Faculty of Physics and Astronomy

University of Heidelberg

Diploma thesis

in Physics

submitted by

Matthias Bach

born in Darmstadt

2009

Utilization of Graphics Processing Units in Applications for High Energy Physics

This diploma thesis has been carried out by Matthias Bach at the
Kirchhoff Institute of Physics
under the supervision of
Prof. Dr. Udo Kebschull

Nutzung von Grafikprozessoren in Anwendungen der Hochenergiephysik

Die Menge an Daten die Experimente in der Hochenergiephysik erzeugen steigt immer weiter an. Klassische skalare Programme können aber nicht mehr davon profitieren, dass Prozessoren immer höher getaktet werden. Um größere Datenmengen verarbeiten zu können sind Anwendungen notwendig, die parallele Architekturen nutzen. Diese Diplomarbeit zeigt auf NVIDIA CUDA basierende Implementierungen und Optimierungen von drei Anwendungen verschiedenen Typs auf Grafikprozessoren. Zelluläre Automaten, welche für die Spurrekonstruktion verwendet werden können, werden durch eine Implementierung des Game of Life repräsentiert die 20 mal schneller als eine SSE nutzende Implementierung auf dem Hauptprozessor ist. Die Laufzeit eines Kalman-Filters, mit dem Spurparameter geschätzt werden, wird auf ein Siebtel gegenüber der auf SSE basierenden Implementierung beschleunigt. Auch der Linpack-Benchmark, als Anwendung zur Lösung von Gleichungssystemen, wird analysiert und so optimiert, dass er auf einer Grafikkarte 67. 2 Gflops erreicht und auf mehrere Karten skaliert.

Utilization of Graphics Processing Units in Applications for High Energy Physics

Experiments in high energy physics are continuing to increase the amount of data produced. Traditional scalar applications however can no longer profit from rising processor clock speeds. To further increase the performance of data processing applications are required to take advantage of parallel architectures. Utilizing NVIDIA CUDA, this thesis presents implementations and optimizations of three applications on the graphics processing unit. Three different classes of applications are covered. Cellular automatons, which can be used for the problem of track finding, are represented by a Game of Life implementation that is 20 times faster than a SSE based implementation on the main processor. For Kalman filters, used for track fitting, a speedup of seven versus the SSE based version is accomplished. Finally the Linpack benchmark, as an application for solving systems of linear equations, is analyzed and optimized to achieve 67. 2 Gflops on one graphics device and scale to more.

Contents

Chapter 1

Preface

New particle accelerators are built in the search for heavier particles and new physics. In search for the Higgs boson the Large Hadron Collider (LHC) at CERN collides protons with an energy of 14 TeV. To research quark-gluon-plasmas, lead ions will be collided at an energy of 1150 TeV[1]. The Large Electron-Positron Collider (LEP), LHC's predecessor at CERN, operated at an energy of only 190 GeV[2]. To compensate lower propabilities of the processes researched beam crossing rates of modern colliders increase. While beam crossing at LEP occurred every 22 μs there are only 25 ns between two beam crossings in the LHC.

With the collision energy the amount of created particles increases. Along their track of flight these particles produce hits in the detector. The increased crossing rates further increase the amount of data that needs to be processed per time. In today's experiments the amount of data produced by a collision exceeds what is economically feasible to be stored permanently. The Alice High Level Trigger at CERN is challenged with the task of reducing incoming data of 16 GiB/s to 2 GiB/s which can be stored[3]. To ensure that no physically interesting events are lost, a full event analysis needs to be performed within 5 ms. This analysis will then be used to decide whether an event is stored or not.

Traditionally the problem of increasing data rates is solved by using more and faster computers. Since the Intel Pentium 4 reached a clock rate of 3.80 GHz clock rates of x86 based processors have no longer increased. Modern processors even use slower clock rates[4]. Due to the power consumption increasing with the square of the clock rate higher clock rates become too inefficient. Modern processors increase available performance by means of multiple cores and improved vectorization capabilities. As processors no longer increase their clock speeds it is no longer possible to wait for faster processors to speed up an application.

Over the last years raw performance numbers as displayed by figure 1.1 have increased faster for graphics processing units (GPU) than for central processing units (CPU) traditionally used for computation. The traditional task of GPUs, rendering multiple triangles to the screen and shading thousends of pixels, is inherently parallel. Therefore, while clocked at moderate speeds, GPUs have tra-

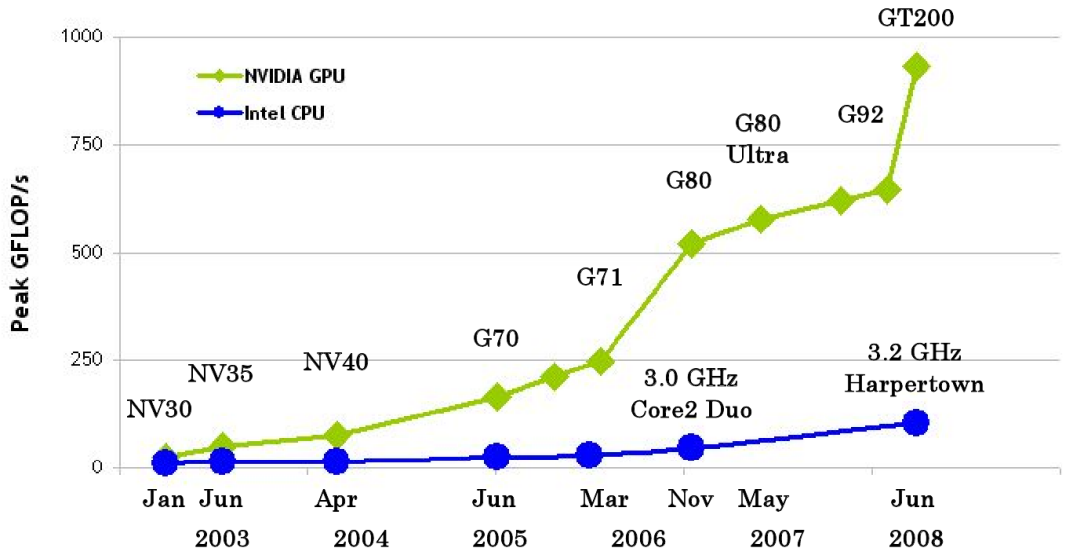


Figure 1.1: Plot comparing theoretical floating point performance of CPUs and GPUs[5]

ditionally been focused on data parallel computations. With the advent of programmable graphics hardware for the consumer market in 2002 general purpose computation on the GPU (GPGPU)[6] emerged, making this processing power available to a range of non-graphical applications like fluid-dynamics[7], k-nearest neighbour search[8], molecular dynamics[9], password recovery[10] and Vandermonde-based Reed-Solomon coding[11].

The following chapters describe the attempt to exploit this performance of the GPU for three classes of applications. Optimizations are discussed and the achieved results compared to existing CPU based implementations. Before discussing the implementations chapter 2 gives a short introduction into the hardware architecture and the programming model used.

One class of applications are cellular automatons. As an example implementation chapter 3 presents a GPU based implementation of the Game of Life[12]. The chapter will also show how this application is related to the track finding problem of high energy physics.

In chapter 4, a Kalman filter is ported to the GPU. The Kalman filter is used for track fitting in high energy physics.

As a representative for the class of applications solving dense fields of linear equations the Linpack benchmark[13] is discussed in chapter 5.

Finally, chapter 6 summarizes the results and gives a short outlook on possible future work on the discussed classes of applications.

Chapter 2

An Introduction to CUDA

2.1 The Hardware

Modern NVIDIA graphics cards are build around an array of streaming multiprocessors[5]. These multiprocessors are multithreaded, focused on a high floating point operation throughput instead of single thread performance. As shown in figure 2.1 cache space is traded for floating point units, relying on zero overhead thread switching to hide memory latencies.

In current GPUs each of these multiprocessors consists of eight scalar processor cores, a special function unit which can execute complex functions like sinus or inverse square root, a double precision unit, a large register file, an instruction decoder and a block of shared memory.

Threads are distributed onto the multiprocessors in groups called blocks. These are also the unit of granularity at which synchronization statements in the code are executed. The blocks are further subdivided into warps, each warp containing 32 threads. At each cycle the multiprocessors takes a warp that is ready for execution and runs it on the scalar processors in a single instruction multiple threads (SIMT) fashion. The shader clock, which is the clock for the scalar processors, is faster than the core clock of the device. Therefore a scalar processor can process more than one thread in one clock cycle of the multiprocessor.

The SIMT execution model is similar to the single instruction multiple data (SIMD) execution model on the CPU. They both use one instruction decoder to feed multiple algorithmic logic units. This means all algorithmic logic units will always execute the same operation. There are however important differences. In the SIMD execution model one thread, meaning one instruction counter and register set, is used for all algorithmic logic units. In contrast in the SIMT model an own thread, meaning an own instruction counter and register set, is used for each algorithmic logic unit. This allows the code executed to be agnostic of the actual physical algorithmic logic unit count, while in SIMD it needs to be aware of the width of the vector registers used to feed them. In addition an algorithmic logic unit in the SIMT model can skip execution of an instruction based on special predicate registers

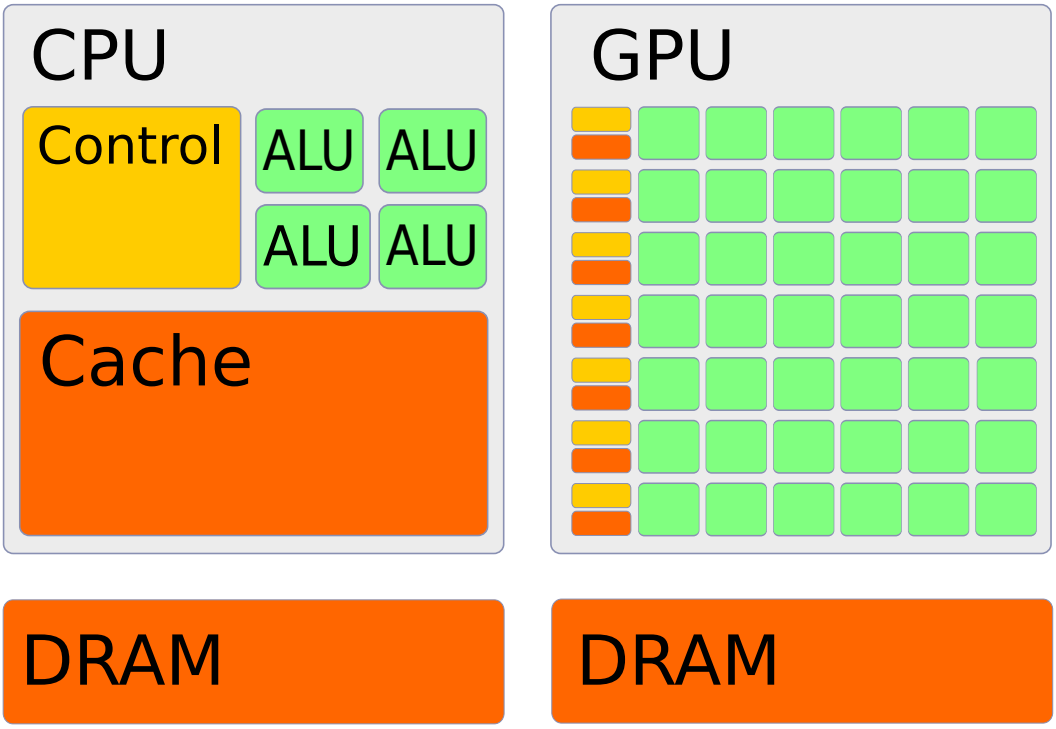


Figure 2.1: Transistor distribution in a CPU versus in a GPU

or if its instruction counter does not match the address of the currently executed instruction. In the SIMD model masking of operands is required to achieve a similar effect.

The memory of the GPU is non uniform and split into global, constant, texture, local, shared memory and register file. This memory structure is shown in figure 2.2. Global memory is the GPU's equivalent to CPU main memory, external to the GPU. Every thread run on the GPU can randomly access global memory for read and write. It is uncached, therefore latency is about 400 to 600 clock cycles. Local memory is a portion of global memory that is addressed local to each thread. Its main purpose is to hold variables that could not be fitted into registers, however on the level of assembler code it is of course freely usable for anything.

The constant and texture memory are cached. These caches are located at the level of groups of multiprocessors[14]. How many multiprocessors belong to one group is listed in table 2.1. The constant memory is limited to 64 KiB with a specified working set of 8 KiB per multiprocessor. It has a broadcast mechanism so that all threads can read from the same address at once. As the name suggests constant memory cannot be written from the device.

Texture memory is a part of global memory that is readonly and cached by accessing it through the texture unit. The cache of the texture unit is a streaming

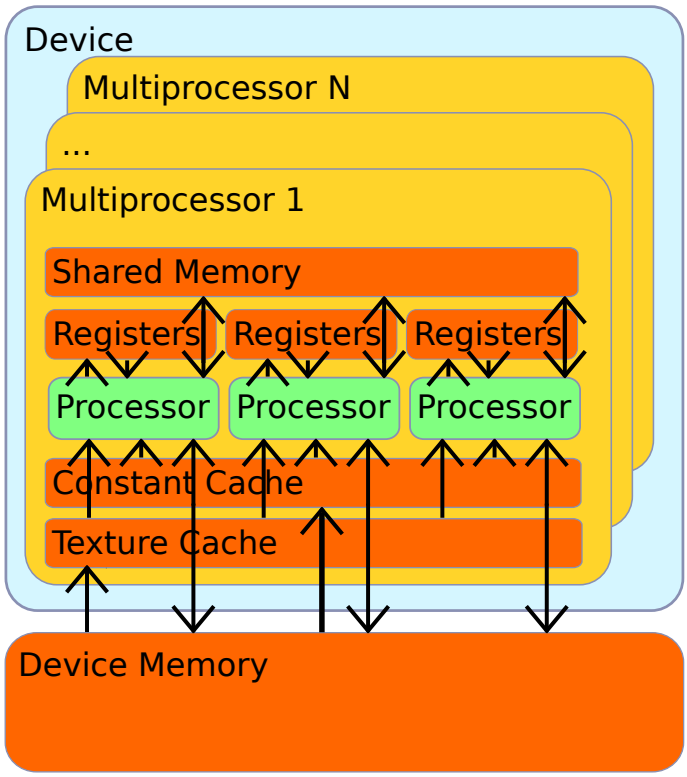


Figure 2.2: Memory structure as specified by CUDA

cache optimized on two dimensional spatial locality between threads. The texture unit is more than a simple cache. It provides additional features like addressing using normalized floats and interpolation of values. It can be used with one, two and three dimensional addressing and is limited to simple types. The working set varies between 6 KiB and 8 KiB depending on the device.

Shared memory and the register file are located on chip in the multiprocessors. Shared memory is partitioned into 16 banks. It is allocated per block and can be accessed as fast as registers as long as different memory banks are accessed by the threads. It is limited to 16 KiB. The register file is dynamically partitioned into register sets for the threads executed on the multiprocessor. Its size depends on the device, exact numbers are given in table 2.1. The registers of a thread are kept in the register file from thread creation until completion of execution. This allows the multiprocessors to switch between warps of threads in every cycle without additional costs. The dynamic partitioning allows varying numbers of threads to be executed synchronously depending on the number of registers used.

CUDA defines several compute capabilities that specify the features supported by the hardware. These capabilities have the format of versions, each capability including the features of the below compute capabilities. Compute capability 1.0 specifies the hardware of the first CUDA capable NVIDIA GPUs. Among other

Model	Multiprocessors	Multiprocessors per Group	Registers per Multiprocessor
8800 GTS 512	16	2	8192
GTX 280	30	3	16384
Tesla S1070	4×30	3	16384

Table 2.1: Technical data for the Multiprocessors of used GPUs

things compute capabilities 1.1 and 1.2 add atomic functions on global and shared memory. Finally compute capability 1.3 adds support for computations in double precision.

On the currently available cards of compute capability 1.3 there is only one double precision unit per multiprocessor. Therefore double precision throughput of the device is at least a factor eight lower than single precision throughput. In addition the hardware is not able to issue instructions to double and single precision units in parallel. It does support this for combining the single precision units and the special function unit[15].

2.2 The Programming Model

To write SIMT applications CUDA defines an extension to C/C++ that allows to write data parallel applications in which the parallelization happens on the thread level. This is similar to OpenMP[16] as both parallelize on the level of threads and work in a shared memory environment. In OpenMP parallelization is done by compiler pragmas and often happens by autoparallelizing loops and similar constructs. In MPI applications are also parallelized on a thread level, but they use messages instead of shared memory for communication. In CUDA the programmer has to write an actual thread function and keep in mind the non uniform memory of the GPU.

From a programmers point of view execution of threads happens in a grid of blocks as shown in figure 2.3. All threads in a grid execute the same function. The dimensions of this grid can be specified by the programmer. For convenience the grid can be two and the blocks three dimensional. Due to the SIMT execution model the threads can take different code paths within the function. Within the blocks threads can cooperate using shared memory and synchronize at barriers given by the `__synchronize()` intrinsic. There is no synchronization between blocks. If multiple grids are launched those are executed sequentially. Therefore synchronization between blocks can be done by splitting the application into two functions launched in separate grids.

CUDA specifies some additional function qualifiers to specify where methods are to be executed. If nothing is specified execution on the host is implied, otherwise

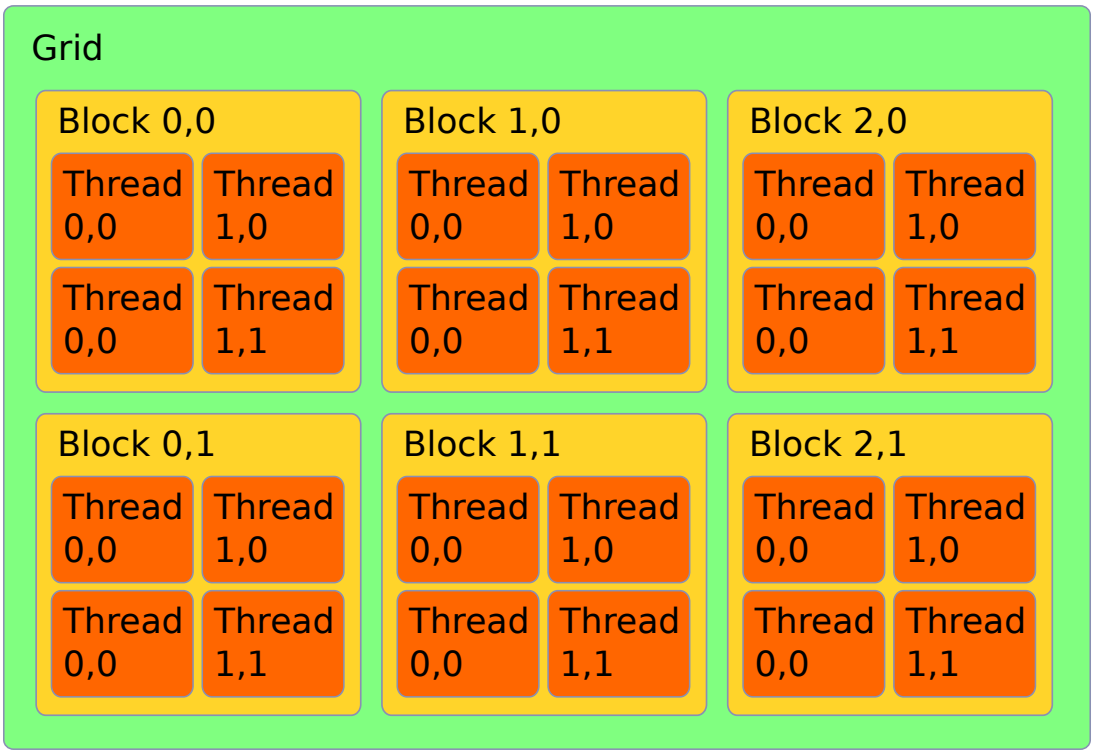


Figure 2.3: A grid of thread blocks

executibility can be specified by the `__host__` keyword. The equivalent for functions to be invoked from and executed on the device is called `__device__`. The `__host__` and `__device__` specifiers can be combined to make a function usable on both platforms. Methods to be executed on the device but invoked from the host are called kernels and qualified by `__global__`. There is no way to invoke a function on the host from the device. The GPU does not have a hardware stack. Therefore function invocations are always inlined.

Similar qualifiers exist to specify the location of variables in GPU memory. `__device__` defines a variable to be located in device memory, by default global memory. `__constant__` allows to declare variables located in constant memory and `__shared__` allows to declare variables located in shared memory. In contrast to other parallel programming models, such as Unified Parallel C[17], CUDA does not offer mechanisms to declare what memory pointers point to. Locations of variables used as function parameters are automatically determined by the compiler.

Each thread has access to the special variables `threadIdx`, `blockDim` and `blockIdx`. These represent the threads position in the grid.

Global memory on the device can be dynamically allocated from the CPU. Pointers to such memory regions can e.g. be handed to the GPU code as kernel parameters.

Transfer of data to the GPU and back needs to be performed explicitly by using using special memory copy functions from the CUDA library.

A special notation exists to invoke kernels. This allows to pass the dimensions of the grid and the blocks. If a `__shared__` array has been declared without a size the amount of shared memory pointed to by this array needs to be specified, too. Finally the stream to execute the kernel in needs to be specified. If no stream is specified the kernel will launch after all previous CUDA calls have finished.

The mentioned streams can be used to declare data dependencies in between CUDA calls. On some devices it is possible to overlap memory copies and kernels. This can be used to hide some of the computation or data transfer time. To use these streams most CUDA library methods can be called asynchronously. Kernel launches and all library calls that operate only on the GPU are always asynchronous.

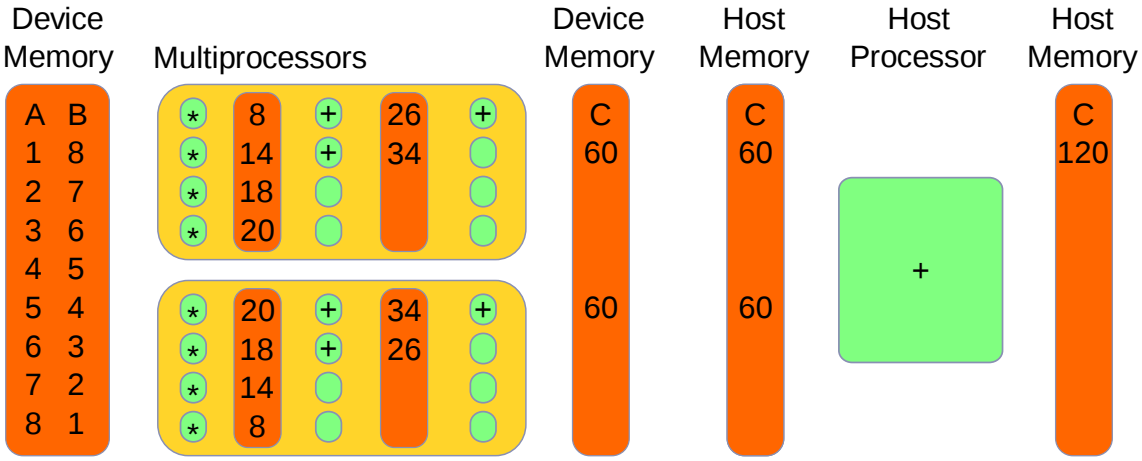


Figure 2.4: Visualization of the algorithm calculating a scalar product

The following short description of a scalar product shows how a program can be implemented using this model. The calculation required is visualized in figure 2.4 for two blocks with 4 threads each. Assuming the operands to be contained in two arrays equivalent memory needs to be allocated on the device and the operands copied. The scalar product consists of an element wise product and a sum. The element wise product is trivial to implement on the device as each thread can calculate the product for one index. The sum requires cooperation in between the threads, which is only available inside blocks. One possibility is to calculate the sum per block in shared memory. The sum for each block can then be stored in global memory and in a last step the host can add those sums up for the final result. To sum the operands in shared memory each thread could store its result of the multiplication at an index corresponding to its `threadIdx`. Then the first half of the threads could add the value at $\texttt{threadIdx} + \texttt{blockDim} \div 2$ to their own values. This can recurse until only one thread is left which will write the result for this block to the global memory. This way the summation is executed in $O(\log_2(\texttt{blockDim}))$ cycles. A schematic

example code is given below.

```
// shared memory for in block cooperation
__shared__ buffer[];

// kernel to be executed in parallel on the gpu
__global__ void gpuScalarProduct(float* op1, float* op2, float out) {
  int id = blockIdx.x * blockDim.x + threadIdx;

  buffer[ threadIdx.x ] = op1[ id ] * op2[ id ];

  for( int offset = blockDim.x / 2; offset /= 2; offset > 1 ) {
    buffer[ threadIdx.x ] += buffer[ threadIdx.x + offset ];
  }

  out[ blockIdx.x ] = buffer[ 0 ] + buffer[ 1 ];
}

// the cpu function that sets up the gpu execution
float scalarProduct(float* op1, float* op2, int N) {
  // ... allocate memory and copy op1 and op2 ...

  // use 128 threads per block
  dim3 blockDim = 128;
  // the number of blocks is given by the size of the vectors
  dim3 gridDim = N / blockDim.x;
  // the buffer needs to be able to store one float per thread
  // remember: shared memory is allocated per block
  size_t shared = blockDim.x * sizeof( float );

  gpuScalarProduct<<< blockDim, gridDim, shared >>>( gpu_op1,
    gpu_op2, gpu_out );

  // ... wait for the kernel to complete and copy data back ...

  // sum the result of the blocks
  int res = 0;
  for( int i = 0; i < gridDim.x; ++i ) {
    res += cpu_out[ i ];
  }

  return res;
}
```

2.3 Performance Characteristics

GPUs are clocked lower than CPUs. The GTX 280 is clocked at 1.3 GHz and the 8800 GTS 512 at 1.6 GHz. Therefore the single thread performance of the GPUs is lower than for CPUs. This is also shown by measurements in section 4.3. When running more than one thread however a speedup proportional to the number of concurrently active threads can be achieved.

For most floating point operations the current GPUs process one warp in four cycles. As the clock of the multiprocessor is only half that of the scalar processors[18] the hardware probably actually executes the warp as two halfwarps, each halfwarp processing 16 threads in two clock cycles. Those two clock cycles would resemble one cycle of the instruction decoder. From a technical point of view this makes sense as computer graphics computations require a lot of operations using 4×4 matrices.

There is no algorithmic integer unit on the device. Integer calculations are therefore emulated in the floating point unit using the mantissa to represent an integer. Operations on 32 bit integers are therefore more costly than floating point operations. A multiplication takes 16 clock cycles. It is possible to use 24 bit operations instead. As the mantissa of a single precision float is 24 bit those will be as fast as floating point operations.

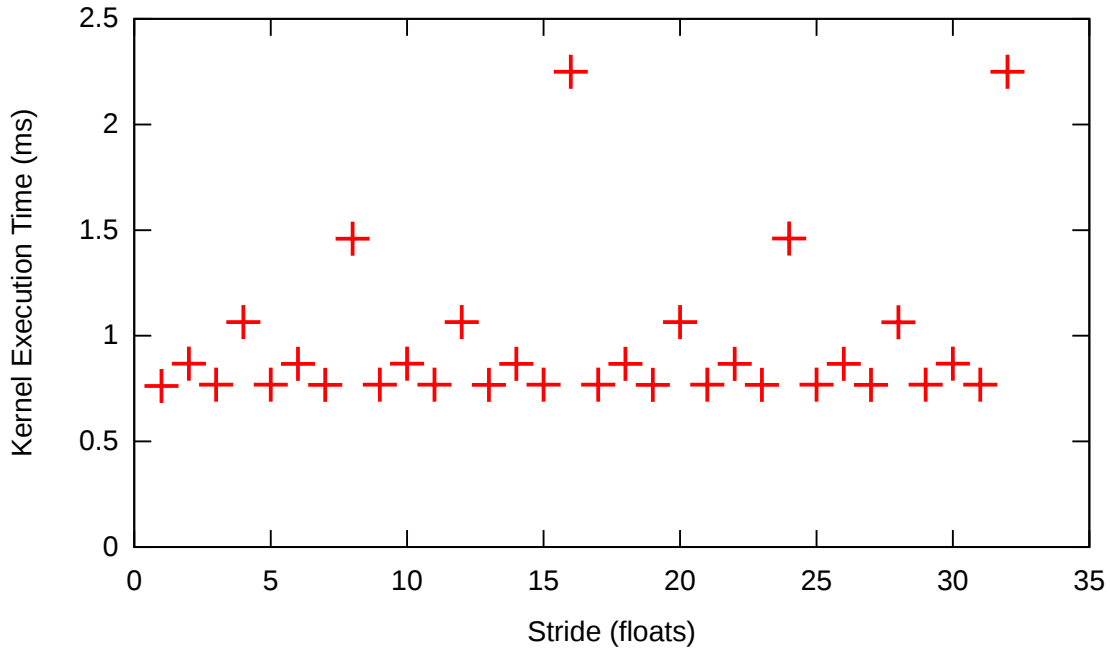


Figure 2.5: Effects of access to shared memory using different offsets between threads

As mentioned in section 2.1 the shared memory is divided into 16 banks. These banks are mapped to addresses so that threads reading consecutive floats from an array will access consecutive banks. If multiple threads in a warp access the same bank

these accesses need to be serialized. Shared memory has a broadcast mechanism, so in the case of accesses to the same address by multiple threads this serialization is not required. The effects of this serialization can be measured and are shown in figure 2.5. Access to shared memory takes two cycles, once again one clock of the multiprocessor. As a warp is processed in four cycles accesses from the first and the second half of the warp cannot collide, these happen in different cycles of the multiprocessor.

Memory requests by a single thread can read four, eight or 16 bytes into a register with a single instruction. As mentioned in section 2.1 the latency for such an instruction is 400 to 600 clock cycles. To optimize throughput memory transactions of multiple threads can be coalesced into single transactions of 32, 64 or 128 bytes. To use this coalescing several criteria need to be met. For hardware implementing compute capability 1.0 or 1.1 all threads of a warp need to access a continuous memory region that is aligned to 64 or 128 bytes. Consecutive threads need to access consecutive memory addresses with a stride of four, eight or 16 byte. Single threads might skip the instruction without breaking the coalescing. On hardware of compute capability 1.2 and higher the restrictions are relieved a little. If the alignment restriction is violated the access is split into two accesses, one for each segment of memory accessed. In addition memory accesses no longer need to be ordered, they only have to be in the same segment, or segments if the alignment criteria is not met. As they require only one or two memory transactions instead of 32 for all threads in the warp coalesced memory accesses are an order of magnitude faster than uncoalesced ones.

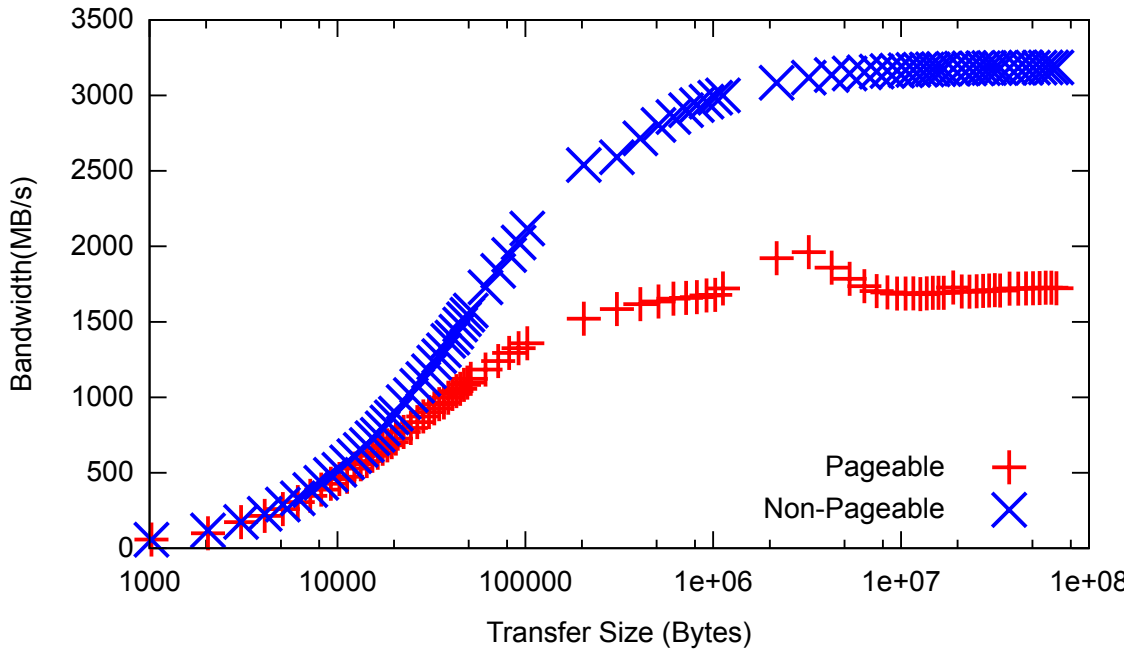


Figure 2.6: PCIe transfer speed achieved with a NVIDIA GeForce 8800 GTS 512

Modern graphics cards are attached to the PC via PCIe. All CUDA capable cards support PCIe-16 which supports a transfer speed of up to 4 GB/s. Figure 2.6 shows some real life measurements executed on the development system. These were executed using the bandwidth test application contained in the CUDA SDK. Variance of the measurements is less than 5 MB/s. Probably due to limitations of the chipset the 4 GB/s cannot be fully reached. Due to setup times small transfers are especially expensive. To achieve maximum speed page locked memory needs to be allocated via the NVIDIA driver. This allows the card to retrieve the data directly from the main memory via DMA. If the data on the CPU is not stored in page locked memory it will first be copied into a page locked buffer managed by the NVIDIA driver. This additional copy process will reduce transfer speed.

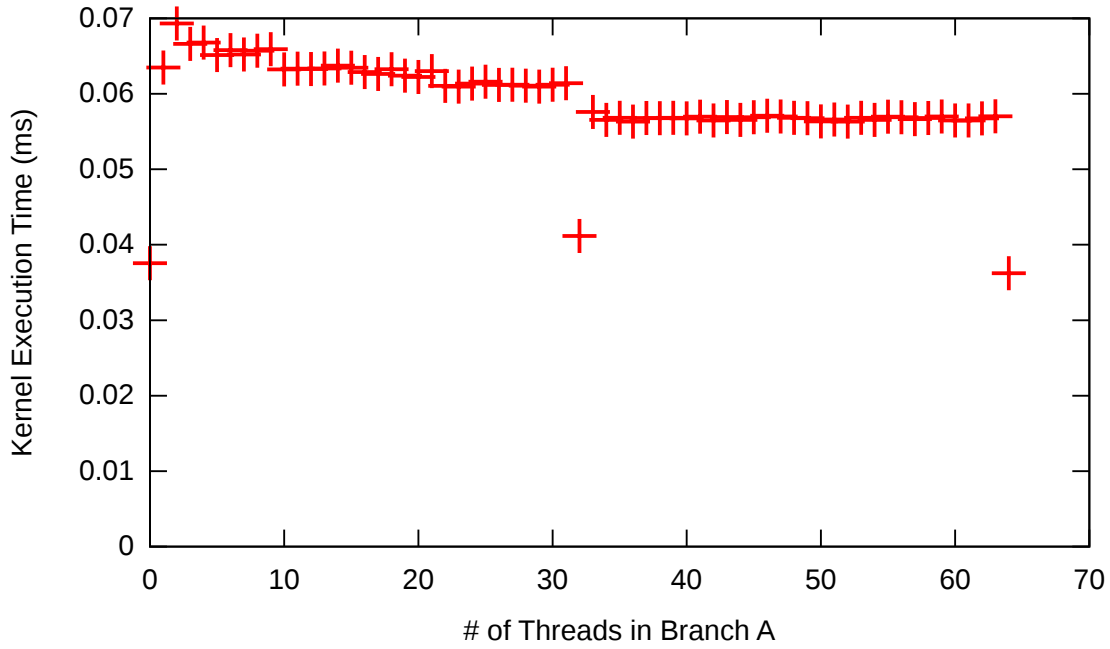


Figure 2.7: Execution speed dependent on number of thread taking a different code path

Due to the SIMT architecture with a shared instruction decoder, threads of the same warp executing different instructions need to be serialized. Figure 2.7 shows how execution time is effected if a specified number of threads takes execution path A, while the rest of the threads takes execution path B. If the criteria is such that whole warps take different execution paths no slowdown is observed as instruction decoding for different warps is independent.

For the measurements presented in figures 2.5 and 2.7 the kernels performed a few thousend iterations. This adds enough statistics too keep the variance of consecutive runs below 1 %. As this is smaller than the size of the dots and the effects shown are of a different scale no errors are given in the figures.

Chapter 3

Track Finding

3.1 Introduction to the Game of Life

The Game of Life, invented by the British mathematician John Horton Conway and first published as "Life" by Martin Gardner in 1970[12], is probably the most popular example of a cellular automaton[19].

A cellular automaton is a model which is discrete in space and time, consisting of a regular grid of cells with a finite number of cells. Each of those is in one of a finite number of states. The value of a cell at the time t is defined by a transition function based only on the values of the cell and its neighbourhood at time $t-1$. The neighbourhood consists of a number of cells defined by the cellular automaton. The transition of all cells in a cellular automaton from $t-1$ to t happens simultaneously.

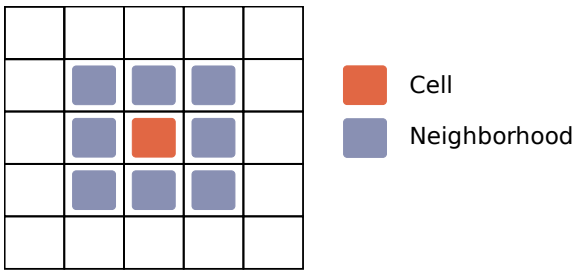


Figure 3.1: The neighbourhood of the red cell is given by its eight adjacent blue cells

In the Game of Life as published by Gardner the cells know only two states, dead and alive. They are square and the grid is two dimensional. The neighbourhood consists of the eight adjacent cells as shown in figure 3.1. The transition function is given by three simple rules.

1. Survival – An alive cell with two or three alive neighbours survives.
2. Birth – A dead cell with exactly three alive neighbours becomes alive.

3. Death – A cell with more than three alive neighbours dies from overpopulation. As does every cell with less than 2 neighbours from isolation.

Even though these rules are pretty simple the system can evolve in complex ways, with a possibility for the system to completely fade out. Often systems reach a stable final state or oscillate forever. Depending on the distributions of living cells when starting the cellular automaton it can take many, even thousends of, timesteps until such a state is reached.

There are many patterns that will frequently occur in the Game of Life, among them blocks, beehives, blinkers and spaceships, of which the most prominent one is the glider. Figure 3.2 shows how these evolve from simple patterns. While blocks and beehives are static objects, the blinker is a pattern that will oscillate with a period of two. The spaceships however will move through the grid until they reach other living cells which will perturbate their path.

The behaviour for a pattern of up to two cells without other living neighbours can be easily predicted, they will fade out in the next iteration. As will any line of cells in which no cell has more than two neighbours, loosing two cells in each iteration until completely fading out. However even from a random distribution one can see the mentioned patterns and more complicated ones appear. There are even patterns like the glider gun that will produce other patterns at a certain rate, without getting destructed themselves.

The game of life will often show global behaviour, e.g. as with the glider gun, while the evolution of each cell follows only local rules. The inherent parallelism, allowing every cell to be calculated independently, makes such systems suitable for GPGPU.

3.2 Game of Life and Track Finding

Detector components such as multiwire proportional chambers or silicon pixel detectors consist of discrete elements and can be viewed as a grid of cells. If a charged particle triggers a wire or a pixel in the detector that cell is alive.

In a very simple interpretation of a track finder one can look at the patterns in figure 3.2. Several characteristic patterns one would expect to find in a detector can be seen. Due to the starvation rule small sets of cells will starve out which can be interpretted as noise filtering, in the same way the overpopulation rule will kill cells where tracks could not be separated. Oscillating elements can be interpreted as a not converging track finding. Finally, patterns leading to beehives can be seen as track candidates. In addition cell birth can be seen as reconstruction of hits lost due to detector inefficiencies.

Track finders based on a modified variant of the Game of Life have been used in the ARES experiment at JINR in Russia and at the M$\bar{\text{M}}$ experiment at PSI in Switzerland[20]. In those experiments a cell consisted of a cluster of multiple hits

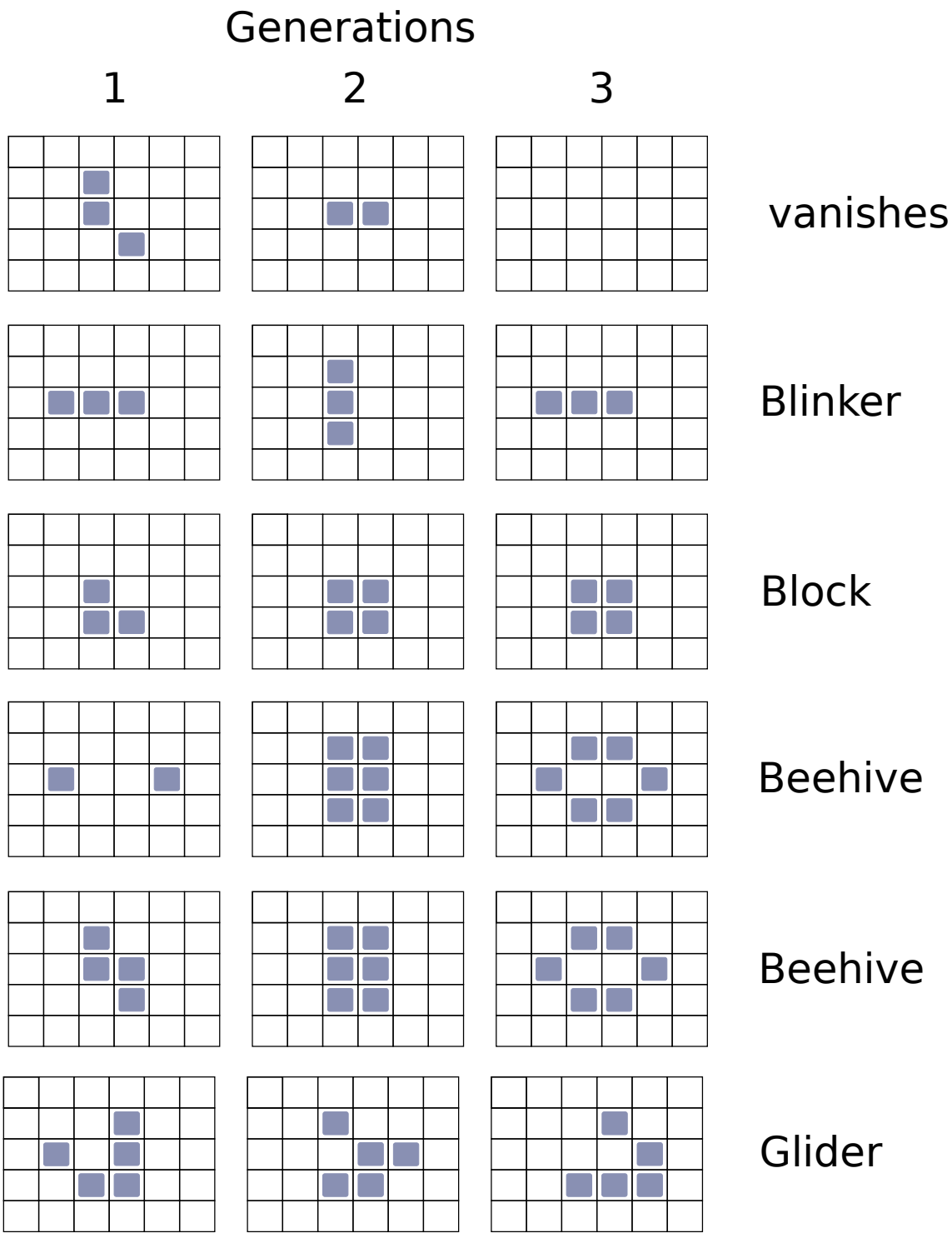


Figure 3.2: Evolution of simple patterns in Game of Life

in a multiwire proportional chamber. The structure of a cluster gives some indication on the direction of the particle trajectory and is used to create neighbourship relations in between the cells. The target and the scintillators around the multiwire proportional chamber are treated as immortal cells, providing hookup points for the physically interesting tracks. The reason for this is the starvation of unsupported lines as mentioned in section 3.1. Due to the immortal cells tracks originating in the target and leaving the experiment through the scintillators will survive, while those that do not originate in the target or do not leave the detector will vanish. This leaves only the physically interesting tracks for further analysis.

In these experiments the automaton is run until it reaches a stable state or becomes cyclic. At that point the automaton has grouped the hits into track candidates using only local information. That local information however cannot solve situations where tracks cross. As a huge part of the combinatorics has already been solved methods using a global analysis can be used to merge the track candidates. In the mentioned experiments a rotor model is used for this part of the reconstruction, merging track candidates based on their curvature and relative position.

3.3 Game of Life on CUDA

As an example for the many different cellular automatons that can be used for track finding the Game of Life was ported to NVIDIA CUDA. This also has the benefit that the solution can be compared to existing high performance SSE versions provided by CERN OpenLab and Intel[21].

When mapping the problem to the hardware two things need to be considered. The transition function needs to be parallelized and the storage format of the grid on the device needs to suit that implementation.

As a cells state is only based on the states of the previous generation the trivial unit of parallelism for the transition function in Game of Life is a cell. Therefore every cell can be mapped to one thread that calculates the transition function for this cell. This leads to a large number of threads, which is optimal for the GPU to hide the memory latencies. As the transition function in Game of Life is very simple register usage is not a problem and does not limit the amount of threads that can be running in parallel.

For the storage format of the data on the device there are two possibilities. Either the full grid is stored as a grid of alive and dead values in memory or a zero suppressed format, storing only the alive cells, could be chosen. The major advantage of the full grid is that it is trivial to calculate the positions at which cells need to be read and written in memory when calculating the transition function. An advantage of the zero suppressed method is that for low occupancies of the grid less memory is required, and all cells that aren't near any living cells will not need to be calculated. This enables larger grids to be calculated on the same hardware, at least for limited occupancies. In the extreme case of an empty grid a full grid

will still need to calculate all cells and require memory the size of the whole grid. In a zero suppressed representation the grid would virtually require no space and no calculations for the iteration.

For each cell the algorithm needs to read nine cells, the cell itself and its eight neighbours, to calculate the new state, which afterwards needs to be written. The number of arithmetic instructions is very low, consisting only of counting the eight neighbouring cells and evaluating the three rules. Chapter 2 pointed out that access to global memory is two orders of magnitude more expensive than calculations. Independent of the implementation the algorithmic density to hide these memory accesses is not reached.

As the memory accesses cannot be hidden properly memory access patterns are important for a fast execution of the algorithm. Therefore starting on a naive, straight forward implementation of the algorithm different optimizations have been developed.

In the naive approach every thread reads the cell and its neighbourhood from global memory. Consecutive threads work on consecutive cells in the grid to allow the memory controller to coalesce the reads. At start-up the grid is copied from CPU to GPU memory, then the iterations are executed without moving the data between the CPU and the GPU. Two buffers on the GPU are used which are swapped in between the iterations. In the end the data is transferred back to the CPU memory.

As mentioned in section 3.1 the calculation of each cell only requires local data. This locality is similar to the case of texture sampling in the rendering applications which the GPUs were originally built for. Therefore the algorithm was modified to read the cells through the texture unit, utilizing its cache which is optimized on streaming access with high locality, to improve throughput from the global memory.

When using zero suppression the position of a cell in memory is no longer trivial. A format was chosen where the grid is compressed on a row level, each row being stored at a defined offset from the previous. This offset specifies the number of alive cells in a row that the algorithm can handle. Setting this offset to the grid width will allow to handle arbitrary numbers of cells at the cost of memory size, the advantage of less calculations for a sparsly populated grid will be kept. It would also be possible to use a two phase approach. The storage offsets for the cells could be calculated using a prefix sum over the number of alive cells per row. This would remove the restriction of a maximum number of alive cells per row, while saving the additional memory as long as not all rows are completely filled. The prefix sum would however add additional computations and this variant has not been implemented.

Using zero suppression one thread is used to calculate each row. This reduces the number of threads used, but in a sparsly populated grid every thread should still only have to compute a few cells. The reason for using only one thread is that when using a block of threads on one row these would have to synchronize where to write their results, adding additional restrictions on the maximum number of cells in a certain area. Especially as shared memory, which the threads could use to communicate, is only 16 KiB. This corresponds to 4000 cells at maximum for each

row, if those are only represented by an integer.

To support cell birth the algorithm must not only calculate the transition of the cells which are already alive, but also all cells which are neighbors of such a cell in the row, as these might become alive. As the threads calculating the neighbouring rows cannot know where to write born cells, it must do the same for cells in this row that are neighbours of living cells in the neighbouring row. Splitting this into multiple threads would reduce the number of cells that could be stored in shared memory further, as one would now need $3 \times cellsPerRow$ threads to perform the calculation, each of which would need a position in shared memory to write to. This would reduce the maximum number of cells per row that can be supported to about 1000. Due to these restrictions the approach of calculation of a single row into threads was not further persued.

To minimize access to global memory the thread calculating a row crawls its own and the neighbouring rows in parallel. It relies on the cells to be stored sorted by their column index. At the first step it will read the column indices of the first living cells for all three rows. Then it checks which column index of the three current cells, one from each row, is the smallest. For this cell it will calculate the transition functions for the three cells potentially effected by the cell. Examples are shown in figure 3.3. Once these are calculated the thread will read the column index of the next living cell in this row. Then again it will proceed to checking for the smallest column index and iterate until all cells are processed. Due to this the thread never needs to move more than one element back and forward to check all the neighbours of the current cell for their state. To prevent calculating cells twice the algorithm keeps track of the last cell calculated. If the column index of the cell to be calculated is not larger than this index the calculation will be skipped.

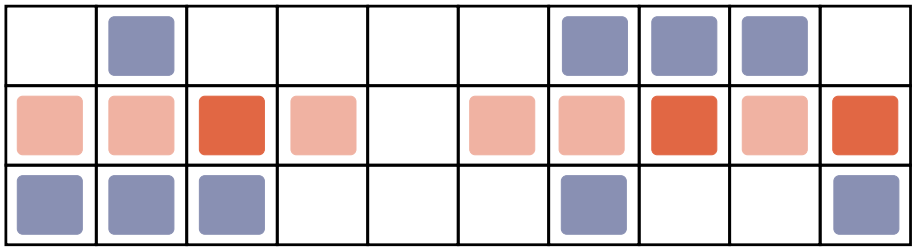


Figure 3.3: The red cells will be calculated by the thread crawling the three rows. Cells shaded in light red are dead and will be calculated because they are neighbours of living cells represented in blue or dark red.

As each thread now processes one row, neighbouring threads will often read and write cells in the same column. Therefore, to allow the memory controller to coalesce memory accesses, it makes sense to rotate the grid into a column-major orientation, as shown in figure 3.4. This way when all threads access e.g. the first element in the row the memory controller can coalesce this access, as memorywise those accesses will occur at consecutive memory addresses.

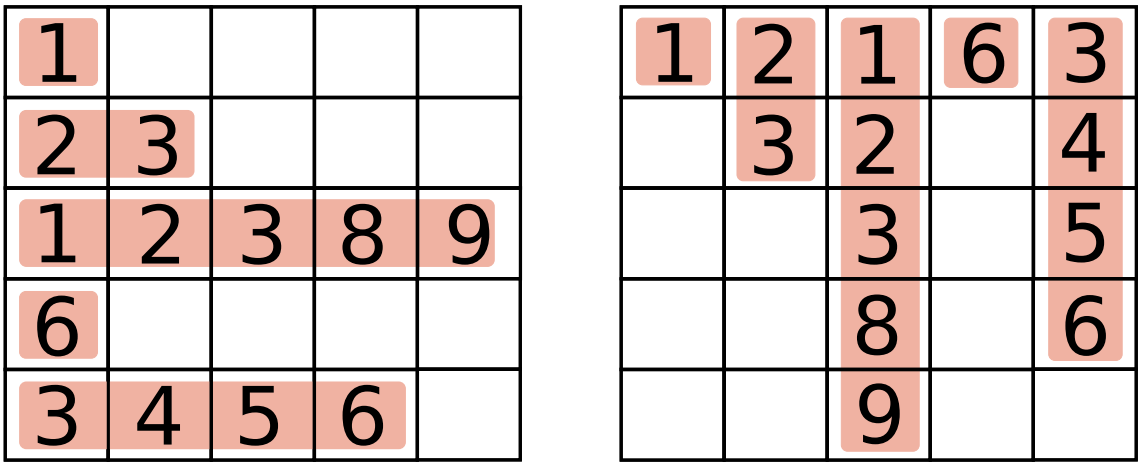


Figure 3.4: Row- and column-major orientation of the grid

Grid Size	Iterations	Total time (ms)	Kernel Time (ms)	PCIe time (ms)
64	2000	52.5	52.0	.5
512	1000	211.6	205.5	5.1
4096	20	582.9	218.7	364.2

Table 3.1: Execution times and PCIe transfer times for texture based version of Game of Life on a GTX 280

As in the full grid case, the texture unit can be used to cache the reading of the cells which happens with a high locality. Some additional accesses to global memory can be avoided if each thread reads the number of alive cells in its row into shared memory. This enables its neighbours to read this value from shared memory instead of main memory, saving two reading access to global memory. Of course special care needs to be taken at the borders of thread blocks.

Figures 3.5 and 3.6 show execution times for the different implementations on a NVIDIA 8800 GTS 512 and a NVIDIA GTX 280 in comparison to a CPU based implementation running on one core of an Intel Core2 Q6600 running at 2. 40 GHz. In those graphs occupancy means the occupancy at the start of the game. A description of the system used to perform the measurements performed in this chapter can be found in appendix A.

For the very small grid of only 64×64 cells the overhead for the GPU is very large and the CPU is as faster than most GPU invocations. Only the full grid texture based version and the zero compressed version manage to compete.

For the larger grids the texture based implementation calculating the full grid is a factor six to 20 faster than the SSE based CPU implementation. This factor largely depends on the number of iterations. As shown by table 3.1, the PCIe transfer only has a small impact on the overall runtime for the 1000 iterations calculated for the

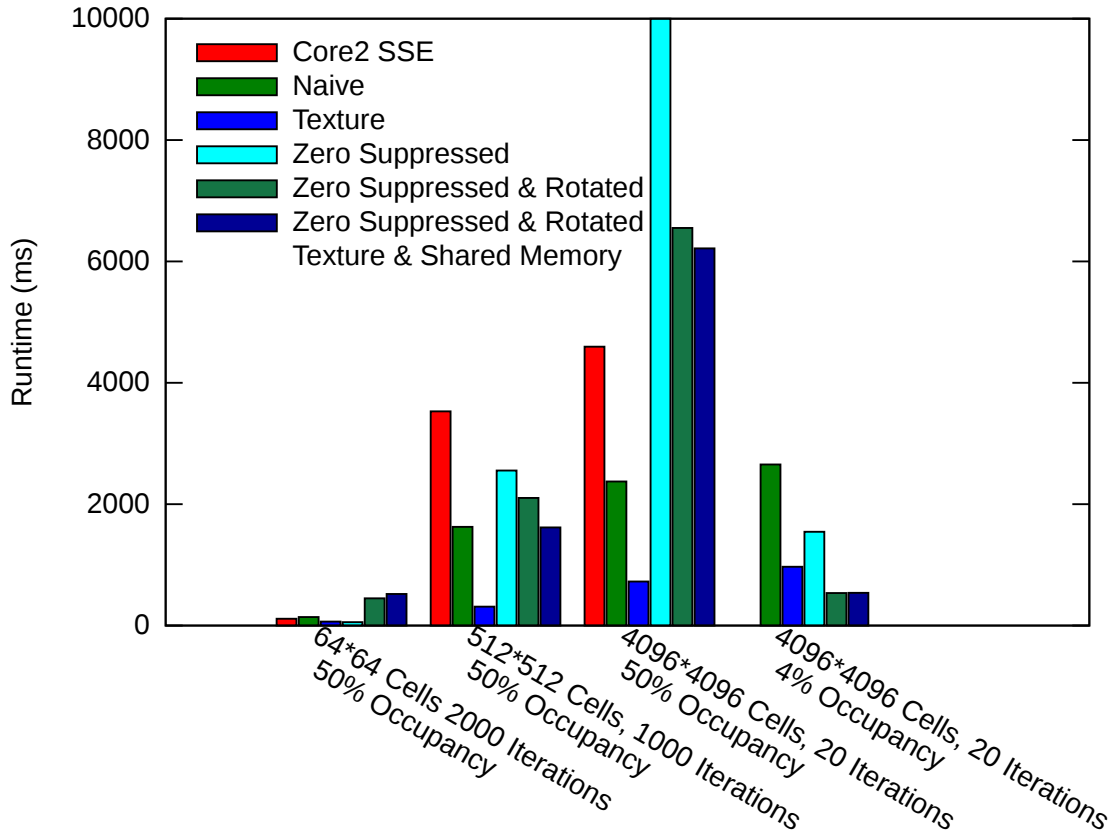


Figure 3.5: Execution times for Game of Life on a NVIDIA 8800 GTS 512

512×512 grid. It increases the runtime for the twenty iterations on the 4096×4096 grid, from about 220 ms needed for the iterations, to about 580 ms including the transfer to and from the card.

In these two graphs the major development regarding the memory controller in between the G92 chip of the 8800 GTS 512 and the GT200 in the GTX280 can be seen. This development is the reason for the different coalescing rules mentioned in chapter 2. The G92 chip only implements compute capability 1.1, while the GT200 chip implements compute capability 1.3. On the 8800 GTS 512 the texture based implementation is about three to five times faster than the naive version. On the GTX280 the naive implementation is as fast as the texture based version. The reason for this can been seen in the example of loading the left neighbour, left meaning having a column index one less than the calculated cell. This read misses the alignment requirements for the coalesced read on the G92 by one byte, causing an own memory transaction for each thread. The GT200 will group this access into two reads, one for the one element which is left of the alignment boundary and another load for the rest of the elements to the right of the alignment boundary.

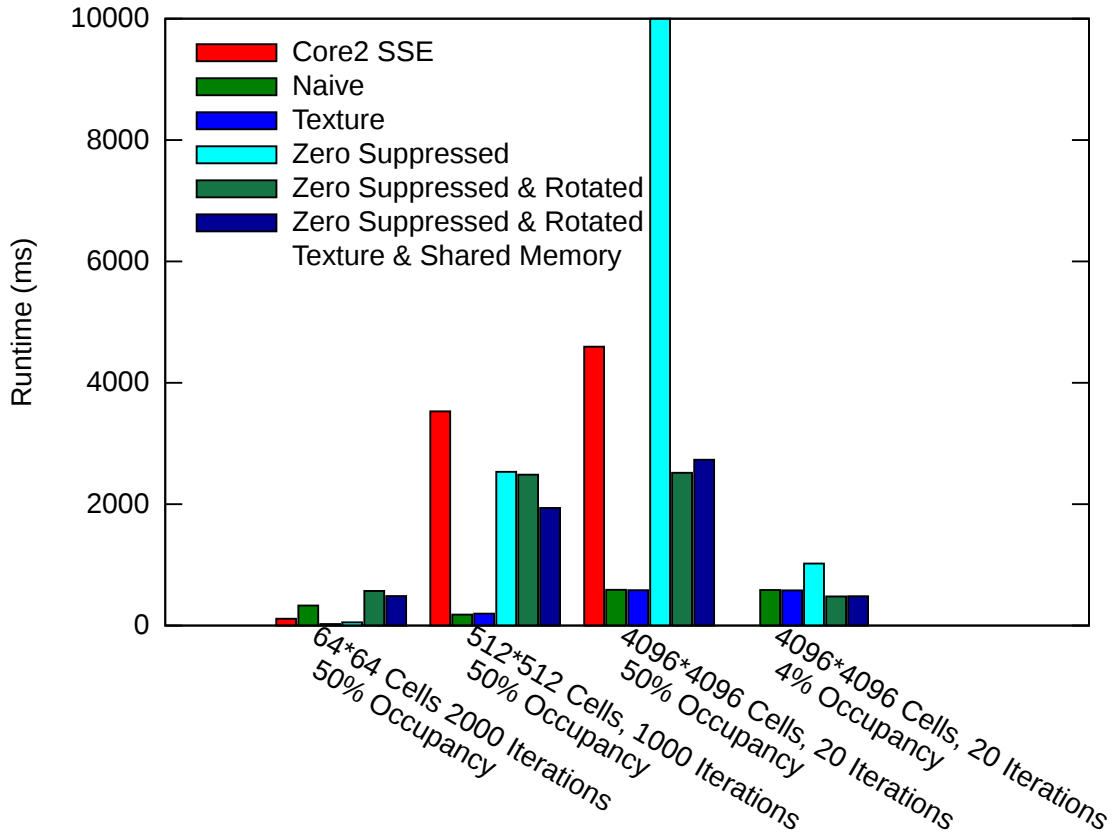


Figure 3.6: Execution times for Game of Life on a NVIDIA GTX 280

The same effect can be seen for the zero suppressed implementations.

For the large matrices the effect of rotating the grid orientation to allow for coalesced accesses can be seen on both cards, with the execution time for the row-major zero suppressed implementation running of the scale for the 4096×4096 grid when the full grid needs to be calculated, and still being a factor of two slower for the grid starting at 5 % occupancy.

Measuring the 4096×4096 grid with multiple starting occupancies, as done in figure 3.7, shows that the zero suppressed implementation is faster than the full grid implementation for starting occupancies of about 5 % and less. As expected the runtime of the full grid implementation does not depend on the number of living cells. The runtime of the zero suppressed version depends on it. For larger occupancies the full grid implementation can profit from its higher level of parallelism, showing how important fine grained parallelism is for high performance implementations on the GPU. For low occupancies the zero suppressed implementation can profit from the lower number of memory accesses required, showing how important efficient memory usage is for high performance implementations.

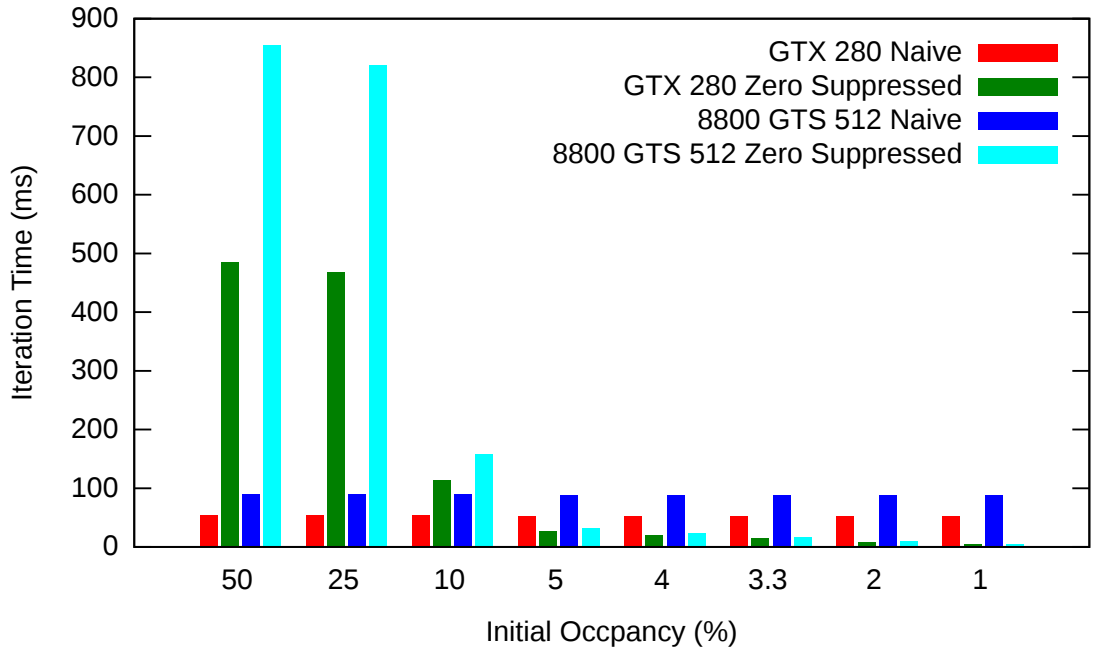


Figure 3.7: Iteration times for multiple starting occupancies on a 4096×4096 grid

The different performance characteristics of the full grid and the zero suppressed implementation show that the version to be preferred depends on the specific use case. For every application a specific ratio between parallelism and optimized memory usage needs to be found to achieve the best performance.

The game of life can be expanded to three dimensions. Such an extension has been implemented and shows the same effects as the discussed two dimensional version.

3.4 Game of Life on AMD Brook+

AMD's solution to GPGPU is called AMD Stream[22]. It is, as NVIDIA CUDA, a two layer approach. AMD Stream consists of the Compute Abstraction Layer on the assembler level and an improved variant of BrookGPU[23], called Brook+. Brook+ is a language for streaming computing. Programming happens on a higher level than in CUDA. Instead of specifying work done by a thread, it is used to define what calculations are required to compute a specific output element from the input. There is no influence on the details of data storage or even data transfer between the CPU and the GPU. There are also tight restrictions on how a data may be accessed. Scattered writes are only supported for special cases.

The AMD hardware is build different than the NVIDIA hardware. It groups its floating point processors in groups of four, together with one special function unit. SIMD style formulation of the problem can be used to optimally use this kind

of hardware layout. There is no special double precision unit. Double precision calculations are performed by combining the four single precision units.

The naive implementation of the Game of Life conforms to the restrictions given by Brook+. A port of this naive implementation was performed to get an idea of the performance of the AMD solution. Due to the limited availability of an AMD graphics card, no optimizations were done. Copying the data between the CPU and the GPU for each iteration it achieved similar execution times to what an equivalent CUDA version would. Execution time for a single iteration of a $512 * 512$ matrix on an AMD Firestream 9250 was 1.7 ms versus 6.3 ms on the NVIDIA GTX 280. The measurement of the CUDA version includes the memory allocations, which are expensive as shown in section 5.2. For the AMD version it is not possible to control whether the allocation is measured or not. Therefore this is an unfair comparison. It does however show that both, AMD and NVIDIA devices, achieve a performance of the same order of magnitude for a naive implementation.

3.5 Cellular Automaton Track Finder for Alice

The High Level Trigger of the Alice[24] experiment at CERN uses a tracker based on the cellular automaton for track finding in the time projection chamber (TPC).

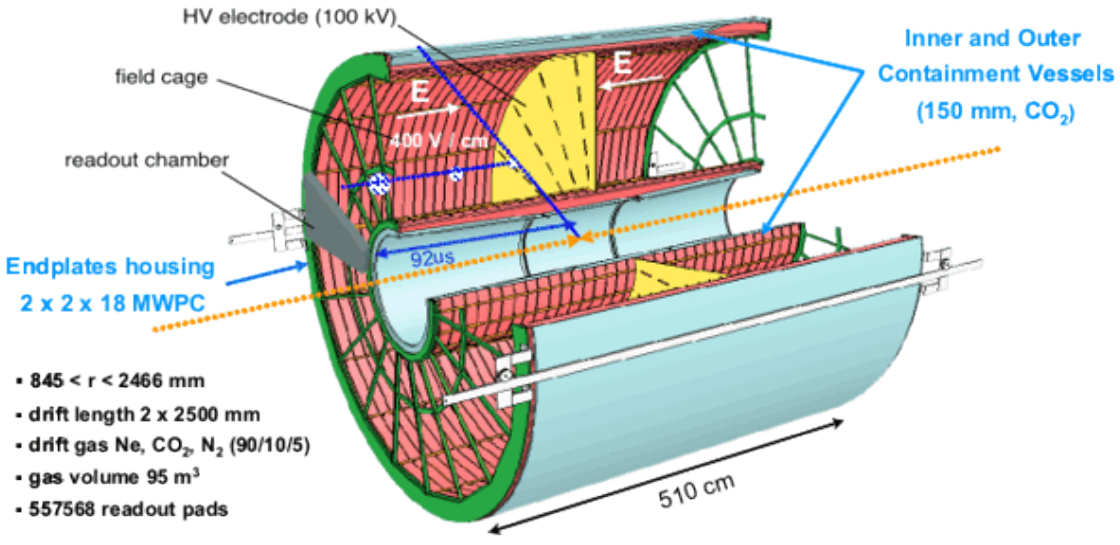


Figure 3.8: Layout of the TPC[25]

The Alice experiment is focused on investigating lead-lead collisions. In those collisions a quark-gluon-plasma is expected to be formed. A state of matter only found at temperatures of 10^{25} K and extremely high densities. The last time these

Event type	CPU Execution Time (ms)	GPU Execution time (ms)
Proton-Proton	50	106
Lead-Lead	7780	8075

Table 3.2: Execution times for finding tracks in all TPC sectors using a first CUDA port

conditions existed in nature was when the universe was only millionth of seconds old.

The TPC[25] is the main tracking detector of Alice. Its layout is displayed in figure 3.8. It is filled with a mixture of Ne, CO_2 and N_2 which can easily be ionized. Charged particles that fly through the TPC leave a trail of ionized atoms. A high voltage applied between the electrode in the middle plane and the field cage in the end plates provides a strong electric field. Due to the resulting electric field the ionized atoms drift towards the end plates. The time projection chamber is divided into 36 segments. These segments are defined by the readout chambers at the end plane of the barrel and the electrode in the middle plane. The readout chambers are equipped with 159 pad rows that register the ionized atoms in two dimension. One from the location of the hit on the row and one from the timing, as the drift time in the chamber depends on the distance from the row where the particle ionized the gas. The detecting row itself provides the third dimension, so that the hits are registered in three dimensions.

The track finder always operates on one of the segments and performs the track finding in multiple steps. The steps performed using the cellular automaton are visualized in 3.9.

1. Hits within a row are grouped into cells.

2. Cells are linked with the nearest cell in the next and in the previous row. A limit to the distance of the cells is applied to only match cells belonging to the same track. If a cell could be matched with two cells in a row it will not be linked. To compensate for inefficiencies cells are also searched two rows away if none is found at a distance of one row.

3. Following the links created in the previous step cells are grouped into track segments.

4. Parameters for the track segments are fitted.

5. Track segments are matched into tracks. This merges the segments disconnected by skipping cells that would have had more than one connection. Such cells appear in track crossings or in circular tracks whose radius is smaller than that of the TPC.

Event type	CPU Execution Time (ms)	GPU Execution time (ms)
Proton-Proton	4.5	30
Lead-Lead	1640	156

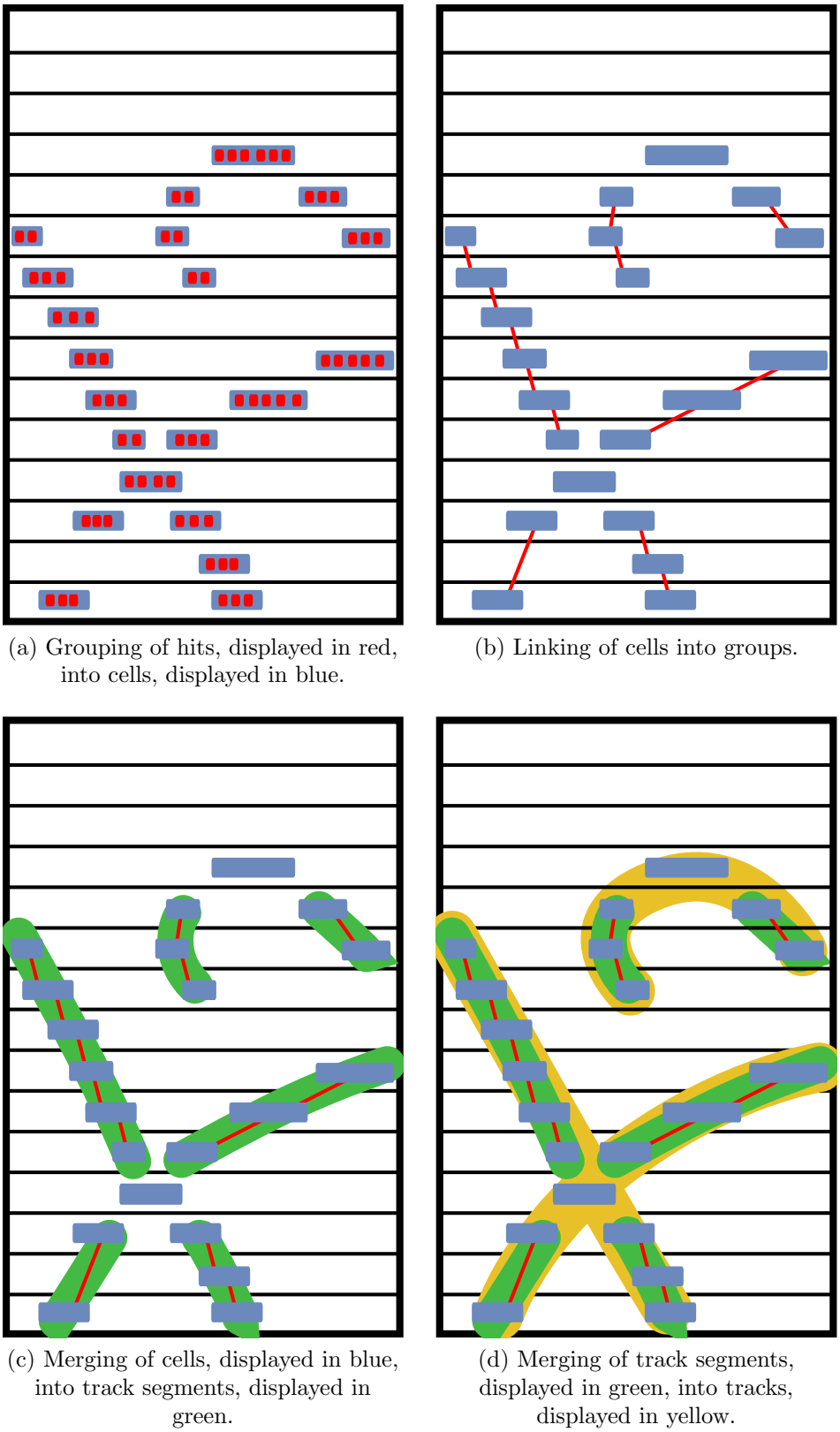
Table 3.3: Execution times for finding tracks in all TPC sectors using the CUDA port by S. Gorbunov

This track finder has prototypically been ported to CUDA. Hit creation was parallelized on a row level, cell linking on a cell level and the rest of the steps on a track level. The port showed that the calculations can be performed on the GPU and that CUDA can be utilized from AliRoot.[1] Performance measurements of this version are presented in table 3.2. The performance suffered especially from low parallelism, 159 threads are not enough to utilize all 16 multiprocessors of a NVIDIA 8800 GTS 512, as threads are scheduled in warps of 32 threads. In addition memory access was very suboptimal, the data was stored in a format consisting of complex structures which were stored at locations dependent on other elements in the row and on the elements in the below rows. This limited potential parallelization and prevented coalesced reads. Also the size of the data structures prevented usage of shared memory to work around this.

Due to these problems optimization of the port would have required modification of the data formats. As at that time the algorithm was still undergoing vivid development, this was put back in favour of the other topics discussed in this thesis.

The results of this research have later found their way into a port to CUDA by the original author of the tracker. S. Gorbunov modified the data format to better suit the GPU and allow for more parallelism. Table 3.3 shows the performance of that port for simulated events. Due to the higher energies mentioned in chapter 1 the lead-lead event produces more particles. This increases the costs of track finding. For those events the GPU can profit from parallelism and is about an order of magnitude faster than the CPU. For the smaller number of tracks produced in proton-proton events the CPU is faster than the GPU. Here the parallelism is unable to outweigh the overhead of offloading to the GPU. As it can also be seen from comparing tables 3.2 and 3.3 the CPU version, which is now based on the same code as the CUDA version, has also improved.

[1] Some notes on how to use CUDA from AliRoot are presented in appendix C.



(a) Grouping of hits, displayed in red, into cells, displayed in blue.

(b) Linking of cells into groups.

(c) Merging of cells, displayed in blue, into track segments, displayed in green.

(d) Merging of track segments, displayed in green, into tracks, displayed in yellow.

Figure 3.9: The steps performed by the cellular automaton based tracker

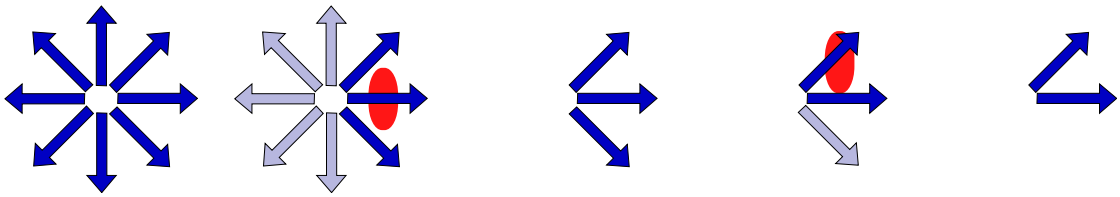
Chapter 4

The Kalman Filter

4.1 Introduction to the Kalman Filter

The Kalman filter[26] is an iterative method to find the optimum estimation $\vec{a}$ and the corresponding covariance matrix C of an unknown vector $\vec{x}$ based on a series of measurements $\vec{m}_i$,$i \in 0 \cdots n$ of $\vec{x}$. In the example of track fitting the vector $\vec{x}$ to be estimated is the trajectory of a particle. The measurements are given by the location of the hits of the particle in the detector.

Kalman filters are applied in many fields. The next sections show the usage of the Kalman filter in particle tracking. Other applications include robot localization[27], weather models[28], noise suppression[29] and trajectory estimation for space vehicles and aircraft[30].



(a) Initial prediction

(b) Update with 1st hit

(c) Estimation becomes prediction

(d) Update with 2nd hit

(e) Estimation after two hits

Figure 4.1: Sketch of the Kalman filter method

The Kalman filter uses a predict and update method which processes the measurements sequentially. This is different to methods like least squares or maximum likelihood that process all measurements at once. For each measurement the Kalman filter uses its current best estimation as a prediction of the vector. It then uses the measurement to update this estimation. Figure **??** sketches this process.

1. The Kalman filter starts with an initial approximation for $\vec{a}_0$ and C_0. Usually this means initializing the covariance matrix C_0 with large positive numbers.

2. Based on the last estimation $\vec{a}_{i-1}$, C_{i-1} the value $\vec{\hat{a}}_i$ and its covariance matrix $\hat{C}_i$ at the measurement i are predicted.

$$\vec{\hat{a}}_n = A\vec{a}_{n-1}$$

$$\hat{C}_n = AC_{n-1}A^T + Q$$

A is the linear operator modeling the propagation in between the measurements. Q models the noise in the propagation from $i-1$ to i, e.g. through scattering.

3. The measurement $\vec{m}_i = H_i\vec{x}_i$ is used to update the estimation. H_i is a linear operator modeling the dependency of the measurement on the real value. E.g. $\vec{m}_i$ might be a location while $\vec{x}_i$ contains location and momentum. The error of the measurement is described by the covariance matrix V_i. K_i is the so called Kalman gain. The Kalman gain is used as a weight for the new measurement in comparison to the previous estimation.

$$K_i = \hat{C}_i H_i^T \left(V_i + H_i \hat{C}_i H_i^T\right)$$

$$a_i = \vec{\hat{a}}_i + K_i \left(\vec{m}_i - H_i \vec{\hat{a}}\right)$$

$$C_i = \hat{C}_i - K_i H_i \hat{C}_i$$

4. If there are more measurements the algorithm continues at the second step.

Once all measurements have been used $\vec{a}_i$ is the optimum estimation for $\vec{x}$ and C_i is the error covariance matrix.

While processing one measurement the Kalman filter does not need any knowledge about the other measurements. This keeps the memory footprint of the Kalman filter leaner than of global fitting methods. In addition, measurements can be added to the estimation without having to reprocess any of the already processed measurements.

Coming back to the example of track fitting an initial estimation could be that the trajectory is somewhere inside the experiment, crossing the collision point, with an upper limit to the speed of the particle and an unknown direction of the trajectory. The first measurement will improve the estimation on the location of the trajectory and give a first estimation on the direction of the particle. Further measurements will improve this guess and add information about the curvature of the trajectory.

4.2 SIMDizised Kalman Filter in CBM

In 2007 S. Gorbunov et al. parallelized the Kalman filter based track fitting algorithm for the CBM experiment and ported it to SSE and the Cell SPE[31]. A

speedup of 10000 measured against the initial version was achieved. As GPUs are highly parallel and provide a high single precision performance porting the algorithm to the GPU was the logical next step.

The CBM experiment at the Facility for Antiproton and Ion Research in Darmstadt is a dedicated fixed target heavy ion experiment. The SIMDizised Kalman Filter was created to help solve the problem of tracking and fitting 500 and more tracks per collision. It has been embedded into the cellular automaton based track finder to be used for track reconstruction in CBM.

The SIMDizised Kalman filter solves two major problems regarding the creation of a fast Kalman filter, one of which is specific to the CBM experiment and one that is a generic problem of numerical stability.

The CBM experiment uses a strong inhomogeneous magnetic field, which has to be taken into account when tracing particles through the detector. The SIMDizised Kalman filter uses a fourth order polynomial to approximate the magnetic field at the silicon pixel stations. The silicon pixel stations are used to detect the particles. In between the stations the field is approximated by a parabola whose coefficients are based on the three closest hits in the current track. This avoids having to use a 70 MB large map of the magnetic field, which on any architecture would be slow to access.

To avoid costly double precision calculations the Kalman filter has been tuned to work with single precision arithmetic only. S. Gorbunov et al. observed numerical instability to only show from the first measurement, where errors of the initial parameters can differ from the errors of the measurements by orders of magnitudes. This does not happen for later steps due to the iterative predict and filter approach of the Kalman filter that gets a better approximation of the track at every step of the iteration. It has been found that neglecting errors of initial parameters that are more than a factor of 4 larger than the mean measurement error makes the algorithm numerically stable and accurate in single precision. This can be achieved without extra calculations.

4.3 Porting the Kalman Filter to the GPU

In the original version operator overloading was used to enable the same implementation of the algorithm to be used on both architectures. This way double maintenance of the code was avoided and the performance of the different architectures could easily be compared. This port keeps to this principle, allowing easy maintenance and comparison of the algorithm for all platforms.

The SIMD execution model, implemented by SSE and Cell, and the SIMT execution model of the GPU are similar in their realization in hardware. There are however important differences in the programming model. In SIMD the parallelization happens below the thread level while in SIMT it happens on the thread level, more like in OpenMP[16] or MPI[32]. The following code example shows how exe-

cution works in SSE. On the thread level an object is created and filled with some data. Afterwards operations are applied to all elements in the vector, whose width needs to be equal to that of the vector register. One thread uses multiple ALUs to process all elements of the vector in one instruction cycle.

```
Vec_t vec1 = { src[0], src[1], src[2], src[3] }
vec1 *= 2
```

In the SIMT model each thread is only attached to one ALU. A thread could only process multiple elements sequentially. To feed the hardware we need to run many threads, each instantiating a scalar object and operating on that. To achieve the same effect as in the previous code on the GPU we would launch 4 threads executing the following code. ThreadId represents an index ranging from 0 to 3 provided by the hardware.

```
Scal_t scal1 = src[ ThreadId ]
scal1 *= 2;
```

An alternative would be to have the GPU directly operate on the global memory, as shown in the following code example. This however would prevent taking advantage of the large register file available on the GPU and prevents the compiler from applying optimizations to the code by command fusion or reordering of commands. In addition, one should keep in mind that memory on the GPU, as described in chapter 2, is uncached and therefore more than a factor of 100 more expensive to access than registers or shared memory.

```
dest[ ThreadId ] = 2 * src[ ThreadId ]
```

Due to this expensive access to global memory the first approach shown was used. The SIMDization was removed by defining the type `Fvec_t`, used to parametrize the common implementation of the algorithm, to `float`. To replace SSE parallelization on thread level was introduced. On the CPU or the SPE one thread sequentially processed chunks of 4 tracks in a SIMD fashion. On the GPU an own thread is started for each track. All this threads are potentially processed in parallel, with the details of the scheduling being up to the GPU.

```
class Fvec_t {
public: float val;
}
```

Before using `float` as a scalar datatype an attempt was made to use vectors with only one element. This had the advantage that operators could be overloaded without problems, which is not possible for the built in type `float`. As it can be seen in table 4.1 the compiler was unable to properly handle this case and uses huge amounts of local memory. The execution time, shown in table 4.2, is larger than that of the SIMDified CPU implementation, which runs at about $0.7\ \mu$s per track. Using `float` as the basic datatype removed this problem.

```
typedef Fvec_t float;
```

As shown by table 4.1 register usage of the algorithm is high through all variants of the implementation. This is a problem as it limits the number of threads that can be in flight on one multiprocessor of the GPU. A GT200 based GPU like the GTX 280 has 16384 registers per multiprocessor and can handle up to 1024 active threads per multiprocessor[5]. Using 91 registers per thread however limits the maximum number of concurrently active threads to 160. This makes it difficult to hide the high latencies in access to global memory and therefore makes it very important to optimally access that memory.

The parameters for the magnetic fields are stored in seven structures which are common to all threads. Due to the nature of the algorithm all threads access the same structure at the same time. Those are therefore perfect candidates for the use of the constant memory, which is cached and has a broadcast mechanism.

The SIMD implementation uses bitmasks stored in the float type `Fvec_t` to implement conditional assignments. Such a construct however is not optimal on the GPU. It is possible to tell the compiler to keep generated PTX assembler code via the parameter `--keep` and make it annotate this code with the original source via the parameter `--opencc-options="-LIST:source=on"`. Scanning the generated PTX code for the local memory access instructions `ld.local` and `ld.store` reveals this masking to be the cause for the remaining local memory usage. Using an own type, that on the GPU is implemented by a scalar `bool` variable, enables the compiler to use the GPU's mechanism of conditional execution based on special predicate registers. As it can be seen in table 4.1 the code now uses four more registers but 56 bytes, worth 14 registers, less local memory. Table 4.2 however shows that this only has a little effect on performance.

Another large speedup for the application can be achieved by minimizing accesses to global memory. Reading all tracks into shared memory and writing the tracks back from shared memory to global memory when ending the kernel gives a first boost. This boost can be increased by making sure that those accesses happen in a way that can be coalesced by the memory controller. Therefore the array of tracks is aliased to an array of integers of the same number of bytes. That array is then transferred between global and shared memory in a way that every i-th thread transfers the i-th integer.

```
for( unsigned int i = ThreadId;
     i < lengthof( g_tracksAsUInt );
     i += NumberOfThreads ) {

  s_uint[ i ] = g_tracksAsUInt[ i ];
}
```

This causes the memory controller to use the minimum number of 128-byte transfers required to fullfill the copy. As a result the execution time halves as it

can be seen in table 4.2. Sadly this further limits the number of concurrent threads active on one multiprocessor. Each thread needs 380 Bytes to store its track in shared memory. As there are only 16 KiB of shared memory per Multiprocessor only 32 threads can run concurrently.

Version	Registers	Local (Bytes)	Constant (Bytes)
One element vectors	88	9472 + 9468	0
`float` as datatype	93	56 + 52	0
Fields in constant memory	84	56 + 52	1560
Flags instead of masks	88	0	1560
Shared memory for tracks	83	0	1560
Coalesce memory access	91	0	1560

Table 4.1: Kalman Filter Memory Usage

Version	Kernel (μs)	Kernel + PCIe Transfer (μs)
One element vectors	6.5	-
`float` as datatype	.26	-
Fields in constant memory	.22	-
Flags instead of masks	.21	-
Shared memory for tracks	.14	-
Coalesce memory access	.10	.57
Pagelock host memory	.10	.33

Table 4.2: Kalman Filter Performance

After the described optimization of the code the kernel was about seven times faster than the SIMD code on the CPU used for comparison. As shown in table 4.2 the PCIe transfer of the hits took nearly five times that long. Page locking the host memory gained some performance. It allowed the GPU to retrieve the data directly from the staging area via direct memory access instead of getting it piped through a buffer in the graphics driver. This sped up the PCIe transfer by the expected factor of two, reducing execution time by 42 %.

For a set of 20000 tracks the application needs to transfer 14. 5 GiB bytes of data over PCIe, the size of one track being 380 Bytes and the size of a structure containing magnetic field data being 156 Bytes, where the tracks need to be transferred twice.

$$N_{transferred} = 20000 \times 380 \text{ Bytes} \times 2 + 7 \times 156 \text{ Bytes} = 15.2 \text{ MB}$$

The total fit time for such a dataset is 6. 73 ms, therefore the transfer rate is 2,26 GB/s. PCIe can transfer 250 MB/s on one lane[33] and the GPUs are connected via 16 lanes. The theoretical maximum transfer rate is therefore 4 GB/s.

This means the application reaches 57 % of the theoretical transfer limit while applying the Kalman filter. To improve performance further when streaming the data through the GPU it could be tried to split the track data into input and output structures, reducing the size of the data to be transferred. This might also enable more threads to be run concurrently as the size of shared memory required per thread might be reduced.

For the measurements performed in this chapter the system described in appendix A was used. An interesting detail that can be seen in figure **??** is that the one year old GeForce 8800 GTS 512, though only 60 % slower in the kernel execution takes nearly twice as long for the PCIe transfer. Contrary to what could be assumed from the documentation that card seems to share the lanes of its PCIe-x16 slot with a second card in a neighbouring PCIe-x8 slot, which limits the theoretical peak transfer rate to 2 GB/s. This bandwith saturates to $15.2\ \text{MB} \div 11.9\ \text{ms} \div 2\ \text{GB/s} = 63\ \%$.

PCIe transfer speed is only an issue when running the fitter standalone. If it is combined with a GPU based track finder, the track data will already be on the GPU, therefore it can usually be neglected for the comparisons. In that case it actually has the benefit of not requiring the data to be transferred back to the CPU for the fit to take place.

Comparing execution times as shown in figure **??** shows that the GPU version of the tracker is approximately 7 times faster than the GPU version running on one core of a modern Intel CPU. A one year old GeForce 8800 GTS 512 is still about 4 times faster than the CPU core running the SSE code.

Device	Plain fit time for 1 track (μs)	Plain fit time per track (μs)
8800 GTS 512	37	0.077
GTX 280	44	0.046

Table 4.3: Computation latency and throughput for the Kalman fit

Table 4.3 gives an interesting visualization on the development of the GPU architectures. For this table plain fit computation time was measured excluding the time for loading the track into shared memory. The 8800 GTS 512, equipped with a clock rate of 1.6 GHz, is faster than the GTX 280, which is clocked at 1.3 GHz, when measuring the latency for one track fitted using one execution unit. Measuring this plain fit execution time with enough tracks to utilize all execution units the GTX 280, equipped with 30 multiprocessors, is faster than the 8800 GTS 512 which is only equipped with 16 of those. The ratios between the throughput and the latency map to the number of threads executed by the devices in parallel. The higher throughput at lower clock speed on the newer card shows how important a good parallelization is to take advantage of architectures that increase their performance by the addition of execution units instead of higher clock rates. It also shows how well the implementation of the algorithm suits this requirement and allows to expect

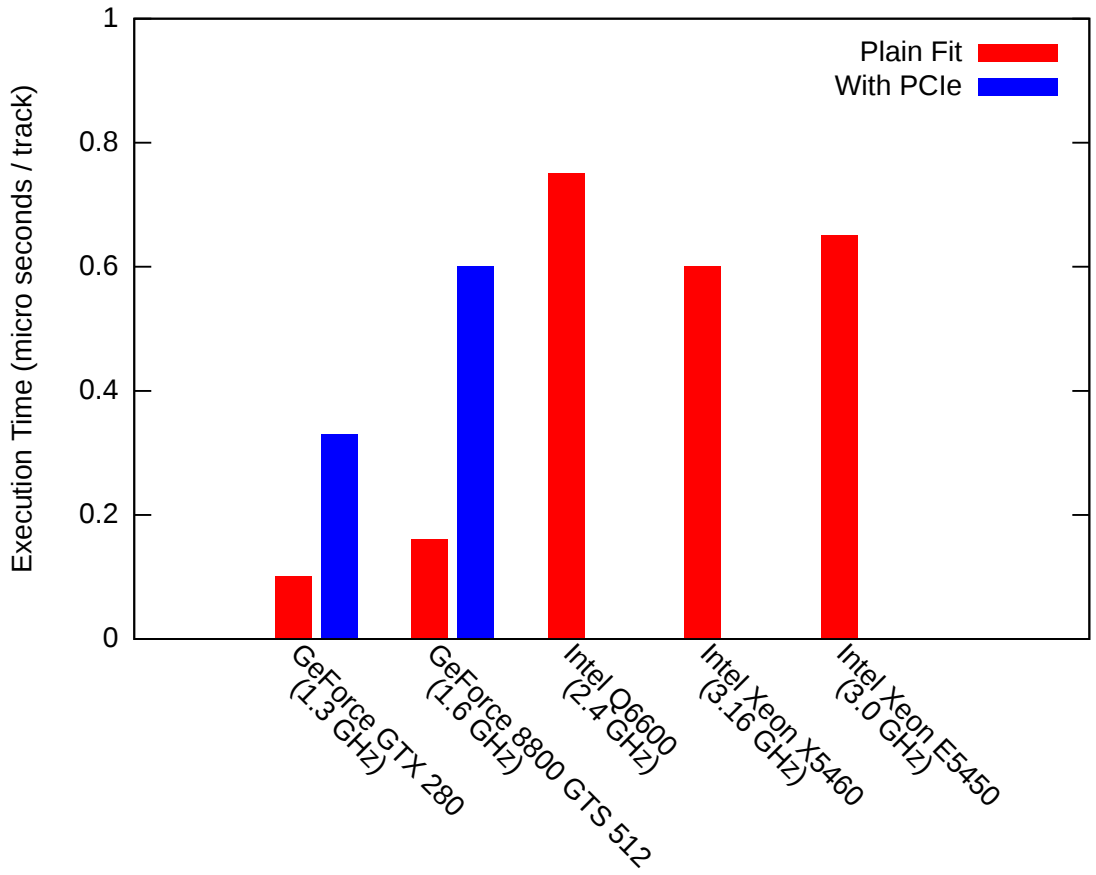


Figure 4.2: Execution times of the Kalman filter on different platforms

improved performance with future hardware which will probably contain even more execution units.

Chapter 5

Linpack

5.1 About Linpack

The Linpack benchmark[13] measures the performance of a system in solving a dense system of linear equations. The track finding and fitting algorithms discussed do not rely on this operation, therefore the Linpack does not necessarily say something about the performance of the system with respect to those algorithms and cannot be used for these applications.

However the Linpack benchmark is used to create the Top500[34] list of the fastest computers and is therefore used to compare performance of different systems. Every system build to run the described algorithms will be compared with other systems using this algorithm as it is the only algorithm that is executed on every system. For proper representation of such systems an optimized version of the benchmark is required. In addition, as the Linpack, opposed to the other algorithms discussed, requires cooperation in between the used processing units it allows to learn about the implementation of algorithms with cooperation in between the graphics devices.

The algorithm implemented by the Linpack solves a linear system of linear equations of order N using LU factorization with row partial pivoting. The linear system is represented by the matrix $\left[A\vec{b}\right]$. The $N \times N$ matrix A represents the coefficients of the system of linear equations, b the constant terms. The matrix $\left[A\vec{b}\right]$ is split into squares with an edge length of NB, so called blocks. Those blocks are cyclically distributed over the processes in the cluster. The process layout can be configured to specify how the blocks are mapped to the nodes. In a $1 \times n$ layout all blocks in a column are processed by the same process. In a $2 \times m$ layout a process will process every mth block in every second column.

In the beginning every process generates the values of its local blocks. Afterwards the Linpack performs the LU factorization. At each step it factorizes a panel of NB columns, which includes finding the pivot element and communicating the row swap. Following the factorization the factorized panel is broadcast to the other processes

for the trailing matrix to be updated accordingly. Finally, once the LU factorization is done, backwards substitution is used to obtain the solution $\vec{x}$. To verify the solution the matrix $[A\vec{b}]$ is regenerated and three residuals are calculated:

$$\| A\vec{x} - \vec{b} \|_\infty \div (eps \times \| A \|_1 \times N)$$

$$\| A\vec{x} - \vec{b} \|_\infty \div (eps \times \| A \|_1 \times \| \vec{x} \|_1)$$

$$\| A\vec{x} - \vec{b} \|_\infty \div (eps \times \| A \|_\infty \times \| \vec{x} \|_1)$$

eps represents the relative machine precision. The solution is considered numerically correct if all these residuals are less then a threshold value of the order of 1.0.

The Linpack uses matrix multiplication for both the factorization and the update step. As matrix multiplication is a problem with a complexity of $O(N^3)$, while e.g. communication is only an $O(N^2)$ problem for a matrix of size N, matrix multiplication has a huge impact on Linpack performance. For reasonable large matrices Linpack performance should approach plain DGEMM performance. DGEMM performs the matrix calculation $C = A \times B + C$. A NVIDIA GTX 280 or one GPU of a Tesla S1070 has a theoretical peak performance of 78 Gflops and can reach up to 74 Gflops in DGEMM, which can be used as limit for the expected Linpack performance on this platform.

5.2 The Standard Method

The standard method to tune the Linpack for a certain machine is to link it with a BLAS library that has been tuned specifically for the architecture of that machine, or even for the machine itself. As described by Fatica[35] this is also the way that NVIDIA persuades.

In a first version the Linpack was simply linked to the NVIDIA CUBLAS library. There was some additional work required as the data needs to be transferred to the GPU before the BLAS call and the result read back into the host afterwards. On one GPU of a NVIDIA S1070 this approach achieves about 21 Gflops, which is only 27 % of the theoretical peak performance that device can offer.

The discrepancy in between the achieved performance of the most naive approach and the theoretical peak performance has several reasons. One of the reasons is the overhead for copying the operands to the device and the results back to the host. This however can only be solved by a major rewrite of the code which is described in section 5.3.

Another reason is the bad device memory management performance of the NVIDIA driver. In the first version memory for the operands and results was always requested from the driver on demand. In an improved version all the the device memory is allocated at program start. Afterwards a simple first fit allocator is used to manage the device memory. This improves the performance to nearly 27 Gflops, now reaching about 36 % of peak performance.

Version	Gflops	% of theoretical Peak
Naive	21,0	26,9
Custom memory allocator	27,7	35,5
Trivial BLAS calls on CPU	28,8	37,0
Optimized DGEMM parameters	34,6	44,3

Table 5.1: Linpack Performance using the Standard Method

Yet another unnecessary bottleneck in the previous versions was moving all BLAS calls to the GPU. Some BLAS calls like DCOPY and DSWAP only copy memory contents. Moving those calls to the GPU only produces additional memory copies without having any potential for a speedup. Having those calls handled by a host BLAS library like Atlas[36] or Intel MKL[37] and pushing only the compute intensive calls like DGEMM and DTRSM to the GPU gives and additional but small speedup to 29 Gflops, which is 37 % of the peak performance.

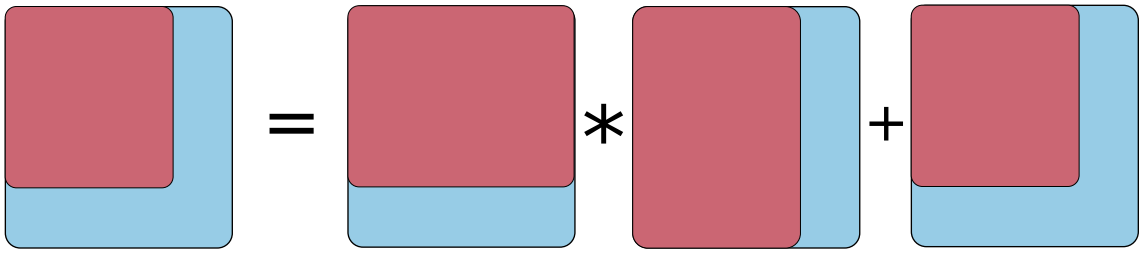


Figure 5.1: Submatrix with optimum DGEMM parametrization

An additional speedup can be gained from the knowledge that performance of the DGEMM contained in the NVIDIA CUBLAS library depends on the invocation parameters. As stated by NVIDIA in in their Forums[38] optimum DGEMM speed is only achieved if $M \bmod 64 = 0$, $N \bmod 16 = 0$ and $K \bmod 16 = 0$, where M is the number of rows of the matrices A and C, N is the number of columns of the matrices B and C and K is the number of columns of A and of rows of B. Therefore splitting up the equation as shown in figure 5.1 to have the largest part of the matrix calculated meeting these optimum parameters gives another performance boost. As matrix multiplication is an $O(n^3)$ problem calculating small parts of the DGEMM with suboptimal parameters doesn't hurt a lot. The largest Matrix C' that can be calculated using the optimum parametrization is of size $M' \times N'$, where $M' = M - M \bmod 64$ and $N' = N \bmod 16$. The parameter K cannot be tuned by choosing submatrices, however in the Linpack K is given by the blocksize NB and the result of recursively dividing NB by 2. Therefore choosing a blocksize $NB = 2^n \times 16$,$n \in \mathbb{N}$ ensures the condition $K \bmod 16 = 0$ to be met.

The best result reached using the traditional approach was 35 Gflops, which is 44 % of the theoretical peak performance of the device. This approach already

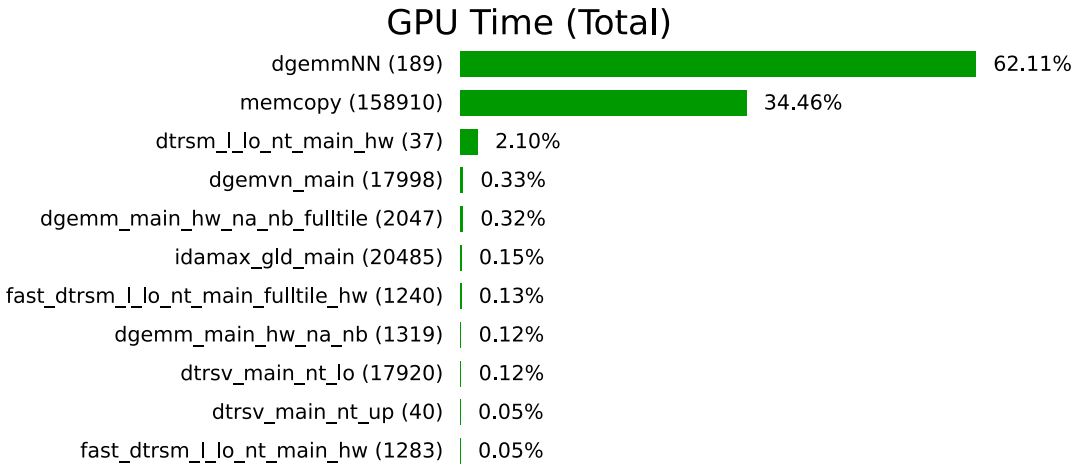


Figure 5.2: Profile of the Graphics Card Usage running Linpack using the Standard Method

requires special tuning in addition to the plain linking against the vendor provided BLAS library to reach this level. The major bottleneck in this case is the data transfer between the host and the graphics device. In figure 5.2 it can be seen that this transfer makes up more than 30 % of the execution time involving the GPU. In addition some parts of the algorithm, e.g. the before mentioned DCOPY and DSWAP are performed on the host and do not show up in the profile.

5.3 The Advanced Method

As shown in section 5.2 the limiting factor for the Standard Method are memory copies in between host and device memory, making up more then 30 % of the GPU execution time. Almdahl's law states that we cannot significantly increase the speed of our program by optimizing parts that only cause a negligible part of the execution time [39]. Therefore figure 5.2 shows that anything else but DGEMM and memory copies can be ignored for the optimizations.

Keeping this in mind the obvious choice would be to start with the largest block of execution time. As mentioned in section 5.1 DGEMM runs at close to peak performance. Therefore not much could be done to improve this part. As this is a benchmark it is also not possible to modify the algorithm to do less calculations. Therefore the best choice for optimization is the second block, memory copies.

In the standard approach the data needs to be moved in between the GPU and the CPU even if consecutive calls to the GPU work on the same data. Those data movements can be avoided by moving the complete matrix $\left[A\vec{b}\right]$ to the device.

Moving the data to the device requires a major rewrite of the code, as every part

of the benchmark has to be aware that the valid data is stored on the GPU. For a fair comparison the matrix is still generated on the CPU and measurements of the execution time include the initial copying from the host to device memory and reverse after solving the system. Due to this approach the host still needs to be able to keep the whole matrix in memory, but as system memory is usually larger than GPU memory on most systems this is not a limitation.

Moving the data to the GPU cannot avoid all memory copies. In addition to the copies at the begin and the end of the benchmark we still need to copy all data that is required by the CPU to perform decisions on the code path, e.g. when choosing the pivot element. Also everything that is supposed to be exchanged in between the processes using MPI needs to be transferred to the host before being sent and written to the GPU after being received.

Even a very first implementation of this method achieves nearly 45 Gflops. As it can be seen in figure 5.3 memory copies no longer make up a significant part of the overall GPU execution time.

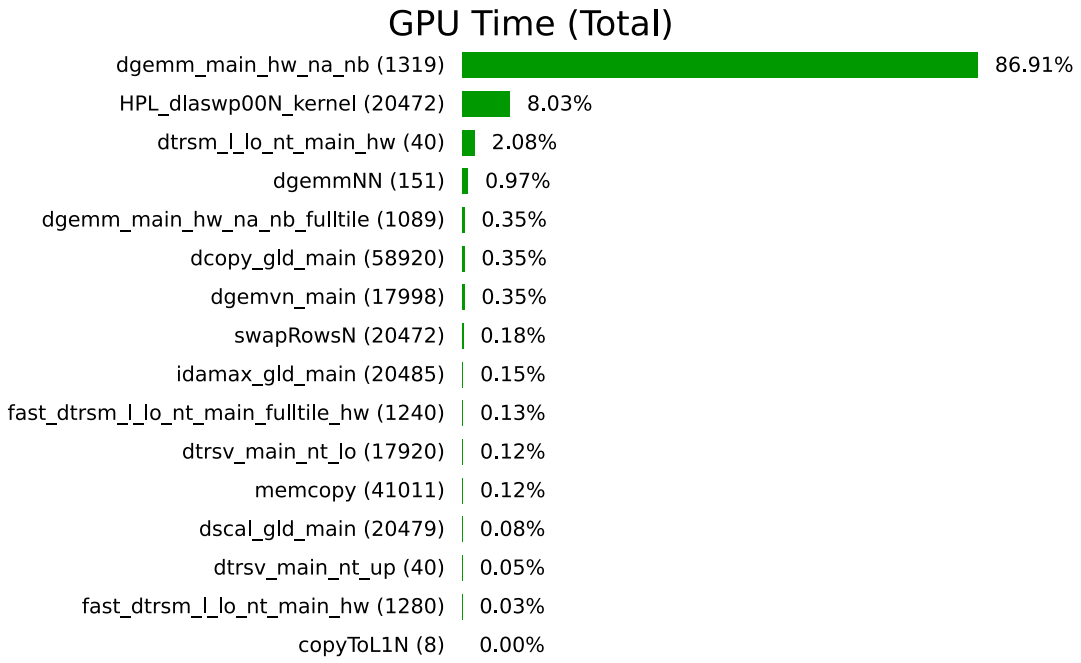


Figure 5.3: Profile of the Graphics Card Usage running Linpack using the first implementation of the advanced method

In addition to this effect row swaps can now profit from the faster RAM on the GPU. However, as seen in figure 5.3 this advantage is not for free, as a naive implementation of the swapping code still makes up 8 % of the execution time. Reworking the row swapping code gains some performance, which can be seen in table 5.2.

As in the standard method the DGEMM invocations can be splitted to achieve maximum performance. In this version the improvement is approximately 66.3 Gflops $\div$ 47.4 Gflops ≈ 40.0 %.

Some additional Gflops can be gained by using the custom memory allocator for host memory, too. Additional improvements to row swapping only show when using more than one process. Some tuning for the data transfer between CPU and GPU to remove restrictions on the pitch in 2D copies causes some minimal performance improvements as a side effect.

Attempts to utilize the CPU to help the GPU during DGEMM by moving the calculation of the small regions appearing in the DGEMM to the CPU proved unsuccessful. Obviously pulling the required data from the GPU and writing back the result takes longer than the GPU needs to operate on these small regions.

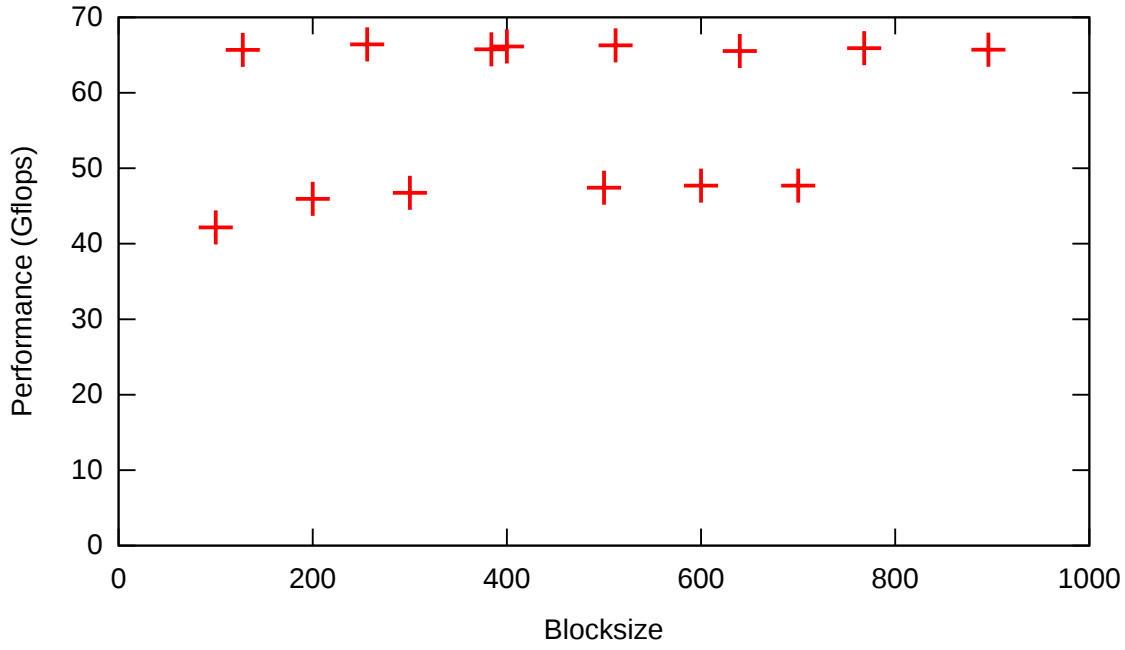


Figure 5.4: Linpack performance on one device for varying block sizes

As it can be seen in figure 5.4 the blocksize NB does not have a large effect on the performance as long as it is a multiple of 16. If this requirement for the optimal DGEMM execution path is violated performance drops by about 30 % to the level achieved before introducing DGEMM splitting.

Figure 5.5 shows that performance increases for larger matrices. This is caused by the fact that DGEMM, which is an $O(N^3)$ problem runs at close to peak performance, while memory copies, swaps and other things done by the benchmark only have a complexity of $O(N^2)$ and therefore loose impact on the overall application performance.

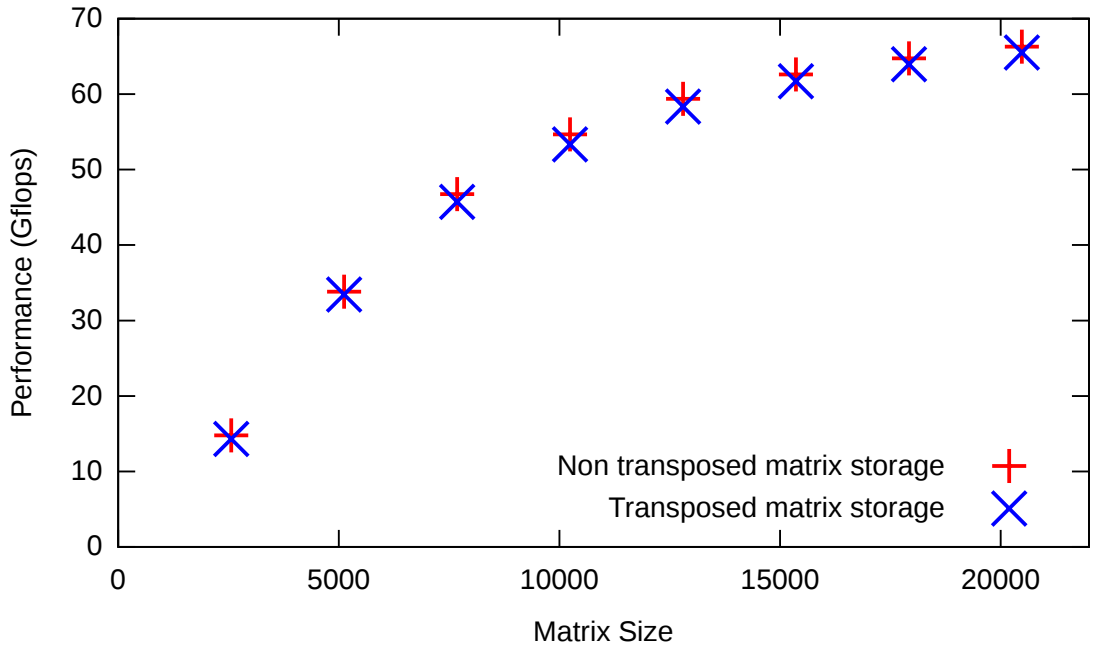


Figure 5.5: Linpack performance on one device for multiple matrix sizes

Additionally figure 5.5 shows that in the one node case it does not make much of a difference whether the not yet factored part of the matrix is stored in transposed or non transposed form. For all matrix sizes the transposed form is about 1 Gflop slower than the non transposed form. Therefore the non transposed storage has been used for all other measurements shown here.

Using this method on one GPU of a Tesla S1070 a performance of 67,2 Gflops could be reached, which is 86. 2 % of the theoretical peak performance of that device. As it can be seen in figure 5.6 In the final version DGEMM makes up more than 90 % of the execution time, followed by DTRSM which only makes up about 3 % and everything else, especially memory copies and swapping, being negligible.

5.4 Running Linpack on Multiple Cards

While the research on the Linpack was performed, the CSC-Scout cluster in Frankfurt was constructed. This provided an opportunity to optimize the Linpack for execution on multiple GPUs.

This cluster features a GPU based architecture shown in figure 5.7. It consists of 18 Dell Precision R5400s that contain 2 Quad-Core 3. 0 GHz Intel Xeon processors with 2×6 MiB L2 Cache and a 1333 MHz FSB. Each of these is equipped with 16 GiB of main memory. Packs of two servers are connected to three Tesla S1070s. Each Tesla S1070 contains four GT200 based GPUs with 4 GiB memory each. Ex-

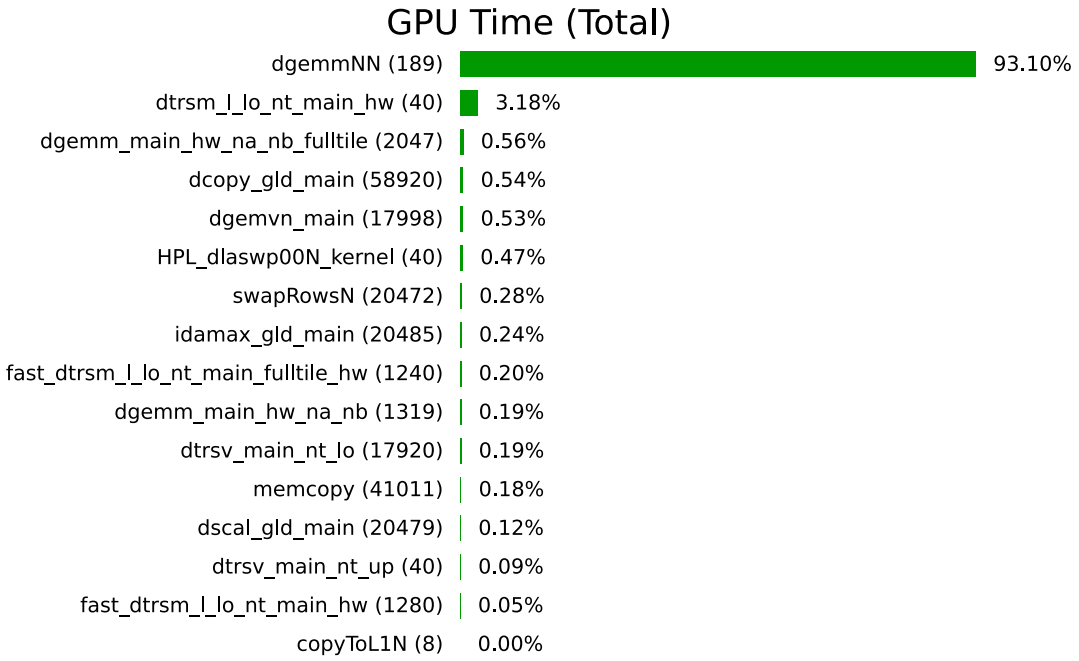


Figure 5.6: Profile of the Graphics Card Usage running Linpack using the final implementation of the advanced method

cept of the memory those cards are equivalent to the GTX 280. Therefore every server is equipped with six GPUs. Ubuntu 7.10 is used as an operating system. The version of CUDA used is 2.0. The installation of the high performance network is scheduled for a later point in time, therefore no full cluster benchmarks could be executed.

One of the problems when using multiple devices and moving towards larger matrix sizes is that two dimensional memory copies with a pitch of more than 262144 bytes occur during the update step. The pitch specifies the offset between consecutive blocks of memory to be copied. Until the given limit the DMA controller on the NVIDIA GPU can handle such a transfer on its own. Using asynchronous memory transfers it is possible to queue the transfers at the controller. This way no performance regression exists, which is shown in table 5.2.

When running the Linpack on multiple devices one process per card is required. The question arises which layout to use for the mapping to reach an optimum performance. Figure 5.8 shows the performance of the Linpack using different process layouts. Four cards and the optimum parameters mentioned in the previous section were used. The measurements show that the performance decreases with the number of process rows used. This can be explained by the fact that swapping the rows becomes more expensive with every process row added. In the optimum case of only one process row every GPU sees all rows. In that case only the row indices need to

Version	Gflops	% of theoretical Peak
First Version	44.5	57.1
Improved Row Swaps	47.4	60.7
Optimized DGEMM parameters	66.3	85.0
Improved Row Swaps for $P > 1$	66.4	85.1
Custom host memory allocator	67.1	86.1
Maximum pitch restriction removed	67.2	86.2
GPU / CPU help	66.3	85.0

Table 5.2: Linpack Performance using the Advanced Method

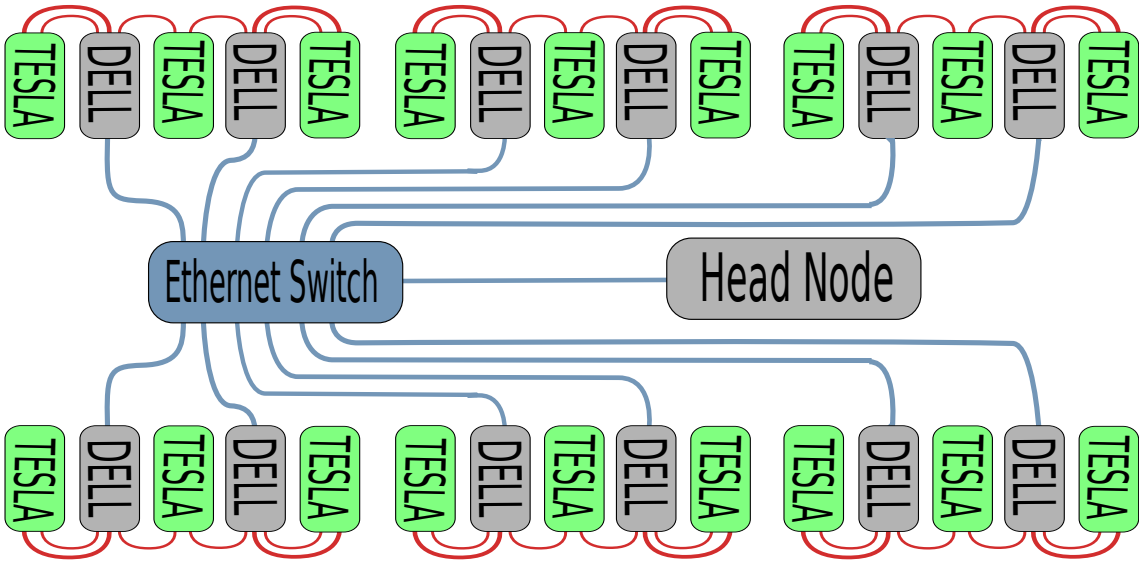


Figure 5.7: Topology of the CSC-Scout cluster

be communicated through the host and each GPU can swap its own part of the rows inside its own memory. In the case of multiple rows the GPUs have to exchange the copies through the host memory which involves slow PCIe transfers and adds an additional copy step. Of course there are limits for the amount of cards that can be used with a one process row layout. At a certain point the process column would be smaller than one block.

Different than in the one node case there is a measurable effect from the storage variant chosen for the update matrix U and the trailing matrix $L1$. As shown by the numbers presented in table 5.3 this effect especially shows for a 2×2 process layout, but it is also measurable in the 1×4 layout. Storing the matrices in the non transposed format obviously allows the memory accesses to be optimized better.

Figure 5.9 shows the performance of the Linpack run on N devices attached to the same computer. As in the one device case we can see the performance improve with the matrix size. The computer has 16 GiB of memory, therefore the matrix

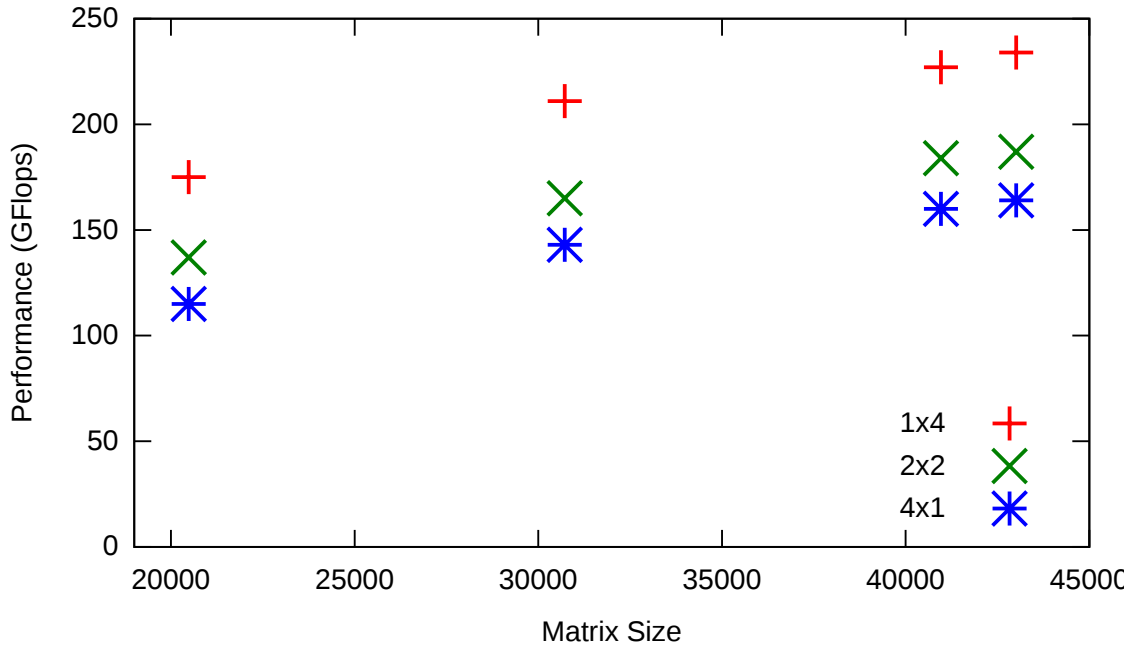


Figure 5.8: Linpack performance for different process layouts

Process Layout	$U^T L1^T$	$U^T L1$	$UL1$	$UL1^T$
1 x 4	221	227	227	222
2 x 2	112	112	183	182
4 x 1	161	162	160	160

Table 5.3: Linpack performance for different storage formats with $N = 41472$, $NB = 768$

size stops at that limit, reaching a peak of 225 Gflops for four devices using all their memory. As one Tesla device has 4 GiB of memory with six Tesla devices a matrix using 24 GiB of memory would be possible, given enough main memory or modifying the benchmark in a way to not keep the whole matrix in CPU main memory, too. Being limited to 16 GiB they peak up to 271 Gflops. However, while the performance for one to four devices clearly separates for all matrix sizes the five and six device runs already seem to hit a performance limit caused by communication overhead.

Until four devices the implementation scales well. Two cards processing an 8 GiB matrix reach 123 Gflops, which is an efficiency of 91.5 %.

$$E_{2 \text{ cards}} = \frac{123 \text{ Gflops}}{67{,}2 \text{ Gflops} \times 2} = 91.5\ \%$$

$$E_{4 \text{ cards}} = \frac{225 \text{ Gflops}}{67{,}2 \text{ Gflops} \times 4} = 83.7\ \%$$

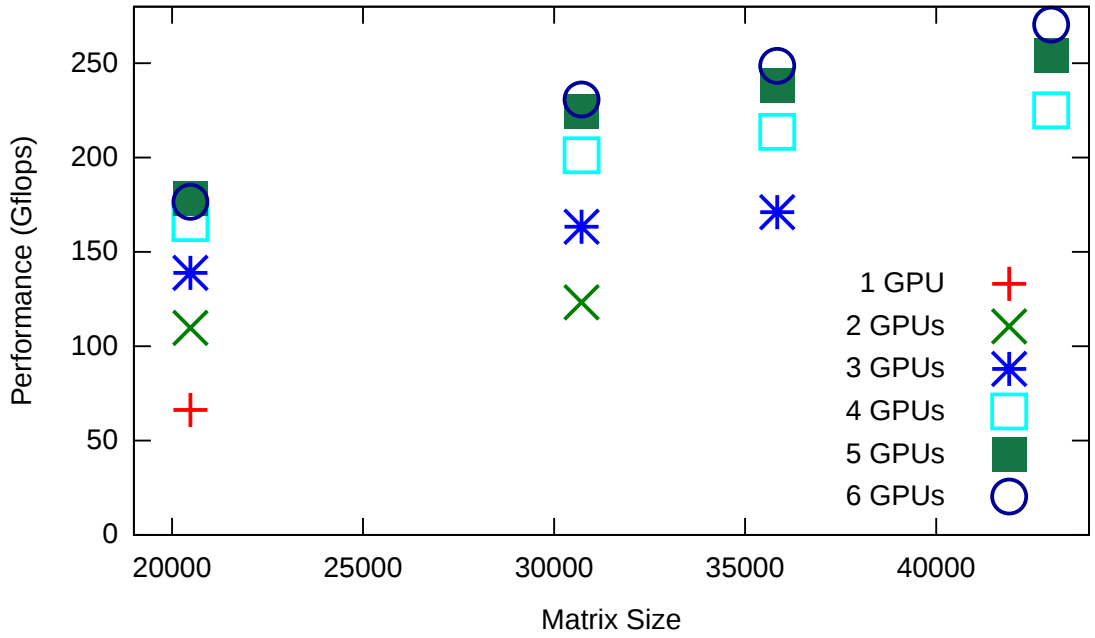


Figure 5.9: Linpack performance for different matrix sizes and number of devices

$$E_{6 \text{ cards}} = \frac{271 \text{ Gflops}}{67{,}2 \text{ Gflops} \times 6} = 67{,}2\ \%$$

As expected the efficiency decreases with the amount of communication required. When using six devices the performance is additionally limited by the devices being unable to use all of their memory. However even the comparison with one GPU using only 2. 5 GiB, reaching 62 Gflops, would give an efficiency of only 71 %.

Overall the advanced method outperforms the standard approach for multiple GPUs, too, reaching more Gflops per card than the standard method in the one device case.

For comparison, using both Xeons of the system together with the Intel MKL a performance of 71. 2 Gflops is achieved for a 41472 × 41472 matrix. This is about 106 % of the performance reached by one GPU and about 26 % of the performance reached by the six GPUs in the system.

Chapter 6

Summary

For all three classes of applications covered in this thesis, cellular automatons, Kalman filters and solving of systems of dense linear equations, applications have been successfully implemented for NVIDIA GPUs using CUDA.

For the game of life, as an example of cellular automatons, a speedup up to a factor of 20 is achieved. On a GPU of compute capability 1. 1 optimizations are required for the maximum speedup. Current hardware achieves the maximum speedup for a straight forward implementation. Further speedup is possible when specializing the algorithm on low occupancies of the grid. AMD devices seem to provide similar performance, but currently have a less flexible programming model.

A prototypical port of a cellular automaton based tracker was unable to outperform the CPU due to lack of parallelism and improper data structures. Later research, influenced by the results of the prototype, showed improved performance of the GPU versus the CPU. Modification of data structures to allow for more parallelism and better memory access patterns enabled a speedup for both architectures.

The implementation of the Kalman filter on the GPU is able to achieve a speedup of seven compared to a high performance SSE version. The way the implementation was performed both still share the same source code for the core algorithm. This ensures equivalent results. For this application PCIe transfer is a limiting factor. The Kalman filter is an estimation method light on memory. Still memory requirements are the limiting factor for the performance of this application, even though memory usage and accesses have been optimized.

Further performance increases might be reached by slimming the data structures used and removing read only data from the structures that represent the fitted track. Integrating this filter with a GPU based track finder is the logical next step.

The work on Linpack shows that an implementation focused on the GPU can achieve a performance of 67. 2 Gflops on one GPU. This is close to the performance reached by two quad core Xeons. Traditional implementations treating the GPU as a pure coprocessor are limited by PCIe transfer speed and only achieve 34. 6 Gflops. The presented implementation scales for multiple GPUs in one system, but efficiency drops if more then four GPUs are used. This work could be continued by testing

the implementation on a cluster like the CSC-Scout and optimizing it for the use on multiple hosts.

GPGPU is a rapidly moving area of research and the upcoming OpenCL[40] standard promises to become an interesting alternative to CUDA for future developments. As an open standard supported by all major companies producing processing units it should allow to easily use and compare different hardware, be it AMD, IBM, Intel or NVIDIA.

The results show that GPUs can be used to cope with the problems of increasing amounts of data. Algorithms properly implemented on the GPU will continue to improve in performance as GPUs increase their number of execution units. GPUs do however require use of highly data parallel algorithms and implementations. In addition PCIe transfer times and efficient memory usage need to be kept in mind. Future many core architectures with even higher degrees of freedom will provide further potential for performance improvements and a new challenge to find the optimum balance between parallelization and efficient memory usage.

Bibliography

[1] CERN Communication Group. *CERN FAQ — LHC: the guide*, January 2008.
URL `http://cdsmedia.cern.ch/img/CERN-Brochure-2008-001-Eng.pdf`

[2] D. Schlatter, P. Perrodo, O. Schneider, and A. Wagner. *Aleph in Numbers*, October 1996.
URL `http://aleph.web.cern.ch/aleph/aleph/SUBDET/aleph_det.html`

[3] ALICE HLT Collaboration. *ALICE High-Level Trigger Conceptual Design*. Technical report, CERN, December 2002.

[4] *Processor Spec Finder*. An overview about the specifications of all CPUs built by Intel.
URL `http://processorfinder.intel.com/`

[5] NVIDIA. *NVIDIA CUDA programming guide*, 2.1 edition, 2008.

[6] *General-Purpose Computation on GPUs*.
URL `http://www.gpgpu.org`

[7] M. Harris. *Simulation of Natural Phenomena Using Graphics Hardware*. Use of programmable graphics hardware for real-time visual simulation of diverse dynamic phenomena.
URL `http://www.markmark.net/cml/`

[8] Vincent Garcia, Eric Debreuve, and Michel Barlaud. *Fast k Nearest Neighbor Search using GPU*. CoRR, abs/0804.1448, 2008.

[9] *Hoomd*.
URL `http://www.ameslab.gov/hoomd/index.html`

[10] *ElcomSoft Files Patent for Revolutionary Technique to Recover Lost Passwords Quickly*, 2007.
URL `http://www.elcomsoft.com/EDPR/gpu_en.pdf`

[11] Sebastian Kalcher. *Optimization of a Distributed Fault-Tolerant Mass Storage System for Clusters*. Master's thesis, University of Heidelberg, 2004.
URL `http://www.kip.uni-heidelberg.de/ti/publications/diploma/2004SebastianKalcher.pdf`

[12] Martin Gardner. *The fantastic combinations of John Conway's new solitaire game "life"*. Scientific American, 223:120 – 123, October 1970.
URL http://www.ibiblio.org/lifepatterns/october1970.html

[13] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. *HPL*. A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers.
URL http://www.netlib.org/benchmark/hpl/

[14] David Kanter. *NVIDIA's GT200: Inside a Parallel Processor*, August 2008. [Online; accessed 06-February-2009].
URL http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242

[15] M. Fatica. *Private communication*.

[16] OpenMP ARB. *The OpenMP API specification for parallel programming*.
URL http://openmp.org/

[17] UPC Consortium. *UPC Language Specifications, v1.2*. Technical report, Lawrence Berkeley National Lab, 2005.
URL http://www.gwu.edu/~upc/publications/LBNL-59208.pdf

[18] *NVIDIA GTX 280*. Detailed description.
URL http://www.nvidia.com/object/product_geforce_gtx_280_us.html

[19] Wikipedia. *Cellular automaton — Wikipedia, The Free Encyclopedia*, 2009. [Online; accessed 29-January-2009].
URL http://en.wikipedia.org/w/index.php?title=Cellular_automaton&oldid=266151627

[20] I. Kisel. *Reconstruction of tracks in high energy physics experiments*.
URL http://www-linux.gsi.de/~ikisel/referat/Kisel_Referat.pdf

[21] H. Bjerke. *Private communication*.
URL http://openlab.cern.ch

[22] AMD. *AMD Stream Computing User Guide*, December 2008.
URL http://ati.amd.com/technology/streamcomputing/Stream_Computing_User_Guide.pdf

[23] *BrookGPU*. An extension of C for stream computing.
URL http://graphics.stanford.edu/projects/brookgpu/

[24] ALICE Collaboration. *Technical proposal for A Large Ion Collider Experiment at the CERN LHC*. Technical report, CERN, December 1995.

[25] *ALICE Time Projection Chamber*. The homepage of the Alice TPC.
URL http://aliceinfo.cern.ch/TPC/index.html

[26] R.E. Kalman. *A new approach to linear filtering and prediction problems*. In *Journal of Basic Engineering*, volume 82, pages 35 – 45. 1960.

[27] Rudy Negenborn. *Robot Localization and Kalman Filters*. Master's thesis, UTRECHT UNIVERSITY, 2003.
URL http://www.negenborn.net/kal_loc/

[28] Dana Mackenzie. *Ensemble Kalman Filters Bring Weather Models Up to Date*. SIAM News, 36(8), October 2003.
URL http://www.siam.org/pdf/news/362.pdf

[29] Jan Kybic. *Kalman Filtering and Speech Enhancement*. Master's thesis, École Polytechnique Fédérale de Lausanne, 1998.
URL http://cmp.felk.cvut.cz/~kybic/dipl/

[30] Leonard A. McGee and Stanley F. Schmidt. *Discovery of the Kalman Filter as a Practical Tool for Aerospace and Industry*. Technical report, NASA, 1985.
URL http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19860003843_1986003843.pdf

[31] S. Gorbunov, U. Kebschull, I. Kisel, V. Lindenstruth, and W.F.J. Müller. *Fast SIMDized Kalman filter based track fit*. Computer Physics Communications, 178:374 – 383, 2008.

[32] *The Message Passing Interface (MPI) standard*.
URL http://www-unix.mcs.anl.gov/mpi/

[33] Wikipedia. *PCI Express — Wikipedia, The Free Encyclopedia*, 2009. [Online; accessed 24-January-2009].
URL http://en.wikipedia.org/w/index.php?title=PCI_Express&oldid=266043649

[34] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. *Top500*. The Top500 list the 500 fastest computer system being used today.
URL http://www.top500.org

[35] M. Fatica. *Accelerating Linpack with GPU*. Handed out by NVidia on request.

[36] *Automatically Tuned Linear Algebra Software (ATLAS)*.
URL http://math-atlas.sourceforge.net/

[37] *Intel® Math Kernel Library 10.1*.
URL http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm

[38] *CUBLAS and coalesced operations, Access patterns inside CUBLAS code?*. Describes the effect of invocation parameters on GEMM speed.
URL `http://forums.nvidia.com/index.php?showtopic=76956`

[39] Gene Amdahl. *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*. In *AFIPS Conference Proceedings*, volume 30, pages 483 – 485. 1967.

[40] *OpenCL - The open standard for parallel programming of heterogeneous system*.
URL `http://khronos.org/opencl/`

[41] ASUS. *P5N32-E SLI Motherboard User Guide*.

Acknowledgements

I want to express my gratitude to everybody who contributed to the successfull completion of this thesis, especially Prof. Dr. Kebschull who invited me to work at the intersection of two fascinating areas of research. I also want to thank Prof. Dr. Lindenstruth for giving me the possibility to take part in the construction of the CSC-Scout cluster and to use this unique system for my measurements.

I want to thank Sebastian Kalcher for being a great supervisor and excellent guidance when entering the field of GPGPU, as well as for a lot of support and for proofreading this thesis. This thanks also goes out to Sergey Gorbunov and Ivan Kisel, who let me base parts of my research on their codes and whom I had many fruitful discussions with. Not to forget the rest of the TI group at KIP that allowed me to have an interesting and fun year.

To Prof. Dr. Ludwig I want to express my gratitude for him taking the time to be the second referee for this thesis.

For proofreading I also want to thank Helen Hofstetter and Martin Schorb. The same goes to Sacher Khoudari and Jennifer Wagner who provided me with a nice LaTeX template, which saved me a lot of time for the layout of this thesis.

Last but not least, I want to thank my friends, the VEX clan and my family for all the support given, especially my wife Tina who head to bear me during the times of coding, debugging and writing.

Appendix A

The Test System

Most of the measurements quoted in this thesis were performed on a system built from the consumer hardware listed in table A.1. Only the measurements in chapter 5 were performed on a different system, the CSC-Scout cluster whose hardware is described in section 5.4.

CPU	Intel Core2 Quad Processor Q6600 (8 MiB Cache, 2.40 GHz, 1066 MHz FSB)
Memory	4 GiB DDR2 800
Motherboard	ASUS P5N32-E SLI
GPU 1	Gainward GTX280 1024MB
GPUs 2 & 3	Sparkle 8800 GTS 512
Power Supply	XILENCE Gaming Edition 1000 Watt modular
Hard Drive	Samsung HD200HJ (200 GB)

Table A.1: Components of the test system

The motherboard is equipped with a NVIDIA nForce 680i SLI chipset. It has three PCIe-16 slots. According to the producer two of these can be run as full PCIe-16 slots while the third can only be used as a PCIe-8 slot[41]. In the test system the first, counting from the CPU, PCIe-16 slot was used for the GTX280. The two other slots were used for the 8800 GTS 512s.

Due to the noise from the airflow required to cool the system it was operated remotely via SSH. There was no monitor or keyboard attached.

Ubuntu 7.10 was used as an operating system during development. The versions of CUDA used were 1.1 and 2.0. For the final measurements the system was upgraded to Ubuntu 8.10 and CUDA 2.1. As the system was only used remotely no X Window System was run on the system.

Appendix B

Monitoring Tesla Systems

A NVIDIA Tesla S1070 unit has a maximum power consumption of 800 W. In installations like the CSC-Scout cluster which was already mentioned in section 5.4 up to 15 of these are installed in one rack.

$$P_{\text{total,max}} = 15 * 800 \text{ W} = 12 \text{ kW}$$

This number does not count the servers which are in the rack, too, and might add another 3 kW. With this much heating power a rack will quickly heat up if there is a problem with the cooling, potentially damaging the expensive hardware. It is therefore essential to monitor the hardware for high temperatures to prevent damage.

NVIDIA provides a tool called `nvidia-smi` to monitor the state of a Tesla unit. On invocation it scans for Tesla devices on the system and dumps a file containing the device state, including GPU and intake temperatures, fan speeds, LED colors and much more. As for an emergency procedure only the highest temperature is interesting a readout script was developed which scans this file and extracts the highest temperature so that it can be reported to monitoring systems like Ganglia or Lemon and be reacted upon by management systems like SysMES.

One of the problems of `nvidia-smi` is that it will completely stall the computer for a few seconds while scanning for Tesla devices. This is of course something that must not happen in cluster where high performance computations shall be performed. To solve this problem `nvidia-smi` should be put in its continuous mode and run as a daemon. When choosing such a solution it is important to verify the log-file age before processing the values.

When using the script in the CSC-Scout cluster it can be observed that the intake sensors of the Tesla devices sometimes report wrong temperatures, even more than 100°C when under load. These errors occur seldom, less then once a week for all 24 Teslas currently installed. As the intake temperature between all Teslas in the climate conditioned racks only varies about 1°C and one Server is connected to three half Tesla S1070 units, each with its own sensor, as a workaround only the second highest temperature is used.

Appendix C

Integrating CUDA into AliRoot

The track finder described in section 3.5 is embedded into the AliRoot framework. To be able to use CUDA inside AliRoot a patch for the AliRoot build system was developed.

The patch adds a new variable `CUSRCS` to the `pkg` file format which is used to specify CUDA source files. It adds the propoer rules to the Makefile so that these files are build with the NVIDIA CUDA compiler and linked into the final application. For the host part of the code it will use the same flags as are used for other C++ files in AliRoot. In addition the variables `CUCC` and `CUFLAGS` are required in the architecture configuration file.

There are a few restrictions when using CUDA inside AliRoot. `CXXFLAGS` must not contain `-pedantic-errors`, as it will by default, because the CUDA headers cannot be used with this option enabled.

CUDA code must not be passed through CINT for linking as CINT will not understand the additional syntax. Therefore the LinkDef.h of a package should not contain any `#pragma link C++ class` definitions pointing to classes using CUDA code. If this is required for some reason the CUDA specific syntax like `__host__` and `__device__` can be hidden from CINT by #defining it away if `__CINT__` is set.

Appendix D

Additional Material on the CD

The accompanying CD contains the additional material in the folder structure as described below.

- `Game of Life`
  - `CUDA` – 2D and 3D versions of the Game of Life implemented in CUDA.
  - `Brook` – Brook+ implementation of the Game of Life.
  - `CPU` – CPU version of the Game of Life used for comparison.
- `CATracker` – The first CUDA port of the cellular automaton based tracker for Alice.
- `Kalman Filter`
  - `simdkf` – The SSE based version of the Kalman filter used for comparison.
  - `CUDA` – The CUDA based version of the Kalman filter.
  - `CUDA_CT` – A CUDA based Kalman filter attempting to implement something similar to CT using templates.
- `Linpack`
  - `standard` – Linpack integrating CUDA in the traditional way
  - `advanced` – Linpack implementation using the advanced method
- `CUDA for Aliroot` – A patch to integrate CUDA into Aliroot version v4-11-Rev-03.
- `Monitoring Script` – A script to retrieve maximum temperatures from the log file produced by `nvidia-smi`.

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 11.2.2009