Compile-time GPU Memory Access Optimizations

Gert-Jan van den Braak
Department of Electrical Engineering
Eindhoven University of Technology
The Netherlands
E-mail: g.j.w.v.d.braak@tue.nl

Bart Mesman
Department of Electrical Engineering
Eindhoven University of Technology
The Netherlands
E-mail: b.mesman@tue.nl

Henk Corporaal
Department of Electrical Engineering
Eindhoven University of Technology
The Netherlands
E-mail: h.corporaal@tue.nl

***Abstract*—In the last three years, GPUs are more and more being used for general purpose applications instead of only for computer graphics. Programming these GPUs is a big challenge; in current GPUs the main bottleneck for many applications is not the computing power, but the memory access bandwidth. Two compile-time optimizations are presented in this paper to deal with the two most important memory access issues. To describe these optimizations, a new notation of the parallel execution of GPU programs is introduced. An implementation of the optimizations shows that performance improvements of up to 40 times are possible.**

I. Introduction

In the past, Graphics Processing Units (GPUs) were special purpose processors which could only be used for graphics processing. When the fixed-function pipeline in the GPU was replaced by a programmable pipeline, the enormous computational performance of GPUs could also be used for other kinds of computation. With the release of "CUDA" by NVIDIA and "Close to Metal" by AMD in November 2006, programming GPGPU (general-purpose computing on graphics processing units) applications became much more accessible. The NVIDIA CUDA SDK enables programmers to write programs which use an NVIDIA GPU for GPGPU applications. However, much time is still needed to optimize program code to utilize the maximum potential of the GPU. The massively parallel architecture which gives the GPU its enormous computational performance, is also responsible for its complex programmability.

As an example, an application which benefits from the parallelism in a GPU is the 2D Discrete Cosine Transform (DCT) [1]. Calculating the DCT of a full grey scale image of 256 by 256 pixels takes almost 10 seconds on a CPU using a single core implementation, but just 48 milliseconds on a GPU using an optimized implementation. This is a speed-up of 200 times, as shown in Table I.

TABLE I
Execution time of a DCT program on a CPU (Intel Q6600) and a GPU (NVIDIA GeForce 8800GT).

	Processing time	**Speed-up**
Naive CPU implementation	239s	0.04x
Single core CPU implementation	9.6s	1x
Four core CPU implementation	2.5s	4x
Basic GPU implementation	3.0s	3x
Optimized GPU implementation	48ms	200x

While porting an algorithm like the DCT from a CPU- to a GPU implementation takes only a few hours; optimizing the algorithm usually requires days to weeks. Before the DCT algorithm could be ported to the GPU, it first needed to be converted into a naive implementation. In the single core implementation all possible cosine values have been pre-calculated and stored in a table. In the naive implementation the cosine values are not stored but calculated over and over again. This matches better with the GPU hardware, since memory accesses can be costly and the GPU has a hardware cosine function, making recalculating the different cosine values easy. However, to optimize the GPU code further, the programmer needs to know many details about the hardware and in particular about the memory hierarchy.

In modern day GPUs the main bottleneck often is no longer the computational performance, but the utilization of the memory bandwidth. Non-optimized code utilizes only about 3% of the available bandwidth. The difference between a non-optimized and an optimized memory access pattern can be very subtle, but has a large impact on performance. This paper presents two optimization steps which improve this utilization. These steps will hide some of the memory hierarchy details to the programmer, enabling him to write code with higher performance without all the labor intensive optimizations. The new optimization steps are embedded into the current compiler and do not require any programmer interaction. With these optimizations a difference in execution time of up to a factor of 40 can be achieved.

This paper is structured as follows. First an overview of the NVIDIA GPU architecture and the CUDA programming model is presented in section II. In section III the main memory issues of an NVIDIA GPU are described. Section IV introduces a new way of describing the parallel execution of CUDA code on a GPU, which helps describing the suggested optimizations to the compiler in section V. An experimental implementation of these optimizations and its results are shown in section VI. Related work is described in section VII. Conclusion and future work are presented in section VIII.

II. Architecture and Programming model

The architecture of an NVIDIA GPU is shown in Fig. 1. It consists of several Streaming Multiprocessors (SMs) which hold several Scalar Processors (SPs). The NVIDIA GeForce

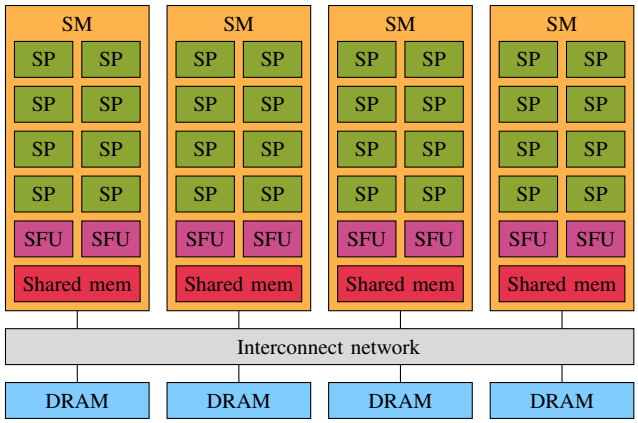

Fig. 1. NVIDIA GPU architecture. The NVIDIA GPU consists of several Streaming Multiprocessors (SMs), four are shown in this figure. Each SM holds several Scalar Processors (SPs), usually eight. Each SM also has two Special Function Units (SFUs) and a small shared memory (Shared mem). All SMs access the off-chip DRAM via the interconnect network.

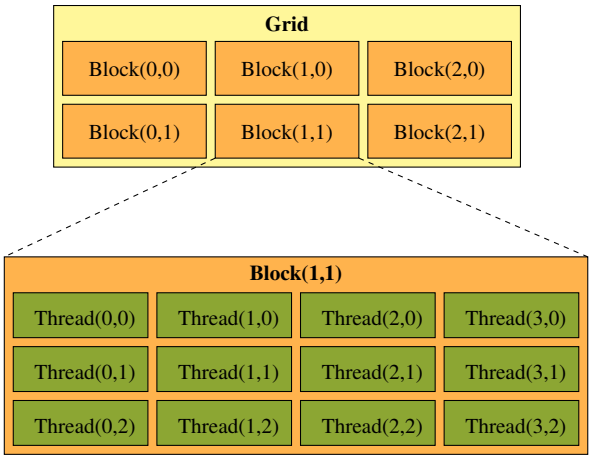

Fig. 2. CUDA thread hierarchy. Kernels are executed in parallel threads, and multiple threads are organized into thread blocks. All thread blocks together form a grid.

8800GT, with compute capability 1.1, used in this paper has 14 SMs each consisting of 8 SPs. In total there are 112 Scalar Processors in this GPU. Each Streaming Multiprocessor also has two Special Function Units (SFUs) and a small on-chip shared memory which makes it possible to share data between SPs in one SM. All Streaming Multiprocessors on the GPU access the off-chip DRAM via an interconnect network.

GPGPU programs running on NVIDIA GPUs consist of two parts, host code which is executed on a CPU and device code which is executed on a GPU. Device code consists of one or more kernels which can be started from the host code. Kernels, when called, are executed K times in parallel by K different CUDA threads. Kernels are called from the host code via the $<<<$ ... $>>>$ syntax. For example, calling a kernel named "MatrixAdd" with three parameters, M number of thread blocks and N threads in each thread block, is done with the following line of code: `MatrixAdd<<< M, N >>>(in1, in2, out);`

The CUDA thread hierarchy is shown in Fig. 2. Kernels are executed one at a time in parallel threads, and multiple threads (up to 512) are organized into thread blocks. All thread blocks together form a grid. Each thread block is executed on one SM and executes independently. The threads within a block are scheduled in warps, which consist of up to 32 threads. Every two clock cycles a half-warp of 16 threads is executed on the 8 Scalar Processors of the SM. All threads within a thread block can be synchronized by calling a synchronization function. Threads within a thread block can be identified by the three component vector threadIdx; each thread block can be identified with the two component vector blockIdx. The dimensions of a thread block are accessible within the kernel via the built-in three component vector blockDim; the dimensions of a grid are accessible via the two component vector gridDim. More detailed information about the NVIDIA GPU architecture and the GPU thread scheduling can be found in [2], [3].

III. Memory issues

For many applications the computations on data do not form the bottleneck, but the memory accesses to these data do. Therefore the memory bandwidth utilization is a better performance indicator than operations per second for these applications. The roofline model [4] combines these two performance indicators into one graph. It shows the attainable performance of a GPU for a given operational intensity (number of operations per DRAM byte accessed).

All measurements referred to in this section are based on a matrix addition kernel which adds two matrices into a third matrix. Each matrix has 512 x 512 elements and the measurements were done on an NVIDIA GeForce 8800GT. The measurements of section III-A use 32-bit integers as elements in the matrices and are indicated by circles in Fig. 3. The measurements of section III-B use 8-bit integers as elements in the matrices and are indicated by stars in the same figure. The roofline model of this GPU is also shown in Fig. 3. The graphs in the figure show the maximum attainable performance of this GPU in Gops/s (operations per second). In contrary to [4], where only GFlops/s (floating point operations per second) are used, all operations, integer as well as floating point operations, are used in this figure since the throughput of integer operations is at most as high as the throughput of floating point operations in this GPU. Also the number of (integer) instructions for calculating the index of the matrices outnumbers the number of operations used for the addition in the matrix addition examples.

A. Coalesced data transfers

Global memory has a latency of 400 to 600 clock cycles for a typical read operation [3]. Therefore coalescing global memory accesses is often one of the most important optimizations [5]. Coalescing is only possible if the memory accesses of the 16 threads in a half-warp are coordinated. The global memory accesses by all threads of a half-warp are coalesced by

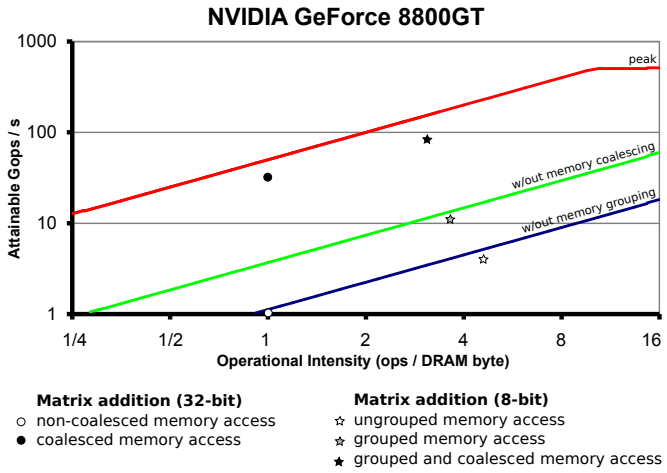

Fig. 3. Roofline model of an NVIDIA GeForce 8800GT. The top line (red) indicates the maximum attainable performance in Gops/s (operations per second) for this GPU. The next line (green) shows the attainable performance when memory accesses can not be coalesced and the bottom line (blue) shows the maximum performance when short-word memory accesses which are not grouped are being used. The indicated points show the measurements of two matrix addition kernels as described in section III.

hardware into one or two memory transactions if they satisfy the following three conditions [3], [5]:

- Threads must access
  - either 32-bit words, resulting in one 64-byte memory transaction,
  - or 64-bit words, resulting in one 128-byte memory transaction,
  - or 128-bit words, resulting in two 128-byte memory transactions;
- All 16 words must lie in the same aligned segment of 64 or 128 bytes for 32- and 64-bit words, and for 128-bit words the data must lie in two contiguous 128 byte aligned segments;
- Threads must access the words in sequence: The k^{th} thread in the half-warp must access the k^{th} word, although not all threads need to participate.

For devices of compute capability 1.2 and higher these conditions have been weakened. For these devices the memory accesses of the 16 threads in a half-warp will be combined to as few memory transactions as possible by hardware. If a half-warp addresses words in n different segments, n memory transactions are issued (one for each segment). The segment size is 32 bytes for 8-bit words, 64 bytes for 16-bit words and 128 bytes for 32-, 64- and 128-bit words [3].

Measurements, indicated by a black and a white circle in the roofline model in Fig. 3, show that the difference in execution time between a kernel with memory accesses which can be coalesced and a kernel with memory accesses which can not be coalesced, can be up to a factor of 30.

B. Grouping memory accesses

Memory accesses to the global memory of words shorter than 32-bit can not be coalesced on GPUs with compute capability 1.0 or 1.1, even if threads access these words consecutively. But it is possible to group four 8-bit memory accesses into one 32-bit memory access, which reduces the total number of memory accesses by a factor of four. This has to be done by the programmer with the extra C-structs CUDA provides to group some variables of the same type together. Measurements show that grouping four 8-bit memory accesses into one 32-bit vector access can lead to a speed-up of a factor of four. The same holds for 16-bit memory accesses; two can be grouped into one 32-bit memory acces or four can be grouped into one 64-bit memory access. Sometimes grouping short word memory accesses together in one large word access also results in the memory accesses becoming coalesced; this can lead to an even greater speed-up. In Fig. 3 the performance of ungrouped memory accesses is indicated by a white star; the performance of grouped memory accesses is indicated by a grey star and the performance of grouped memory accesses which can be coalesced is indicated by a black star.

```
1 forall( i=1:4 )
2   a(i) = a(i)+1
3   b(i) = a(i-1)+a(i+1)
4 endforall
```

(a)

```
1 dopar( i=1:4 )
2   a(i) = a(i)+1
3   b(i) = a(i-1)+a(i+1)
4 enddopar
```

(b)

initial value		after forall loop		after dopar loop
a(0)	1			
a(1)	2	b(1)	5	4
a(2)	3	b(2)	8	6
a(3)	4	b(3)	10	8
a(4)	5	b(4)	11	10
a(5)	6			

(c)

Fig. 4. Forall- and dopar loop example from [6]. The forall loop is shown in (a), the dopar loop is shown in (b) and the results are listed in (c).

IV. Parallel loop notation

To clarify the optimization steps to be discussed in section V, a new notation for the execution of CUDA kernels is introduced. This new notation uses parallel loops and makes the parallel execution of kernels in threads and thread blocks more visible than the normal $<<<$ ... $>>>$ syntax.

Two types of parallel loops are used, the forall loop and the dopar loop. In a forall loop each statement is completely executed for each iteration of the loop and data from all iterations is synchronized before the next statement is started. In a dopar loop each loop iteration is executed completely in parallel. The code within each loop iteration is executed sequentially. The initial state of all the data seen by each iteration is the initial state before the dopar loop was initiated [6]. See Fig. 4 for an example of the parallel loops.

When a kernel is being called from the host-code, it is executed $M \times N$ times in as many CUDA threads, where M is the number of thread blocks and N is the number of

```
__global__ void Kernel( )
{
  Kernel body
}

//invoke kernel
Kernel<<< M, N >>>( );
```

(a)

```
dopar ( i=0 : M-1 )
   forall ( j=0 : N-1 )
      Kernel body
   endforall
enddopar
```

(b)

Fig. 5. CUDA kernel and invoking as in the original syntax (a) and in the new syntax with parallel loops (b).

```
__global__ void MatrixAdd(int *in1, int *in2, int *out)
{
  int t1 = in1[threadIdx.x * gridDim.x + blockIdx.x];
  int t2 = in2[threadIdx.x * gridDim.x + blockIdx.x];
  out[threadIdx.x * gridDim.x + blockIdx.x] = t1 + t2;
}
```

(a)

```
__global__ void MatrixAdd(int *in1, int *in2, int *out)
{
  int t1 = in1[blockIdx.x * blockDim.x + threadIdx.x];
  int t2 = in2[blockIdx.x * blockDim.x + threadIdx.x];
  out[blockIdx.x * blockDim.x + threadIdx.x] = t1 + t2;
}
```

(b)

Fig. 6. CUDA kernel for matrix addition. In (a) the memory accesses cannot be coalesced; in (b) the memory accesses can be coalesced.

threads inside each block. A typical way to describe such a kernel is shown in Fig. 5-a. The kernel code itself is indicated by the "__global__" keyword, and the kernel is launched via the $<<<$ ... $>>>$ syntax. Each thread block is executed completely independent from all other thread blocks, and thread blocks can be executed in any order. Threads within a block can cooperate by sharing data through shared memory and can also synchronize their execution by calling a synchronization function. How the kernel is executed in parallel in threads and thread blocks is not clear from this notation, nor is the possible communication between the threads in a thread block.

The execution of a kernel on a GPU in many CUDA threads as described above, can also be described as two tightly nested loops which have the kernel as the loop body. This notation is shown in Fig. 5-b. The outer loop, which runs over all M thread blocks, is a dopar loop, since all thread blocks are executed independently and in any order. The inner loop is a forall loop which runs over all N threads inside a block. Since the execution of all threads within a block is not independent, due to communication via the shared memory and the synchronization function, the inner loop can not be expressed as a dopar loop.

V. Compile-time optimizations

The measurements in section III showed that coalescing memory accesses has the largest impact on the performance of GPU code which is limited by memory bandwidth. That is why the first proposed compile-time optimization step is to automatically change memory accesses into memory accesses which can be coalesced when this is possible. This is described in section V-A. Since coalescing memory accesses is not possible on memory accesses of short words, the compile-time grouping of multiple accesses is described as the second compile-time optimization in section V-B.

A. Compile-time coalescing

In Fig. 6 a typical CUDA kernel for a matrix addition program on a GPU is shown. Without knowledge about the underlying hardware in the GPU, there is no functional difference between the two. The kernel in Fig. 6-a uses memory accesses which cannot be coalesced, while the kernel in Fig. 6-b uses memory accesses which can be coalesced. This leads to a difference in execution time of a factor of 30, as described in section III-A.

When the parallel loops are put around the kernel-code, as shown in Fig. 7, it becomes clear why the second kernel executes so much quicker than the first kernel. In Fig. 7-a the memory accesses of two consecutive threads have a stride of M, the number of thread blocks, while the same memory accesses have a stride of one in Fig. 7-b. Since memory accesses can only be coalesced when the stride is equal to one, it is clear that the hardware can coalesce the memory accesses in Fig. 6-b and Fig. 7-b, but not in Fig. 6-a and Fig. 7-a.

Analyzing if a global memory access is coalesced can be done by a compiler calculating the stride of the memory access. If the stride of the memory access is equal to one between consecutive threads, the access is coalesced. Determining the stride of the memory access can be done by calculating the effect on the index of the memory access when the threadIdx.x variable is increased by one. If this effect is one word, then the access is coalesced, otherwise it is not. Also the effect on the index of the other variables, like threadIdx.y and blockIdx.x, can be calculated. If one of these variables shows an effect of one word, then swapping this variable with threadIdx.x will lead to a coalesced global memory access.

Changing the index of a single global memory access is usually not allowed. When the data from multiple memory accesses are used in a calculation, the indices of all these accesses must be changed in the same way. And when a variable like threadIdx.x or blockIdx.x is used as a part of the data, this also has to be changed. Usually it is easier to swap the variable which has an effect of one word on the index with threadIdx.x to make the memory access coalesced. Two types of variable swapping can be distinguished, intra-loop variable swapping and inter-loop variable swapping, which will both be discussed below.

1) Intra-loop variable swapping: Variables within the three component vector threadIdx can always be swapped, since

```
1 dopar ( i=0 : M-1 )
2    forall ( j=0 : N-1 )
3       int t1 = in1[ j * M + i ];
4       int t2 = in2[ j * M + i ];
5       out[ j * M + i ] = t1 + t2;
6    endforall
7 enddopar
```

(a)

```
1 dopar ( i=0 : M-1 )
2    forall ( j=0 : N-1 )
3       int t1 = in1[ i * N + j ];
4       int t2 = in2[ i * N + j ];
5       out[ i * N + j ] = t1 + t2;
6    endforall
7 enddopar
```

(b)

Fig. 7. Kernel for matrix addition, shown with the parallel loops. In (a) the memory accesses cannot be coalesced; in (b) the memory accesses can be coalesced.

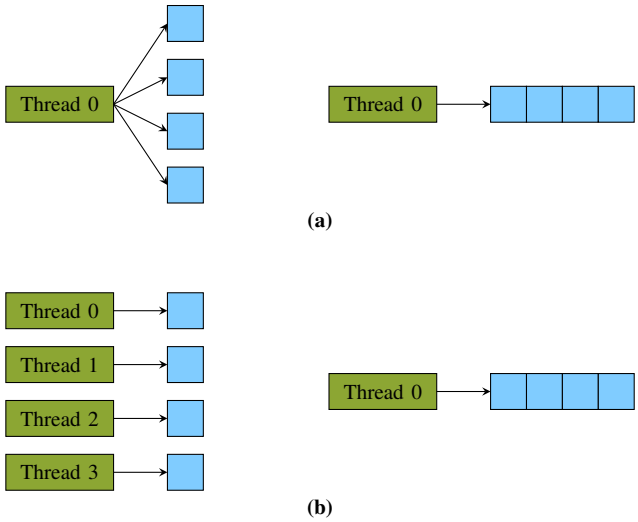

Fig. 8. Intra-thread memory access grouping (a) and inter-thread memory access grouping (b).

the threadIdx vector is used to calculate the ID of each thread within a thread block. All threads in a thread block are executed as in a forall loop, so the order in which they are executed does not matter. For the code inside the kernel swapping variables in the threadIdx vector is the same as renaming these variables and thus has no effect. Also the total number of threads in each block does not change. The same holds for swapping variables within the two component vector blockIdx. The blockIdx vector is used to calculate the block ID of all blocks in the grid, which can be executed in any order. For the code inside the kernel the swapping of these variables is the same as renaming them. Also the total number of thread blocks in the grid does not change.

2) Inter-loop variable swapping: When a variable from the threadIdx vector is exchanged with a variable from the blockIdx variable, some conditions have to be met. The number of threads in a thread block changes, just as the total number of thread blocks. Also the order in which the threads are executed changes in a way which may affect the outcome of the computations. Exchanging variables from the threadIdx vector and the blockIdx vector is similar as performing a loop interchange on the outer dopar loop and the inner forall loop. Interchanging these two loops is only allowed when the following two conditions (to be discussed below) are met:

1) Threads do not have a true data dependence[1] relation between each other;
2) After the loop interchange the number of threads in a thread block is less than or equal to the maximum number of threads in a thread block.

The inner forall loop can be expressed as a dopar loop when threads do not have a data dependence relation between each other. This means that each thread can be executed independently and that the inner- and outer loop can be interchanged. A data dependence can only occur via the shared memory. It is not possible to share data via the global memory. Writes to the global memory are not guaranteed to be seen by other threads within a specified time and when multiple threads write to the same memory location it is not specified which write will be preserved. Sharing data via the local memory (not discussed in this paper) or registers is also not possible, since these are private to each thread. So a data dependence between threads can only exist via the shared memory.

The maximum of each variable in the three component vector blockDim is 512, 512 and 64 respectively, and the maximum number of threads in each thread block is 512 [3]. The maximum number for each variable in the two component vector gridDim is 65536 [3]. For loop interchange to be allowed, the number of threads in a thread block must be less than or equal to 512 after the loops are interchanged. When this condition is met, the loops can be interchanged, otherwise the loop interchange is not allowed.

B. Compile-time memory access grouping

Coalescing memory accesses is only possible when 32-, 64- and 128-bit words are used. Shorter, 8- and 16-bit word, accesses can not be coalesced. The only way to enable these shorter word memory accesses to be coalesced is to group them into vector accesses which can be coalesced. Even if grouping shorter word accesses does not lead to memory accesses which can be coalesced, it can still improve the execution time of a kernel since the total number of memory accesses is reduced, as shown in section III-B.

Two possible cases where short word memory accesses can be grouped are distinguished. The first one is intra-thread memory access grouping, which involves multiple short word accesses within one thread which can be grouped in a vector access. This first case is shown in Fig. 8-a. The second one is inter-thread memory access grouping, which involves grouping multiple short word accesses from consecutive threads into a vector access. This can be achieved by grouping multiple

[1]True dependence (flow dependence) occurs when a variable is assigned or defined in one statement and used in a subsequently executed statement [6].

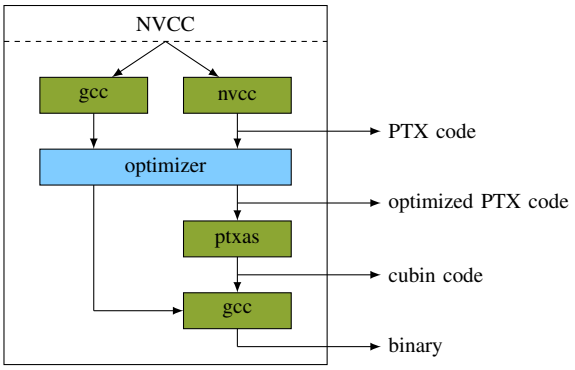

Fig. 9. CUDA compilation trajectory including the optimization tool.

consecutive threads together. This second case is shown in Fig. 8-b.

1) Intra-thread memory access grouping: Memory accesses within a kernel which can be grouped into a vector, can be found apart, or in a loop. To be able to group separate memory accesses, the stride of the accesses must be one word and it must be possible to calculate the index during compile time. Loops can be unrolled to create separate memory accesses which can then be grouped. The body of the loop can be replaced by several copies of it; the ideal number of copies, the unrolling factor, is either two or four, since these are the allowed vector sizes. The only thing which has to be taken into account, is the loop count which must be a multiple of the unrolling factor, otherwise guard code must be added to prevent the loop body from executing too many times.

2) Inter-thread memory access grouping: Several threads can be grouped together, to find multiple memory accesses which can be grouped. Since threads are executed in parallel, it is not always possible to group multiple threads into one thread. Threads are executed in a forall loop which means that executing one thread in series with another thread is only allowed when there is no dependence between these threads. Dependencies can only arise from the use of shared memory. Executing multiple threads as one thread by weaving them together and executing their instructions interleaved, is only possible when there is no conditional code and no branches in the code, due to limitations of the GPU hardware.

VI. Implementation & test results

A tool has been developed to measure the results of the optimizations of the CUDA compiler as proposed in section V. The optimization tool has been incorporated with the original CUDA compiler. The original CUDA compiler is described in section VI-A and the added optimization stage is introduced in section VI-B. The results of the optimization tool are presented in section VI-C and are also summarized in Table II. Also shown in Table II are the results which can be achieved by hand-optimizing the GPU code. In five of the six examples the optimization tool achieves the same speed-up as a programmer could achieve by hand-optimizing the code.

A. CUDA compilation trajectory

The main compiler for CUDA is called NVCC which controls the entire compilation process by calling various programs, see Fig. 9. The first stage of the compilation process is to split the program code into host code, which runs on a CPU, and device code, which runs on a GPU. The host code is compiled using a standard C-compiler. The device code is compiled into PTX code (an intermediate assembly language) using the nvopencc compiler, after which the PTX code is compiled to device-dependent cubin code[2] via the ptxas compiler. In the final stage the PTX and cubin code are combined with the compiled host code into one executable. This makes it possible to run the same executable on any kind of NVIDIA GPU. In case the cubin code does not match the specific hardware it is compiled for, the PTX code is compiled at run-time to cubin code which does match. More information about the compilation process can be found in [3], [7].

B. Optimization tool

The optimization tool is placed in the second stage of the compilation process. Here the host- and device code are already split, and the device code is compiled to intermediate PTX code. The host code still contains CUDA specific code, but the modifications that need to be made to the host code are minimal. A full source-to-source compiler is not required for these modifications, only a simple parser and C-code generator. Because the PTX code is fully documented in [8], modifying the PTX code is a better choice than modifying the cubin code, which is not documented, but only partly reverse engineered [9].

Other programming languages also compile to PTX, like OpenCL [10], or will do so in the future, like FORTRAN [3]. This means that a PTX optimizer can be used for many high-level programming languages and not only C, in contrary to a source-to-source compiler. The changes which have to be made to the host code are minimal, and can be easily implemented for high-level languages other than C.

The implemented optimization tool uses more strict conditions to determine if an optimization is allowed than the conditions described in section V. To check if there is a data dependence between threads, the tool checks if shared memory is used, since a data dependence between threads can only exist via the shared memory. An improved version of the optimization tool will track data dependencies trough the shared memory. To determine if and which variables need to be swapped in order to make the memory accesses coalesced, the tool calculates the index of each memory access by following every path in the control flow graph. This should lead to a single value for the index, otherwise the optimization step is canceled. To check if the number of threads in a block is larger than the maximum (which can occur when variables are swapped), a run-time check is added to the host code.

[2]Cubin code stands for CUDA binary code. This is the binary code that will be executed on the GPU.

This makes it possible to decide at run-time if the optimized kernel can be executed or that the original one must be used.

The single index of each memory accesses is also used to determine if various short word memory accesses have a stride of one word and can be grouped together. If possible, short word memory accesses within a kernel are grouped together. Otherwise the tool tries to group multiple threads to group short word memory accesses. The tool can only combine multiple threads into one thread by executing them in series. This means that the synchronization function should not be present and that the threads should not have a data dependence, which is checked as described above.

The optimization stage will copy the kernel, and apply all optimizations on this copy. The host code is modified so it can check at run-time which kernel to invoke, the original one or the modified one. One of the checks which is made at run-time is if the maximum number of threads within a thread block is not violated by the optimized kernel. If it is violated, the original kernel is executed, otherwise the modified and optimized kernel is executed. For example, the number of threads in a thread block is fixed in the Sobel edge detection algorithm (which will be discussed below), but the number of blocks is based on the size of the input image. This means that it is only possible to check the number of threads in a thread block (after inter-loop variable swapping is applied) at run-time.

C. Test results

The optimization tool is able to change the memory accesses as shown in Fig. 6-a to memory accesses as shown in Fig. 6-b. The first executes in 2.06ms, and the second in 0.067ms. This means that the optimization tool can achieve a speed-up of 31 times on this kernel by interchanging the blockIdx.x and threadIdx.x variables.

When the 32-bit memory accesses in Fig. 6 are replaced by 8-bit memory accesses, the optimization tool can group memory accesses in both kernels by grouping threads. The tool will also interchange blockIdx.x and threadIdx.x in the kernel of Fig. 6-a. Without the optimization tool the execution times are 1.020ms and 0.634ms; after the optimization tool is applied this is reduced to 0.024ms for both kernels, which is a speed-up of 43 and 26 times.

Applying the optimization tool on the Discrete Cosine Transform (DCT) example of section I results in a speed-up of four times (3.0s vs 0.75s). The optimization tool has unrolled the inner loop four times and grouped four 8-bit memory accesses. The speed-up of 62 times of the hand optimized code is achieved by grouping sixteen instead of four 8-bit memory accesses and by using the texture cache (not discussed in this paper).

The Sobel edge detection algorithm is a commonly used algorithm to determine the edges in a grey-scale image. The original run-time of this algorithm on a grey-scale image of 1920 by 1080 pixels is 12.1ms. After the optimization tool is applied, the run-time reduces to 2.84ms. The optimization tool has grouped four threads together to be able to group multiple memory accesses together. This optmization leads to the speed-up of just over four times.

TABLE II
TEST RESULTS OF THE OPTIMIZATION TOOL.

	Original	**With tool**	**Speed up**	**By hand**	**Speed up**
Matrix addition 32 bit	2.06ms	0.067ms	31x	0.067ms	31x
Matrix addition 8-bit	1.02ms	0.024ms	43x	0.024ms	43x
DCT	3.0s	0.75s	4x	48ms	62x
Sobel edge detection	12.1ms	2.84ms	4x	2.84ms	4x
Image rotation 1	3.29ms	1.64ms	2x	1.64ms	2x
Image rotation 2	3.40ms	1.27ms	3x	1.27ms	3x

The same grey-scale image is used for the two image rotation algorithms. In the first algorithm, all input pixels are read only once, and the corresponding output pixels are calculated. This results in some output pixels being written more than once, or not at all. The second image rotation algorithm writes each output pixel exactly once, and calculates which input pixel it has to read for each output pixel. In the first algorithm, the optimization tool can group four memory reads into one, which leads to a speed-up of two times. In the second algorithm, the optimization tool can group four memory writes into one, which leads to a speed-up of almost three times.

VII. RELATED WORK

There is a lot of research done on advanced loop transformations. However, most of this research focuses on e.g. data lifetime reduction, optimizing loop addressing and memory layout for general purpose systems with cache based memory hierarchies. Memory access coalescing has been described in 1994 by Davidson and Jinturkar in [11]. Although their work focusses on coalescing narrow memory references into wide ones on scalar processors, some of the issues of automated coalescing data accesses still apply to GPUs. Also the possibilities and limitations of compile-time coalescing memory accesses are discussed, as well as the options of changing the memory access pattern at run-time, which still leads to a significant speed-up.

The forall and dopar parallel loops have been described by Wolfe in [6]. Although this book mainly focusses on the compilation process of sequential code to parallel machines, data dependence in parallel loops is also discussed, just as loop interchanging with parallel loops. The dopar and forall loops as described in [6] are used in this paper to describe the parallel execution of CUDA threads.

Helping the programmer in optimizing the memory accesses has also been attempted by CUDA-lite [12]. CUDA-lite applies a source-to-source compiler, but programmers must add annotations to guide the compiler to do the memory optimizations. The help the programmer gets from CUDA-lite by not having to know the details of the GPU's architecture is largely canceled by the need of adding the annotations. In contrary, our approach does not require any programmer interaction, since all required information is obtained from the source code.

Source-to-source translation from C code with OpenMP directives to CUDA code has been done in [13]. To run the OpenMP programs efficiently on a GPU, several optimization steps, which improve global memory access, are done. The two compile-time optimization techniques used in [13] are parallel loop-swap and loop-collapsing. Parallel loop-swap is comparable to compile-time coalescing as described in section V-A.

VIII. Conclusion

In this paper two memory optimization steps are proposed for inclusion in a high performance GPU compiler. GPUs are praised for their enormous computational performance. In practical use however the actual performance is limited by the utilization of external memory bandwidth, but typical memory utilization for hand-written code is only about 3%. Manual optimization requires expert architecture knowledge and several weeks of rewriting effort. The memory utilization can be significantly improved by grouping memory transactions according to certain criteria derived from the architecture. Although the difference between two representations of an algorithm can be subtle from a programmers perspective, it can account for a factor 40 of difference in execution time. Rather than suggesting a certain writing style, a step to the existing NVIDIA GPU compiler is added, thereby hiding the memory optimizations and complexity from the programmer.

The integrated optimization tool changes memory accesses so they can be coalesced by the GPU hardware and groups several memory accesses in a thread or groups several threads. It has been shown that for several applications speed-ups between 2 and 40 times can be realized. It should be emphasized that the code quality never decreases as a result of the added compiler steps. Furthermore, readability, maintainability and portability of the program code remains because the optimization step is applied on the intermediate (PTX) assembly code.

Future work

The current optimization tool optimizes global memory coalescing and short word memory access grouping, but does not improve other memory issues, like partition camping and shared memory bank conflicts. The current implementation applies all optimizations whenever possible, but does not take any drawbacks into account, like a reduced number of threads when multiple threads are grouped together. Since the improvements are large compared to the drawbacks, this is allowed for the current implementation. An implementation which optimizes also other memory issues, should look more careful at the balance between the profits and the drawbacks. This could be done by adding an analytical model of the GPU [14] to the optimization tool, which can aid in making the decision on when an improvement to the code is beneficial.

References

[1] M. Nixon and A. Aguado, *Feature Extraction & Image Processing*. Newnes, 2002.

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[3] *NVIDIA CUDA™ Programming Guide - Version 2.2*, NVIDIA Corporation, 2009.

[4] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[5] G. Ruetsch and P. Micikevicius, *Optimizing Matrix Transpose in CUDA*, NVIDIA Corporation, 2009.

[6] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[7] *The CUDA Compiler Driver NVCC - Version 2.0*, NVIDIA Corporation, 2006.

[8] *PTX: Parallel Thread Execution ISA Version 1.2*, NVIDIA Corporation, 2008.

[9] W. van der Laan, "http://wiki.github.com/laanwj/decuda/," 2008.

[10] *OpenCL Programming Guide for the CUDA Architecture*, NVIDIA Corporation, 2009.

[11] J. W. Davidson and S. Jinturkar, "Memory access coalescing: A technique for eliminating redundant memory accesses," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994, pp. 186–195.

[12] S. Ueng, S. Baghsorkhi, M. Lathara, and W. Hwu, "CUDA-lite: Reducing GPU Programming Complexity," in *LCPC 2008*, 2008.

[13] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2009, pp. 101–110.

[14] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, 2009.