Universiteit Leiden
Opleiding Informatica

Comparison of the effectiveness of shared memory optimizations for stencil computations on NVIDIA GPU architectures

Name:	Geerten Verweij
Date:	12/08/2016
1st supervisor:	Kristian Rietveld
2nd supervisor:	H.A.G. Wijshoff

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Stencil computations are commonly used in for example scientific simulations of heat distribution, fluid dynamics and seismic activity. The effectiveness of shared memory optimization in stencil computations on different NVIDIA GPU architectures is investigated in this thesis. Experiments on the previous GPU architectures Fermi and Kepler show that shared memory optimizations result in increasing efficiency as expected based on previous work, where the Kepler architecture GPU gets a greater performance increase than the Fermi architecture GPU. For two newer GPU architectures, Maxwell and Pascal, extensive GPU analysis is done to discover why the Maxwell GPUs almost doubles in speed with shared memory optimization while the Pascal architecture GPU only shows a speedup of about 10%. A conclusive explanation for this difference is not found which calls for future research to be done on the use of shared memory in stencil kernels on Pascal architecture GPUs.

Acknowledgements

My thanks go out to Orson Peters for helping me setting up a testing machine, to Leiden University for providing access to a lot of hardware, to Kristian Rietveld for guiding me and having a lot of patience and to Anika Noort for helping me regaining my focus on an academic future.

Contents

Abstract i

Acknowledgements iii

1 Introduction 3

2 Definitions 4

2.1 The Stencil . . . 4

2.2 The GPU . . . 5

2.3 Shared Memory optimization . . . 6

3 Related Work 7

4 Experimental Setup 8

4.1 The kernel code . . . 8

4.2 The benchmarking . . . 9

4.3 The hardware . . . 9

5 Evaluation 11

5.1 Shared memory use in Fermi and Kepler . . . 11

5.2 Shared Memory Optimization . . . 12

5.3 Maxwell speedups versus Kepler speedups . . . 12

5.4 Pascal speedups versus Maxwell speedups . . . 13

5.4.1 Non-optimized stencil code . . . 13

5.4.2 Optimized stencil code . . . 15

6 Conclusions 17

6.1 Future Work . . . 18

Bibliography 18

List of Tables

4.1 List of GPUs that have been tested. 10

5.1 L1 cache and shared memory usage. 12
5.2 Execution time and speedup of the shared memory optimization on different NVIDIA GPUs . 12
5.3 Shared Memory Performance of Optimized Stencil Kernel In Kepler and Maxwell 13
5.4 Memory Benchmark Results. 13
5.5 Non-optimized stencil code nvprof analysis. 14
5.6 Optimized stencil code nvprof analysis. 16

List of Figures

2.1 A 5-point stencil 4
2.2 the formula for the update value in point x,y,z 4
2.3 GP104 SM Diagram 5

4.1 Each column represents the 10th, 125th and 256th slice in the cube. Each row represents 100th, 1000th, 3000th and 10000th timestep 10

Chapter 1

Introduction

Stencil computations are used in many numerical solvers and physical simulations. Their field of application lies mostly within scientific computing for example in seismic computing, image processing and computational fluid dynamics. In this thesis, we will focus on a stencil kernel that solves a heatwave equation to simulate the spread of heat through a space. Multiple instances of the stencil kernel will calculate the heat of each data point in parallel. This results in a fast estimate of the heatwave equation.

Graphics Processing Units (GPUs) are very suited for executing stencil computations, because they have a lot of simple cores that can all do the same stencil computation on a different part of the data in parallel. In this thesis we will analyze the performance of stencil computations on different NVIDIA GPUs. In particular, we will study the effects of a code optimization that enables the use of shared memory present on GPUs. The results on older hardware will be compared to results in previously published work and will be used to validate our benchmarks. The results on more recent hardware will be analyzed to determine whether these optimizations are still fruitful on recent hardware and to find opportunities for further performance optimization specifically for these newer GPU architectures.

Chapter 2 explains some definitions needed to understand the contents of this thesis. In Chapter 3 we will discusses the related works that brought us to researching this topic and to which we compare our results. In Chapter 4 the setup of our experiment is explained. In Chapter 5 we evaluate and discuss the test results. Chapter 6 concludes the thesis.

Chapter 2

Definitions

2.1 The Stencil

A stencil is a calculation that can be applied to multiple data points in parallel. Stencils can have different sizes and shapes depending on what is needed for the application. Figure 2.1 shows a simple example with a small cross-shaped 5-point stencil on a 2D array. It takes the four data points surrounding the middle point and calculates the new value for the middle array element. This calculation is done for all elements in the array. This example is in 2D, but this thesis we use a 3D stencil with a formula from Micikevicius's work [Mico9] shown in Figure 2.2 on a 3D array of data points. This is a stencil only computation for measuring GPU performance. Considering GPU memory pressure and computational load it is similar to the finite difference discretization of the wave equation which is the basis for the reverse time migration algorithm in seismic computing. k is the order of the stencil, x,y,z are the coordinates of the point for which the new value is calculated and t is the time-step. In case of an order 8 stencil, four points in each of the six possible directions are used. Combined with the middle point, 25 data points are used to calculate one new value. The stencil kernel used in this thesis is thus called a 25-point stencil. This stencil is not chosen

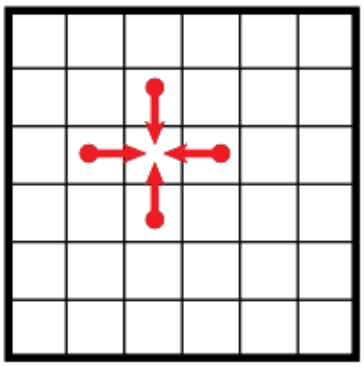

Figure 2.1: A 5-point stencil

$$D^{t+1}_{x,y,z} = c_0 D^t_{x,y,z} + \sum_{i=1}^{k/2} c_i \left(D^t_{x-i,y,z} + D^t_{x+i,y,z} + D^t_{x,y-i,z} + D^t_{x,y+i,z} + D^t_{x,y,z-i} + D^t_{x,y,z+i} \right)$$

Figure 2.2: the formula for the update value in point x,y,z

for this thesis because of the output of the computations but because it has been used for performance measurements before, many different stencils exist with other formulas, sizes, shapes and applications.

2.2 The GPU

The graphics processing unit is very well suited for doing stencil computations. It consists of many small multiprocessors each with their own shared memory and a global memory for the whole GPU. Figure 2.3, from the GTX1080 whitepaper [NVI16], shows a single streaming multiprocessor (SM) of a GP104 chip which is in the GTX1080. The GP104 chip has 20 of these SMs which all share the same L2 cache and DRAM. In this single SM there are 128 CUDA cores. NVIDIA's CUDA gives us the options to write kernel code that utilizes this architecture. A kernel defines code for one thread that will be executed by one core in a multiprocessor. Those threads are then grouped into thread blocks which can all access the same shared memory of the multiprocessor (seen at the bottom of the figure). This shared memory is faster than the global memory but much smaller. In the Fermi architecture this shared memory is used as a cache for the global memory and in the Kepler architecture this optimization is not implemented as discussed in Maruyama's and Aoki's [MA14]. The speed of the stencil computation is limited by the memory bandwith of a GPU. This is because GPUs in general can do more floating points operations per second than they can load floating points per second. In the article of Su et al about GPU performance they take the NVIDIA Tesla K20 as an example. It has to do 45 floating-point operations for each floating point it loads to get to peak performance [SCWZ15].

Figure 2.3: GP104 SM Diagram

2.3 Shared Memory optimization

To prevent the limited global memory speed of a GPU to be the bottleneck of stencil computations shared memory can be used. The data that is used by the threads in one thread block overlaps because they use data next to the data point they are trying to calculate. Because of this overlap a thread can pre-load it's own data point to shared memory and all the threads in the same thread block can read that data point from shared memory, instead of global memory. Then all the threads in that block have quicker access to most of the data compared to when they need to load it all from global memory. This reduces the pressure on the global memory. The thread block is a 16x16 plane in x,y. The stencil is a cross in x,y,z-direction. The data in the z-direction does not overlap between threads. The data in the x,y-plane does overlap between threads in the same thread block. We implemented two kernel codes, one that reads all data from global memory and one that loads partially from global and partially from shared memory. The optimized kernel using shared memory loads the data point of each thread into shared memory for other threads to use, this is the 16x16-plane overlap. But because the stencil reaches beyond the edge of the thread block we also load in that data in to shared memory. This is similar to [Mic09].

Chapter 3

Related Work

Implementations and optimizations of stencil kernels have been described in literature before. The following works inspired this thesis by providing examples for stencil kernels and optimizations. They also gave the insight on how GPUs work in the context of stencil kernels. The main contribution of this thesis is to investigate the effectiveness of the shared memory optimization on the newer Maxwell and Pascal architectures, which has not been described in the literature up till now.

In [Mic09] an initial implementation of stencil kernels and a shared memory optimization on NVIDIA Tesla GPUs is described. To be able to deal with data sets that exceed the memory capacity of a single GPU a multi-GPU approach is proposed. Within this thesis we use the 3-D stencil kernel as described in this paper to show the effectiveness of shared memory optimization on newer architectures such as Maxwell and Pascal. The paper shows the performance speedup of stencil computation when using their GPU approach compared to a CPU approach and the linear scaling of their multi-GPU approach.

In [MA14] it is concluded that shared memory optimization is more effective on the Kepler architecture than on the Fermi architecture. We try to validate this conclusions and we will also extend their findings by adding Maxwell and Pascal GPUs to the comparison.

In [SCWZ15] a model to predict GPU performance on stencil computations is provided. Because it uses double precision floating points, it is not directly applicable for this thesis but it does give a good insight in a GPU's memory hierarchy and performance.

In [VINS14] different optimizations of stencil kernels are compared using the Fermi and Kepler architectures. They also compare desktop and mobile GPUs. A conclusion that we try to validate is that "shared memory usage has become essential for double precision stencil based computation on Kepler GPUs.". The paper also shows a performance decrease linear to power consumption between mobile and desktop Kepler GPUs.

Chapter 4

Experimental Setup

In this chapter the different parts used in the experiment on both the software and hardware side will be explained. First the kernel codes that we try to benchmark are explained. The benchmarking and benchmark tools are described. And finally the choice of hardware and a list of GPUs is provided.

4.1 The kernel code

The main stencil code that is considered in this thesis is based on the code found in [Mic09]. The main change to this code is an out-of-bounds boolean that makes sure the kernel only calculates the value of elements that are at least $k/2$ elements away from the edge. In the original implementation this was not done and the calculations are incorrect because data that is out of bounds is used. In the original paper this incorrectness was not seen as a problem, because the focus was to determine the maximum attainable performance.

Two different kernel codes are implemented; one that uses the shared memory of the streaming multiprocessor and one that does not. This is done to show the difference of performance gain from explicitly using shared memory in the kernel code. The kernel code it self is not enough to run the stencil computations. Therefore we used the Parboil stencil benchmark framework [LWC12]. This framework provides us with a CPU context to place the kernel code in and the means to measure the performance of the kernel code. In CPU code the data and kernel code is loaded onto the GPU and the kernel code is then repeated for n timesteps. The CPU code also performs all the input and output after the kernel is done and it keeps a timer to measure the execution time of the kernel. For this thesis we only measured the time the kernel takes to complete a simulation and so CPU performance is ignored. The CPU code initializes the GPU to use 16x16 threads in one thread block. These thread blocks then propagate through the z-axis of the 3D data array. This data array is initialized to represent a box at 40 degrees Celsius with edges that are minus 273 degree Celsius.

The right side of the box is also set at 40 degrees Celsius. Because of the out-of-bounds boolean mentioned before these edges remain their initial temperature. The resulting simulation is a box that is cooling down towards the 5 cool sides but remains hot on the right side. A visualisation of example output, created using a 512x512x512 points cube, can be see in Figure 4.1. In this figure it becomes apparent that all edges have started to cool down after executing 100 timesteps. The 10th slice is a lot cooler because it is closer to the minus 273 degree edge. At this timestep there is not a big difference between the 125th and 256th slice. After 10000 timesteps a clear difference between the temperature at the edge, halfway to the middle and the middle of the cube becomes apparent. It is also shown that the right side of the cube retains heat from the unchanging 40 degree edge.

4.2 The benchmarking

The benchmarking was done by running the two different kernel codes 10 times on each GPU. Because we are solely interested in the running time on the GPU, only the kernel run-time was measured which excludes any variation in CPU run-time. Each measurement was repeated 10 times and the execution times were averaged after verifying that the difference between the measurements was negligible.

Nvprof is used to further analyze the performance of the GPUs. This was done separately of the benchmarking since it greatly impacts the run-time of the kernel. Nvprof is a CUDA profiling tool from NVIDIA which can measure different metrics and events within the GPU while executing kernel code. We used it to get insight in memory usage and possible causes of performance differences between the GPUs.

4.3 The hardware

The GPUs have been tested in different Linux-based systems. They all had different hardware but for this experiment we can ignore the CPU hardware since the speed of the CPU is not measured. Only one CPU operation is involved while the GPU kernel time is measured and that is swapping the pointers to the input and output arrays. This operation is negligible compared to the many operations the GPU does between this operation which is processing the entire array for one time step. The individual results hardly deviated from the average of 10 measurements so if the CPU happened to be busy sometimes it hardly made an impact on kernel performance. A list of different GPUs that have been surveyed can be found in Table 4.1.

Name	Architecture	Chip	Release year	Memory Bandwidth (GB/s)	CUDA Cores
GTX480	Fermi	GF100	2010	177.4	480
GTX660	Kepler	GK106-400-A1	2012	144.2	960
GTX980ti	Maxwell	GM200-310	2015	336.5	2816
TitanX	Maxwell	GM200-400	2015	336.5	3072
GTX1080	Pascal	GP104-400	2016	320	2560

Table 4.1: List of GPUs that have been tested.

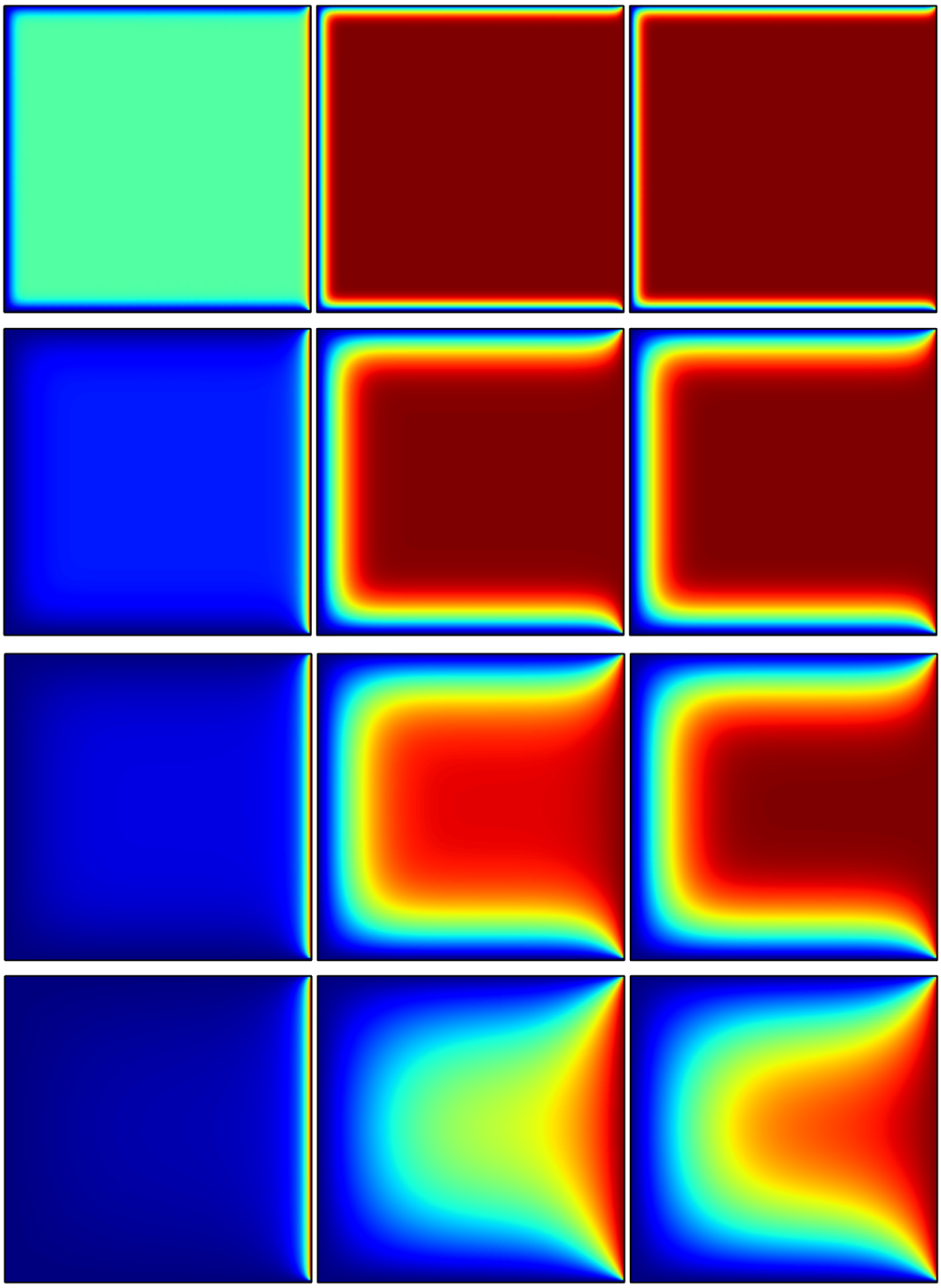

Figure 4.1: Each column represents the 10th, 125th and 256th slice in the cube. Each row represents 100th, 1000th, 3000th and 10000th timestep

Chapter 5

Evaluation

In this chapter the results of the benchmarks are presented and discussed. We will try to explain these results using documentation on the hardware, previous work on stencil kernels and the nvprof analysis.

5.1 Shared memory use in Fermi and Kepler

An important difference between the Fermi and Kepler architectures is that the Fermi architecture caches DRAM loads in the L1 cache and that Kepler does not. Maruyama and Aoki confirm that the Fermi architecture uses the L1 cache by default for DRAM loads to reduce DRAM pressure [MA14]. They also state that because Kepler does not do so, shared memory optimizations are essential for this architecture.

To validate this claim, we ran the stencil kernel with and without shared memory support on a Fermi (GTX480) and Kepler (GTX660) GPU. The results can be seen in Table 5.1. From this table can be seen that the GTX480 had about 16M L1 cache load hits on average per kernel call. In this kernel there are no explicit shared memory calls and the profiling tools showed no shared memory hits. When running the kernel that does have explicit shared memory calls we saw about 160k L1 cache hits and about 7M shared memory load hits. The L1 cache and the shared memory are actually the same memory on the hardware. When doing the same analysis on the GTX660 which has a Kepler GPU we saw no L1 cache usage with any of the two kernels and only shared memory usage with the kernel that explicitly uses it. We saw the exact same amount of shared memory hits as with the GTX480. This analysis proves us that Fermi uses the L1-cache/shared-memory by default and Kepler does not. This would cause us to expect more performance gain from shared memory optimization in Kepler than in Fermi. This will be discussed in the next section.

GPU	kernel implementation	L1 cache load hits	shared memory load hits
GTX480	without shared memory	16090392	0
GTX480	with shared memory	163706	7225344
GTX660	without shared memory	0	0
GTX660	with shared memory	0	7225344

Table 5.1: L1 cache and shared memory usage.

5.2 Shared Memory Optimization

To evaluate the effectiveness of the shared memory optimization on different NVIDIA architectures, the performance of the stencil kernels with and without shared memory usage was measured on a 512x512x64 points array for 100 time steps. The calculation per point was that of Figure 2.2 with $k = 8$. Table 5.2 shows execution time for the kernels with and without shared memory optimization and the speedup when applying shared memory optimization for each tested GPU. The GTX660 with Kepler architecture shows a greater speedup than the GTX480 with Fermi architecture, 1.642 and 1.316 times faster respectively, confirming our expectation mentioned in the previous section. Maxwell shows greater speedups compared to Kepler. Pascal shows the smallest speedup of all the architectures. In the next sections we will elaborate on the possible causes of these differences in speedups.

GPU	Architecture	Non optimized execution time	Optimized execution time	speedup
GTX480	Fermi	0.438178 s	0.3328737 s	1.31635
GTX660	Kepler	0.8197876 s	0.4992285 s	1.64211
GTX980ti	Maxwell	0.2158845 s	0.1123426 s	1.92166
TitanX	Maxwell	0.205462 s	0.1083434 s	1.89630
GTX1080	Pascal	0.1085029 s	0.097338 s	1.11470

Table 5.2: Execution time and speedup of the shared memory optimization on different NVIDIA GPUs

5.3 Maxwell speedups versus Kepler speedups

The GTX980ti and TitanX with Maxwell architecture show a greater speedup than the GTX660 with Kepler architecture, 1.921 and 1.896 respectively versus 1.642, when using shared memory. The NVIDIA Maxwell Tuning Guide tells us "As with Kepler, global loads in Maxwell are cached in L2 only" [NVI15]. So, because the L1 cache is not used by default, a significant speedup is expected when applying shared memory optimization with Maxwell. This is a greater speedup with Maxwell than with Kepler likely due to the larger dedicated shared memory in Maxwell, which causes better occupancy. Furthermore, the Maxwell architecture has a higher shared memory bandwidth for 32-bit elements, such as the 32-bit floats from the kernels we use, as can be seen in Section 1.4.3 of the Maxwell Tuning Guide [NVI15]. Table 5.3 shows the results of nvprof analysis on the kernel code with shared memory optimization. The GTX980ti Maxwell GPU has a greater shared memory throughput, greater shared memory utilization and a greater shared memory efficiency than

the GTX660 Kepler GPU. This is to be expected considering the changes from Kepler to Maxwell stated in the Maxwell Tuning Guide and it is clear that the shared memory optimization is even more effective on the Maxwell architecture than on the Kepler architecture.

GPU	Architecture	Sh. Mem. Load Throughput	Sh. Mem. Utilization	Sh. Mem. Efficiency
GTX660	Kepler	578.28 GB/s	mid(5)	30.39%
GTX980ti	Maxwell	2000 GB/s	high(7)	47.96%

Table 5.3: Shared Memory Performance of Optimized Stencil Kernel In Kepler and Maxwell

5.4 Pascal speedups versus Maxwell speedups

In Table 5.2 we can see that the GTX1080 with its Pascal architecture is almost 100% faster than the GTX980ti Maxwell GPU, when not using the shared memory optimization. When using the shared memory optimization the Pascal GPU is only about 15% faster than the Maxwell GPU. This is a very interesting result because the specifications of both GPUs suggest the newer GTX1080 to be faster than the older GTX980ti but not twice as fast. At the same time the shared memory optimization has a far smaller impact on the Pascal GPU than both the Maxwell GPU's and all earlier architecture generations mentioned in this thesis. The 10-15% speedup from Maxwell to Pascal using shared memory optimization is quite low considering the increase in theoretical GFlops and improvements to the memory architecture. Table 5.4 shows the theoretical compute throughput, memory bandwidth, shared memory performance and L2 cache throughput measured using a specialized benchmark from Konstantinidis [EK16]. The memory bandwidth of the GTX1080 is actually slightly lower than the GTX980ti and the theoretical compute throughput of the GTX1080 is 57% higher than the GTX980ti. When it comes to shared memory the GTX1080 shows a 83% higher throughput than the GTX980ti. The L2 cache throughput is 52% faster with the GTX1080 than the GTX980ti. None of these results seem to be representative of our stencil benchmark results as the GTX1080 is faster than expected when not using shared memory optimization and slower than expected when using shared memory optimization.

	GTX980ti	GTX1080
Compute throughput (theoretical single precision FMAs)	6060.03 GFlops	9523.20 GFlops
Memory bandwidth	336.48 GB/sec	320.32 GB/sec
Shared memory throughput (32bit)	2455.11 GB/sec	4484.07 GB/sec
Shared memory operations per clock (32bit)	570.42 (per SM 25.93)	602.70 (per SM 30.13)
L2 cache throughput	1696.48 GB/sec	2571.30 GB/sec

Table 5.4: Memory Benchmark Results.

5.4.1 Non-optimized stencil code

To find out why the GTX1080 is almost 100% faster than the GTX980ti when not optimizing with shared memory we had to do some deeper analysis. We ran the benchmarks again with nvprof on both GPUs to see

where this increase in performance might come from. Table 5.5 shows several metrics and events measured with nvprof on the non-optimized stencil code. These results show a lot of similarity. The global hit rate, global load transactions, L2 read transactions and global load transactions per request are all the same or almost the same. This suggests that the coalescing of memory accesses is done in the same way by both GPUs. The difference in device memory read throughput is most likely a result of some other bottleneck since the throughput is much lower than the theoretical maximum shown in Table 5.4 and most of the data is read from the L2 cache. The L2 cache throughput from the memory benchmark is 52% higher in the GTX1080 than the GTX980ti and when we run the stencil code it is 60% higher in the GTX1080 than the GTX980ti. Since the GTX980ti can be faster when using shared memory instead of the L2, it is likely that the L2 cache is its bottleneck but the peak of L2 cache speed is not reached. The GTX980ti does have a higher amount of stalls due to data requests, 14.9% versus 11.6% in the GTX1080. This can lead to more warps queueing up for data at the L2 cache which can lead to less effective occupancy. This could explain the lower L2 cache utilization in the GTX980ti and less performance relative to its maximum potential.

	GTX980ti	GTX1080
Shared Memory Load Transactions Per Request	0.000000	0.000000
Local Memory Load Transactions Per Request	0.000000	0.000000
Global Load Transactions Per Request	8.700000	8.700000
Shared Load Transactions	0	0
Local Load Transactions	0	0
Global Load Transactions	68419584	68419584
L2 Read Transactions	38797690	38896305
Global Hit Rate	43.30%	43.30%
Local Hit Rate	0.00%	0.00%
Requested Global Load Throughput	513.94GB/s	817.96GB/s
Global Load Throughput	633.86GB/s	1008.8GB/s
L2 Throughput (Reads)	633.87GB/s	1011.4GB/s
Local Memory Load Throughput	0.00000B/s	0.00000B/s
Shared Memory Load Throughput	0.00000B/s	0.00000B/s
Global Memory Load Efficiency	81.08%	81.08%
Issue Stall Reasons (Execution Dependency)	56.28%	53.83%
Issue Stall Reasons (Data Request)	14.91%	11.59%
Shared Memory Efficiency	0.00%	0.00%
L2 Cache Utilization	High (8)	Max (10)
Shared Memory Utilization	Idle (0)	Idle (0)
Device Memory Read Transactions	2497662	2606510
Device Memory Read Throughput	40.806GB/s	67.774GB/s
Device Memory Utilization	Low (2)	Mid (4)
Active cycles	46812743	38216784
Active warps	1482118322	1210565281
Instructions per warp	5.9054e+03	5.4994e+03
Branch Efficiency	100.00%	100.00%
Warp Execution Efficiency	99.61%	99.58%
Warp Non-Predicated Execution Efficiency	98.65%	98.56%
Instruction Replay Overhead	0.000044	0.000028

Table 5.5: Non-optimized stencil code nvprof analysis.

5.4.2 Optimized stencil code

The GTX1080 is about 15% faster than the GTX980ti when optimizing with shared memory. Table 5.6 shows the result of the same analysis as the previous section but then applied to the optimized stencil code. The Shared Memory Load Throughput of the GTX1080 is lower than the GTX980ti but the GTX1080's execution speed is still faster, probably due to the faster L2 cache and compute throughput of the GTX1080 . Instructions per warp decreases in the GTX980ti when applying the optimization but it does not decrease for the GTX1080 when applying the optimization. Instruction Replay Overhead remains about equal for the GTX980ti when applying the optimization but it increases for the GTX1080 when applying the optimization, which can explain the lack of decrease in instructions per warp increase for the GTX1080. Active cycles decreases a lot more in the GTX980ti when applying the optimization than the GTX1080. Active warps decreases in the GTX980ti when applying the optimization but with the GTX1080 it greatly increases. The ratio of active warps and active cycles is the same for the GTX980ti and the GTX1080 in both the optimized and non-optimized situation. This suggests that the optimized kernel is less effectively executed by the GTX1080 compared to the GTX980ti. However the shared load bank conflicts and shared load transactions are the same in the GTX980ti and GTX1080. Issue Stall Reasons (Data Request) increase for both the GTX980ti and the GTX1080 when applying the optimization, but it increases a lot more for the GTX1080. Putting all this together it seems that the GTX1080 handles the bank conflicts in shared memory loads very badly. An increase in bank conflicts leads to an increase in instruction replays and thus in increases in active cycles and active warps. The fact that the shared memory efficiency, branch efficiency, warp execution efficiency and warp non-predicated execution efficiency is equal for both GPUs, further confirms this. Finally, given that the same amount of instructions per warp are executed, but warps are active for a larger number of cycles when using the shared memory optimization suggests the GTX1080 is not used as efficiently as the GTX980ti.

	GTX980ti	GTX1080
Shared Memory Load Transactions Per Request	2.000000	2.000000
Local Memory Load Transactions Per Request	0.000000	0.000000
Global Load Transactions Per Request	4.705882	4.705882
Shared Load Transactions	14450688	14450688
Local Load Transactions	0	0
Global Load Transactions	7864320	7864320
L2 Read Transactions	4850007	4928908
Global Hit Rate	38.33%	38.33%
Local Hit Rate	0.00%	0.00%
Requested Global Load Throughput	120.77GB/s	120.04GB/s
Global Load Throughput	148.94GB/s	148.10GB/s
L2 Throughput (Reads)	148.96GB/s	150.47GB/s
Local Memory Load Throughput	0.00000B/s	0.00000B/s
Shared Memory Load Throughput	2e+03GB/s	1764.6GB/s
Global Memory Load Efficiency	81.08%	81.05%
Issue Stall Reasons (Execution Dependency)	23.56%	16.69%
Issue Stall Reasons (Data Request)	40.39%	52.22%
Shared Memory Efficiency	47.96%	47.96%
L2 Cache Utilization	Low (3)	Low (3)
Shared Memory Utilization	High (7)	Mid (5)
Device Memory Read Transactions	2451465	3060942
Device Memory Read Throughput	75.290GB/s	93.442GB/s
Device Memory Utilization	Mid (5)	Mid (5)
Shared load bank conflict	7225344	7225344
Shared load transactions	14450688	14450688
Active cycles	23673599	36897631
Active warps	1104885204	1721897227
Instructions per warp	5.5644e+03	5.4864e+03
Branch Efficiency	100.00%	100.00%
Warp Execution Efficiency	99.33%	99.32%
Warp Non-Predicated Execution Efficiency	90.52%	90.90%
Instruction Replay Overhead	0.000043	0.000039

Table 5.6: Optimized stencil code nvprof analysis.

Chapter 6

Conclusions

Stencil computations can benefit from higher GPU performance for higher precision and/or shorter execution time. It is important to tailor the stencil code to suit the target architecture to be able to obtain optimal performance and also to keep up with the hardware improvements and changes that are made over the years. One optimisation found in previous work is the use of the faster shared memory instead of global memory. This optimization was beneficial for the Fermi architecture GPUs and more beneficial for Kepler GPUs. We tried to confirm this in this thesis and extend this to two newer GPU architectures; Maxwell and Pascal.

We can confirm that shared memory optimization in stencil kernels provides a significant performance increase. The Kepler architecture GPU did indeed show a greater increase in performance than the Fermi architecture GPU because Fermi architecture GPUs already use the shared memory/L1 cache by default.

The Maxwell architecture GPUs benefited even more from shared memory optimization than the Kepler architecture GPU , likely because the Maxwell architecture has a larger dedicated shared memory, causing better occupancy, and a higher shared memory bandwidth for 32-bit elements. It is very useful to remember this when using GPUs with a Maxwell architecture for stencil computations.

The Pascal architecture GPU also benefited from shared memory optimization but the performance gain was very small compared to the improvements seen in the Maxwell architecture GPUs. This was not expected when considering the greater potential compute throughput and shared memory throughput in the Pascal architecture GPU compared to the Maxwell architecture GPUs. We expect the cause to be that the Pascal architecture GPU handles shared memory bank conflicts in a much less efficient way than the Maxwell architecture GPUs because of the higher amount of active warps in the Pascal GPU while running the shared memory optimized stencil kernel.

6.1 Future Work

Because the Pascal architecture GPU shows relatively very good performance for the non optimized stencil kernel it would be very interesting to see if it is possible to further improve the execution time with shared memory optimization, just like with the Maxwell GPUs. To do this, more research is needed to the exact cause of the lack of performance increase when applying shared memory optimization to stencil kernels on a Pascal architecture GPU.

Bibliography

[EK16] Y Cotronis E Konstantinidis. A quantitative performance evaluation of fast on-chip memories of gpus. *24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2016.

[LWC12] Christopher Rodrigues Li-Wen Chang. Parboil stencil benchmark. *Parboil Benchmarks*, 2012.

[MA14] Naoya Maruyama and Takayuki Aoki. Optimizing stencil computations for nvidia kepler gpus. *HiStencils 2014*, 2014.

[Mic09] Paulius Micikevicius. 3d finite difference computation on gpus using cuda. *GPGPU2, March 8, 2009, Washington D.C., US.*, 2009.

[NVI15] NVIDIA. Tuning cuda applications for maxwell. *maxwell-tuning-guide*, 2015.

[NVI16] NVIDIA. Nvidia geforce gtx 1080 whitepaper. *NVIDIA-GeForce-GTX-1080-Whitepaper*, 2016.

[SCWZ15] Huayou Su, Xing Cai, Mei Wen, and Chunyuan Zhang. An analytical gpu performance model for 3d stencil computations from the angle of data traffic. *The Journal of Supercomputing*, 71(7), 2015.

[VINS14] Anamaria Vizitiu, Lucian Itu, Cosmin Nita, and Constantin Suciu. Optimized three-dimensional stencil computation on fermi and kepler gpus. *IEEE High Performance Extreme Computing Conference*, 2014.