High Performance Stencil Code Algorithms for GPGPUs

Article *in* Procedia Computer Science · December 2011

DOI: 10.1016/j.procs.2011.04.221 · Source: DBLP

2 authors:

Andreas Schäfer
Google Switzerland GmbH
27 PUBLICATIONS **149** CITATIONS
SEE PROFILE

Dietmar Fey
Friedrich-Alexander-University of Erlangen-Nürnberg
283 PUBLICATIONS **1,118** CITATIONS
SEE PROFILE

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2011) 1–10
Procedia Computer Science

International Conference on Computational Science, ICCS 2011

High Performance Stencil Code Algorithms for GPGPUs

Andreas Schäfer[1], Dietmar Fey

Chair for Computer Science 3 – Computer Architecture
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany

Abstract

In this paper we investigate how stencil computations can be implemented on state-of-the-art general purpose graphics processing units (GPGPUs). Stencil codes can be found at the core of many numerical solvers and physical simulation codes and are therefore of particular interest to scientific computing research. GPGPUs have gained a lot of attention recently because of their superior floating point performance and memory bandwidth. Nevertheless, especially memory bound stencil codes have proven to be challenging for GPGPUs, yielding lower than to be expected speedups.

We chose the Jacobi method as a standard benchmark to evaluate a set of algorithms on NVIDIA's latest Fermi chipset. One of our fastest algorithms is a parallel wavefront update. It exploits the enlarged on-chip shared memory to perform two time step updates per sweep. To the best of our knowledge, it represents the first successful application of temporal blocking for 3D stencils on GPGPUs and thereby exceeds previous results by a considerable margin. It is also the first paper to study stencil codes on Fermi.

Keywords: stencil codes, GPU, high performance computing, temporal blocking, Jacobi solver, CUDA

1. Introduction

Stencil codes are space and time discrete simulations. The system's state at a certain point of time is given by a regular matrix whose elements are generally referred to as simulation cells. The state of a cell in the next time step can be deduced from its own previous state and the states of the cells in its neighborhood. The shape of this neighborhood is called stencil. Common stencils in the 3D case are the six-point von Neumann neighborhood (only direct neighbors in the cardinal directions) and the 26 point Moore neighborhood (which includes the diagonal members as well). For example, cellular automata [1] are a subset of stencil codes.

Stencil codes can be characterized by their FLOPs per byte ratio. The Jacobi iteration shown in Figure 1 performs two multiplications and six additions per update. When using double precision floating point numbers, for each update at least 8 bytes have to be read and written. This leads to a ratio of 0.5 FLOP/byte. On architectures with less than half a FLOP/byte of memory bandwidth, this code would be compute bound, meaning that its speed is limited by the

Email addresses: andreas.schaefer@informatik.uni-erlangen.de (Andreas Schäfer), dietmar.fey@informatik.uni-erlangen.de (Dietmar Fey)
[1]Corresponding author

arithmetic throughput. Virtually all current architectures, however, have a significantly higher ratio. According to the roofline model [2] memory bandwidth will therefore be the limiting factor. This is unless caches can be exploited to reuse intermediate results. We chose the Jacobi iteration as a benchmark because its low FLOP/byte ratio makes it especially challenging to optimize and furthermore its widespread use allows us to compare our results to previous publications.

```
for (int z = 1; z < DIM_Z - 1; ++z)
    for (int y = 1; y < DIM_Y - 1; ++y)
        for (int x = 1; x < DIM_X - 1; ++x)
          grid_b[z][y][x] = alpha * grid_a[z][y][x] + beta *
              (grid_a[z - 1][y][x] + grid_a[z + 1][y][x] +
               grid_a[z][y - 1][x] + grid_a[z][y + 1][x] +
               grid_a[z][y][x - 1] + grid_a[z][y][x + 1]);
```

Figure 1: Sequential Jacobi iteration. Each update calculates a weighted average of the old cell with its neighbors in the six cardinal directions. This requires two multiplications and six additions.

Our research is embedded into LibGeoDecomp [2] (Library for Geometric Decomposition Codes), a generic library which can automatically parallelize user supplied stencil codes [3]. It is based on C++ class templates. A user specifies his custom stencil code in a C++ class whose instances represent single simulation cells. LibGeoDecomp then uses a topology discovery library [4] to detect which hardware components are available of the given machine. For this paper we have extended LibGeoDecomp to harness GPGPUs via NVIDIA's CUDA toolkit. Figure 2 shows the results of two simulation runs using different models. We have used VisIt [5] for the visualization. Apart from the rather academic Jacobi solver, it is also being successfully used for more sophisticated simulation models, such as simulating dendritic growth [6] in solidifying molten metal alloys [3].

Compared to standard multi-core CPUs, today's GPGPUs offer a significantly higher floating point peak performance since their chips spend more space on the die on ALUs and less on caches. Depending on the model, their memory interface offers a performance of 100-200 GB/s. Yet, these benefits come at a high cost: the memory typically has a latency of a couple of hundred cycles and the shaders (processing cores) have to be programmed in a vectorized fashion. Especially NVIDIA's current range of Fermi Tesla cards, which have helped the Chinese supercomputer Tianhe-1A to assume the #1 spot on the current Top 500 list, appeal to scientific computing: the whole chip has a double precision peak performance of 515 GFLOPS, the 14 vector processors each feature 128 kB of registers, paired with 64 kB of combined shared memory/L1 cache local to each vector processor, as well as 768 kB of shared L2 cache. Figure 3 outlines Fermi's architecture. Given the huge architectural differences to previous GPGPUs, it is not immediately clear which algorithms will be most efficient on the current chipset.

As previous publications have shown, GPGPUs are well suited for stencil codes [7], but in order to achieve a high degree of efficiency, a deeper understanding of the GPGPUs processor is imperative. The next section investigates different design options to optimize the data flow and computations. Using this knowledge we have developed a number of algorithms which try to harness different properties of the chip. Finally we provide benchmark results to validate the performance of the proposed algorithms.

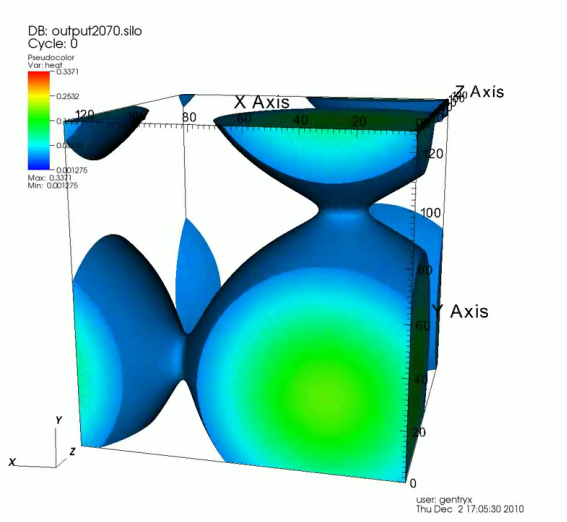
2. Performance Modelling

The performance of a stencil code depends on multiple factors, but initially it's not clear which is the deciding one, e.g. if memory bandwidth is more important than double precision performance. In this section we employ a number of micro benchmarks to figure out how to achieve the GPGPU's maximum memory bandwidth, hide the several hundred cycles of memory latency and saturate the arithmetic pipelines.

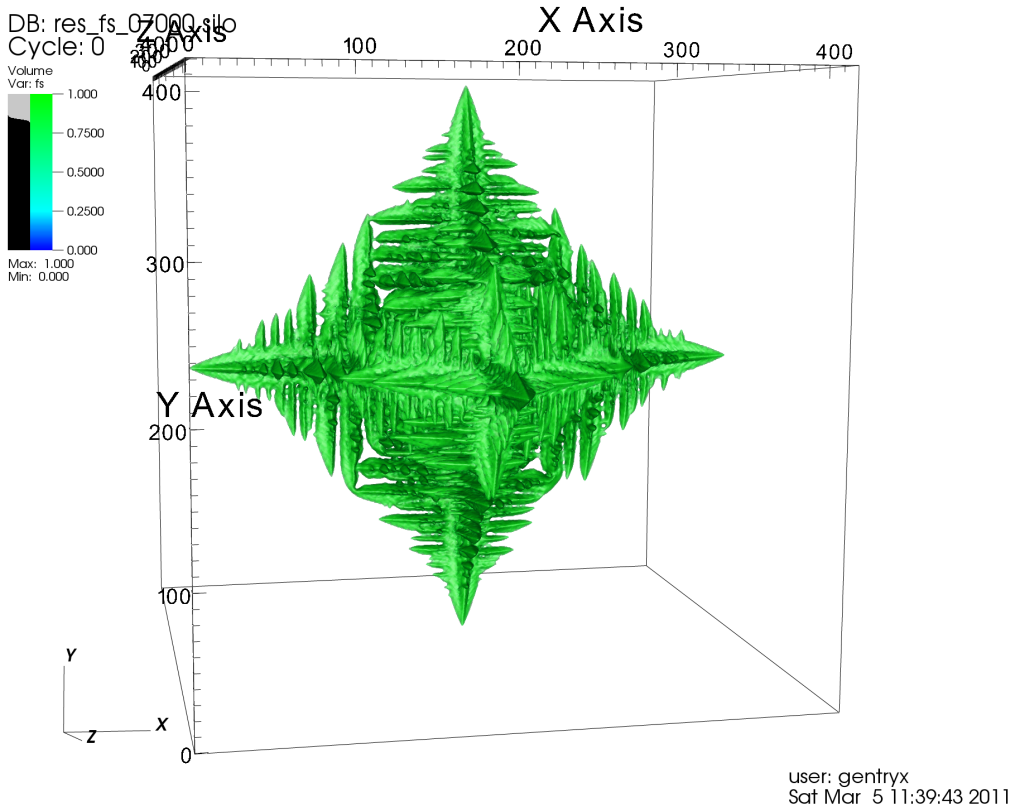
Our testbed consists of an NVIDIA GeForce GTX 480 consumer GPGPU, and a Tesla C2050 which is directly targeted at scientific computing. Table 1 summarizes the cards' basic properties. Interestingly the consumer card

[2] `http://www.libgeodecomp.org`

[3] Dendrite simulation results courtesy of the Department of Metallic Materials, Friedrich-Schiller-Universität Jena, Germany



(a) Heat Dissipation



(b) Dendrite Simulation

Figure 2: Two different use cases for LibGeoDecomp. For the Jacobi heat dissipation a hot cube was inserted into a colder volume. A toroidal topology was used, therefore heat can flow from one boundary surface to the other. The dendrite simulation below shows the solid volume of a crystallization process in molten metal. The dendrite has just started to solidify. From its primary arms secondary arms begin to protrude. Both plots were created with VisIt, for which LibGeoDecomp can write data using the HDF5 based Silo format.

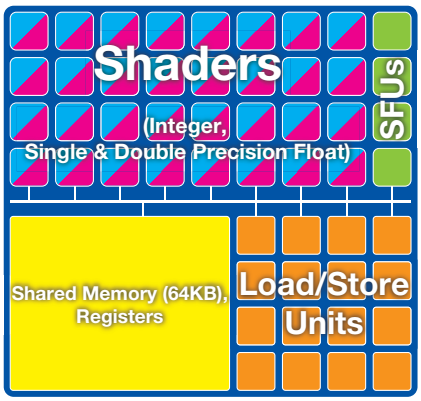


Figure 3: Schematics of a single Fermi multiprocessor. It consists of 32 shaders, 16 load and store units, 4 Special Function Units (SFUs), 32768 registers (32 bits wide), and 64 kB of combined L1 cache and shared memory. It can be configured as either 16 kB cache and 48 kB shared memory, or the other way round.

features a higher number of shaders (480 vs. 448) which are also higher clocked (1.4 GHz vs. 1.15 GHz), and yet the Tesla card has the upper hand in terms of double precision throughput: each of its vector processors, often dubbed multi processors, can retire 16 fused multiply-add instructions (FMADD, $a = b + c \cdot d$) in double precision per clock cycle. The consumer card is artificially limited to 4 FMADDs per cycle. On the other hand, the GeForce has a significantly higher memory bandwidth (177 GB/s vs. 144 GB/s).

The reason why GPGPUs can contain such a high number of processing cores is that each core is much simpler than traditional CPUs. Fermi's shader are based on an in order design. To hide both, memory latency (approx. 400 cycles) and arithmetic latency (read after write latency is roughly 18 cycles), it is generally recommended to use hundreds of threads. Another way to increase throughput is to use instruction level parallelism (ILP) [8]. This means that because the ALUs are pipelined, independent instructions can be issued in direct succession.

With our first benchmark we want for find out how high the register bandwidth of the cards in our testbed is. For this each threads performs a number of double precision FMADDs as outlined in Figure 5 a. To improve ALU saturation, we use an ILP of 4, meaning that four independent FMADDs per thread can be executed. Per FMADD three operands of eight bytes each have to be read and one has to be written. For the GeForce we have measured a bandwidth of 2688.3 GB/s which corresponds to 167.8 GFLOPS for the whole card, meaning that the throughput is limited by the card's double precision performance (168 GFLOPS). The Tesla C2050 peaks at 8103.8 GB/s (corresponding to 506.1 GFLOPS, the theoretical optimum would be 515.2 GFLOPS). Table 1 summarizes all benchmark results. The actual register bandwidth may be even higher, but is not accessible due to the ALUs then becoming a bottleneck.

For the second benchmark we use the same benchmark, but have this time placed the operands in the fast on chip shared memory. A commonly found misunderstanding is that the shared memory is as fast as the register space. However, we have measured a bandwidth of 1343.2 GB/s and 1020.3 GB/s for the Tesla respectively. We believe that this is because all accesses to the shared memory are serviced by the 16 load and store units on each multi processor. Each can deliver 4 bytes per cycle, resulting in a maximum throughput of 1344 and 1020.4 GB/s. If all operands could be read from shared memory, this would mean that a maximum of 21 Giga Lattice Updates (GLUPS) could be achieved on the GeForce, and 15.9 GLUPS on the Tesla.

The memory bandwidth can be measured by a simple 3D array copy, as shown in Figure 5 c. To measure the L2 cache bandwidth, we have reduced the size of the arrays so that the total size of the arrays fits into the 768 kB cache. The stride of the data access has to be carefully tuned to ensure that we are not measuring L1 bandwidth. Measuring L1 bandwidth provided an interesting insight: even with a reduced size of data accessed, L1 cache performance did not exceed L2 bandwidth, if the results were directly written back. But throughput did increase using a benchmark similar to the one in Figure 5 a, with `add1`, . . ., `add4` and `mult1`, . . ., `mult4` placed in memory. Defining the pointers as volatile and inspecting the resulting assembler code with `decuda` ensured that the values are actually fetched from memory. With this read only benchmark we have seen the L1 to match the shared memory bandwidth.

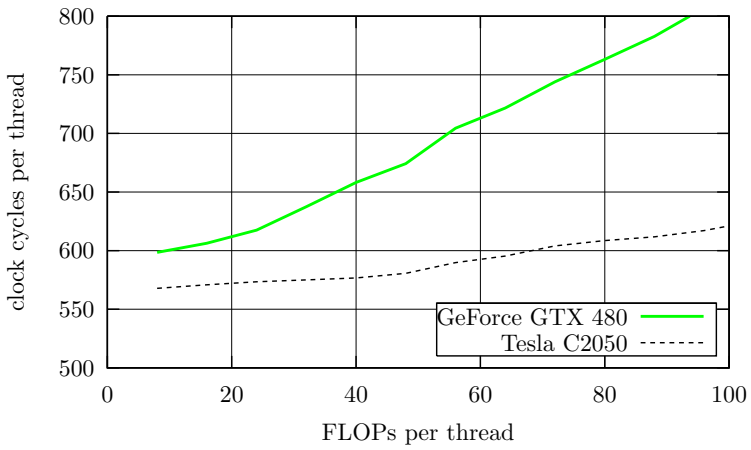


Figure 4: Overlapping floating point operations and memory loads. The GeForce can hide a lot less calculations in its memory latency. Its curve starts to rise linearly from about 25 FLOPs, the Tesla's curve remains mainly flat until about 50 FLOPS, but even then rises much slower thanks to the card's superior double precision performance.

Loads from memory do not immediately block a thread, as long as the loaded data is not required in an ALU. Our next micro benchmark explores how many FLOPs can be overlapped with memory accesses. Listing 5 b outlines the algorithm: it is a combination of an array copy and the previous benchmark to measure the register bandwidth. We did then measure how many net cycles each thread took for execution. As long as `LENGTH` is small enough, the time did increase only marginally. But as the loop took longer and longer, the time for calculation surpassed the memory latency. Figure 4 shows that on the GTX 480 we can perform only a few FMADDs without loosing memory bandwidth. On the Tesla C2050 we could perform 50 FMADDs, and even then, memory performance did not decrease as badly as the GeForce card.

From these measurements we conclude the following: as expected, one should place as many operands as possible in registers. Shared memory should be used to synchronize multiple threads, but, rather unexpected, it could become a bottleneck if used exclusively for all operands. Unlike standard CPUs, GPU algorithms should not block read/write data in the L1 cache, since it is optimized for read accesses. Similarly, blocking for L2 cache not very promising because its bandwidth does not far exceed the memory bandwidth.

3. Algorithm Design

Using the insights gathered in the previous section, we will now review different design options for stencil algorithms. The general approach is to break down the total volume into smaller chunks (Fig. 6). Each chunk is assigned to a CUDA thread block. How the cells of a chunk are distributed among the threads of a thread block depends on the algorithm. As a baseline we have chosen a naïve implementation which loads all seven values for a single cell update directly from memory. On previous generation GPGPUs one would use the texturing units to cache redundant accesses. Thanks to the L1 and L2 caches, this is not required on Fermi GPGPUs, so that mostly only one accumulated memory read and write per update hit the memory. This decomposition is very fine granular, leading to high number of threads which is generally the by NVIDIA suggested approach [9]. Alternatively to blocking the data in L1, one could manually cache data in shared memory, but as our micro benchmarks have shown, this does not promise much of a speedup.

The naïve algorithm has sub par ratio of setup costs to calculation: for eight FLOPs a lot of index calculations have to be carried out. An improvement over this is to update not just one cell per thread, but multiple threads. The *cached plane sweep* algorithm, similar to those proposed in [10, 7], puts all threads of a block on a 2D plane and lets this plane sweep through the chunk (Fig. 6 b). Unlike previous publications, we have optimized the memory layout of our algorithms not for shared memory, but L1 cache reuse. The amortized memory costs of this algorithm are 16 bytes per update: one read and one write. The accesses to neighbor values in the same plane can be mostly served from L1 cache. Cells in the direction of the moving plane are directly available in registers.

```
#pragma unroll
for (int i = 0; i < LENGTH; ++i) {
    temp1 = add1 + temp1 * mult1;
    temp2 = add2 + temp2 * mult2;
    temp3 = add3 + temp3 * mult3;
    temp4 = add4 + temp4 * mult4;
}
```

(a) Register Bandwidth

```
double x = array[index];
#pragma unroll
for (i = 0; i < LENGTH; ++i) {
    temp1 = add1 + temp1 * mult1;
    temp2 = add2 + temp2 * mult2;
    temp3 = add3 + temp3 * mult3;
    temp4 = add4 + temp4 * mult4;
}
array[index] = x + temp1 + temp2 + temp3 + temp4;
```

(b) Overlapping Memory Access and Caculation

```
int x = BLOCKSIZE_X * blockIdx.x + threadIdx.x;
int blockIdxRealY = blockIdx.y & (DIM_Y / BLOCKSIZE_Y - 1);
int blockIdxRealZ = blockIdx.y / (DIM_Y / BLOCKSIZE_Y);
int y = BLOCKSIZE_Y * blockIdxRealY + threadIdx.y;
int z = BLOCKSIZE_Z * blockIdxRealZ + threadIdx.z;
int index = x + y * DIM_X + z * DIM_FRAME;

newGrid[index] = oldGrid[index];
```

(c) Memory Bandwidth Test

Figure 5: Micro Benchmarks used to assess Fermi's key properties.

Model	GeForce GTX 480	Tesla C2050
Multi Processors	15	14
Shaders	480	448
Shader Clock	1.4 GHz	1.15 GHz
DP FLOPS	168	515.2
RAM capacity	1.5 GB	3 GB
Memory Bandwidth	177	144
FLOPs/byte	0.95	3.57
DP GFLOPS (FMADD)	167.9	506.1
Register Bandwidth	2689.4	8103.8
Shared Memory Bandwidth	1343.2	1020.3
L1 Cache Bandwidth	1129.6	1000.3
Memory Bandwidth	142.9	113.0

Table 1: GPGPU properties. The upper half contains the details as given by NVIDIA, the lower half contains our benchmark results. All bandwidths are given in GB/s. For better performance, measurements on the Tesla were done with ECC switched off. When switched on, memory bandwidth is reduced by about one eighth.

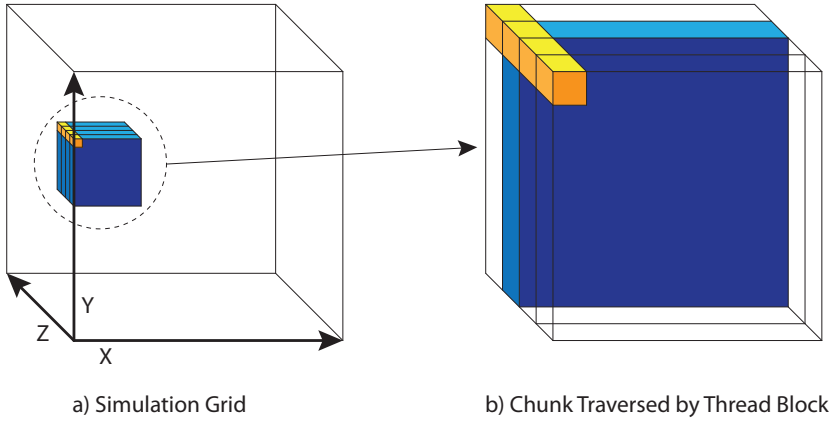


Figure 6: Decomposition of the simulation space into smaller chunks. Each chunk is assigned to a thread block. The blue plane in Subfigure b indicates the moving plane as it sweeps through the chunk in z direction. The yellow blocks are all cells handled by a single thread.

Still, the plane sweep algorithm is limited by the GPU's memory bandwidth. One drawback of the recent Tesla C2050 is that its memory bandwidth is just on par with the 2008 GeForce GTX 280. In order to get beyond the memory bandwidth barrier, CPU implementations perform temporal blocking [11]. Since Fermi GPUs do not have a high bandwidth cache that could serve read *and* write accesses, we have to resort to the shared memory. Still, this memory can only be used to synchronize threads on one multi processor. Even when configuring the chip to use 48 kB shared memory and 16 kB L1 cache, this means that just about 6000 cells can be stored in shared memory (a small portion of shared memory is reserved by CUDA for thread parameters). When choosing a block size of 32×32, just five planes can be placed into shared memory. This could turn out to be a show stopper since for a single update three planes are required, meaning that we would be short of one plane to perform two updates.

For our implementation of the *pipelined wavefront* we exploit the von Neumann neighborhood's property that neighboring cells outside of the plane's direction of travel have to be read only from one layer. Thereby we can reduce the number of required planes to two. The following pseudo code illustrates how our algorithm works. i refers to the offset of the plane from the chunk origin in the z direction.

1. Fill the pipeline so that the following loop invariant is fulfilled.
2. Set $i = 0$
3. While $i <$ `LENGTH` do
   (a) Loop invariant:
      i. Planes i, $i + 1$, and $i + 2$ at time step t are present in registers. Plane $i + 1$ should be additionally available in shared memory.
      ii. The planes $i - 1$ and i at time step $t + 1$ should be in registers, too. From these, plane i has to be in shared memory.
      iii. Plane $i + 3$ at timestep t should be in flight, meaning that a load from main memory has already been issued.
   (b) Issue a load of plane $i + 4$
   (c) Using planes i, $i + 1$, and $i + 2$ at $t + 0$ to update plane $i + 1$ to step $t + 1$.
   (d) Now plane $i + 0$ can be updated from $t + 1$ to $t + 2$, the result is written back to memory.
   (e) Store plane $i + 2$ at $t + 0$ to shared memory
   (f) Store plane $i + 1$ at $t + 1$ to shared memory
   (g) Increment i
   (h) Synchronize all threads of the thread block
4. Perform the final updates to empty the pipeline.

The last step 3h, which synchronizes all operations, is required to make sure that the content of the shared memory is well defined. Using two shared memory buffers, we would theoretically need an additional synchronization step before step 3e in the loop, since we overwrite the buffers in the two following steps. By using four buffers, we can eliminate this step. Loading plane $i + 4$ in step 3b is a practice that has been derived from the micro benchmarks from Section 2. Even though it is only required in the next loop iteration, this allows us to overlap calculation and memory traffic, even when using fewer threads. The drawback of the algorithm is that even though we load planes of 32×32 elements, we can write back only planes of 28×28 elements because we cannot update the border elements due to missing neighbor elements. Also, the initial effort to fill the pipeline reduces efficiency.

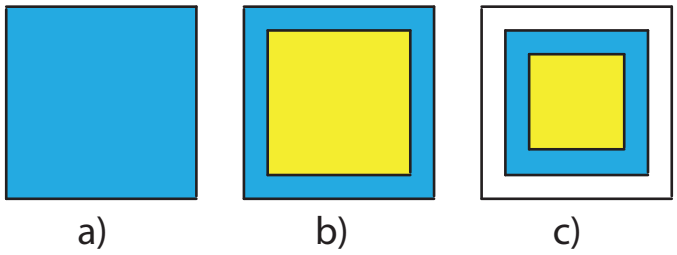


Figure 7: Memory loads (a) and the following two update steps (b and c) of the *pipelined wavefront* algorithm. A single plane gets loaded to shared memory and two update steps are performed on it, using other planes which are kept in registers (not shown). Border cells cannot be updated because some of their neighboring cells are missing.

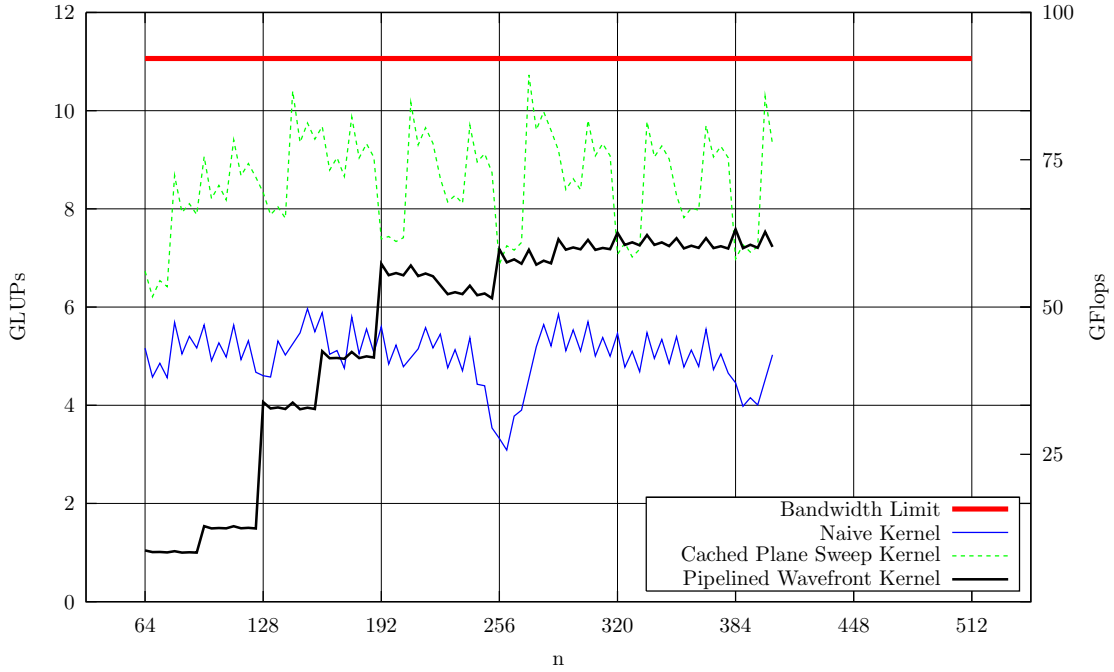
4. Evaluation

Given that NVIDIA does not share much details on the exact architecture of their chips (e.g. how thread scheduling works, how long the pipelines are, which operations may overlap and so forth), the theoretical consideration from the previous section have to be validated by benchmarking. Figure 8 contains a plot of the baseline, cached plane sweep, and pipelined wavefront algorithm for different cubic matrices ranging from 64^3 to 512^3 elements. For our tests we assume that the whole grids fit into the DRAM of the GPGPU. If they do not, parts of the grid have to be streamed from host RAM to the GPU, ideally overlapping with the calculation. This, however would be the same for every algorithm and since our goal was to differentiate between the algorithms, we did not include this in our measurements.

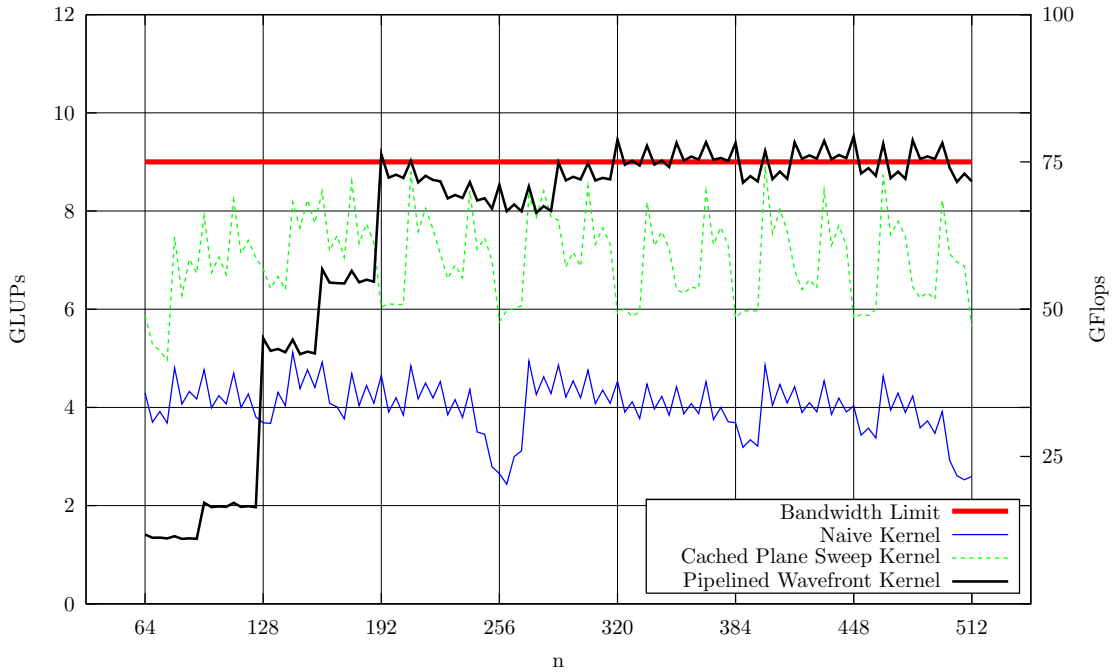
All algorithms were tested with a block size of $32 \times 16 \times 1$ threads which we found to be optimal during our experimentation. A width of 32 threads is required since Fermi's L1 cache lines and memory controller are optimized to load 128 bytes in a row. The *naïve* as well as the *cached plane sweep* updated blocks of $32 \times 16 \times 8$ cells while the *pipelined wavefront* turned out to run faster when updating $32 \times 32 \times 64$ cells. Each thread would then update a block of $1 \times 2 \times 64$ cells. Because of this coarser granularity, this algorithm does not work well on small matrices. For each update it would on average have to load just three cells from shared memory, 0.5 from memory (since we are performing two updates per sweep) and write 0.5 cells back. The remaining accesses can be served from registers since they come from neighboring cells of the same thread.

Interestingly, the wobbling of the *cached plane sweep* algorithm's curve is not caused by noise during the measurement, but are repeatable. We found the performance to peak when the width of the matrix was a multiple of 128 bytes, but for powers of two, most notably 256 and 512 elements, there were unexpected drops in throughput. The best performance was achieved when the width of the matrix, divided by 128 bytes was prime number (e.g. 304 elements, which corresponds to 2432 bytes). When the matrix width is a power of two, all loads from memory end up in the same L1 cache set. We do therefore attribute the performance drops to cache thrashing.

The peak performance on the Tesla C2050 were 8.68 GLUPS (69.4 GFLOPS) for the *cached plane sweep* and 9.52 GLUPS (76.1 GFLOPS) for the *pipelined wavefront* algorithm. For sufficiently large matrices, the *pipelined wavefront* is the fastest algorithm on the Tesla card. This is more than twice as fast as previous results on GPUs [7, 12], but then again those results are not quite comparable, since they were obtained on older hardware. On the GeForce



(a) GeForce GTX 480



(b) Tesla C2050

Figure 8: Jacobi performance on a 3D grid of size n^3 cells, measured in Giga Lattice Updates Per Second (GLUPS). Each update corresponds to 8 FLOPs. Both, the *cached plane sweep* and the *pipelined wavefront* kernel clearly outperform the naïve algorithm. The performance of the pipelined wavefront algorithm is less dependent from the matrix size. On the GeForce we could only benchmark matrices of up to 400^3 elements since its DRAM of 1.5 GB cannot hold much larger matrices. For all algorithms except the pipelined wavefront we have configured the on-chip memory to yield 48 kB shared memory and 16 kB of L1 cache.

consumer card the wavefront is slower than the plane sweep algorithm. We assume that this is because the algorithm performs some redundant calculations and this GPGPU is artificially limited in its double precision performance.

5. Conclusion

We have presented a number of different approaches to conduct stencil computations on GPGPUs, based on a range of micro benchmarks. The micro benchmarks suggest that if threads on the same multi processor need to synchronize data that is not just being read, but written, too, they should use the shared memory. For data that is only being read, the L1 cache is sufficient. Due to its limited throughput, L2 cache is not a promising candidate for cache blocking.

For the Jacobi iteration (and possibly other stencil codes with few FLOPs per update and whose cells are of moderate size (8 bytes, possibly 16)), the *pipelined wavefront* algorithm is the fastest. Since it performs two updates per sweep, it can – in certain conditions – even surpass the bandwidth limit. We believe that further profiling could improve its performance, since in its current state the throughput is lower that to be expected (ideal would be a twofold performance increase compared to the *cached plane sweep* algorithm). However, it is not applicable for larger models, because then the planes to be loaded would not fit into shared memory. For the caching kernels the matrix should be padded, so that its width is a multiple of 128 bytes, but not a power of two.

Acknowledgments

We would like to thank Klemens Reuther and Markus Pitzing of the Department of Metallic Materials, Friedrich-Schiller-Universität Jena `http://www.matwi.uni-jena.de/metalle/englisch/index.htm`, for providing the dendrite simulation model and data as well as the fruitful discussions we had while working on its parallelization.

[1] J. Von Neumann, Theory of Self-Reproducing Automata, University of Illinois Press, Champaign, IL, USA, 1966.
URL `http://portal.acm.org/citation.cfm?id=1102024`

[2] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (2009) 65–76. doi:http://doi.acm.org/10.1145/1498765.1498785.
URL `http://doi.acm.org/10.1145/1498765.1498785`

[3] A. Schäfer, D. Fey, Libgeodecomp: A grid-enabled library for geometric decomposition codes, in: Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, Berlin, Heidelberg, 2008, pp. 285–294.

[4] A. Schäfer, D. Fey, Pollarder: An Architecture Concept for Self-adapting Parallel Applications in Computational Science, in: Computational Science - ICCS, Vol. 5101 of LNCS, Springer, 2008, pp. 174–183.

[5] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, B. J. Whitlock, N. Max, A contract-based system for large data visualization, in: Proceedings of IEEE Visualization 2005, 2005, pp. 190–198.

[6] D. F. W. Kurz, Fundamentals of Solidification, Enfield Publishing And Distribution Company, 1998.

[7] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 4:1–4:12.
URL `http://portal.acm.org/citation.cfm?id=1413370.1413375`

[8] V. Volkov, Better performance at lower occupancy, in: NVIDIA GPU Technology Conference, 2010.

[9] NVIDIA, Cuda c programming guide, 2010.
URL `http://developer.nvidia.com/object/cuda\_download.html`

[10] P. Micikevicius, 3d finite difference computation on gpus using cuda, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, ACM, New York, NY, USA, 2009, pp. 79–84. doi:http://doi.acm.org/10.1145/1513895.1513905.
URL `http://doi.acm.org/10.1145/1513895.1513905`

[11] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, H. Fehske, Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization, Computer Software and Applications Conference, Annual International 1 (2009) 579–586. doi:http://doi.ieeecomputersociety.org/10.1109/COMPSAC.2009.82.

[12] A. Nguyen, N. Satish, J. Chhugani, C. Kim, P. Dubey, 3.5-d blocking optimization for stencil computations on modern cpus and gpus, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–13. doi:http://dx.doi.org/10.1109/SC.2010.2.
URL `http://dx.doi.org/10.1109/SC.2010.2`