple smaller workloads, which can be calculated seperately. [3] While CPUs are designed to calculate bigger workloads with less threads, GPUs are designed to calculate smaller workloads on hundreds of threads. In its original form, each pixel could be calculated individually on a GPU, which implies heavy multi threading. [4]

There are some advantages and disadvantages of GPGPU usage:

1) Threading: A problem usually contains multiple smaller workloads, that can be calculated independently on GPUs. This calculations can be processed parallel to save time. Modern GPUs have more than 2500 cores with up to 1.7 GHz, which can execute code parallel. Other papers determine speedups up to 20 times. [5] [**?**]

2) Data Transfer Time: Transfering data from memory into GPU memory is usually a bottle lag. Calculations, which uses data from secondary storage or produces a lot of data, have to transfer data to the host system, that moves the data to CPU memory or on secondary storage. NVIDIA GTX 1080 has a data transfer rate of 10 Gbps, which is 1.25 GB/s. Based on this data it takes a 12,8 seconds to refill the entire 8 GB of storage on the graphics card. [5] [**?**]

3) Complexity: Complexity is a huge difficulty. The greater the degree of complexity, the more difficult and slower development progresses. Often code is not optimized if the consumer can deal with the waiting time, because development of accelerated solutions would cost much money. [**?**]

C. NVIDIA CUDA

NVIDIA CUDA is a parallel computing platform and model, which was developed by NVIDIA to speed up applications by usage of NVIDIA GPUs. For example CUDA can be plugged as library into C programs to make use of a CUDA-capable GPU. [7]

NVIDIA CUDA, further referred as CUDA, makes it easy to access GPU for general purpose programming by providing functions for different programming languages like C, C++ and Python. It provides simple functions and/or annotations to enable GPU usage.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Fig. 1. Code with CUDA Support to add two vectors, accelerated with GPU - Taken from NVIDIA Programming Guide

In Figure 1, we see a simple vector addition, that is accelerated by using GPU with CUDA. You have to define a kernel function, which is a simple c function, which is called N times by N different CUDA threads. This kernel function is executed on GPU. `threadIdx.x` contains, in this simple example, just the number ($0 \leq \texttt{threadIdx.x} < \texttt{N}$) of the thread, which called the function. In Figure 2, we see the mathematical equation, which is done by the code in Figure 1. We can calculate every c_i independently in sepeate threads, which makes it possible to speed up the process. This example is trivial, however matrix operations, that are more complex, can be speed up with the same approach. [8]

$$\begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Fig. 2. Mathematical Equation for Vector Addition

D. *Memory Seperation in CUDA*

NVIDIA CUDA introduces six different types of memory, which are up to four more types than in other programming languages. Different types of memory has different attributes.

1) Register: Register Memory is quite similar to CPU Register Memory and is used as storage for local variables, etc. It is limited to 16kb and everything, which exceeds this limit, will be pushed on Local Memory. [10]

2) Local Memory: Local Memory is an abstraction of CUDA, which makes it possible to store information in Global Memory, which is only available from current thread. [10]

3) Shared Memory: Shared Memory is much faster than the Global Memory. It is visible to all threads, which are in the same thread block. It is intented to share data across multiple threads, that are in the same thread block. [9] [8]

4) Global Memory: Global Memory is the slowest memory. It is visible to all threads, can be read and written from all threads and can be as General Purpose memory. It is very similar to CPU memory, which makes it easy to use to share data across multiple thread blocks. [9]

5) Constant Memory: Constant Memory is quite similar to the Global Memory, but can only be read. It can help to reduce overhead by caching information. [9] [10]

6) Texture Memory: Texture Memory is quite similar to the Global Memory, but can only be read. It provides some caching methods, which can be used to accelerate certain applications, but this is not used in the further explanations. [8]

III. Body

Waiting for Memory is one of the most time-consuming states in executing a program. Compared to CPU programming time can be saved in GPU programming by utilizing different memory types, which were introduced in the prior section. Fundamentally data, that is created and used by GPU threads, could be read and written in the global memory. This native approach would be slow, because only one thread could use the data at a time.

It would be better if data that is only used by the thread would also be stored in the memory that can only be used by the thread to reduce memory accesses to global memory and to shorten the wait time for other memory requests for global memory. Unfortunately it is not possible to determine the usage of different variables at compile time, because we are not able to test a complex program deterministically, and as a consequence it is not possible to optimize the memory access time this way.

Fortunately, there are a few other techniques to accelerate GPGPU programs.

A. *Thread-Block Merging*

Thread merging is a technique where different thread blocks are merged into one thread block to reduce memory access times.

Thread Blocks have a shared memory segment, where they can save data for execution. If multiple thread blocks uses the same data from the global memory to execute its threads, we can merge them into one thread block to reduce the memory access for global memory.

In Figure 3, we see an example for merging two thread blocks. Both thread blocks contain a data segment, which is equivalent from global memory. After the merge the data segment is only fetched one time and the number of threads in the thread block adds up.

B. *Thread Merging*

Thread merging is a technique where different threads are merged into one thread to reduce memory access times. In exchange, actual computing time gets lost.

With Thread merging, there is a possibility to merge different threads to reduce global memory access. The goal of

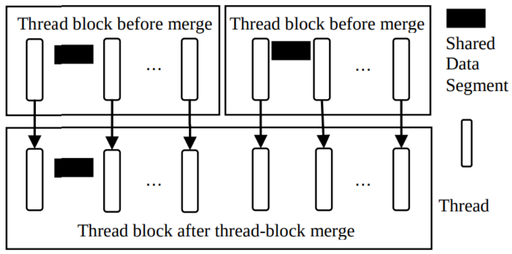

Fig. 3. Improved Memory Reuse by Thread Block Merging - Taken from A GPGPU Compiler for Memory Optimization and Parallelism Management

thread merging is to reuse data. In the best case, each block of data has to be fetched only once. In addition to thread block merging, which only allow reuse of Shared Data Segments, thread merging allows the reuse of shared registers too.

In Figure 4, we see an example for merging two thread blocks into one thread block by merging threads. Again, both blocks contain a shared data segment, which is equivalent. Additionally a shared register per thread is equivalent. By merging two threads into one, the shared data segment and the shared register just have to be fetched once.

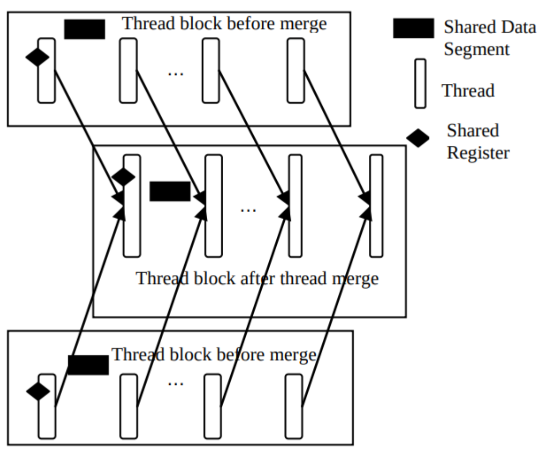

Fig. 4. Improved Memory Reuse by Thread Merging - Taken from A GPGPU Compiler for Memory Optimization and Parallelism Management

C. Data Prefetching

Data prefetching is a technique to optimize loops by prefetching the data, before they are actually needed.

Data prefetching optimizes the iteration over data structures by prefetching the data, that is needed in the following iteration step. In the current iteration step i the data from iteration step $i + 1$ gets fetched. This step allows you to overlap memory access time and computing time.

In Figure 5, we see example code with data prefetching. To understand the concept the tmp variable is important. In the beginning tmp holds the first part of data. In the following iteration steps it requests the part of data, that is used in the following step. That is an asynchronous event, that run next to the calculations of the current iteration step.

```
for (i=0; i<w; i=(i+16)){
   __shared__ float shared0[16];
   shared0[(0+tidx)]=a[idy][((i+tidx)+0)];
   __syncthreads();
   int k;
   for (k=0; k<16; k=(k+1))   {
        sum+=(shared0[(0+k)]*b[(i+k)][idx]);
   }
   __syncthreads();
}
```

(a) Before inserting perfetching

```
/* temp variable */
float tmp = a[idy][((0+tidx)+0)];
for (i=0; i<w; i=(i+16)) {
   __shared__ float shared0[16];
   shared0[(0+tidx)]=tmp;
   __syncthreads();
   if (i+16<w) //bound check
       tmp = a[idy][(((i+16)+tidx)+0)];
   int k;
   for (k=0; k<16; k=(k+1))  {
       sum+=(shared0[(0+k)]*b[(i+k)][idx]);
   }
     __syncthreads();
}
```

(b) After inserting perfetching

Fig. 5. Example Code for Data Prefetching - Taken from A GPGPU Compiler for Memory Optimization and Parallelism Management

In Figure 6, we see a timeline, that shows the overlapping execution times from fetching data and executing code after the optimization. There is no time saving in the first iteration, because it is not possible to execute code without loaded data. After the first iteration, data can be loaded and code can be executed at the same time.

Fig. 6. Example Timeline for Example Code in Figure 5 - Taken and Modified from NVIDIA CUDA Programming Guide

D. Partition Camping

Partition Camping is a problem, which is related to the memory architecture of NVIDIA CUDA. It describes the problem, that data, that is used in different threads, uses the same partition in global memory, which implies long memory access times.

The global memory of graphics cards is divided into partitions. Partitions can only be read or written by one thread at a time. If a record is loaded into global memory and is to be processed by multiple threads, then different parts of the record are processed by different threads. If all parts of the data set are located on a few partitions, then many threads need to access a partition, so memory access is slower. This is called partition camping.

If the data is evenly distributed across all available partitions, a single partition has fewer threads to access, which speeds up memory access.

In Figure 7, we see $\mathrm{SM\text{-}1}, \mathrm{SM\text{-}2}, \ldots, \mathrm{SM\text{-}30}$, which are different accessors.

In the table titled "Without Partition Camping" we see that the data for the different accessors are distributed evenly over all partitions.

In the table titled with "With Partition Camping" we see that the data is all in one partition and that all accessors would have to access one partition, resulting in long waiting times.

In conclusion, it can be said that in this example it is very easy to see that the individual partitions are used by considerably fewer accessors, which reduces the waiting time for memory.

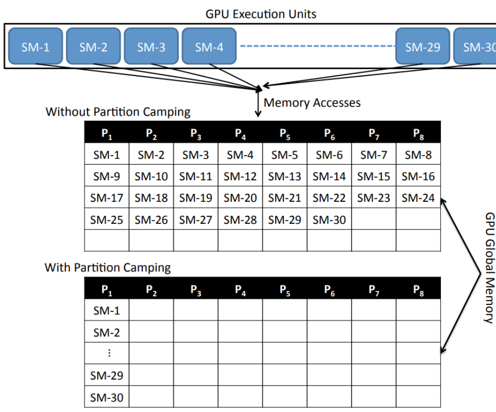

Fig. 7. Partition Camping Example - Taken from Bounding the Effect of Partition Camping in GPU Kernels

IV. Evaluation

In this section we compare the different optimizations for GPGPU, which we have explained before. In doing so, we will look at various points such as use cases, efficiency, difficulty and error cases. Note that all optimizations occur at compile time and do not mean any additional work for the developer of the application.

A. Use Cases

Cases for data prefetching and partition camping occur quite frequently. Nearly every loop that accesses data in global memory can be optimized with the help of data prefetching. The costs of the additional registers required for the temporary variable are relatively low.

The effect of partition camping always occurs when processing relatively large amounts of data and is therefore also optimized.

However, the use cases for thread merging and thread block merging are less frequent, since the circumstances and prerequisites are considerably more complex. Although cases in which these optimizations work can be constructed in this way, the application of this optimization is probably less in comparison to data prefetching.

B. Efficiency

The efficiency of the various optimizations can be measured by the increase in speed and number of memory accesses.

Thread merging and thread block merging reduce memory access by the number of n threads merged. This arithmetically increases the speed by a factor of n. According to REFERENCE, an acceleration of 10 times is possible in some cases.

The speed increase is not directly calculable with the elimination of partition camping and data prefetching, since it depends on many different factors.

The speed increase for the two other optimizations is much lower in detail, since only increases of the speed are possible in the same example by 2 to 4 times. These two optimizations do not reduce memory accesses.

C. Difficulty

Difficulty describes the effort of implementation for the application developer and the developer of the compiler. In this case, where the compiler automatically optimizes the application, there is no difficulty for the developer of the application, who does not have to work on the optimization.

The difficulty of data prefetching is, according to REFERENCE, easy to classify. The sample code in Figure 5 shows that the optimization is trivial to implement, apart from the bounding checks.

Implementing the elimination of partition camping is a bit more difficult to evaluate. There are several approaches to better disseminate data that produce different results. However, an initial implementation is relatively simple. Suppose there are n different partitions. Then the data of each partition can be split m times, so that it can be distributed to n partitions. Each partition contains an average of m/n data blocks.

Unfortunately, there is little information about the difficulty of implementing Thread Block Merging and Thread Merging. The detection is also not explained further, so that no conclusions can be drawn.

D. Error Cases

V. Related Work

VI. Conclusion

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.
[2] CUDA Zone - https://developer.nvidia.com/cuda-zone
[3] Yi Yang, Ping Xiang, Jingfei Kong, Huiyang Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management", 2010
[4] Victor W Lee, Changkyu Kim, Jatin Chhugani et. al "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU", 2010
[5] NVIDIA GeForce GTX 1080, https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/
[6] NVIDIA GeForce GTX 1080 White Paper, https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf
[7] GPU Accelerated Computing with C and C++ - https://developer.nvidia.com/how-to-cuda-c-cpp
[8] Programming Model - Kernel http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels

[9] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, Wen-mei W. Hwu, ”CUDA-Lite: Reducing GPU Programming”, 2008 Complexity
[10] Peter Bakkum, Kevin Skadron, ”Accelerating SQL Database Operations on a GPU with CUDA”, 2010
[11] Ashwin M. Aji, Mayank Daga, Wu-chun Feng, ”Bounding the Effect of Partition Camping in GPU Kernels”, 2011