CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator

Jakob Siegel, Juergen Ributzka and Xiaoming Li
Department of Electrical and Computer Engineering
University of Delaware, Newark, DE, USA
Email: jakob@udel.edu, juergen@udel.edu, xli@ece.udel.edu

Received: 22/02/2010; Accepted: 07/04/2010

ABSTRACT

Modern GPUs open a completely new field to optimize embarrassingly parallel algorithms. Implementing an algorithm on a GPU confronts the programmer with a new set of challenges for program optimization. Especially tuning the program for the GPU memory hierarchy whose organization and performance implications are radically different from those of general purpose CPUs; and optimizing programs at the instruction-level for the GPU.

In this paper we analyze different approaches for optimizing the memory usage and access patterns for GPUs and propose a class of memory layout optimizations that can take full advantage of the unique memory hierarchy of NVIDIA CUDA. Furthermore, we analyze some classical optimization techniques and how they effect the performance on a GPU.

We used the Gravit gravity simulator to demonstrate these optimizations. The final optimized GPU version achieves a 87× speedup compared to the original CPU version. Almost 30% of this speedup are direct results of the optimizations discussed in this paper.

Keywords: GPGPU; CUDA; N-body; memory layout; optimization

1 INTRODUCTION

Scientific computations have always been in need of more computational power than available. However, obtaining high computational power is traditionally very expensive and, if available, the full usage of it requires extensive programming efforts. The limited availability of computational power to the HPC community has changed recently. Off-the-shelf hardware such as IBM Cell/B.E. processors in video game consoles and Graphic Processing Units (GPUs) in consumer PCs provides equivalent or even higher computational

power than what was offered by traditional CPU-centered high-performance computing solutions. With theoretical peak performances of up to 1 TFLOPS[1] for the NVIDIA G200 series, such GPUs are becoming a common tool to accelerate computational expensive algorithms used in scientific computing.

Until recently, it was a challenging task to implement an algorithm efficiently to run on a GPU, because the functionality of such a device was plainly geared toward graphics acceleration and did not offer an interface to perform non-graphics related operations. Hence, scientific applications had to be implemented using functions and APIs such as OpenGL [1] that are intended only for graphic tasks. In the past, the necessity of doing computation in OpenGL and the lack of downward compatibility of newer generations of graphic hardware greatly restrained the usage of GPUs for scientific computing, even that the computational power of many GPUs would have offered a cheap way to gain additional FLOPS.

The introduction of NVIDIA CUDA and the accompanying CUDA driver and C language extension [2] made this computational power of GPUs easier to utilize. In particular, it took GPU programming to a higher level that normal programmers feel more familiar with. The CUDA driver and C language extension simplify the usage of the GPU as a co-processing device.

However, even though the problem of writing a program that can *work* on a GPU seems to have been solved, the question of how to tune a program to make it *work well* on a GPU is only rudimentarily understood and insufficiently investigated. Most notably, the program optimization for GPU faces two major challenges: the radically different organization of GPU memory hierarchy and the re-evaluation of traditional instruction level optimizations in the new context of GPUs.

It is the programmer's responsibility to maximize the throughput by optimizing the memory layout. Second, the GPU runs programs in a SIMT[2] manner where each instruction stream is a thread, so that the optimization for the GPU memory hierarchy must be tuned for the collection of memory access streams from multiple threads. Program optimizations for the CPU memory hierarchy usually assume that a single process occupies all memory levels. The consideration of multiple-thread memory accesses makes the optimization for the GPU memory hierarchy more complicated than that for the CPU.

---

[1] single precision floating point operations per second

[2] Single Instruction Multiple Thread

Instruction level optimizations for the CPU usually assumes a program can occupy all CPU resources such as registers. However, the main focus of instruction level optimization for CUDA programs is to conserve hardware resources to allow for a higher occupancy of the available hardware for all threads. Therefore, traditional instruction level optimizations such as loop unrolling must be re-evaluated in the context of GPUs. Loop optimizations are of special interest, since most of the algorithms that qualify to be implemented in CUDA are loop based.

In this paper we propose a technique to optimize *global memory* accesses especially for larger structures that exceed the alignment boundaries of the CUDA architecture. Furthermore, we quantitatively analyze the performance increase on a GPU by fully unrolling the innermost loop of a program and propose a way to predict the impact on the performance. Then we applied these techniques to a CUDA version of the Gravit gravity simulator.

The remaining part of this paper is organized as following: Section 2 overviews the NVIDIA GPU and CUDA programing framework. Section 3 introduces the Gravit application and the algorithms used. Section 4 discusses the proposed memory layout optimizations for CUDA and Section 5 shows the related experimental results. The loop optimizations are described in Section 6 and Section 7 shows the experimental results for the Gravit application. At last, Section 8 and Section 9 discuss related work and conclude this paper.

2 CUDA OVERVIEW

CUDA (Compute Unified Device Architecture) is NVTDIA's programming model that uses GPUs for general purpose computing (*GPGPU*). It allows the programmer to write programs in C with a few extensions that are designed to allow the programming of the CUDA architecture. These extensions enable programmers to directly access the different levels of the memory hierarchy that are quite different from the common CPU memory/cache model.

The CUDA memory hierarchy is composed of a large on-board *global memory,* which is used as main storage for the computational data and for synchronization with the *host memory.* Unfortunately, the *global memory* is very slow and not automatically cached[3]. To fully utilize the available memory bandwidth between the *global memory* and the GPU cores, data has to be accessed consecutively. To alleviate the high latencies of the *global memory,* a

[3] caches are not existent except for a small texture- and constant cache

on-chip *shared memory* can be used. *Shared memory* is explicitly managed by the kernel and special care has to be taken when it is accessed. When the same *shared memory* banks are accessed by multiple threads at the same time, a memory access conflict will occur and the reads to the same memory bank will be serialized. There are two other types of memory available, texture- and constant memory, which will not be discussed here.

In addition to the CUDA memory hierarchy, the performance of CUDA programs is also affected by the CUDA tool chain. The CUDA tool chain consists of special GPU drivers, a compiler which is based on the Open64 compiler [3], a debugger, a simulator, a profiler and libraries which can be used with most GPUs of the GeForce series. This paper mainly studies optimizations that are specific to the CUDA memory hierarchy including the CUDA register file. A more detailed description about CUDA and its supporting hardware can be found in [2].

In this paper we used two different NVIDIA GPUs which are based on different compute models. The compute models mostly differ in what hardware resources are available (see Table 1). Besides the differences in hardware resources, *the global memory* in model 1.3 is organized differently. The memory access for 1.3 is organized in segments of 32, 64 or 128 bytes. Therefore it is only important to guarantee that accessed data is placed in the same segment and does not have to be accessed sequentially to guarantee the lowest possible latency.

Table 1: Overview of the differences between the used compute models.

NVIDIA GPU	**G80**	**G280**
Compute capability	1.0	1.3
number of multiprocessors	16	30
max number of threads per block	512	512
warp size (threads per warp)	32	32
register per multiprocessor	8192	16384
shared memory per multiprocessor	16KB	16KB
max number of active blocks per multiprocessor	8	8
max number of active warps per multiprocessor	24	32
max number of active threads per multiprocessor	768	1024

3 GRAVIT OVERVIEW

Gravit [4] is a multi-platform gravity simulator written by Gerald Kaszuba. Gravit is released under the GNU General Public License (GPL). It implements simple Newtonian physics using the Barnes-Hut N-body algorithm [5]. Although the main goal of Gravit is to be as accurate as possible, it also creates beautiful looking gravity patterns. We select the gravity simulation as our target problem, because the gravity simulation is an important type of physics simulation and optimization techniques developed for Gravit will be relevant for many other N-body algorithm based computation problems.

3.1 Algorithms used in Gravit

The Gravit simulates the gravity force and the movement of N particles in a closed system. The most computationally intensive part of the Gravit program is the calculation of the external forces on single particles in the system. In general the absolute force F on a particle is the sum of the external force F_e, nearest neighbor force F_{nn} and the far field force F_{ff},

$$F = F_e + F_{nn} + F_{ff} \tag{1}$$

In the Gravit implementation there are two different ways to calculate those forces. One is to use the Barnes-Hut Tree Code algorithm that is widely used in astrophysics and has been thoroughly parallelized for standard multi-core systems. It addresses the expensive far-field force calculations in the following clever "divide-and-conquer" way.

1. Build an octree
2. For each sub-square in the octree, compute the center of mass and total mass for all the particles it contains,
3. For each particle, traverse the tree to compute the force on it.

This approach has a complexity of $O(N \log N)$ which for a general purpose computer is better than the second approach which is a pretty simple but way more computational intense $O(N^2)$ algorithm.

The second approach computes the force on a single particle p_i simply by calculating and adding up the forces exceeded on p_i by all other particles p_0 to p_{i-1} and p_{i+1} to P_{N-1} in the system. Even that this second approach looks like a waste of resources and computational power, it is a perfect algorithm to be implemented on a GPU.

3.2 The N-body problem and the GPU

The Barnes-Hut Tree Code algorithm helps to improve performance on a general purpose machine. Because of its heavily recursive nature it is not an algorithm that allows for an (easy) implementation on the CUDA architecture, because program parts that are written for CUDA (called *kernel* functions) are limited in many ways compared with general programming for the CPU. This includes for example, no recursion, limited synchronization, no interblock communication, no dynamic memory allocation during runtime. To implement an algorithm like the Barnes-Hut Tree Code algorithm on the GPU, the recursion has to be transformed into an iterative equivalent.

The second more direct approach is also the more computational intense method. It represents a straight forward way to calculate the force on each particle for $F[i]$ out of N particles, where $F(j, i)$ represents the force on particle i due to particle j (see Algorithm 1).

Algorithm 1 Pseudo code for the far field force calculation. F[i] is the total force on particle i and F(j,i) the force j acts on i.

```
1: for i = 1 to N do
2:     F[i] = 0
3:     for j = 1 to N do
4:         if i ≠ j then
5:             F[i] = F[i] + F(j, i)
6:         end if
7:     end for
8: end for
```

This seemly more computational intense $O(N^2)$ algorithm can be implemented more efficiently on CUDA. For this algorithm we can have one thread for each single particle to calculate and add up all the forces on itself. We use the $O(N^2)$ implementation in this paper.

4 MEMORY LAYOUT OPTIMIZATIONS

For most known computer architectures the memory wall represents a major performance issue. A GPU is no exception to this rule. A kernel running on a GPU usually works on a massive amount of data that resides in *global memory* and has to be loaded into the local *shared memory* of the GPU processor core. A good memory layout is the key to improve the performance of a CUDA kernel. Besides reducing the overall access to *global memory,* it is of equal

```
1 typedef struct particle_s {
    float px, py, pz;
3   float vx, vy, vz;
    float mass;
5 } particle_t;

7 particle_t S[N];
```

Figure 1: Original layout of the input data structure. This layout leads to seven single non-coalesced reads from *global memory.*

importance to guarantee for coalesced reads. This means that all threads of one warp, which access the *global memory* in parallel should access consecutive data elements. In the following section we will discuss the transformations we performed on the original memory layout and how those affected the performance.

4.1 Original Memory Layout: Array of Structures (AoS)

In the Gravit application, we have to store the position and the velocity in 3D space as p_x,p_y,p_z and v_x,v_y,v_z, as well as a scalar value for the mass m for every single particle. In the original layout those values are stored in a structure with seven elements as shown in Figure 1.

For this layout every thread of the warp will read one value of the structure at a time. This results in seven non-coalesced reads from *global memory* for every time the elements of such a structure have to be accessed by a thread, as shown in Figure 2. This is one of the worst case scenarios for CUDA. We used the results of this layout as the baseline for the following tests.

4.2 Structure of Arrays (SoA)

Since coalesced reads seem to be of great importance to improve memory access time, the straightforward way to achieve this is to change the layout from an array of structures to a structure of arrays as presented in Figure 3. This means that every value that in the previous version was stored as an element of

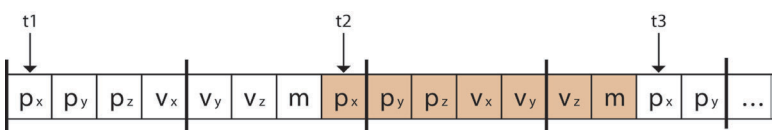

Figure 2: Each thread has to issue seven reads where the reads for the threads of one warp-half are not coalesced.

```
typedef struct particle_s {
  float px[N], py[N], pz[N];
  float vx[N}, vy[N], vz[N];
  float mass[N];
} particle_t;

particle_t S;
```

Figure 3: The Structure of Arrays layo guarantees for coalesced reads when threads are reading from the single arrays. One sad still needs to issue seven reads to retrieve all relevant data.

the structure now is saved in an array of scalar values. We now have an array for each dimension of all the positions as well as for the velocities and the masses. Overall there are seven arrays of scalar values. The structure-of-array layout guarantees that all reads from *global memory* are coalesced, since all threads of the omxie warp-half will access consecutive single floating point values in *global memory,* as shown in Figure 4.

4.3 Array of Aligned Structures (AoAS)

One way of improving access to structures residing in *global memory* is using the alignment attribute offered by CUDA (Figure 5) or using already aligned build in types like *float4,* which is defined as an aligned structure with four float values x,y,z and w. CUDA can handle 64 bit and 128 bit aligned data and read or write such data with one single read or write instruction. By aligning the original structure to 128 bit, an eighth hidden 32 bit value is added to the

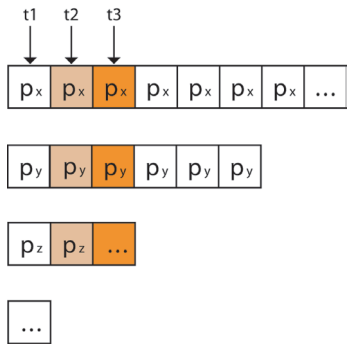

Figure 4: Each thread has to issue seven reads where the reads for each type of data element are coalesced.

```
1 typedef struct particle_s __align__(16) {
    float px,py,pz;
3   float vx,vy,vz;
    float mass;
5   // plus hidden 32bit padding element
  } particle_t;
7
  particle_t S[N];
```

Figure 5: Aligned structure with one padding element.

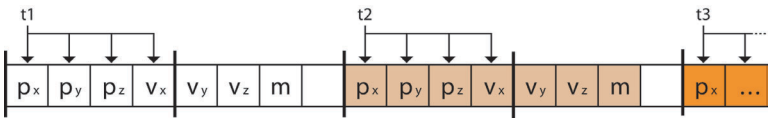

Figure 6: For all seven elements each thread now only has to issue two 128bit reads, but the reads are not coalesced.

structure (see Figure 5). Hence the overall number of reads can be reduced to two 128 bit reads, as shown in Figure 6. This is the commonly proposed solution to efficiently handle larger structures in CUDA. Even though the addition of a padding value results in a slight increase of memory usage, the number of issued *global memory* accesses are drastically reduced.

4.4 Structure of Arrays of Aligned Structures (SoAoAS)

Overall it can be said that there are two major optimizations that can either be applied to larger structures residing in *global memory,* where each will increase the overall performance.

- Accessing consecutive elements to guarantee for coalesced reads.
- Alignment of data structures to allow for fewer reads.

As shown in the previous section each of those optimizations can be applied to the data set discussed in this paper. Both ways will have a positive impact on the performance which one in general is the better approach still has to be determined. To benefit from both methods we propose a combination of those approaches. By organizing aligned structures that do not exceed the alignment boundary in multiple arrays we first can reduce the overall number of issued reads by using 64 or 128 bit accesses and we can guarantee that all the memory accesses of the single threads in a warp half are coalesced. For this specific

```
typedef struct posmass_s __align__(16){
  float px,py,pz,mass;
} posmass_t;

typedef struct velocity_s __align__(16){
  float vx,vy,vz;
  // plus 32bit hidden padding element
} velocity_t;

typedef struct particle_s __align__(16){
  posmass_t p[N];
  velocity_t v[N];
} particle_t;

particle_t S;
```

Figure 7: Aligned structure with two arrays of aligned structures. Instead of the alignment specifier the aligned build in type *float4* could have been used instead.

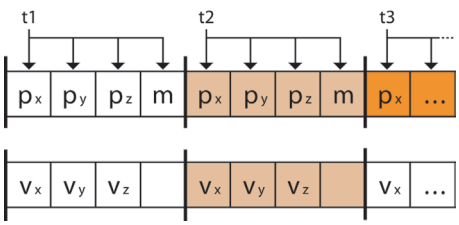

Figure 8: For all seven elements each thread now only has to issue two 128bit reads. With the Structure of Array of Aligned Structures layout the structures in the two arrays can be read coalesced.

example we went from the original implementation with seven single non-coalesced reads to two 128 bit coalesced reads. The resulting memory layout can be described as a "structure of arrays of aligned structures" (SoAoAS) (Figure 7). Furthermore we grouped data elements with similar access frequencies into the same structure to guarantee that no data element is loaded that is not needed at the time. The improved memory access pattern is shown in Figure 8.

5 EXPERIMENTS AND RESULTS

The average memory access time is the most direct metric of the effectiveness of our memory layout optimization. To measure the actual memory access time, we strip down the kernel to containing only the read operations from *global memory*. We used the clock() function to get the clock cycles needed to perform these operations. To prevent the compiler from removing the loads or executing

the clock() function before the loads are completed we had to add instructions that use the loaded values before measuring the time. Thus, the overall test kernel layout can be described as the following:

1. setup all the variables needed
2. get the clock cycle
3. load data from *global memory* using the pattern and layout we want to examine
4. sum up all the data we retrieved from *global memory*
5. get the clock cycle, calculate difference and write result into *global memory* for review.

To measure the effect of the memory optimizations on the benchmark kernels, we used input data similar to the data used in the Gravit project. Since the idea behind those optimizations was to use the best possible memory layout and access pattern for a real-world application and not just for benchmarking. All the tests were run on a Pentium Core 2 Duo running Ubuntu 8.03 with 2.4GHz and 2 GB of RAM, a G8800GTX GPU using the versions 1.0, 1.1 and 2.2 of CUDA driver/compiler.

5.1 Results for the Memory Access Benchmarks

For the memory access benchmarks we can see that coalesced reads and alignment increase the memory bandwidth usage of the program. The baseline version discussed in Section 4.1 shows the worst performance compared to all the other layouts and patterns we suggested. As can be seen in Figure 9 the runtimes for the benchmarks 4.2, 4.3 and 4.4 outperform the unoptimized benchmark 4.1. All the optimizations show promising results as can be seen in Figure 10, e.g. the speedup from benchmark 4.1 to benchmark 4.4 is approximately 50%. Even the the change from seven non-coalesced reads to seven coalesced reads 4.2 gave a speedup of roughly 10%. The biggest improvement resulted from reducing the overall number of reads by switching to an aligned data layout, which allows for 64 or 128 bit reads. Our results show that a combination of alignments and a layout that allows for coalesced reads gives the best results for large-structure based input data. When compiled and run with the 1.1 or 2.2 version of CUDA it can be observed that NVIDIA significantly changed how unoptimized memory accesses are handled up to a level that not coalescing or aligning the data does not effect the overall performance to the same extend as it did with CUDA version 1.0. It is interesting to observe that the impact on the performance has a complete

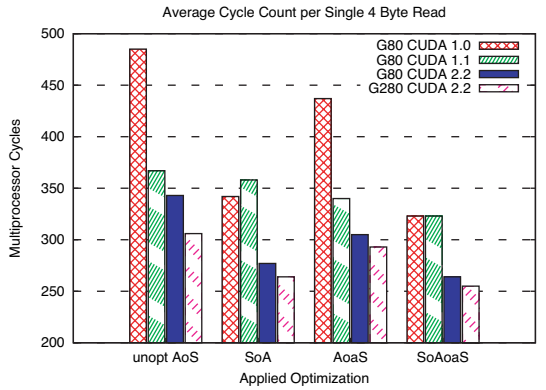

Figure 9: Comparison of the average cycle count to read one of the 4 byte data elements of the structure using the different memory layouts and access patterns. The values obtained are calculated as:

$$\textit{avg cycles per read} = \frac{\textit{cycles needed to read the whole structure}}{\#\ \textit{of 4 byte elements in structure}}$$

different pattern for CUDA 1.1. For the latest CUDA 2.2 revision the impact of the single optimizations has a pattern similar to CUDA 1.0 (see Figure 9). Why CUDA 1.1 was effected differently cannot be determined with the available tools and documentation. Inspection of the generated assembly code did not reveal any obvious changes done by the compiler as well as memory snapshots did not show any automated alignment. For the latest CUDA 2.2 revision our

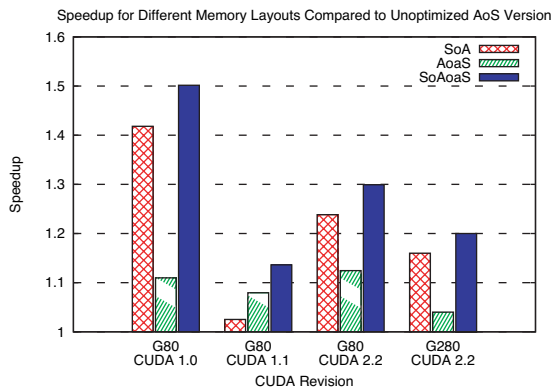

Figure 10: Comparison of the speedup for the different memory layouts and access patterns. The baseline is the unoptimized AoS version discussed in Section 4.1.

final benchmark 4.4 still shows an improvement of roughly 30%. This suggests that the combination of structure alignment and coalesced reads still gives the best performance for large data structures.

6 OPTIMIZING GRAVIT

To see how well the structure of array of aligned structures (SoAoaS) approach works in a real-world application we implemented the Gravit function that calculates the far field forces in CUDA using the optimized data layouts. We used the original Gravit AoS layout for *global memory* as a baseline. To obtain performance results we ran the application and measured the overall runtime from copying the data to the device, through the kernel invocation till after copying the results back. We repeated this procedure for problem sizes from 40,000 to 1,000,000 particles. The data of the benchmarks discussed in section 4 represents the layout we encountered with Gravit. Therefore all the different memory layouts could be applied, where also the SoAoAS approach was the most efficient (see Figure 8). For this data layout we also considered grouping the data in a way that elements with similar access frequencies are stored together. Here the positions vectors and the masses are always needed at the same time; the velocities are read less frequently. This leads to the structures shown in Figure 7. In general the procedure we suggest should be:

1. Group data in portions with similar access frequencies.
2. Split structures that exceed the alignment boundaries into smaller structures of 64 or 128 bits that can be aligned.
3. Organize the aligned structures in arrays to allow for coalesced reads.

6.1 Loop Unrolling

Most of the algorithms that are suited for being implemented in CUDA are heavily loop based. Not surprisingly, loop optimizations, if applied appropriately, might be very beneficial for CUDA kernels. Loop optimizations are mainly designed to enhance the program's memory hierarchy performance such as loop tiling and improve instruction-level-parallelism of a program such as loop unrolling. Among the loop optimization, those that target at memory hierarchy performance cannot be directly applied to GPU programs because the memory hierarchy performance on GPU must be optimized within the context of a collection of threads instead of optimizing only one thread as assumed by most existing loop optimizations for memory hierarchy. Among the loop optimizations for instruction-level-parallelism, loop unrolling is generally

thought as a simple optimization. On a common CPU, loop unrolling allows the compiler to do extensive instruction reordering and scheduling to hide especially the latencies of slow memory operations. This leads to a speedup of the overall application. However, loop unrolling is worth being re-examined under the context of GPU programming because the general pattern of programming for GPUs reveals additional guidelines as to which loop level to unroll and the performance improvement from loop unrolling on GPUs involves factors that are considered less important for loop unrolling on a CPU.

In many cases, the part of a program running on the GPU has similar patterns of code. A CUDA kernel handling an $O(N^2)$ algorithm like the N-body problem, the Gravit algorithm being an example, generally has the following structure:

1. The thread setup code *S* that sets up the environment for one single thread. Here all the information is organized that will be needed just by this single thread. This portion of the code is executed once for every thread. If every thread works on one single data element we have *N* threads.
2. The thread block setup code *B* or input data fetcher. The portion of the code that fetches the data used by all or most threads in a single block and stores it into shared memory. This portion of code is commonly executed $\frac{N}{K}$ times where N is the problem size and *K* is the number of data elements per slice the input date is split in. For problems with single dimensional input data like Gravit, *K* usually equals the block size since every thread works on one element and every thread in a thread block loads one element per slice.
3. The usually pretty small inner loop *P* which is executed N^2 times and handled by the thread represents the essential part of the algorithm where most of the calculations are performed.

Let's assume that *S,B,P* represent the fraction of instructions in those code segment and that

$$S + B + P = 1.0 \tag{2}$$

which represents all the instruction of the kernel with one iteration for each loop. Usually the innermost loop *P* is very small, containing only a few instructions, and has a small number of iterations compared to the problem size, e.g., K times in the Gravit algorithm. Furthermore, the innermost loop *P* if unrolled likely will not reveal any possibilities for instruction reordering, like the case in the Gravit algorithm. Therefore, if we still use the mindset of loop unrolling for CPU, the innermost loop of a GPU kernel will not be thought as a

good target for loop unrolling. However, the benefit/cost analysis for loop unrolling is different for GPU kernels. We can come up with a formula that will give us an idea of how the overall instruction load changes for a problem of size N and a block size of K.

$$Speedup = \frac{S_1 + \frac{N}{K} \times B_1 + N \times P_1}{S_{ur} + \frac{N}{K} \times B_{ur} + N \times P_{ur}} \approx \frac{P_1}{P_{ur}} \tag{3}$$

Where S_1, B_1P_1 represent the fraction of instructions for the original code and S_{ur}, $B_{ur}P_{ur}$ for the fully unrolled code. For large Ns we can neglect the impact of the setup code S. Even the block setup code B only effects the overall instruction count by not more than $\frac{N}{K}$, where K is the block size which should be $K \geq 128$ which makes it less significant than P. The actual instruction count for on iteration off P is typically very small. Consequently, if we can reduce the amount of instructions of the innermost loop we will see that the overall number of instructions will get reduced by almost the same fraction.

For our example, the Gravit simulator has an innermost loop that does not reveal any possibilities for instruction reordering. However, when we unroll the loop starting from unrolling it four times to fully K (here 128 times), the result is rather surprising. With a fully unrolled loop we gain a speedup of 18%. The nature of this speedup is plainly the reduction of overall instructions that have to be performed per iteration of the inner most loop. It is a pretty common thing for a CUDA kernel that the "kernel" of the algorithm that has to be performed for every data element consist of only a few instructions. In the case of Gravit the innermost loop consists of a little more than 25 instructions including the instructions needed for the loop. By fully unrolling the loop we get rid of one compare, an add, a jump plus an additional add to calculate the address offset that now is hard coded. By just looking at the portion of code this doesn't look like a huge improvement compared to the increase in code size. But if we look at the overall problem size and the importance of this innermost loop, we will see that this code segment is executed N^2 times and represents more than 95% of the overall execution time. By fully unrolling this portion of the code we just reduced the overall number of instructions for this part by nearly 20%.

Furthermore, the impact of register pressure on loop unrolling is probably not a major factor when deciding whether to unroll a loop and if yes how many times to unroll on CPU. However, the impact of register pressure in the application of loop unrolling is zoomed on GPU. Given the large number of

threads that try to run at the same time on a GPU, the limited number of registers of a GPU often reduces the number of threads that can be in active state simultaneous. More precisely, because not enough registers can be allocated to the hundreds of threads on a GPU, the execution of many threads have to be backlogged. Improving the GPU occupancy of a program has shown to be crucial to maximize the parallelism and improve speedup. One huge benefit of fully unrolling a loop in CUDA is the removal of the loop iterator which frees up one register. This reveals a chance to increase the occupancy of the device and therefore performance.

7 RESULTS FOR GRAVIT

The above analysis of loop unrolling in the context of GPU has been fully reflected in our application of loop unrolling to Gravit. By fully unrolling the loop we reduced the number of instructions of one single iteration by roughly 18% and we gained an overall speedup of 18% by doing so. Further more it shows that optimizing the inner most loop of the algorithm in many cases is more beneficial than extensive *global memory* optimizations. Since access to global memory for most kernels will take place in section *B* which will not affect the overall performance for more than a few percents where optimizations in the inner loop might affect performance by a magnitude more. Furthermore, in the case of Gravit we reduced the maximum number of registers used by a single thread from 18 to 17. In addition we observed that that by manually applying invariant code motion we were able to reduce the register pressure further for the innermost loop. Through this and switching to a block size of 128 threads we were able to increase the GPU occupancy from 50% to 67% resulting in another 6% of speedup compared to the baseline GPU implementation. The results for all those optimizations are shown in Figure 11. Whereas the fully optimized version represents a 87× speedup compared to the original serial CPU version.

7.1 Multi GPU Implementation

In addition we implemented a multi GPU version to utilize more than one GPU to calculate the N-body problem. The multi GPU version is based on the fully optimized version discussed in the previous section. To run that version on more than one GPU a POSIX Thread (Pthread) has to be generated for every GPU we want to use. In the test system we only had two G80 GPUs available. When the algorithm is performed on more than one GPU, each GPU will have to access the position and mass of every particle

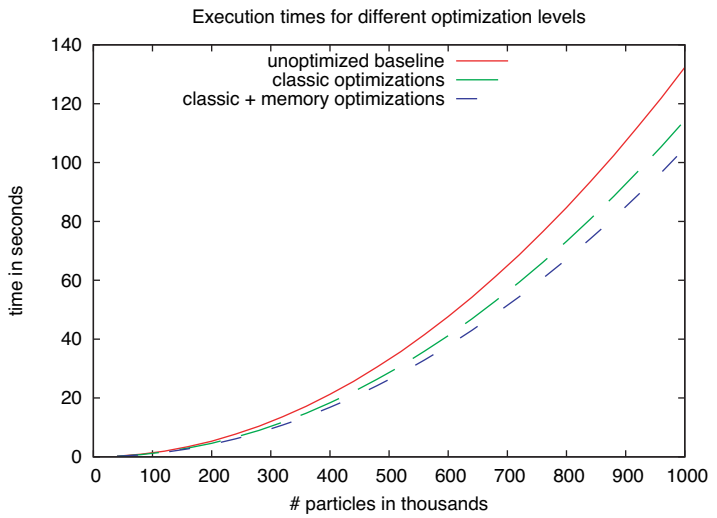

Figure 11: Comparison of the different optimization levels for Gravit with different problem sizes.

in the system. Therefore this data has to be copied to any device that will be used. On the other hand each card only needs the velocities of the particles it is actually calculating the far field forces for. For the data in a system with two GPUs this means that the first card will get all the positions and masses but only the first half of the velocities, the second card will also get all positions and masses but only the second half of the velocities. Over all making the code suitable for multiple cards did not involve that many changes to the actual kernel code. The major changes took place on the host side where the Pthreads had do be generated and the data has to be split and after the calculation combined again.

The multi GPU approach adds some overhead that has to be considered. First of all the generation of the Pthreads, which is a complex task that adds a large amount of overhead, and having to copy and update data on multiple cards for every iteration. This overhead becomes less and less significant for bigger problem sizes. To visualize this overhead we calculated the speedup based on the execution time of only one, the first, iteration. This includes the time needed to generate the Pthreads and to copy all the initial data. Therefore the curves would be way steeper and would level out much faster the more iterations we would run, since the Pthreads only have to be generated once. But even for this single iteration the multi-GPU version running on two G80 devices outperforms

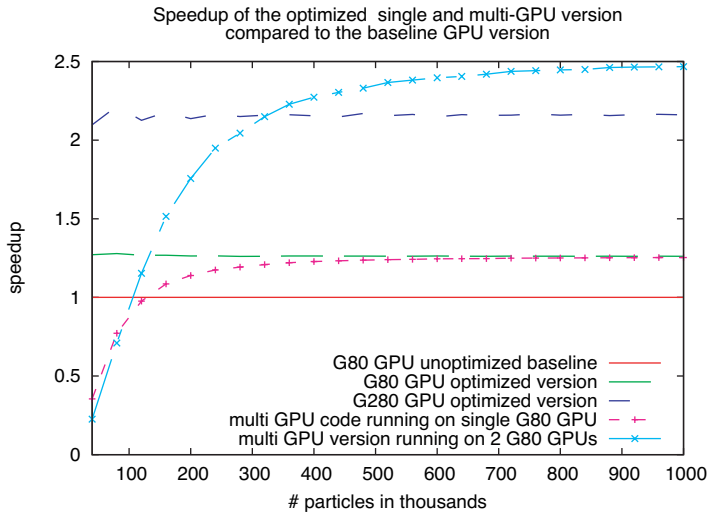

Figure 12: The speedup of different implementations vs. the original GPU implementation. The curves for the multi-GPU version show the overhead added by the generation of the Pthreads and the additional host-device memory transfers. The graph is based on the execution time for the first iteration.

the single G80 implementation for a problem size of 130,000 particles. For large problem sizes the overhead is negligible and the second cards almost doubles the performance. The two card solution still outperforms NVTDIA's G280 GPU which runs 30 instead of 16 multiprocessors (Table 1) and can have more threads in active state at the same time. The G280 reaches a speedup of approximately 2.1× versus the multi-GPU version running on two G80 cards reaches almost 2.5× compared to the unoptimized single G80 version.

7.2 Extremely Large Problem Sizes

The size of the problem handled by a GPU is pretty much only restricted by the size of the global memory that is available and the time the user wants to spend till the computation is completed. The largest problem size for Gravit in our optimized version that can be handled on the device by a G80 GPU is roughly 22 million particles. It would be possible to handle even bigger data sets by saving intermediate results and swapping the input data. The overhead introduced by those swapping operations would be minimal compared to the already enormous execution time. The swapping only would have to be performed once every 14 hours for the single GPU version when one slice of

Table 2: Execution time of one iteration for the largest possible problem size of 22 million particles on a G80 GPU.

version	single G80	two G80s	single G280
time in h	14.22	7.43	8.30
speedup	1.0	1.91	1.71

the input data is handled and would probably not take more than some milliseconds. We ran this problem size on the optimized single GPU version as well as on a two GPU multi-GPU version. For the larger problem sizes the only modification that had to be performed in the code was to add a function that dynamically determined the grid size. Since the grid x-dimension is limited to 65536 and the program handles 256 particles per block this would limit the problem size to 16.7 million particles. Therefore larger problem sizes have to be handled by a 2 dimensional grid.

8 RELATED WORK

Memory layout optimizations and loop optimizations have been studied extensively for general purpose CPUs. The many memory layout optimization studies [6, 7, 8] relate to the memory layout optimizations proposed in this paper, because all approaches focus on improving program performance by changing the layouts of programmer-defined data structures. Loop tiling is probably one of the most important program transformations to improve the cache locality of a program and is studied in [9, 10, 11, 12, 13]. Studies of loop unrolling include those that analyze the benefit/overhead ratio and predict the best unrolling factors [14, 15].

Work on loop optimizations for GPUs concentrating on loop unrolling was presented in [16]. Their experiments concentrated on optimizing matrix multiplication for the GPU platform. The N-body problem has been researched and implemented in many applications during the years. Optimized versions can be found in the SPEC benchmarks under AMMP [17] which uses a fast multipole method. Other implementations include [18, 19] from which the CUDA implementation is the most interesting one, which was completely reimplemented solely for the CUDA system. Our focus, however, was not to implement a program from scratch. Our intention was to accelerate existing programs, by extracting computational extensive kernels and move them to the

GPU. This allowed us to improve the performance of an existing application without burdening the programmer with extensive code changes.

9 CONCLUSIONS

Even though GPUs have become an integral part of high performance computing, optimizations for those architectures still lack sufficient investigation. Among them, the memory layout optimizations for the specific memory organization of a GPU are expected to significantly accelerate programs on GPUs. Particularly, applications that frequently access data stored in large structures suffer from unoptimized access patterns. Our work presented a novel idea on how to efficiently use the heterogeneous memory space of these architectures. We developed and tested our approach in a real world application, Gravit. Our layout optimization technique consists of splitting large structures into smaller sub structures that are aligned and stored consecutively in global memory. For CUDA 1.0, our results show a 50% speedup when using this approach compared to an unoptimized layout. Even for improved frameworks like the CUDA version 2.2, our method still provides 30% speedup. Furthermore we analyzed how common loop optimizing techniques affect the performance of heavily loop based CUDA kernel functions. Because of the blocking of computation to match the GPU architecture, large loops that could not be completely unrolled in the original application now can be fully unrolled. Our tests show that when completely unrolling a loop in CUDA and applying simple manual optimizations like invariant code motion, we can gain surprising speedups. The main reason is that the nature of most CUDA implementations determines that the innermost loop just holds a few instructions but works on huge data-sets. Therefore, the relatively few instructions used by the innermost loop still represent a large percentage of the overall instructions executed for the algorithm. In addition, fully unrolling a loop frees registers which might lead to an increase of the overall GPU occupancy[4]. Future work for our memory layout is to study how the basic principles can be tuned for different GPU models or new developing environments such as OpenCL, which we believe will equally benefit from a combination of the optimization techniques proposed in this paper.

[4]ratio of threads in active state to possible maximum number of active threads

REFERENCES

[1] M. Woo and M. Sheridan, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[2] NVIDIA, "Compute Unified Device Architecture-Programming Guide Version 3.0," 2010.

[3] "Open64. http://www.open64.net."

[4] "Gravit home page, [online], http://gravit.slowchop.com."

[5] J. Barnes and P. Hut, "A hierarchical O (N log N) force-calculation algorithm," *Nature,* vol. 324, no. 6096, pp. 446–449, 1986.

[6] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in *PLDI '99: Proceedings of the ACM SIGPLAN1999 conference on Programming language design and implementation,* (New York, NY, USA), pp. 13–24, ACM Press, 1999. Separate a class into "hot" class and "cokTclass.

[7] T. Chilimbi, M. Hill, and J. Larus, "Cache-conscious structure layout," in *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation,* pp. 1–12, ACM New York, NY, USA, 1999.

[8] C. Lattner and V. Adve, "Automatic pool allocation: improving performance by controlling data structure layout in the heap," *SIGPLAN Not,* vol. 40, no. 6, pp. 129–142, 2005.

[9] S. Coleman and K. s. McKinley, "Tile Selection Using Cache Organization and Data Layout," in *Proc. of Int. Conference Programming Language Design and Implementation,* pp. 279–290, June 1995.

[10] M. Lam, E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," in *Proc. of the Int. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 63–74, October 1991.

[11] P. Panda, H. Nakamura, N. Dutt, and A. Nicolau, "Augmenting Loop Tiling with Data Alignment for Improved Cache Performance," *IEEE Trans, on Computers,* vol. 48, pp. 142–149, February 1999.

[12] G. Rivera and C. Tseng, "Data Transformations for Eliminating conflict Misses," in *Proc. of Int. Conference Programming Language Design and Implementation,* pp. 38–49, June 1998.

[13] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pin-gali, P. Stodghill, and P. Wu, "A Comparison of Empirical and Model-driven Optimization," in *Proc. of Programing Language Design and Implementation,* pp. 63–76, June 2003.

[14] R. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimizations of Sofware and the ATLAS Project," *Parallel Computing,* vol. 27, no. 1-2, pp. 3–35, 2001.

[15] P. Kisubi, P. Knijnenburg, and M. O'Boyle, "The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling," in *Proc. of the International Conference on Parallel Architectures and Compilation Techniques,* pp. 237–246, 2000.

[16] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, and W. Wenmei, "Program Optimization Space Pruning for a Multithreaded GPU," in *Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization,* pp. 195–204, ACM New York, NY, USA, 2008.

[17] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady, "SPEComp: A new benchmark suite for measuring parallel computer performance," *Lecture Notes in Computer Science,* pp. 1–10, 2001.

[18] J. P. Lars Nyland, Mark Harris, *Fast N-Body Simulation with CUDA,* ch. 32, pp. 677–695. GPU Gems, Addison-Wesly, 2007.

[19] A. Dellson, G. Sandberg, and S. Mohl, "Turning FPGAs Into Supercomputers," *Cray User Group,* 2006.