C Programming and Performance Engineering

Lab Work
Hybrid CPU + GPU Programming and Performance Engineering using OpenMP + OpenACC + CUDA

January 28, 2018

Spyridoula Chrysikopoulou-Soldatou and Jeremy Williams

1. 2D Laplace equation - GPU

First of all, the following commands had to be typed before each session:

The PGI toolset was installed:	***module add pgi64/17.4***
The CUDA toolset was installed:	***module add cuda/7.5***
Show info about available GPUs:	***nvidia-smi***
Show GPU device info in order:	***pgaccelinfo***
Set order of GPU devices:	***export CUDA_VISIBLE_DEVICES=0,1***
Check order is as desired:	***pgaccelinfo***

All the codes were compiled with the command: ***pgcc -fast -acc -ta=tesla:cc30 -Minfo=accelcode.c -o codename*** and executed with ***1000*** iterations, with the use of ***perf stat*** and ***pgprof*** in order to measure basic performance metrics. Every code is resulted from the previous one, with some alterations, and it has been checked that provides correct results.

1.1. Baseline code

Firstly, the baseline laplace2d.c code was modified in order to include OpenACC directives. Analytically, the "**#pragma acc data copyin(A,Anew)**" was inserted before the "while loop", in order to move the contents of both matrices from host to device before the convergence loop. Furthermore, the "**#pragma acc kernels**" were inserted before each of the outmost loop of "for loops". The use of acc kernels provides massive parallelization. This directive basically tells the compiler to analyze all the code that is assigned to. As a result, the compiler "understands" the code in the annotated accelerated region and does all the parallelizing code.

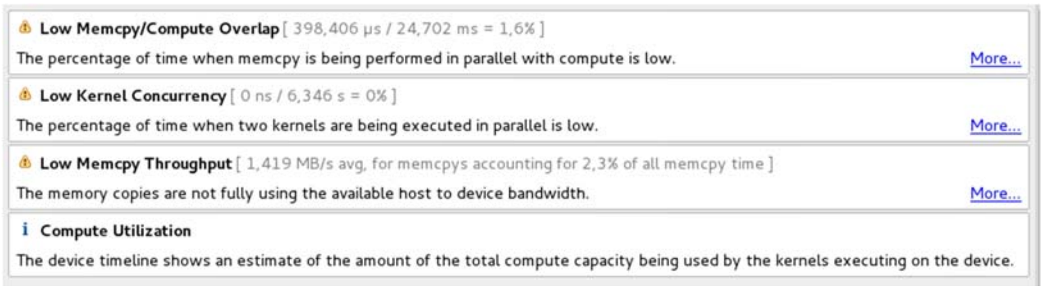
The most relevant performance metrics for the annotated baseline code are the elapsed execution time (5,88 seconds time elapsed), the total number of executed machine instructions (22,57 Giga), and the IPC rate (1,29 insns per cycle).

The command pgprof was used to measure the execution time of each kernel:

Time(%)	Time	Calls	Avg	Min	Max	Name
46.26%	2.17641s	1000	2.1764ms	2.1549ms	2.4731ms	main_88_gpu
26.43%	1.24358s	1000	1.2436ms	1.2360ms	1.3222ms	main_82_gpu
22.98%	1.08131s	1000	1.0813ms	1.0751ms	1.1417ms	main_93_gpu
3.74%	175.75ms	1000	175.75us	174.37us	185.66us	main_89_gpu_red

The kernel computing the error takes 46.26% of the execution time on GPU, the kernel computing the Anew takes 26.43%, the kernel copying Anew to A 22.98% and the kernel doing the final reduction takes only 3.74% of the execution time.

Afterwards, the NVIDIA Visual Profiler (**nvvp**) tool was used in order to provide information about understanding the performance of the GPU application. The results of the GPU usage are shown below:



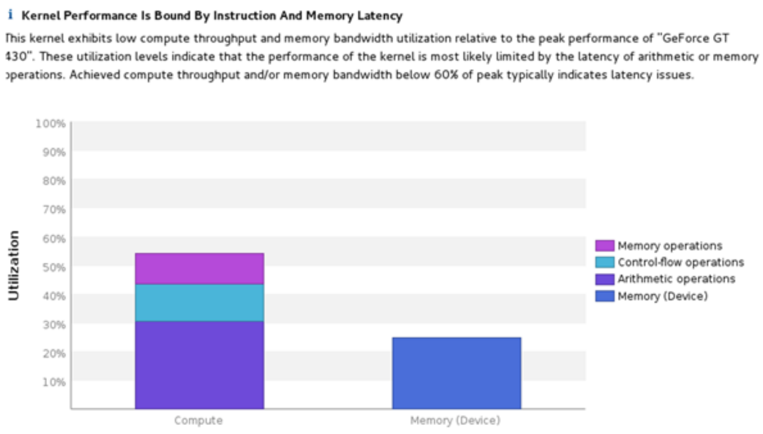
⚠ **Low Memcpy/Compute Overlap** [398,406 µs / 24,702 ms = 1,6%]
The percentage of time when memcpy is being performed in parallel with compute is low. More...

⚠ **Low Kernel Concurrency** [0 ns / 6,346 s = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

⚠ **Low Memcpy Throughput** [1,419 MB/s avg, for memcpys accounting for 2,3% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

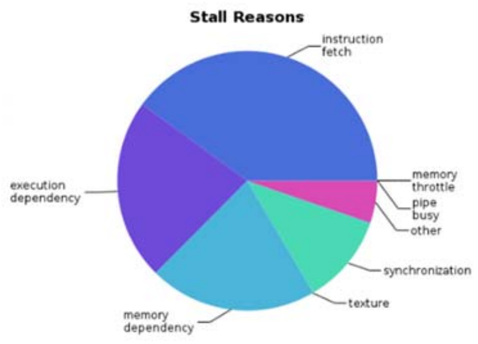
ℹ **Compute Utilization**
The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device.

When examining the kernel that computes the error, which has a rank of 100 in the optimization importance, the results obtained are the following:



ℹ **Kernel Performance Is Bound By Instruction And Memory Latency**
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GT 430". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

⚠ **Instruction Latencies May Be Limiting Performance**



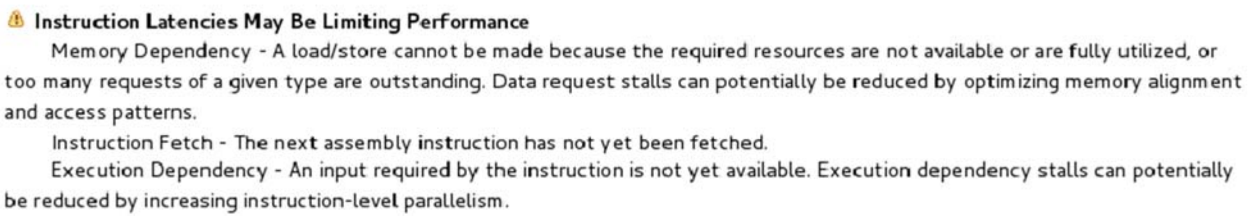
The compute units are 55% utilized, while the memory units are around 25% utilized. The memory resource with the higher utilization percentage is the device memory.

The utilization of both resources is low (both compute and memory resources). As a result it can be assumed that the performance is bounded by latency, and specifically, as introduced by the profiler, instruction and memory latency.

Latency issues indicate that the hardware resources are not used efficiently, since most warps (threads in groups of 128 parallel threads) are stalled by a dependency on a data value from a previous math or memory instruction.

The pie chart shows that the three primary stall reasons in this kernel are instruction fetch, execution dependency and memory dependency.

Some information given by the program is:



⚠ **Instruction Latencies May Be Limiting Performance**
Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.
Instruction Fetch - The next assembly instruction has not yet been fetched.
Execution Dependency - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.

Regarding the instruction fetch, it signifies that the next instructions stalled to be fetched from the memory address that is currently stored in the program counter and stored into the instruction register. At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.

Instruction fetch stalling could be caused by the fact that the Instructions are stored in global memory and that the SM has to load instructions before executing them.

As a result, the bottleneck of the program is memory and execution dependency in the computation of the error. That is probably because Anew and A are computed first, and are stored in the device memory. As a result, each tread has to access the memory to compute the error.

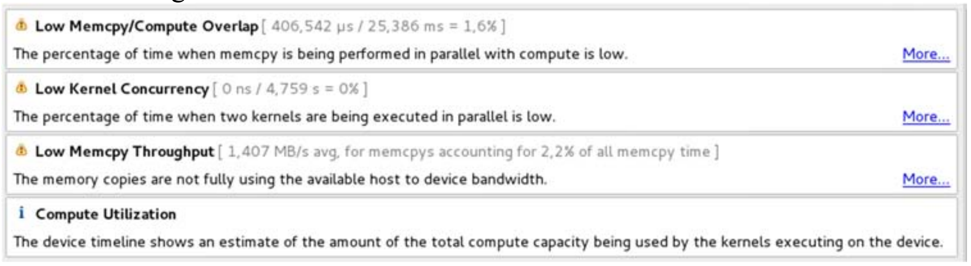
1.2.Loop fusion – GPU – OpenACC

The two for loops that computed the Anew and error were fused in the same loop, and the new code was compiled and executed.

The most relevant performance metrics are the elapsed execution time (5,32 seconds time elapsed), the total number of executed machine instructions (18.64 Giga), and the IPC rate (1,24 insns per cycle).

The kernel computing the Anew and the error takes 67.37% of the execution time on GPU, the kernel copying Anew to A 27.47%, while the kernel doing the final reduction takes only 4.47% of the execution time.

The results of the GPU usage are shown below:



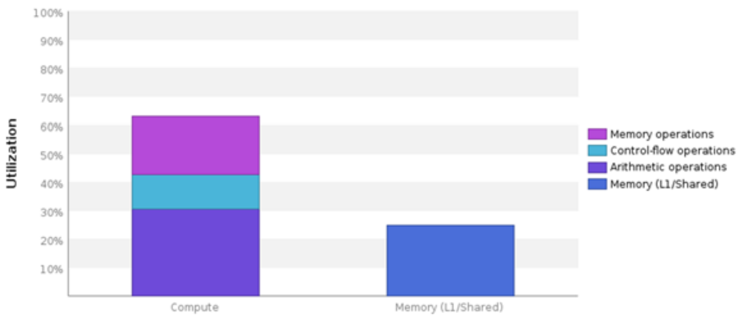
Low Memcpy/Compute Overlap [406,542 µs / 25,386 ms = 1,6%]
The percentage of time when memcpy is being performed in parallel with compute is low. More...

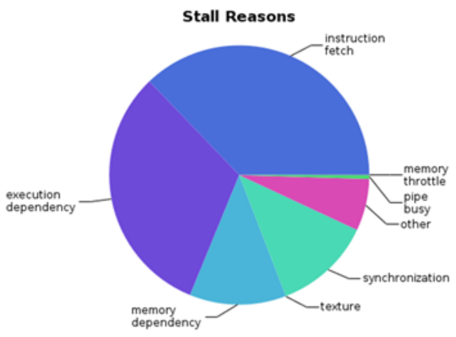
Low Kernel Concurrency [0 ns / 4,759 s = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Memcpy Throughput [1,407 MB/s avg, for memcpys accounting for 2,2% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

Compute Utilization
The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device.

Kernel Performance Is Bound By Instruction And Memory Latency
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GT 430". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



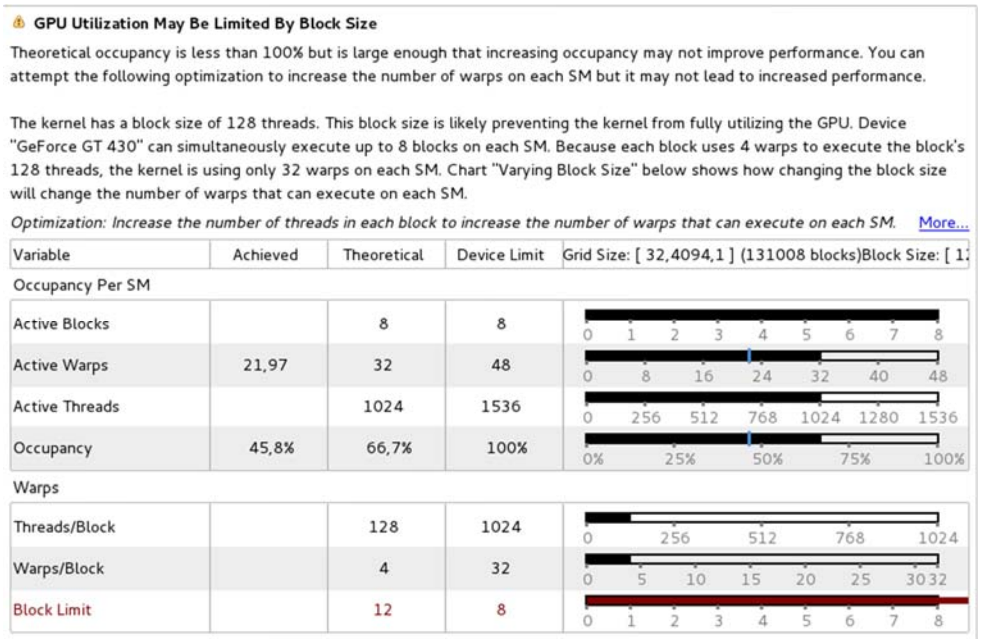


The compute resources are used of around 65% of their capacity, while the memory resources have an utilization of around 25%. Based on the previous results, now that the Anew and error are being computed in the same loop, the memory resource with the higher utilization percentage is the L1 cache, which is private for each multiprocessor and the Shared memory.

The utilization of memory resources is low. As a result, the performance is bounded again by instruction and memory latency. The main stall reasons in this kernel are instruction fetch and execution dependency.

Again is assumed that most warps (threads in groups of 128 parallel threads) are stalled by a dependency on a data value from a previous math or memory instruction.

According to the NVIDIA Visual Profiler, the main probable reason for the low GPU utilization in this case is the block size of the kernel, because is likely preventing the kernel from fully utilizing the GPU. And as a result, that the performance of the kernel is limited by the amount of parallelism in each gang. The achieved *Occupancy* is 45,8%, while the Theoretical is 66.7%. Occupancy is a measure of how much parallelism is running on the GPU versus how much theoretically could be running.



GPU Utilization May Be Limited By Block Size

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

The kernel has a block size of 128 threads. This block size is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GT 430" can simultaneously execute up to 8 blocks on each SM. Because each block uses 4 warps to execute the block's 128 threads, the kernel is using only 32 warps on each SM. Chart "Varying Block Size" below shows how changing the block size will change the number of warps that can execute on each SM.

Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM. More...

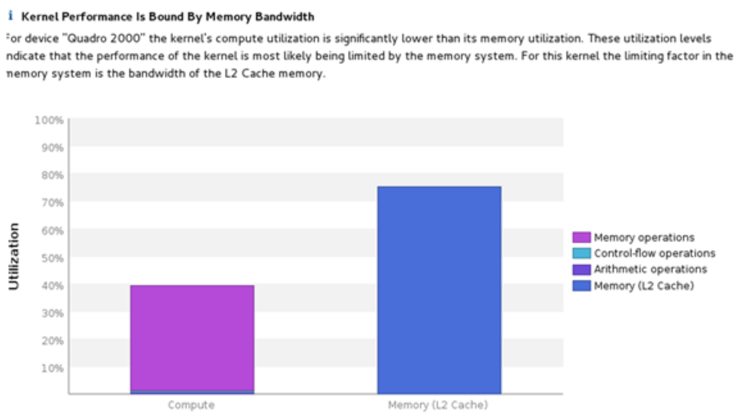
Variable	Achieved	Theoretical	Device Limit	Grid Size: [32,4094,1] (131008 blocks)Block Size: [1
Occupancy Per SM				
Active Blocks		8	8	0 1 2 3 4 5 6 7 8
Active Warps	21,97	32	48	0 8 16 24 32 40 48
Active Threads		1024	1536	0 256 512 768 1024 1280 1536
Occupancy	45,8%	66,7%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		128	1024	0 256 512 768 1024
Warps/Block		4	32	0 5 10 15 20 25 30 32
Block Limit		12	8	0 1 2 3 4 5 6 7 8

The *Block Limit* metric is highlighted in red. The occupancy table is showing that the GPU *streaming multiprocessor (SM)* can theoretically run 48 *warps* (groups of 128 threads), but has only 8 blocks to run. Looking at the *Warps/Block* and *Threads/Block* rows of the table, can be seen that each block contains 4 warps, although it could run many more. This is because the compiler uses a vector length of 128.

As a result, according also to the optimization proposed by the program itself, if the number of threads increases this problem will be solved.

For this reason, an alteration in the program consist of replace the acc kernels before each nested for loop with the necessary pragmas. Analytically, the **"#pragma acc parallel loop gang vector_length(256)"** was put in front of the outmost loop and the **"#pragma acc loop vector"** before the innermost for loop, and the program was executed again.

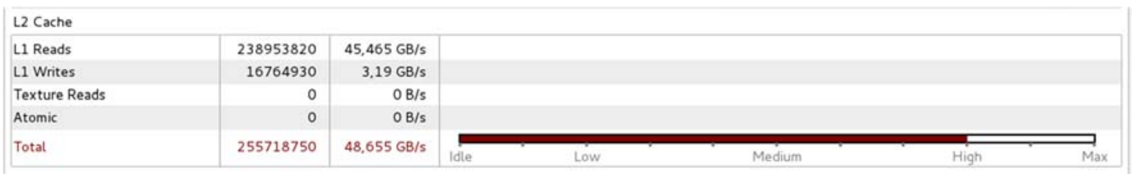
This time, the elapsed execution time is a lot greater, 177,9 seconds, while the total number of executed machine instructions is 832 Giga and the IPC rate 1,42 insns per cycle.
The results of the NVIDIA Visual Profiler are being presented below:


Kernel Performance Is Bound By Memory Bandwidth

or device "Quadro 2000" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels ndicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the nemory system is the bandwidth of the L2 Cache memory.

This time, the results indicate that the compute resources are used 40% of their capacity, while the memory resources achieve an utilization of around 75%. That means that the computation cost is smaller than the cost of moving data of the matrices between the GPU computation resources and the L2 Cache memory (shared by all the SMs), which is the memory resource with the higher utilization percentage is the L2 cache.

The diagnostic of the performance tool is that the performance of the kernel is most likely limited by the bandwidth of the device memory. Therefore, it seems that the computation of the error and Anew is not a performance problem.


L2 Cache

L1 Reads	238953820	45,465 GB/s
L1 Writes	16764930	3,19 GB/s
Texture Reads	0	0 B/s
Atomic	0	0 B/s
Total	255718750	48,655 GB/s

Idle
Low
Medium
High
Max

When getting more information about the memory bandwidth for all types of memory of the device, can be seen again that only of L2 Cache limits the performance and that there is an achieved 80% of the peak bandwidth.

1.3. Loop Interchange – GPU – OpenACC

Afterwards, the loops of i and j have been interchanged.

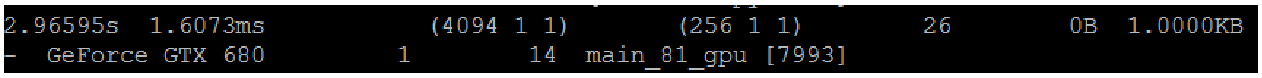
First the program was ran with the acc kernels directives (128 threads). Again the profiler showed the same initial bottleneck as before, the block size, and the optimization proposed by the program was again the increase of the threads.

The program was executed for 256, 512 and 1024 threads. For each one of them, according to the profiler the bottleneck is again instruction and memory latency, but caused by execution dependency and instruction fetch.

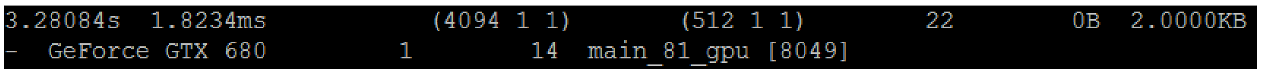
Of course in all cases the more execution time on GPU takes the kernel that computes the Anew and the error. Below are compared the execution times of the program when ran with 128 threads, 256, 512 and 1024 threads:

Number of threads	**128**	**256**	**512**	**1024**
Execution time (sec)	54	40	43	45

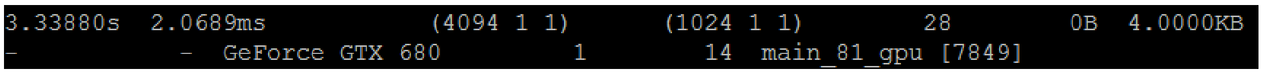
It can be observed that after the threshold of 256 threads, the execution time is increasing. If we activate the GPU activities, and focus on the computation of the error and the matrix Anew, we receive the following results for 256 (first), 512 (second) and 1024 (third) threads:



```
2.96595s  1.6073ms           (4094 1 1)        (256 1 1)        26         0B  1.0000KB
-   GeForce GTX 680         1          14  main_81_gpu [7993]
```

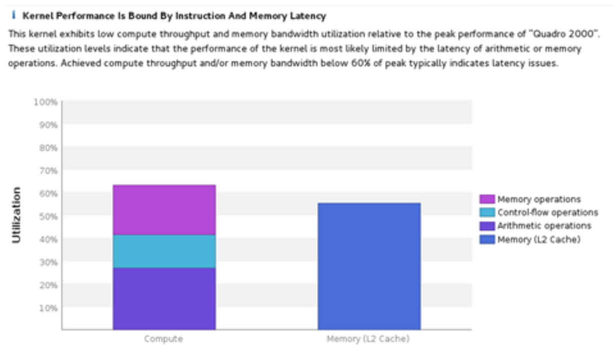


```
3.28084s  1.8234ms           (4094 1 1)        (512 1 1)        22         0B  2.0000KB
-   GeForce GTX 680         1          14  main_81_gpu [8049]
```



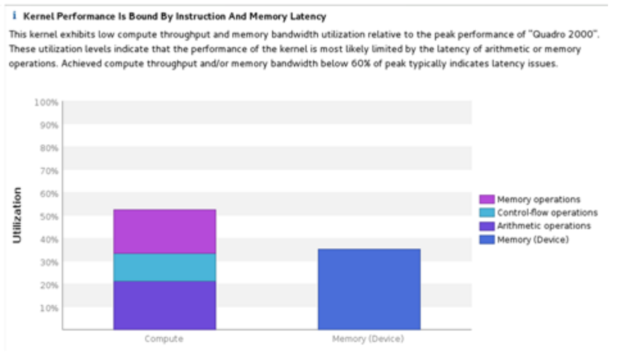
```
3.33880s  2.0689ms           (4094 1 1)       (1024 1 1)        28         0B  4.0000KB
-                -   GeForce GTX 680           1          14  main_81_gpu [7849]
```

As a result, the most optimal number of threads regarding the execution time is 256 threads.
Below are presented the results of the kernel performance in the case of 256and 1024 threads respectively.



Kernel Performance Is Bound By Instruction And Memory Latency

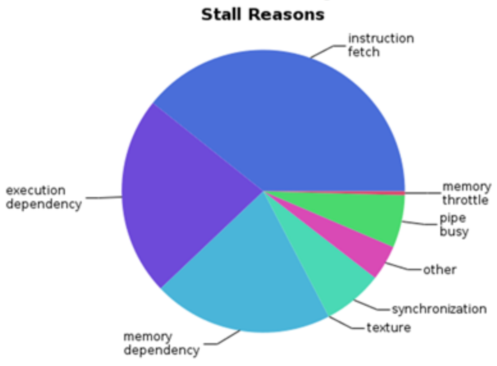
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Quadro 2000". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Quadro 2000". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

The compute resources in the case of 256 threads are used more, of around 65% and the memory 55%, while in the case of 1024 threads it is obvious that the utilization of the resources is lower. It is obvious that again the option of 256 threads is better, as when having more threads they are not used properly.



The main stall reasons are the instruction fetch and the execution depedency.

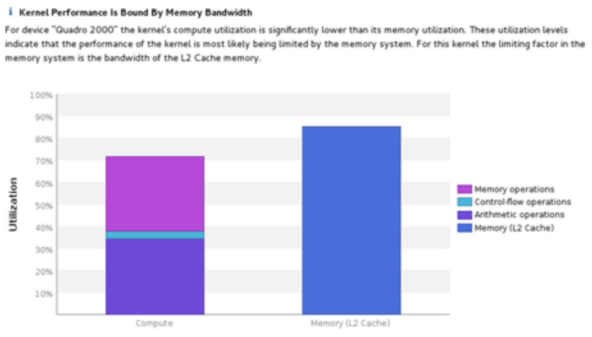
1.4. Move sqrtf out of loop – GPU – OpenACC

In this version of the code, the sqrtf() was removed from the inner loop and was moved outside.
Again in this version of the code, 256 threads were selected.

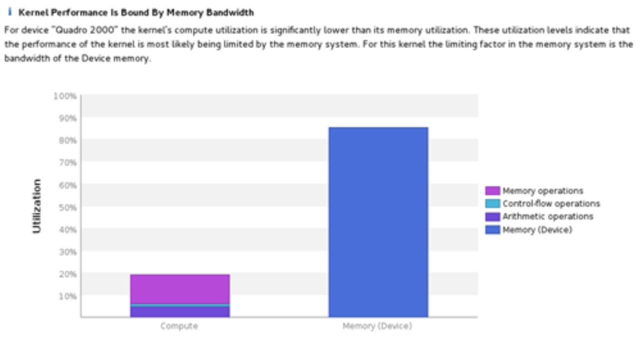
The most relevant performance metrics are the elapsed execution time (3,84 seconds time elapsed), the total number of executed machine instructions (12.91 Giga), and the IPC rate (1,18 insns per cycle).

The kernel computing the Anew and the error the takes 59.50% of the execution time on GPU, the kernel copying Anew to A 39.07%, while the kernel doing the final reduction takes only 1.03% of the execution time.

Below are presented the results for the two most time consuming (and with the most optimization importance) kernels.


Kernel Performance Is Bound By Memory Bandwidth

For device "Quadro 2000" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the L2 Cache memory.


Kernel Performance Is Bound By Memory Bandwidth

For device "Quadro 2000" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the Device memory.

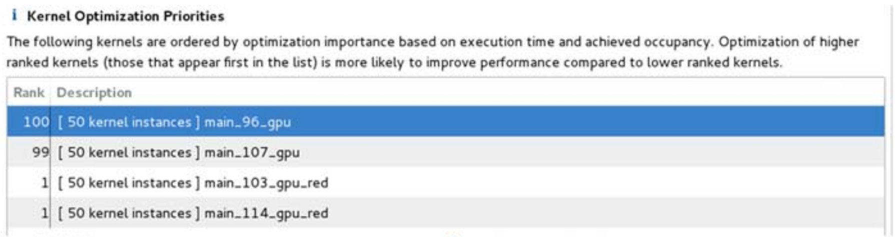
The above results are as expected. In the first case, the computation resources are used more, as there are computations to be done, and also there is need for access to the memory (to the specific points of the matrix A). In the second case, the kernels perform copies. So it is logical that the computation cost is much smaller than the cost of moving the data of the matrices from and to the device memory.

1.5. Double Buffer – GPU – OpenACC

The double Buffer, refers to avoiding copying from Anew to A by reversing roles of A and Anew on every iteration.

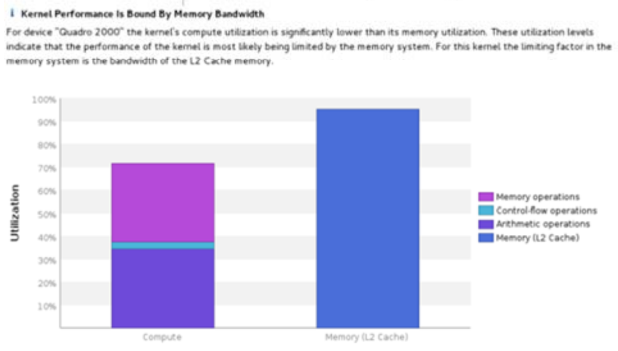
The most relevant performance metrics are the elapsed execution time (2,82 seconds time elapsed), the total number of executed machine instructions (8.13 Giga), and the IPC rate (1,02 insns per cycle).

As expected, the two kernels, the one computing A and the one computing Anew take 48.80% of the execution time each one, they have the same optimization performance and their bottleneck is the same, memory bandwidth.

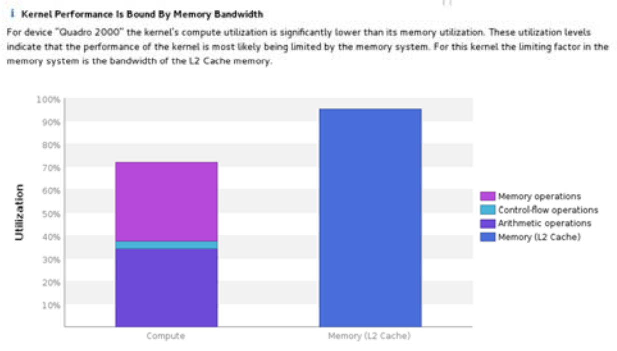

Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[50 kernel instances] main_96_gpu
99	[50 kernel instances] main_107_gpu
1	[50 kernel instances] main_103_gpu_red
1	[50 kernel instances] main_114_gpu_red


Kernel Performance Is Bound By Memory Bandwidth

For device "Quadro 2000" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the L2 Cache memory.


Kernel Performance Is Bound By Memory Bandwidth

For device "Quadro 2000" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the L2 Cache memory.

1.6Results

The execution time is computed for 10,000 iterations of the convergence loop. The execution times in all cases are extrapolated by multiplying by 10 the time for executing 1,000 loop iterations (or multiplying by 100 the time for executing 100 loop iterations in the CPU cases).

Version	Time (seconds)	Performance Bottleneck
Baseline – CPU – PGI compiler	5383	Inefficient memory accesses (IPC=0.09)
Optimized – CPU – PGI compiler	376	Not vectorized (226 G instructions)
Optimized – CPU – GCC compiler	201	Single-core usage (75.9 G instructions)
Multi-thread(3) – CPU – GCC compiler	162	DRAM bandwidth
Baseline – GPU – OpenACC	59	Instruction and memory latency
Loop fusion – GPU – OpenACC (acc kernels-128 threads)	53	Memory bandwidth
Loop fusion – GPU – OpenACC (acc kernels -128 threads)	1779	Instruction and memory latency
Loop fusion – GPU – OpenACC (acc parallel-256 threads)	45	Instruction and memory latency
Loop Interchange – GPU – OpenACC (acc kernels-128 threads)	54	Instruction and memory latency
Loop Interchange – GPU – OpenACC (acc parallel-256 threads)	40	Memory bandwidth
Move sqrtf out of loop – GPU –OpenACC (acc parallel-256 threads)	38	Memory bandwidth
Double Buffer – GPU – OpenACC (acc parallel-256 threads)	28	Memory bandwidth
Double Buffer – GPU – OpenACC (acc kernels-128 threads)	42	Instruction and memory latency

When comparing the performance results of versions Baseline-CPU and Baseline-GPU, it is obvious that the difference is significant. The GPU code is 91 times faster than the CPU on 10000 iterations. Comparing the most optimized versions of GPU with the Baseline-CPU, the speedup is 192.25x.

In conclusion, it can be said that in the beginning, when having only the baseline code without any alterations, the simplest OpenACC design gives the better performance for the GPU used, and without doing an exhaustive analysis of all the possibilities. The program loop is simple, the addressing of the matrix is simple, and then the compiler can do its work very efficiently.
On the other hand, while starting meliorating and optimizing the code, the acc kernels directive do not work as efficiently. The best number of threads is not selected, but instead the programmer has to direct the compiler.

2. Diffusion - GPU

All the codes were compiled with the command ***pgcc -fast -acc -ta=tesla:cc30 -Minfo=accel-DREAL=double code.c -o codename*** *a*nd executed with the standard problem size of 128x128x128, with the use of *perf stat* and *pgprof* in order to measure basic performance metrics.

The baseline version was annotated with OpenACC directives, in order to generate the best-performing massive parallel version for GPU. The most important alterations on the code were to annotate the "for loops" with the following directives:

#pragma acc data copy(F1[0:(nx)*(ny)*(nz)],F2[0:(nx)*(ny)*(nz)])

- This directive was put before the beginning of the loops, in order to tell the compiler the size of F1 and/or F2.

#pragma acc parallel loop num_gangs() num_workers() vector vector_length() present(F1,F2)

- This directive was put before in the beginning of the outmost for loop, so that the highest performance possible will be provided, since it determines how the threads access memory. A correct annotation replaces the loop reordering optimization.

#pragma acc loop worker

- This directive was put before the middle for nested loop.

#pragma acc loop gang

- This directive was put before the last for nested loop.

As a result, there is no need for loop reordering after this annotations.

The next table summarizes the performance results for different versions of the code:

Diffusion Version (128x128x128)	Time	IPC	Instructions(G)
Baseline – CPU – ICC compiler	270 seconds	0,22	200
Optimized – CPU – ICC	13 seconds	1,27	48
Opt. Multi-thread (3) – CPU – ICC	9,5 seconds	0,57	50
Baseline – GPU –num_gangs(4) num_workers(16) vector vector_length(64)	3,37 seconds	1,15	10
Baseline – GPU –num_gangs(4) num_workers(8) vector vector_length(128)	2 seconds	1,09	9,7
Baseline – GPU –num_gangs(4) num_workers(4) vector vector_length(128)	4,43	1.21	15

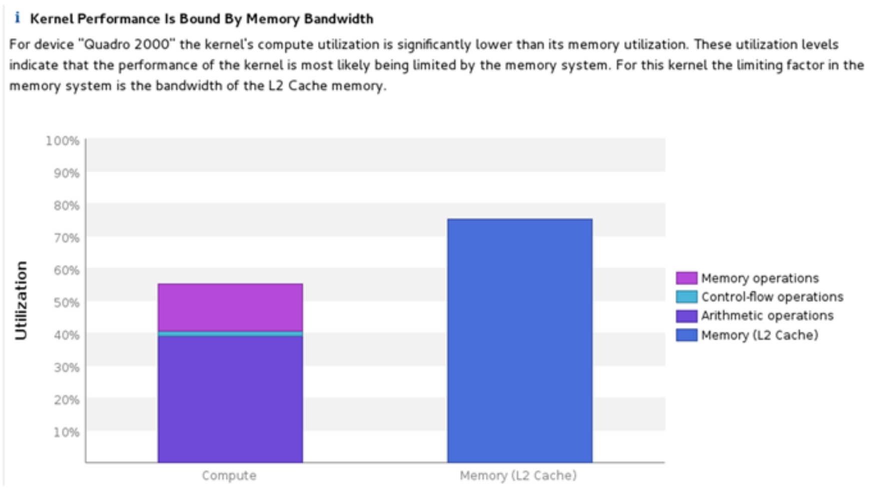
Afterwards, the results of the best performing version of the code (***Baseline – GPU* – num_gangs(4) num_workers(8) vector vector_length(128))** are presented.

The command "*pgprof*" was used to measure the execution time of each kernel:

Time(%)	Time	Calls	Avg	Min	Max	Name
98.28%	1.74096s	2097	830.21us	819.86us	891.57us	diffusion_26_gpu
0.78%	13.806ms	5	2.7613ms	2.7510ms	2.7653ms	[CUDA memcpyHtoD]
0.73%	12.968ms	10	1.2968ms	2.3040us	2.5928ms	[CUDA memcpyDtoH]
0.21%	3.6584ms	1	3.6584ms	3.6584ms	3.6584ms	init_65_gpu

The kernel calculating the matrix (in the diffusion function) takes the 98.28% of the execution time on GPU. The NVIDIA Visual Profiler (**nvvp**) tool was used in order to provide information about the performance of the GPU application.

When examining the diffusion kernel, which of course has a rank of 100 in the optimization importance, the results obtained are the following:

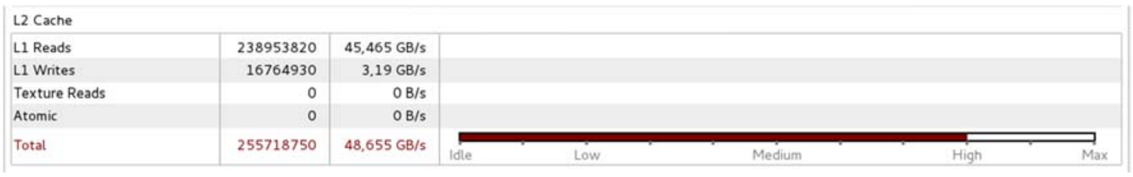

i **Kernel Performance Is Bound By Memory Bandwidth**

For device "Quadro 2000" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the L2 Cache memory.

According to the above figure, compute resources are used 55% of their capacity, while the memory resources achieve an utilization of around 75%. That means that the computation cost of the west, east, north, south, top and down integers is smaller than the cost of moving data. Each thread needs to access each time matrix point, so that the F2 will be computed.

Even though the biggest percentage in the computing resources is used in Arithmetic operations, the cost of moving data between the GPU computation resources and the L2 Cache memory (shared by all the SMs) is larger.

When getting more information about the memory bandwidth for all types of memory of the device, can be seen again that only of L2 Cache limits the performance and that there is an achieved 80% of the peak bandwidth. As a result, the results indicate that the bottleneck of the program is memory bandwidth.



L2 Cache

L1 Reads	238953820	45,465 GB/s	
L1 Writes	16764930	3,19 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	255718750	48,655 GB/s	Idle Low Medium High Max

When comparing the performance results of versions Baseline-CPU and Baseline-GPU, it can be seen that the GPU code is 82.5 times faster than the CPU on a 128x128x128 matrix. Furthermore, when comparing the most optimized CPU version with the baseline GPU one, the GPU is almost 5 times faster.

In consequence, as seen in the 2D Laplace and the Diffusion problems, the significance of accelerators in these type of algorithms is huge, as the GPU provides a lot better results, with less code alterations.

References

1. Notes on Programming Massively-Parallel Architectures (GPU and accelerators, Juan Carlos Moure Lopez, Modelling for Science and Engineering, Parallel Programming, Autonomous University of Barcelona
2. OpenACC API 2.5, Reference guide, 2015 openacc-standard.org
3. http://docs.nvidia.com/cuda/profiler-users-guide/index.html
4. http://on-demand.gputechconf.com/gtc/2014/presentations/S4165-cuda-optimization-nvidia-nsight-ee-case-study.pdf
5. https://en.wikipedia.org/wiki/Instruction_cycle
6. https://devblogs.nvidia.com/cuda-7-5-pinpoint-performance-problems-instruction-level-profiling/