***Introduction***

Ever since the start of the GPGPU era, NVIDIA has focused more and more on implementing architectural changes in their graphics cards for leveraging their use in high-performance computing environments. Probably the biggest step towards the general purpose computer platform was made when the Fermi architecture was launched (2009). This architecture boosted significant performance improvements and was better suited for both single and double precision floating point arithmetic. Iteratively, Kepler (2012), and, then, Maxwell (2014) platforms suffered changes for means of adapting to the starving market of graphics accelerators.

The thesis used as a baseline for all comparisons in this assignment covered and used two different platforms: Fermi and Kepler (Tesla C2065 and GeForce GT 640). A sensitive issue about the two mentioned cards is that they were not intended for the same market segment: while the Tesla card was built with the purpose of scientific, accurate, high-performance computing, the GeForce card is a low-end media graphics card meant for every consumer. Therefore, the architectural specifications differ by a large margin in almost all areas: compute capability (CUDA compute generation), memory organization and bandwidth, multiprocessor structure, floating-point support etc. Probably the biggest difference between the two cards is the price (more than an order of magnitude for Tesla).

The above-mentioned cards were used to evaluate the performance of a neural network simulation. The conclusion should show advantages and disadvantages of the two benchmarked accelerators and set a common ground for further development in the field.

For our part of the work, we will insert into the picture the Maxwell architecture and benchmark it against the results presented in the thesis. Moreover, we will try to detail some of the changes in the new architecture and explain how can these changes impact the performance of the original algorithm. Last, using the profiled application, we will try to adapt the algorithm for improving execution time, by trying to eliminate observed bottlenecks.

***NVIDA architectures: Maxwell vs Kepler:***

The latest graphics accelerator architecture, codenamed Maxwell, is the current state-of-the-art reference for both graphics and GPGPU market. Since its launch it has been marketed as a very power efficient step of the Kepler architecture, optimizing performance and area for better utilization of the compute resources. Some of the changes over the Kepler architecture are:

-The new Streaming Multiprocessor (SMM) has 33% fewer CUDA cores. However, NVIDIA states that, because of “40% higher delivered performance per CUDA core”, the new SMM block will deliver performances within 10% of Kepler.

-Maximum number of threads per block has been increased from 16 to 32.

-Functional improvements in FPU, scheduling and instruction throughput allows for higher practical occupancy of the GPU when massive parallel workloads are dispatched.

**Architectural comparison between accelerators used**

For a proper comparison between the benchmarked cards, we will quickly review their technical specifications (Table 1). From the presented numbers, we can easily draw the conclusion that the GT 640 will be outpaced by the newer Maxwell cards. The GTX860M is a an identical chip as the one used on the virtual machine (Similde), with the difference that it is integrated in a laptop and has slightly slower memory bandwidth. We would expect that the two Maxwell cards to perform on par, with small performance differences coming from the system setup (CPU, RAM memory setup, PCI etc)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **GT 640** | **GT 750 Ti** | **GTX 860M** |
| *Architecture* | GK107 (Kepler) | GM107 (Maxwell) | GM107 (Maxwell) |
| *Clock speed (MHz)* | 902 | 1020 | 1029 |
| *Memory* | 1 GB | 2 GB | 2 GB |
| *Memory bandwidth* | 28.8 GB/s (DDR3) | 86.4 GB/s (GDDR5) | 80 GB/s (GDDR5) |
| *SM units* | 2 | 5 | 5 |
| *CUDA cores* | 384 | 640 | 640 |

Table 1 Technical comparison between the benchmarked platforms

Task 3

For means of comparison, the original CUDA code has been benchmarked on the Virtual Machine (SIMILDE). In the tables below we have results from the Kepler GeForce card (GT 640 – from the thesis, page 54) and the Maxwell GeForce card (GT 750 Ti) – tables 2 and 3. Table 4 shows the performance increase (green)/decrease (red) in percentage of the newer graphics card over the reference one.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Paper benchmark Double Precision GeForce GT 640 (Kepler)** | | | | |
| **Input size (cells)** | **Block size (threads/block) (seconds)** | | | |
| 32 | 64 | 256 | 1024 |
| 64 | 19 | 16 |  |  |
| 256 | 19 | 16 | 17 |  |
| 1024 | 26 | 22 | 21 | 35 |
| 4096 | 88 | 66 | 81 | 82 |
| 9216 | 191 | 127 | 175 | 202 |

Table 2 Benchmarks taken from the thesis for the Kepler platform (double precision

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Benchmark using the SIMILDE GTX 750 Ti (Maxwell)** | | | | |
| **Input size (cells)** | **Block size (threads/block) (seconds)** | | | |
| *32* | *64* | *256* | *1024* |
| 64 | 14.42 | 15.422 |  |  |
| 256 | 21.305 | 21.531 | 22.82 |  |
| 1024 | 47.353 | 48.126 | 48.124 | 37 |
| 4096 | 171.819 | 168.398 | 170.55 | 129.558 |
| 9216 | 311.474 | 305.498 | 296.957 | 225.851 |

Table 3 Benchmarks performed on the lab Maxwell platform.

We can see that, for a small number of input cells, the Maxwell card outperforms the GT640, as expected (Table 4). However, for hundreds or more of input cells, the performance gap increases between the two cards, favoring the older model. These results are confusing because we couldn’t find any argument for the results published in the thesis. However, on closer inspection we blamed it on the current configuration of the system in which the GTX 750 Ti runs.

The server is running on an Intel Xeon CPU, but older model, and the data transfer is done via PCI-Express version 1, a bus that communicates to the GPU with much lower speed than the one that was used with the GT 640 probably. This could explain why, for very small number of cells, the GTX 750 Ti outperforms (memory transfers are very small) and why, when the input size increases, the memory is becoming a major bottleneck. This theory is further supported by the fact that, after 4096 input cells the performance gap starts to decrease (the parallel fraction of the algorithm becomes more predominant and advantages the more powerful GPU).

As mentioned before, Maxwell should show better resource occupancy values than its predecessor, but the presented bottleneck of the current configuration may make correct results visualization harder. We will focus on this issue in the next task, when benchmarking on another platform.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Paper benchmark Double Precision GeForce GT 640 (Kepler)** | | | | |
| **Input size (cells)** | **Block size (threads/block) (seconds)** | | | |
| 32 | 64 | 256 | 1024 |
| 64 | 31.76% | 3.75% |  |  |
| 256 | 10.82% | 25.69% | 25.50% |  |
| 1024 | 45.09% | 54.29% | 56.36% | 5.41% |
| 4096 | 48.78% | 60.81% | 52.51% | 36.71% |
| 9216 | 38.68% | 58.43% | 41.07% | 10.56% |

Table 4 Performance increase of the current platform over the baseline (thesis). Red represents decrease in performance

Task 4

Improving the algorithm has posed several challenges, not only because the current implementation is already achieving similar performance to the theoretical limit, but also because the difference between the two benchmarked platforms are small (one generation difference). However, we have observed that the cache hit (L2 cache) is around 70 %, a figure that doesn’t surprise when talking about a data-flow algorithm like the one studied. Therefore, we have focused on eliminating some minor bottlenecks that we found in the code:

* Because divergent branches cause stalls within an entire warp, if-statements and conditional loops should be avoided as much as possible. We have found a code snippet that could be improved and adapted it accordingly. The modifications that we operated on the code (before – Code Snippet 1 and after – code snippet 2), showed almost 20 % improvement for the execution time of the neighbor\_kernel function.

Code snippet 1 neighbor\_kernel(..) Original code with if-statement included in loop

int jmin = j-1, jplus=j+1, kmin = k-1, kplus=k+1, n=0;

cellStatePtr[dev\_fetch(j,k) + (n++)] = fetch\_double(t\_cellVDendPtr, jmin, kmin);

cellStatePtr[dev\_fetch(j,k) + (n++)] = fetch\_double(t\_cellVDendPtr, jmin, k);

cellStatePtr[dev\_fetch(j,k) + (n++)] = fetch\_double(t\_cellVDendPtr, jmin, kplus);

cellStatePtr[dev\_fetch(j,k) + (n++)] = fetch\_double(t\_cellVDendPtr, j, kmin);

cellStatePtr[dev\_fetch(j,k) + (n++)] = fetch\_double(t\_cellVDendPtr, j, kplus);

cellStatePtr[dev\_fetch(j,k) + (n++)] = fetch\_double(t\_cellVDendPtr, jplus, kmin);

cellStatePtr[dev\_fetch(j,k) + (n++)] = fetch\_double(t\_cellVDendPtr, jplus, k);

cellStatePtr[dev\_fetch(j,k) + (n++)] = fetch\_double(t\_cellVDendPtr, jplus, kplus);

//Get neighbor V\_dend

n = 0;

for(p=j-1;p<=j+1;p++){

for(q=k-1;q<=k+1;q++){

cellStatePtr[dev\_fetch(j,k) + (n++)] = fetch\_double(t\_cellVDendPtr, p, q);

if(p==j && q==k)

n=n-1;

}

}

Code snippet 2 neighbor\_kernel(..) Modified code with unrolling

* Another change applied to the code was directed towards increasing data locality when accessing the used arrays, especially the *cellStatePtr.* The modifications meant changing the structure of the two main kernels and moving a part of the processing from the *compute kernel* to the *neighbor kernel*. The modifications can be studied in the *zip* file with the updated code, starting with line 360.

For assessing the performance improvement for our updated code we have used the virtual machine and our Cuda-enabled laptops, which run on a 3.5 Ghz i7 mobile processor with PCI-Express version 3 bus between the RAM and the graphics card (GTX 860M, 2 GB GDDR5 memory – as presented in table 1).

On the laptops, we have used the Windows environment (Nsight Visual Studio) for benchmarking and profiling the application. This mention is important also because the compiler used is the one packaged with the Visual Studio, and therefore should show differences in compiled optimizations than the GCC (at least for the CPU side).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Updated code GTX 750 Ti*** | | | | |
| **Input size (cells)** | **Block size (threads/block) (seconds)** | | | |
| *32* | *64* | *256* | *1024* |
| 64 | 6.834 | 3.926 |  |  |
| 256 | 12.5 | 12.932 | 12.833 |  |
| 1024 | 36.753 | 35.89 | 35.457 | 37.84 |
| 4096 | 129.188 | 135.03 | 129.31 | 129.685 |
| 9216 | 226.717 | 227.08 | 226.8 | 226.9 |

Table 5 Execution speed when running the updated code

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Original code GTX 860M*** | | | | | ***Updated code GTX 860M*** | | | |
| **Input size (cells)** | **Block size (threads/block) (seconds)** | | | | **Block size (threads/block) (seconds)** | | | |
| *32* | *64* | *256* | *1024* | *32* | *64* | *256* | *1024* |
| 64 | 21.51 | 22.48 |  |  | 11.67 | 11.02 |  |  |
| 256 | 22.9 | 23.27 | 26.06 |  | 14.54 | 14.34 | 14.67 |  |
| 1024 | 32.66 | 33.4 | 34.28 | 22.7 | 21.24 | 21.14 | 21.23 | 23.33 |
| 4096 | 93.25 | 92.89 | 96.84 | 51.1 | 51.41 | 51.84 | 52.16 | 51.67 |
| 9216 | 178 | 177.53 | 175 | 98.6 | 98.79 | 98.1 | 98.4 | 97.9 |

Table 6 Results and comparison of both the original and updated algorithm on GTX 860 M

Our updated code does not improve performance for a very big number of cells (it can be seen that for 9216 input cells, the results are similar to the original implementation), but brings the execution time down for lower numbers, where speedups of 100 % have been achieved in some cases (64 – input size, 8 x 8 threads per block – Table 6).

Moreover, the improvements seem to make the execution time agnostic to the number of threads per block, meaning that a fine tuning is not needed anymore. We assumed this statement from the fact that the execution time barely changes when resizing the blocks.

References:

## [1] [Tuning CUDA Applications for Maxwell](http://docs.nvidia.com/cuda/maxwell-tuning-guide/" \l "abstract) http://docs.nvidia.com/cuda/maxwell-tuning-guide/#axzz3c0dziLNY