Technische Universiteit Delft
Faculteit Elektrotechniek, Wiskunde en Informatica
Delft Institute of Applied Mathematics

Het versnellen van CONTACT-programmatuur met behulp van de Graphics Processing Unit.

Engelse titel:
Speeding-up the CONTACT package by means of the Graphics Processing Unit.

Verslag ten behoeve van het
Delft Institute for Applied Mathematics
als onderdeel ter verkrijging

van de graad van

BACHELOR OF SCIENCE
in
TECHNISCHE WISKUNDE

door

MICHIEL DE REUS

Delft, Nederland
Augustus 2010

Copyright © 2010 door Michiel de Reus. Alle rechten voorbehouden.

BSc verslag TECHNISCHE WISKUNDE

"Het versnellen van CONTACT-programmatuur met behulp van de Graphics Processing Unit."

(Engelse titel: "Speeding-up the CONTACT package by means of the Graphics Processing Unit.")

MICHIEL DE REUS

Technische Universiteit Delft

Begeleiders

Prof.dr.ir. C. Vuik

Dr.ir. E.A.H. Vollebregt

Overige commissieleden

Dr. J.G. Spandaw

Augustus, 2010

Delft

Preface

As the final part of the bachelor programme Applied Mathematics at Delft University of Technology, a bachelor project must be completed. This thesis is the result of one semester of research in the field of GPU computing. Being a mathematics student, numerical analysis has my interest and I also have a lot of affinity with computer science. This is the reason why this particular project, 'Speedups of the CONTACT-software through the GPU', immediately took my interest. GPU computing makes it possible to do floating point calculations at very high speeds. The biggest problem is that the algorithms have to be suitable for the tasks a GPU can perform, which in practice means they have to be adapted quite a lot. The speedup is also highly dependent on the scale the algorithm can be parallelized. Because of the fact this field is relatively new, it took my special interest.

In addition this was one of few projects in cooperation with a company, which, I think, also contributes to the value of it. The company I worked with is VORtech BV. VORtech is situated in Delft, and is a combination of a software house and an engineering firm in applied mathematics. They combine a thorough knowledge of mathematical modeling, simulation techniques and computational algorithms with state-of-the-art software development methods. VORtech spun off from Delft University of Technology in 1996. Since then, it has seen a steady development, which still continues to this day. They service a broad range of customers, including both government departments and many commercial clients. VORtech supports the CONTACT computer programme that was developed by professor J.J. Kalker at Delft University of Technology. This software is considered to be an accurate model for contact mechanics and is used in various application fields. VORtech finances an active research programme to further extend and improve the software.

I want to specially thank professor Vuik and Edwin Vollebregt for guiding me through the project. Edwin is one of the founders of VORtech, and has a lot of knowledge of the CONTACT programme. Whenever I had questions about the algorithms involved, he helped me out. Professor Vuik helped me with the more general numerical issues, and the programming on the GPU in general. Furthermore the feedback they gave on my drafts helped me to improved the final result. A final word of thanks to all the people who posted on the CUDA GPU Computing forums of NVIDIA[1]. They helped me solving a lot of minor issues in a short amount of time.

[1] CUDA GPU Computing Subforums: http://forums.nvidia.com/index.php?showforum=62

Contents

Chapter 1

Problem formulation

CONTACT is a computer programme designed to calculate deformations in three-dimensional frictional contact problems. The programme is based on algorithms designed by professor J.J. Kalker.[1]

When two surfaces roll over each other, both surfaces deform due to tractions in the normal and tangential directions. The three-dimensional frictional contact problem determines the contact area of the two surfaces, and divides this area into a slip and an adhesion area. This gives information on which are the wearing parts of the surfaces. For example this information is used to determine where abrasion takes place on a train wheel and rail.

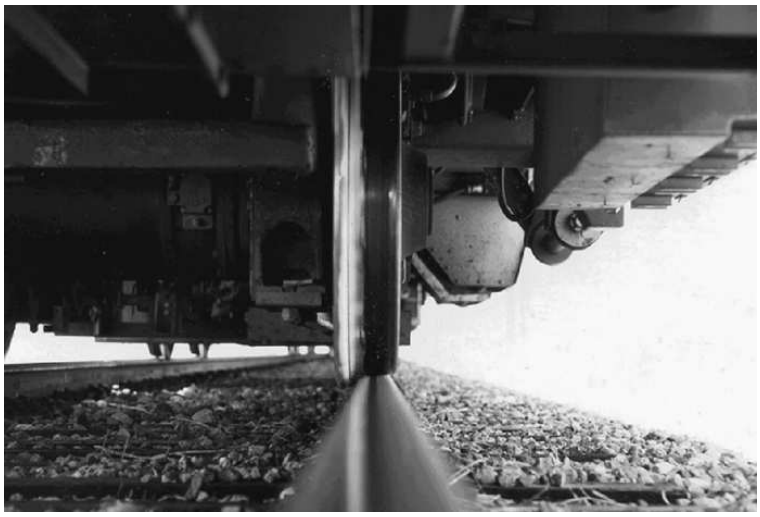


Figure 1.1: Train wheel on the rail, from www.emeraldinsight.com.

To determine which points are in adhesion and which are in slip, an iterative process makes sure all deformations are in equilibrium, under the condition of certain non-linear frictional constraints.

[1] See http://www.kalkersoftware.org/ for more information on the programme and papers describing the used algorithms.

During the iterative process, the Conjugate Gradients method and a certain Gauss-Seidel method are used. The first method consists of a full matrix-vector product during each iteration, the second method consists of a single inner product during each iterations. It are these operations that we will investigate in detail during this project.
The GPU is known for its parallel power, since it consists of a lot of seperate processors, each capable of performing floating point operations simultaneously. When calculating matrix-vector products and inner products, most calculations are independent of each other, and therefore might be suitable for the GPU.
The goal of this Bachelor project is to investigate possible speedups by using the parallel arithmetic capabilities of the Graphics Processing Unit, instead of just the Central Processing Unit, when calculating these matrix-vector products and the inner products.

Chapter 2

The frictional contact problem

The general problem considered is the three-dimensional frictional contact problem with homogeneous, but not necessarily the same, materials. In this problem, two surfaces roll over each other. The contact area is assumed to be concentrated. This means that the contact area is much smaller than the typical size of the objects. The contact surfaces can then be taken flat. The forces exerted on both surfaces will result in deformation of both materials. The degree of deformation depends on the materials used and the total force exerted.

During the solution process both the normal and the tangential contact problem are solved. The procedure is an iterative one, which will alternate between solving the normal contact problem and solving the tangential contact problem, using the solution of the last problem solved as input for the next. This process is repeated until all the forces are in balance, or in practice, until some tolerance is met. During this process, the contact area is updated continuously, resulting, in the end, in the true contact area.

The main equation, governing the deformations is

$$u_i(\underline{x}) = \sum_j \int_C A_{ij}(\underline{x}, \underline{y}) p_j(\underline{y}) ds, \text{ for } i = x, y, n. \tag{2.1}$$

$A_{ij}(\underline{x}, \underline{y})$ is the influence coefficient of the force at $\underline{y}$ in the direction j, on the deformation at $\underline{x}$ in the direction i. $p_j(\underline{y})$ is the traction, or force per unit area, at $\underline{y}$ in the direction j, and ds is the surface element. $u_i(\underline{x})$ is the resulting deformation at $\underline{x}$ in the direction i. In general the possible directions are x, y and n. The integral is taken over C, the contact area. Of course points outside the contact area will not contribute to the deformation. C can also be taken as the potential contact area, setting $p_j(\underline{y}) = 0$ outside the contact area.

One important property of the influence coefficients is that they are invariant under translation. This means that points with the same relative position to each other will have the same influence on each other. This makes it possible to reduce the number of independent influence coefficients dramatically, resulting in less memory usage. The details about the storage structure will be discussed later.

2.1 One-dimensional grid

The first step to solve the problem, is to discretize the contact area. If we look at the one-dimensional problem, the contact area is an interval $[0, L] \subseteq \mathbb{R}$. Equation 2.1 will change to

$$u(x) = \int_0^L A(x, y)\, p(y)\, dy. \tag{2.2}$$

This equation needs to be discretized in order to perform the calculations.
The interval is divided into nx equal-sized intervals x_I, $1 \leq I \leq nx$, of size $\delta x = \frac{L}{nx}$. The tractions are assumed to be constant in each interval, and the influence coefficient is determined by integrating over the inverval. The integral in Equation 2.2 changes to a sum over all the intervals. The resulting sum is

$$u(x_I) = \sum_{J=1}^{nx} A(x_I, x_J)\, p(x_J), \text{ for } 1 \leq I \leq nx. \tag{2.3}$$

This sum can be written as a matrix-vector product. If $\mathbf{u}$ and $\mathbf{p}$ are defined so that $u_I = u(x_I)$ and $p_I = p(x_I)$, and the matrix A is given by $A_{IJ} = \int_J A(x_I, y) \cdot dy$, then Equation 2.3 can be written as

$$\mathbf{u} = A\mathbf{p}. \tag{2.4}$$

As stated earlier, influence coefficients are invariant under translations. This means, for the one dimensional case, that $A(x_I, x_J) = A(x_K, x_L)$ if $x_I - x_J = x_K - x_L$. So an array B_K can be defined by

$$B_K = B_{J-I} = A(x_I, x_J), \text{ for } -n + 1 \leq K \leq n - 1. \tag{2.5}$$

The total number of independent influence coefficients will now be $2n - 1$. Remark that in general $B_K \neq B_{-K}$. The total number of independent influence coefficients is much smaller than without this symmetry, $2n - 1$ instead of n^2. For $n = 128$ for example, the number of independent influence coefficients is 255 instead of 16384. In the two dimensional case the difference will be even bigger.
The original sum can now be replaced by

$$u_I = \sum_{J=1}^{n} A_{IJ}\, p_J = \sum_{J=1}^{n} B_{J-I}\, p_J, \text{ for } 1 \leq I \leq n. \tag{2.6}$$

This will be used later, when implementing the calculations on the computer.
Geometrically this sum can be visualized by stretching the B vector from left to right, and aligning the $\mathbf{p}$ vector above it, with the rightmost element of $\mathbf{p}$ shifted $I - 1$ elements to the left and then taking the inner product of elements aligned above each other. See Figure 2.1 for an example. The blue elements of $\mathbf{p}$ are multiplied with the green elements of A, aligned above each other, and after addition, this results in the red element of $\mathbf{u}$.

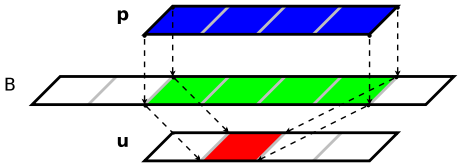


Figure 2.1: Example of an one dimensional inner product.

2.2 Two-dimensional grid

In the two-dimensional case a rectangular potential contact area is defined. This potential contact area is divided into $nx \times ny$ rectangular elements of size $\delta x \times \delta y$. The elements will be labeled by $\underline{x}_I$, with $I = (iy-1) \times nx + ix$. ix and iy represent the x and y coordinates of a grid point. The tractions now have three components, two components in the plane of the grid and one component in the normal direction. For simplicity we will only consider the normal direction. The tangential tractions will be covered later. This does not affect the process, because during the solving of the normal contact problem, the tangential effects are assumed to be constant, and are covered by the term h_I^*, as we will see later. Again the tractions and influence coefficients are assumed to be constant in each particular element. This will change the integral over the potential contact area to a sum over the individual elements, so Equation 2.1 will change to

$$u(\underline{x}_I) = \sum_J A(\underline{x}_I, \underline{x}_J) p(\underline{x}_J). \tag{2.7}$$

$A(\underline{x}_I, \underline{x}_J)$ is the influence coefficient of the traction at element $\underline{x}_J$ on element $\underline{x}_I$. If we again define $A_{IJ} = \left(\int_J A(\underline{x}_I, \underline{x}') d\underline{x}'\right) \cdot \delta x \cdot \delta y$, $p_J = p(\underline{x}_J)$ and $u_I = u(\underline{x}_I)$, this sum can be written as

$$u_I = \sum_J A_{IJ}\, p_J. \tag{2.8}$$

As in the one dimensional case, the influence coefficients are invariant under translation. If the source and the target have the same relative position, the influence coefficients are equal. This means that $A_{IJ} = A_{KL}$ if $ix - jx = kx - lx$ and $iy - jy = ky - ly$. As in the one dimensional case, an array B can be introduced so that

$$B_K = A_{I,J} \Rightarrow B_{(kx,ky)} = A_{(ix,iy),(jx,jy)}, \tag{2.9}$$

with $kx = jx - ix$ and $ky = jy - iy$. Remark that K is no longer a grid point, but a vector, representing the position of grid point J relative to I. Because $ix, jx \in \{1, ..., nx\}$, we have $kx \in \{-(nx-1), ..., 0, ..., nx-1\}$, which results in a total number of possible values for kx of $2nx - 1$. In the same way we have $iy, jy \in \{1, ..., ny\}$, so the total possible values of ky is $2ny - 1$. The dimensions of B are therefore $(2nx-1) \times (2ny-1)$. So the number of independent influence

coefficients is $(2nx-1)\times(2ny-1)$ instead of $(nx\cdot ny)^2$. Equation 2.8 can now be written as a matrix-vector product,

$$\mathbf{u} = A\mathbf{p}, \tag{2.10}$$

exactly as in the one-dimensional case.
There is also a geometric interpretation in terms of B and $\mathbf{p}$. If B and $\mathbf{p}$ are represented as two dimensional arrays, the different inner products of the matrix-vector product can be calculated by aligning $\mathbf{p}$ on top of B, and choosing the appropriate two-dimensional translation. So $\mathbf{p}$ is shifted over B in two directions. u_I can be calculated by aligning $\mathbf{p}$ so that p_I is on top of B_0. Values of B and $\mathbf{p}$ on top of each other are now multiplied, and the total sum is the inner product needed in the matrix-vector product. See Figure 2.2 for an example. As in the one dimensional case, the blue elements of $\mathbf{p}$ are multiplied with the green elements of B, aligned directly above each other, and after addition, this results in the red element of $\mathbf{u}$.

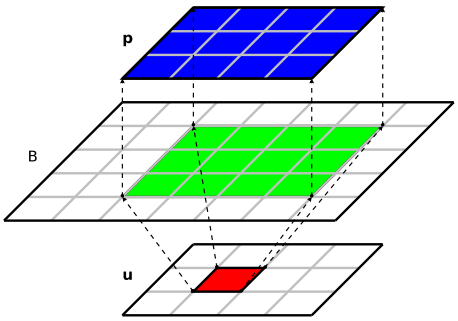


Figure 2.2: Example of a two dimensional inner product.

2.3 Normal contact problem

With the grid defined, the normal contact problem can now be solved. $e(\underline{x})$ is defined as the distance between the two bodies, in deformed state, and $h(\underline{x})$ is the distance in undeformed state. As stated earlier $u(\underline{x})$ is the deformation in the normal direction, so

$$e(\underline{x}) = h(\underline{x}) + u(\underline{x}). \tag{2.11}$$

As stated at the beginning of this section, the normal and tangential contact problem are solved in an iterative way, alternating between the two. Because of this, during the solving process of the normal contact problem, tangential tractions are assumed to be constant. In practice this means that deformations due to tangential tractions can be absorbed in the term representing the distance in undeformed state. We will therefore write $h^*(\underline{x})$ instead of $h(\underline{x})$. This results in

$$e(\underline{x}) = h^*(\underline{x}) + u(\underline{x}). \tag{2.12}$$

If $e(\underline{x}) > 0$ the traction is zero since there is no contact, so $p(\underline{x}) = 0$. If $e(\underline{x}) \leq 0$, there is a traction acting. The deformation as function of the traction is now given by Equation 2.1.
For the discretized problem, two sets of grid points are defined. Grid points for which $e_I > 0$ are set 'inactive'. At these points, the bodies are not in contact, so there is no traction. Points for which $e_I \leq 0$ are set 'active'. At these points there is contact and thus a traction. The set of active points is C, the contact area, and the set of inactive points is E, the exterior area. Note that these sets will change during the iterations. The problem we now want to solve, is to find the active grid points, and the tractions p_I at these grid points, so that the distances in deformed state are zero for active grid points. In this case the bodies will only be in contact at the surfaces, and they will not interpenetrate each other, which, of course, would be nonsense in reality. Equation 2.12 will change to

$$e_I = h_I^* + u_I = 0, \tag{2.13}$$

for all active grid points $I \in C$.
The deformations u_I are given by Equation 2.8. If we substitute this in Equation 2.13, we get

$$e_I = h_I^* + \sum_J A_{IJ} p_J = 0, \text{ for all } I \in C. \tag{2.14}$$

Remark that it does not matter whether the sum is taken over all grid points J, or $J \in C$, since $p_J = 0$ if $J \notin C$. We can write this equation as

$$A\mathbf{p} = -\mathbf{h}^*. \tag{2.15}$$

We now have a large linear system in p_J, for all active grid points $I \in C$. This system is solved by the iterative Conjugate Gradients method is used instead. This method will iterate, until the relative norm of the update of the solution vector is within a certain tolerance. When this tolerance is met, the iteration is stopped and the sets of active and inactive grid points are updated. From the resulting p_I, new u_I and e_I can be calculated. All grid points $I \in C$ with $p_I < 0$ are moved to the exterior set E, and all grid points $I \in E$ with $e_I < 0$ are moved to the contact set C.
The normal contact problem is iterated until the contact area is constant, so when C and E do no longer change.
For the first iteration of the Conjugate Gradient method, a starting vector is needed. It is assumed that $p_I^{(0)} = 0$ for all I in the potential contact area, so from Equation 2.14 we have

$$e_I^{(0)} = h_I^*. \tag{2.16}$$

2.3.1 Conjugate Gradients

The Conjugate Gradients method works only for positive semi-definite systems. In general, the system we get from the normal problem possesses this property, so the method will converge. The most important property of the Conjugate Gradients method, from our point of view, is that during each iteration a full matrix-vector product has te be calculated, which in general is very

time-consuming. There are several possible implementations of the iterative Conjugate Gradients method. We have for example Algorithm 2.1 from [3].

Algorithm 2.1 Iterative conjugate gradients algorithm.

$\mathbf{u}_0 := 0$
$\mathbf{r}_0 := \mathbf{b}$
for $k = 1, 2, ...$ **do**
  if $k = 1$ **then**
5: $\mathbf{p}_1 = \mathbf{r}_0$
  else
    $\beta_k := \frac{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}{\mathbf{r}_{k-2}^T \mathbf{r}_{k-2}}$
    $\mathbf{p}_k := \mathbf{r}_{k-1} + \beta_k \mathbf{p}_{k-1}$
  end if
10: $\alpha_k := \frac{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}{\mathbf{p}_k^T A \mathbf{p}_k}$
  $\mathbf{u}_k := \mathbf{u}_{k-1} + \alpha_k \mathbf{p}_k$
  $\mathbf{r}_k := \mathbf{r}_{k-1} - \alpha_k A \mathbf{p}_k$
end for

As can be seen on lines 10 and 12, the matrix-vector product $A\mathbf{p}_k$ has to be calculated. In Chapter 4 this matrix-vector product is discussed in detail.

2.4 Tangential contact problem

The tangential contact problem is defined through the friction law. The discretization is similar to the normal contact problem, resulting in the same grid as in Section 2.2. Since the constraints are non-linear, the Conjugate Gradients method cannot be applied. A specific version of the Gauss-Seidel method is used, to solve the non-linear problem.

During each iteration of the Gauss-Seidel like algorithm, the frictional constraints are taken care of, by shifting grid points from the slip area to the adhesion area and vice versa. During each iteration, four inner products need to be calculated. These inner products determine the total slip distance in the x and y direction in element I, due to the tractions in the x and y direction. There is also an additional term, consisting of the distance due to tractions in the normal direction and the rigid shift of the bodies.

The total deformation in both the x and y direction for grid point I is given by

$$\begin{cases} s_{Ix} = W_{Ix}^* + \sum_J A_{IJ}^{xx} p_{Jx} + \sum_J A_{IJ}^{xy} p_{Jy} \\ s_{Iy} = W_{Iy}^* + \sum_J A_{IJ}^{yx} p_{Jx} + \sum_J A_{IJ}^{yy} p_{Jy}. \end{cases} \tag{2.17}$$

We use the same indexing as in Section 2.2, so we have $I = (iy - 1) \cdot nx + ix$ and $J = (jy - 1) \cdot nx + jx$.

The influence coefficients have the same translational invariance as in the normal contact problem, so again we can define matrices B, with the geometric

interpretation explained in Section 2.2. This will change Equation 2.17 to

$$\begin{cases} s_{Ix} = W_{Ix} + \sum_J B_K^{xx}\, p_{Jx} + \sum_J B_K^{xy}\, p_{Jy} \\ s_{Iy} = W_{Iy} + \sum_J B_K^{yx}\, p_{Jx} + \sum_J B_K^{yy}\, p_{Jy}, \end{cases} \tag{2.18}$$

with $kx = jx - ix$ and $ky = jy - iy$. Remark that the inner products stated in Equation 2.18 are exactly the inners product from Figure 2.2. In Chapter 5 the inner products are discussed in detail.

Chapter 3

CUDA

In this chapter we will first review some historical background behind scientific computing on graphics cards, after which the way CUDA works will be described in more detail. The main goal is that the implementations in the next chapter can be understood.

3.1 Historical background

Nowadays almost all desktop computers come with a graphics accelerating card. These cards are specially designed to perform the calculations necessary to render three-dimensional images. Because most of these calculations can be performed independently of each other, these cards have a highly parallel character. The same calculations are performed on different sets of data. For this reason, a lot of space on the chip, normally used for control logic (fetching and decoding instructions etcetera), can be dedicated to performing calculations.
This combined with a huge market demand for better and faster graphics cards, especially from the video game industry, resulted today in high performance cards, with calculation performance comparable with supercomputers. Figure 3.1 illustrates the exponential growth of the graphics card performance, compared to general processors. In the figure, NVIDIA graphics cards are compared to Intel processors.

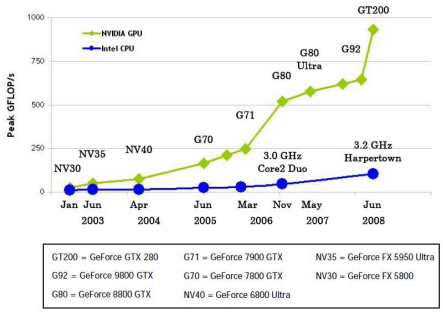


Figure 3.1: Intel CPU's compared to NVIDA GPU's, from [1]

In the past, all the stages performed by the graphics card were hardwired. These were the so-called fixed-function graphics pipelines. The application created the input data, and the graphics card performed a set of fixed operations on these data, resulting ultimately in a three-dimensional picture on the screen. One example is the stage applying the textures to the objects, built up from a large set of triangles.
This was the way graphics cards operated from the early 1980s to the late 1990s. During this time, the different stages were optimized, and new graphical algorithms resulted in ever increasing performance. But by the graphics industry growing larger, the application developers became more and more sophisticated, and the demand for more features grew with them. These demands could not be realized using the built-in fixed functions, so the next step was a programmable processor.
In 2001 NVIDIA took the first step with their GeForce 3. The developers could access the instruction set of the floating-point vertex engine. At the same moment, DirectX and OpenGL came with new releases to use these new features, allowing developers to use their own algorithms. The most interesting stages to be programmable were the stages where the different render algorithms take place. These are the stages using the most computational power, in particular, the vertex shader and pixel shader.
Because these stages performed so well on independent floating-point operations, it quickly took the attention of scientists in the field of scientific computing. They saw possibilities to port certain algorithms, gaining a huge performance boost for a relative low price. The problem was that, because the graphics cards were meant for graphical purposes, the interface was not suitable for general problems. So scientific problems needed to be adapted to 'graphical problems', meaning for example, that data needed to be translated to the same format as how vertices are stored on the GPU. Also the memory interface did not allow to read from, or write to random locations. So it took a huge effort to program the graphics card for general purposes. Because of these difficulties, only a few scientists demonstrated some useful applications.
To make scientific computing on the graphics card more accessible, NVIDIA introduced CUDA in November 2006. CUDA is an acronym for Compute Unified Device Architecture, and gives developers the opportunity to program the GPU using an extended version of the C programming language. Since the hardware architecture has to support CUDA, only the more recent models of the NVIDIA GPU's are compatible with CUDA.

3.2 The programming model

In the CUDA programming model, a distinction is made between the host and the device. The host consists of the CPU, together with the internal memory[1]. The device consists of the GPU, with different kinds of memory residing on the graphics card. When writing a CUDA program, separate code is written to run on the host and the device. The host code initializes the device for performing

[1] We will call the traditional internal memory the host memory.

the calculations. This means memory on the device is allocated and data needed for the calculations is transferred from the host memory to the device memory. After this is done, the host program calls a function intended to run on the device, a so-called *kernel*. It is the kernel that performs the calculations, using the data that is transferred earlier.
During the execution of a program, it is possible to invoke the same kernel more than one time, or to invoke different kernels. To every kernel call, a specific *grid* is attached. A grid is set of *blocks*, with a one- or two-dimensional structure, defined by the programmer. Every block consists of a set of *threads*, which are organized in a one-, two- or three-dimensional structure, again this structure is defined by the programmer. This hierarchy is important, because threads in different blocks are more limited to access the same memory locations than threads in the same block. Threads can identify themselves by built-in variables, containing the index of the block they are part of, and their own index, inside the block they are part of. This gives for example a mechanism to determine which memory index should be referenced. A more detailed explanation will follow later.
Another important consequence of the way threads are divided into blocks, is the way the threads are scheduled by the device. To better understand this, the lay-out of the device needs to be explained.

3.3 Lay-out of the device

A device consists of different parts. Figure 3.2 shows an overview. Every device is equipped with a number of *Streaming Multiprocessors*, and a certain amount of *global memory*. Multiprocessors are processing units operating independent of each other. The number of multiprocessors depends on the device model and can vary from 1 on the older notebook models to 120 on the high-end Tesla models. The total amount of global memory also varies, usually between 512 MiB for the older models and 2 GiB for the new high-end ones. A small portion of the global memory is accessible through the constant cache and the texture cache. These parts are usually called the *constant memory* and *texture memory*. These parts cannot be written to from the device. Because they are cached, memory read transactions from the same locations have a much lower latency than the normal global memory. The constant memory has a capacity of 16 KiB (per multiprocessor). The constant memory and texture memory will not be used during this project.
Every multiprocessor consists of 8 *Streaming Processors*, an *Instruction Unit*, 16 KiB of *shared memory* and a number of *registers*, each capable of storing 4 bytes of data. The total number of registers depends on the device and can vary between 8192 and 16384. Since the shared memory and registers are on-chip, they can be accessed very fast. However their size is very limited, because they are used by all the threads active on that multiprocessor. The differences between the different kinds of memories will be explained in detail later.
Since each multiprocessor has only one Instruction Unit, every processor in a multiprocessor executes the same instructions, and thus calculations, at the

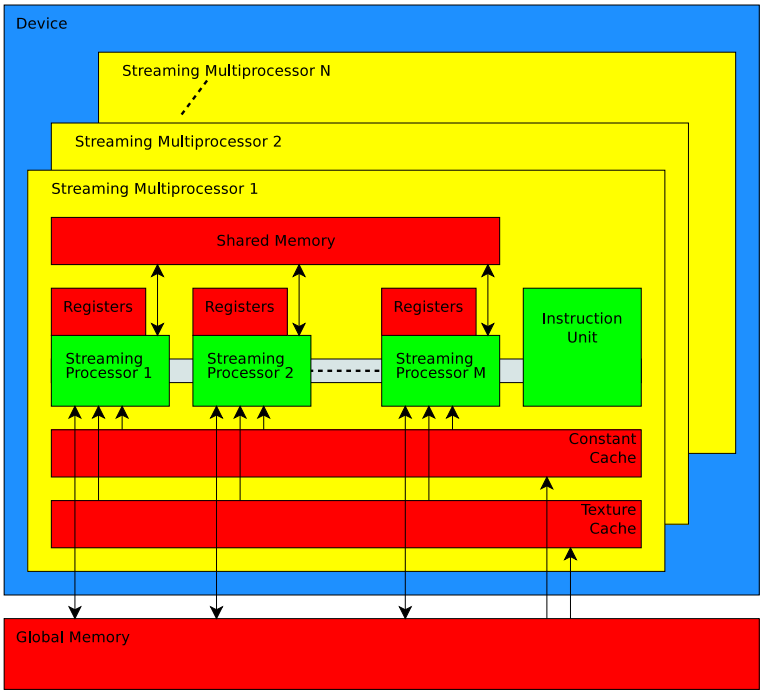


Figure 3.2: Layout of a typical GPU, adapted from [1]

same moment. The data on which these instructions are executed can of course differ. This model is called SIMT (Single-instruction multiple-threads) by NVIDIA. It is very similar to the SIMD-model (single-instruction multiple-data).

Now the role of threads and blocks can be explained. During execution, up to 8 blocks are assigned to a particular multiprocessor. These are called the active blocks. The actual number of blocks can be lower, depending on the number of threads per block and the amount of memory used per block. For example older models have a maximum of 768 active threads, so if the number of threads per block is 128, there are only 6 active blocks per multiprocessor. Furthermore if each block uses 4 KiB of shared memory, there will be only 4 active blocks, due to the limited amount of shared memory. For the use of registers the same applies. The reason we want as much active threads as possible, is that because when one set of threads is waiting until some high latency operation is finished, the (hardware) scheduler can assign a different set of threads for execution. This way the total time spent waiting for these high latency operations will be minimized.

The *occupancy* is defined as the number of active threads, compared to the maximum number of active threads possible. So if we have blocks of 256 threads, we can achieve an occupancy of 1, if memory usage allows three active blocks per multiprocessor. If we only have two active blocks, because each block uses a lot resources, the occupancy will be $\frac{2}{3}$, since we only have 512 active threads. The threads are executed in groups, called *warps*. Although the number of pro-

cessors is only 8 per multiprocessor, a warp consists of 32 consecutive threads, from the same block.[2] This means that all 32 threads in a warp will execute the same instruction at a specific moment. One important consequence of warps executing the same instructions, is branching. When the code contains conditional statements, and the threads in a warp 'diverge', they will execute different statements. But as mentioned earlier this is not possible since there is only one Instruction Unit, so the execution will be performed sequentially. In general this will cost more execution time. When the threads converge again, execution will continue parallel. Remark that because warps consist of 32 threads from the same block, the number of threads per block should be a multiple of 32 for maximum performance.
A warp is divided in two *half-warps*. Each consisting of 16 consecutive threads. Remark that from a programmers point of view, the division of threads into warps is completely transparent. This means that for the correctness of the code, you do not have to be aware of the existence of warps. But huge performance gains can be realized when the code is designed with this division in mind.

3.4 Extension of the C language

As already mentioned, CUDA is an extension of the C programming language. This means that extra keywords and special syntax are added, which are then compiled by the NVIDIA compiler. Without getting into too much detail, some of these extension are discussed.
The most important part is the kernel. The kernel is the function that is executed on the device. When launching the kernel, the CUDA driver needs to know certain parameters. We will be using only three of them, the number of blocks in the grid, the number of threads per block and the amount of (dynamic) shared memory used by each block. A kernel launch has the form

```
Kernel<<<dimGrid, dimBlock, Sharedmem>>>(...);
```

`Kernel` is the name of the function. `dimGrid` and `dimBlock` are the dimensions of the grid and the blocks. The parameters are of type `dim3`, which is a structure of three unsigned integers, representing the structure of the grid and the blocks. `Sharedmem` is the amount of shared memory used, in bytes. The type is `size_t`, which basically is an unsigned integer. The dots represent the additional arguments used by the kernel, like pointers to global memory locations, or the size of a particular array. When defining (or declaring) the kernel, the keyword `__global__` has to be placed in front, and the type is `void`. So we get something of the form

```
__global__ void Kernel(...);
```

The keyword `__global__` specifies that this function can be called from both the device and the host. It is executed on the device. Other possible keywords are

[2] This has to do with the different pipeline stages during the executing of an instruction.

`__device__` which means the function can be called from the device only, and is executed on the device, and the `__host__` keyword, which means the function can be called from the host and is executed on the host. This is also the default keyword. Inside the kernel several built-in variables are accessible. These can be used by a thread to identify itself, as in calculating its unique index, and determining what the the dimensions of the grid and blocks are. The used built-in variables are `threadIdx`, `blockIdx`, `blockDim` and `gridDim`. For example the unique index of a thread in a grid can be calculated by `blockIdx.x * blockDim.x + threadIdx.x;` given that blocks have a one-dimensional structure. Otherwise extra terms consisting of `y` and `z` members have to be added. One important other CUDA function that is used is `__syncthreads()`. This function makes sure execution continues only when all threads in that particular block have reached that statement. This is important when data is shared between different threads. It ensures that memory updates are in sync. We will see the importance later on.
These are just the basic CUDA keywords used. Later on other functions will be introduced, for example when allocating memory on the device, and copying data. Also the use of shared memory will require special syntax.

3.5 Compute capability

To make it easy to get information on what the capabilities of a specific device are, NVIDIA assigned a 'compute capability' to every CUDA compatible device. This compute capability consists of a major revision number and a minor revision number. The major revision number defines the core architecture. The minor revision number corresponds to small improvements of the core architecture, and newly introduced features. At the moment of writing, all the cores are of major revision number 1. The minor revision numbers can vary from 0 to 3. For example only devices with compute capability 1.3 support double precision floating-point numbers. Devices with lower compute capability only support single precision floating-point numbers. Furthermore compute capability 1.2 and 1.3 have 16384 registers per multiprocessor while 1.0 and 1.1 only have 8192 registers per multiprocessor. There are also differences in the way memory accesses are 'coalesced'. We will see this later in more detail. We will not get into too much detail regarding the compute capabilities, and all the code written is compatible with the lowest compute capability 1.0.[3]

3.6 Different memory types

As mentioned earlier, there are different types of memory. In this section a detailed description of the global memory, shared memory and the registers will be given and the syntax needed to use these different memories will be

[3] Although the code is compatible with any compute capability, there can be performance differences between different compute capabilities. For example because of different ways of memory coalescence.

explained. We will also look at the effect of the number of blocks assigned per multiprocessor due to the amount of memory used by each block and thread.

3.6.1 Global memory

Of the different types of device memory, the global memory has the biggest capacity. As mentioned earlier it can range from 512 MiB to 2 GiB. For the purposes of this project this will be more than enough. The global memory is used to store the bulk of the data supplied by the host. Because this memory is not located on-chip, it has a relatively high latency. Meaning that transactions from and to the global memory will cost a lot of clock ticks and should be avoided as much as possible. Also, for maximum performance, when moving data between the host and the device, it is better to move big data chunks instead of moving multiple small data chunks.
Allocating memory, freeing memory and copying data from the host memory to the global memory on the device and vice versa is done by special CUDA API calls. These functions are invoked from the host code, so before or after a kernel is invoked. There are different kinds of functions. For moving data for example, there are synchronous and asynchronous functions available. We will be using the synchronous functions only.
All the CUDA API calls return an error code. In the case of no error, the value `cudaSuccess` will be returned. In case of an error, a code describing what went wrong is returned, for example `cudaErrorMemoryAllocation` is returned when allocating memory fails. For obvious reasons we will check these return values when a CUDA function is invoked.
Allocating memory is done by invoking the function

```
cudaError_t cudaMalloc(  void ** devPtr,
                         size_t size      ).
```

The location of the allocated memory is stored in `*devPtr`. The second argument, `size` is amount of memory to be allocated, in bytes. Freeing memory is done by invoking

```
cudaError_t cudaFree(  void * devPtr  ).
```

where `devPtr` is the location of the memory chunk to be freed. Once memory is allocated, data can be copied to and from that location. This is done by the function

```
cudaError_t cudaMemcpy(  void * dst,
                         const void * src,
                         size_t count,
                         enum cudaMemcpyKind kind  ),
```

where `dst` is a pointer to the destination of the data. This can be located either in the host memory or the global memory. `src` is a pointer to the source of the data. This can also be either in the host or the global memory. `count` is the amount of data to be copied, in bytes. Finally, `kind` gives the direction in which

the memory has to be copied. The possible values are `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost` and `cudaMemcpyDeviceToDevice`. These values speak for themselves. We will mainly use the second and third mode. Again different kinds of error codes can be returned in case of an error.
When the data is copied to the global memory, it can be accessed from within a kernel. When invoking that kernel, the location of the chunk of data (in the global memory) is passed as a pointer in the argument list of the kernel, and normal C statements can be used to access it. Remark that the global memory has one memory space, so every thread can access every location. Still the global memory should not be used to communicate between threads of different blocks. This is because the order of execution of the different blocks is not defined. Even for threads in the same block it is very important to keep an eye on synchronization issues. A thread writing to a specific memory location, can change the input of another thread, because the order of thread execution is not defined. All we know for certain is that threads in the same warp are executed simultaneously. If a programmer does not anticipate this, and does not use `__syncthreads()` properly, this may lead to undefined results.
Special care is needed by the use of these pointers. The host memory and the global memory consist of different memory spaces, so a particular pointer points to particular locations both in the general and the global memory. When using a pointer, pointing to allocated memory in the global memory space, in the context of the host memory the results will be undefined, and an exception is probably raised. Vice versa it will lead to undefined results, since non-initialized data is used. For this reason we will be using prefixes in the names of the variables, making clear to which type of memory it points. We will use the `h` to point to host memory and the `d` to point to device memory. For example `float* hA` will point to some location in the host memory, while `float* dA` will point to a location in the device memory.
For maximum performance, it is important to use the correct alignments when moving data. For example, for device capability 1.0 and 1.1, when a half-warp, consisting of 16 threads, access consecutive 4-byte memory locations in a single 64-byte segment, aligned at a 64-byte boundary, this will result in a single memory transaction. This is called *memory coalescence*, and the memory transaction is called *coalesced*. In contrast to a non-coalesced memory transaction, for example when the alignment is incorrect. It is very important to ensure coalescing happens when possible, because it can have drastic influence on the total performance. When a memory transaction is non-coalesced, is it serialized in 16 seperate transactions. See Figure 3.3 for an example. Both access patterns are coalesced and will result in one memory transaction. In the access pattern on the right-hand side, the warp has diverged because of some conditional statement. This will lead to bandwidth loss, because some data that is read is not used, but the pattern is still coalesced.
In Figure 3.4 there are two examples of access patterns that will lead to non-coalesced memory transactions. The left access pattern is non-sequential, and will lead to 16 memory transactions on devices of compute capability 1.0 and 1.1. The right access pattern is misaligned, because the first thread does not

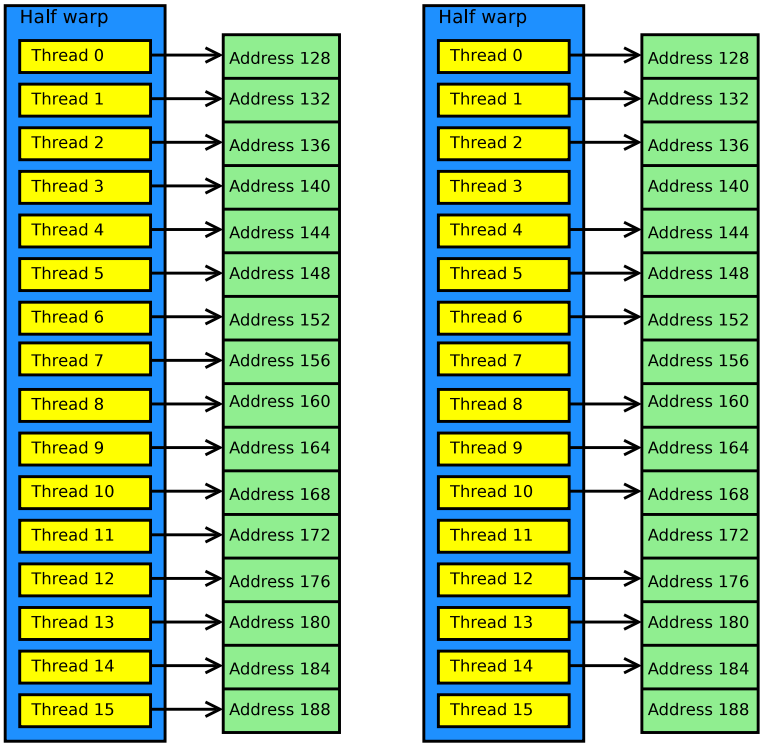


Figure 3.3: Examples of coalesced global memory transactions

access a location on a 64-byte boundary. This access pattern will also lead to 16 memory transactions. This illustrates that it is very important to make sure transactions are coalesced.
On devices with a higher compute capability, misaligned memory transactions will result in two memory transaction instead of 16, one for each segment data is requested from.
In the next chapter, the first CUDA implementation will show the use of the global memory. Also the effect of coalescing will be shown.

3.6.2 Shared memory

The next type of memory is the shared memory. Shared memory is shared among threads of the same block. This makes it the easiest and fastest way of data exchange between threads. The current devices consist of 16 KiB shared memory per multiprocessor. This amount is shared between all the active blocks. As mentioned earlier, when blocks use a lot of shared memory, this might lead to a decrease of the number of active blocks per multiprocessor. In the case of a block exceeding the total amount of 16 KiB, the kernel will not launch at all.
The shared memory can be used in two ways, either static or dynamic. Static means that the amount of shared memory per block is known when the code is compiled and dynamic means it is only known when the kernel is invoked.

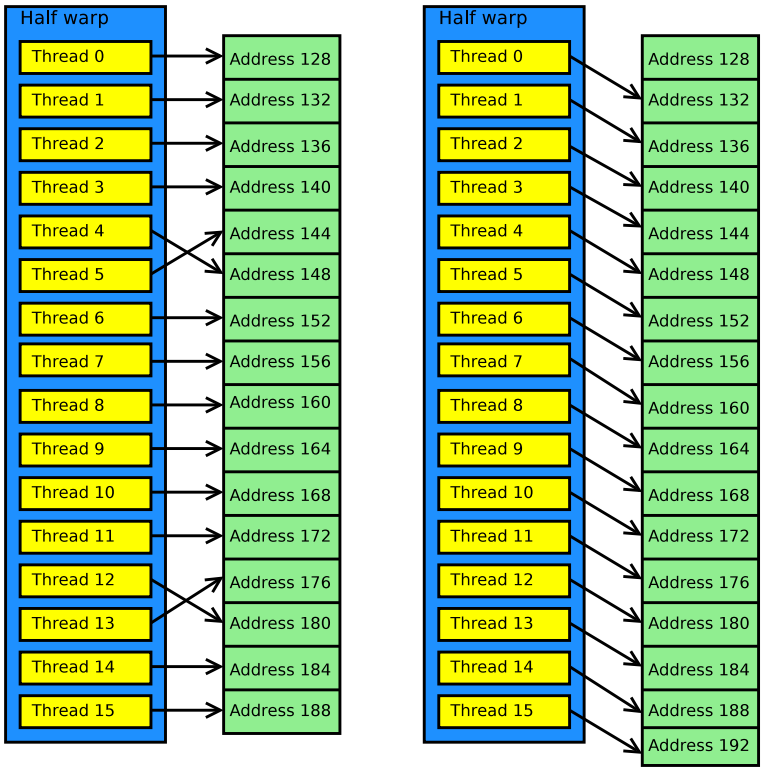


Figure 3.4: Examples of non-coalesced global memory transactions

So the total amount per block can be determined during runtime. These two different ways of using the shared memory need different syntax.
To make use of the shared memory, variables have to be declared as shared explicitly. This is done by the type qualifier `__shared__`. When using static shared memory, a variable can be declared within the kernel, with the `__shared__` type qualifier. For example

```
__shared__ float values[16];
```

For each block a different instance of the array `values` will be allocated. Threads from the same block will access the same instance of `values`, and can read and write from and to it. As mentioned earlier, special care is needed to avoid sync issues.
When using dynamic shared memory, the total amount of used shared memory is given by the programmer as a parameter when the kernel is launched. This amount of memory is then allocated by the CUDA engine as a linear array. It is the responsibility of the programmer to make sure this linear array is divided in the correct structure. A special statement,

```
extern __shared__ float array[];
```

has to be placed in the global scope, and the kernel has to be invoked by a statement like

```
Kernel<<<dimGrid, dimBlock, Sharedmem>>>(...);
```

where `Sharedmem` is the desired amount of shared memory, in bytes. The CUDA engine makes sure that for each block this amount is allocated and it can be accessed by the threads through `array`.
When using shared memory it is important to understand how this memory is organized for maximum performance. The total shared memory is divided into 16 memory modules, or banks. These banks can be accessed simultaneously. The memory is divided in such a way that successive 4-byte words fall in successive banks. During each cycle, each bank can deliver one 4-byte word. If multiple threads try to read different 4-byte words from the same bank, the requests are serialized, resulting in a lower performance. This is called a bank conflict. Since there are 16 banks and 32 threads in a warp, the first half-warp is served first, and then the second half-warp is served. This means that there cannot be bank conflicts between threads in a different half-warp. When accessing an array of shared memory, it is important to make sure the access pattern does not result in bank conflicts. See Figure 3.5 for an example. The left access pattern will not lead to any bank conflict, since every bank serves one word to one thread. The right access pattern will lead to a two-way bank conflict, since half of the banks have to serve two different words.
The shared memory also has a broadcast mechanism. This means that one of the words can be broadcast to all the threads in the half-warp. See Figure 3.6 for an example. The right access pattern will lead to a single memory transaction, since the word from Bank 0 will be broadcast. The left access pattern will lead to two memory transactions.

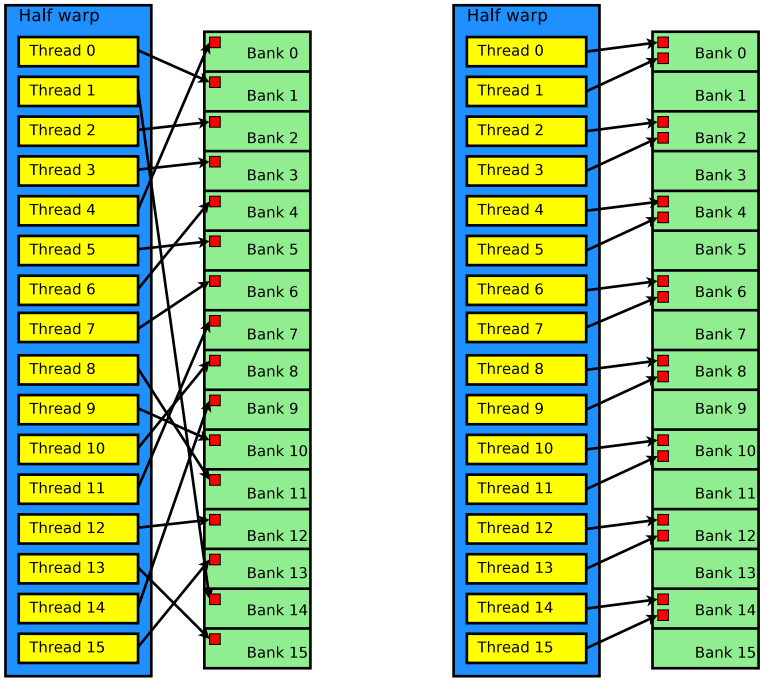


Figure 3.5: Examples of access patterns with and without shared memory bank conflicts: The left access pattern will not load to a bank conflict, the right access pattern will lead to a two-way bank conflict.

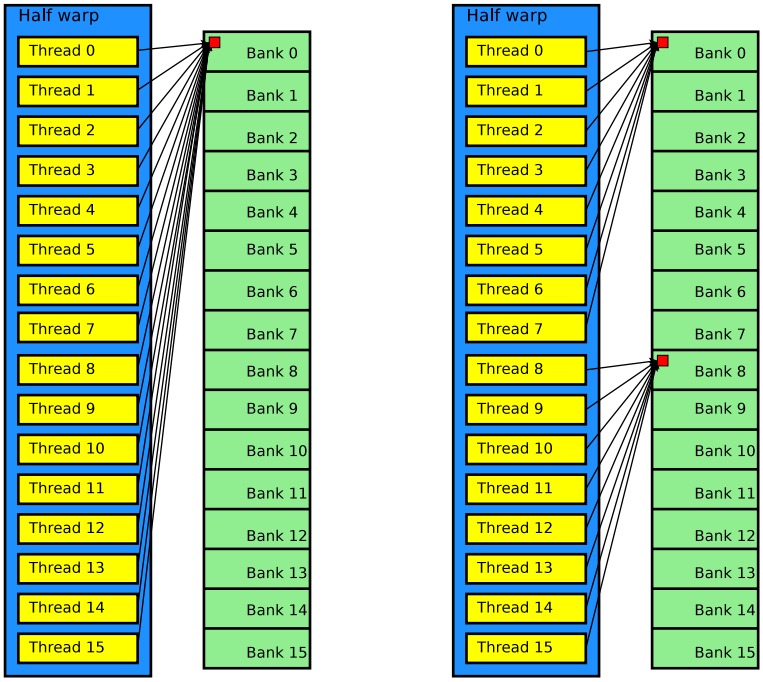


Figure 3.6: An example of how the shared memory broadcast mechanism works: The left access pattern will lead to one single memory transaction, while the right access pattern will lead to two sequential transactions.

When no bank conflicts occur, accessing shared memory is as fast as accessing registers.

3.6.3 Registers

Registers are the fastest memory (together with shared memory with no conflicts). When using registers, an instance of a variable is made for every thread. This means that when in a kernel n register variables are used, the total number of registers used per block will be $n \times m$, where m is the number of threads per block. It is important not to exceed the total number of registers of 8192 or 16384, otherwise the kernel will not launch. It is possible to make sure the compiler uses a maximum number of registers for the kernel. This is achieved by mapping some variables to the global memory (which in this context is called the 'local memory'). Although this might lead to more active threads, it costs a lot of extra time to access these variables.

Registers are used for variables of standard types which are declared inside the kernel. Small structures and arrays are also mapped to registers. To avoid register memory bank conflicts, the best results are achieved when the number of threads per block is a multiple of 64. These conflicts are resolved by the compiler, and the programmer has no influence on it.

3.7 Timing

To measure the performance of a certain implementation, time measurements have to be taken. Both the host and the device can be used to do the timing. When using CUDA for timing, we have to work with CUDA-events. Although this might be a bit more accurate, the differences are small. This is the reason that the timing mechanism of the host is used.

To get the most accurate results, the `gettimeofday()` function, from the <sys/time.h> header is used. This function returns the current time, in seconds and microseconds. Timing can be achieved by simply taking the difference of two different results of `gettimeofday()`.

The resolution of the measurements is usually about 10 ms, but accurate information cannot be given without getting in too much detail. For example the resolution is both dependent on the hardware, and the software platform running on. For our purposes, a resolution of 10 ms is sufficient, and we will not get into more detail.

For accurate timing it is necessary to realize that some CUDA functions behave asynchronous. For example, when a kernel is invoked, the CPU continues executing the next statements, and will not wait for the kernel to finish. This can be solved by invoking the function `cudaThreadSynchronize()`. This function will not return until all previous CUDA calls are finished. So before taking any time measurement, `cudaThreadSynchronize()` should be called. In particular before the first measurement, since the CUDA engine might be performing certain startup routines in the background.

3.8 Accuracy

Floating point accuracy is an important concept in scientific computing. Calculations have to be performed accurate enough, and rounding errors should not build up, resulting in significant differences. One important consequence of programming on the GPU is that floating point operations can produce slighly different results than on the CPU. This is a consequence of different methods of rounding off, for example because this might save space on the chip, making them cheaper. To make sure the results are accurate enough, each experiment run on the GPU is also run on the CPU, and the results are compared. This comparison is done by calcuting the relative norm of the difference vector of the two results. So we check if

$$\frac{\|\mathbf{u}_{\text{CPU}} - \mathbf{u}_{\text{GPU}}\|}{\|\mathbf{u}_{\text{CPU}}\|} < \epsilon,$$

for some small $\epsilon \approx 10^{-5}$. This also validates the algorithms on the GPU.
We will not get in more detail on rounding off differences.

Chapter 4

Matrix-vector products

In Section 2.3 we see that the normal contact problem is solved using the iterative conjugate gradients method. During the execution of this method a large matrix-vector product has to be calculated during each iteration. This matrix has a special geometric structure, which results in the matrix B. See Figure 2.2 for the structure of this matrix.
In this chapter we will investigate a number of implementations of this matrix-vector product, using CUDA. Because a lot of calculations can be performed independent of each other, a huge speed-up might be possible when using the GPU. The first implementation is a direct port of the CPU version, using only global memory. Then we will use shared memory, to gain performance. All the implementations are written in the C programming language, with the CUDA extensions.
During the experiments, the performance is given by the number of floating point operations executed divided by the execution time. The number of floating point operations executed during a matrix-vector product is the number per inproduct, multiplied by the number of rows. The total number of floating point operations per inproduct is $2 \cdot n$, we have n multiplications, followed by n additions. The number of rows is also n, bringing the total number of floating point operations to $2n^2$.

4.1 Direct implementation

In this section we will derive an algorithm for calculating the matrix vector product $\mathbf{u} = A\mathbf{p}$, in terms of the matrix B. This algorithm will be implemented on both the CPU and GPU.
From Equation 2.9 we have

$$B_{(kx,ky)} = A_{(ix,iy),(jx,jy)},$$

with $kx = jx - ix$ and $ky = jy - iy$. As stated earlier, we have $kx \in \{-(nx - 1), ..., 0, ..., nx - 1\}$ and $ky \in \{-(ny - 1), ..., 0, ..., ny - 1\}$.
We can now rewrite Equation 2.8 in terms of B, resulting in

$$u_I = \sum_J A_{IJ} p_J \quad \Rightarrow \quad \begin{array}{rcl} u_{(ix,iy)} & = & \sum_{jy} \sum_{jx} A_{(ix,iy),(jx,jy)} p_{(jx,jy)} \\ & = & \sum_{jy} \sum_{jx} B_{(kx,ky)} p_{(jx,jy)}, \end{array} \tag{4.1}$$

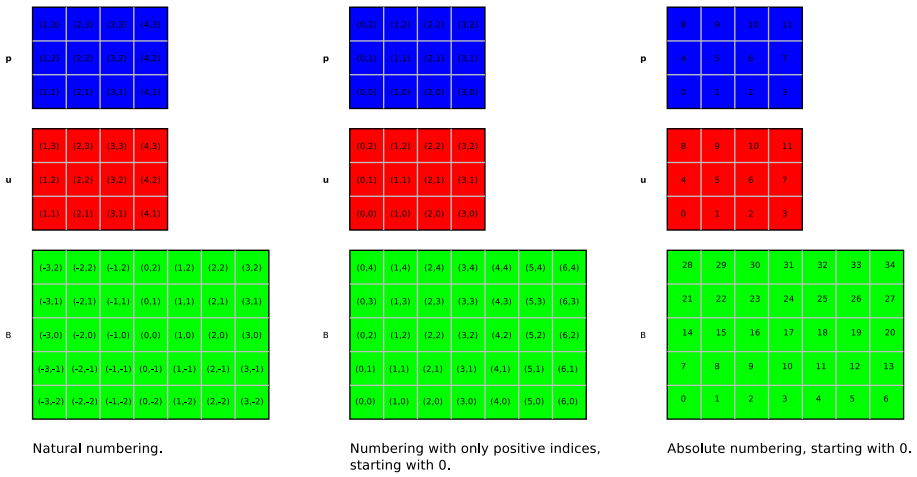


Figure 4.1: The different ways of numbering, for a grid of size 4×3.

with $kx = (jx - ix)$ and $ky = (jy - iy)$.
When implementing in C, we want positive indices, starting with 0. For ix, iy, jx we simply subtract 1 from the values. We then have $ix' = ix - 1$, $iy' = iy - 1$, $jx' = jx - 1$ and $jy' = jy - 1$, with $ix, jx \in [0, nx - 1]$ and $iy, jy \in [0, ny - 1]$. For kx and ky, we add an offset $nx - 1$ to kx and an offset $ny - 1$ to ky, resulting in

$$kx' = kx + (nx - 1), \text{ with } kx' \in \{0, ..., nx - 2\} \text{ and}$$
$$ky' = ky + (ny - 1), \text{ with } ky' \in \{0, ..., ny - 2\}.$$

From now on, we will always use this zero-based indexing, unless states otherwise.
Because we use linear (one-dimensional) arrays in C, we have to transform the double indices to a single index. In the arrays, the horizontal lines of $\mathbf{u}$, $\mathbf{p}$ and B will occupy consecutive memory locations. This results in an absolute index $kk = ky' \cdot bx + kx'$ for B, and $ii = iy \cdot nx + ix$ for $\mathbf{u}$ and $\mathbf{p}$. See Figure 4.1 for an example of the different ways of labeling the grid points. Algorithm 4.1 gives the resulting algorithm for calculating $\mathbf{u} = A\mathbf{p}$.
This algorithm can be implemented in C directly. For the source code, see Appendix A.

We can now port this algorithm to the GPU. In doing so, we have to choose how we divide the total work over the different blocks and threads. One possible method to proceed, is to divide $\mathbf{u}$ into rectangular tiles, and let each block calculate the values of $\mathbf{u}$ in one specific tile. Each thread in a block can then calculate a specific element of $\mathbf{u}$. We denote the size of a tile by tx and ty. The total number of tiles is denoted by Tx and Ty. See Figure 4.2 for an illustration of this division. Here the grid has size 8×4, and is divided into 4×2 tiles each of size 2×2. So we have $nx = 8, ny = 4$, $Tx = 4, Ty = 2$ and $tx = ty = 2$. Of course the dimensions of a tile must be chosen in such a way, that the whole grid can be tiled.
When using a division like this, each tile of $\mathbf{u}$ is calculated by a different block,

Algorithm 4.1 Algorithm to calculate $\mathbf{u} = A\mathbf{p}$ in terms of B.

```
    for iy = 0 to ny − 1 do
      for ix = 0 to nx − 1 do
        ii = iy · nx + ix
        u_ii = 0
 5:     for jy = 0 to ny − 1 do
          ky = jy − iy + ny − 1
          for jx = 0 to nx − 1 do
            kx = jx − ix + nx − 1
            jj = jy · nx + jx
10:         kk = ky · bx + kx
            u_ii = u_ii + B_kk · p_jj
          end for
        end for
      end for
15: end for
```

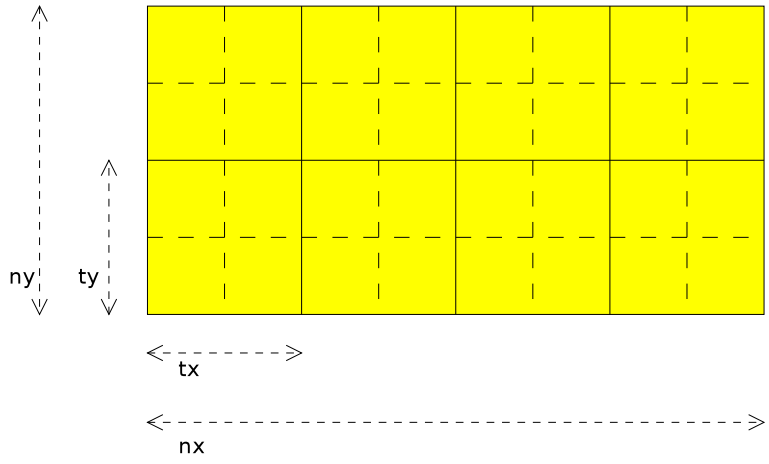


Figure 4.2: Example of a division in tiles of the grid.

and in each tile, each component of **u** is calculated by a different thread. Since the threads are executed in parallel, we do not need the two outer-most loops of Algorithm 4.1. Each thread calculates its unique ix, iy and ii values and proceeds with the inner-loops. This way each iteration of the outer-most loop of Algorithm 4.1 corresponds to a different block, and each iteration of the first nested loop corresponds to a different thread in a particular block.
Algorithm 4.2 gives the steps that need be to performed by each thread when using a tiling like this.

Algorithm 4.2 Algorithm when using tiles to divide **u**. This algorithm is executed by each thread in the grid.

$iy = \texttt{blockIdx.y} \cdot \texttt{blockDim.y} + \texttt{threadIdx.y}$
$ix = \texttt{blockIdx.x} \cdot \texttt{blockDim.x} + \texttt{threadIdx.x}$
$ii = iy \cdot nx + ix$
$u_{ii} = 0$
for $jy = 0$ to $ny - 1$ **do**
  $ky = jy - iy + ny - 1$
  for $jx = 0$ to $nx - 1$ **do**
    $kx = jx - ix + nx - 1$
    $jj = jy \cdot nx + jx$
    $kk = ky \cdot bx + kx$
    $u_{ii} = u_{ii} + B_{kk} \cdot p_{jj}$
  end for
end for

When stating an algorithm like this for the GPU, it is executed by each thread of all blocks. We assume that all threads in a block simultaneously execute a particular line of code. This way memory synchronization problems cannot occur. It is important to be aware of the fact that when implementing an algorithm in CUDA the lines of code are not executed simultaneously, but in warps, and we have to use the `__syncthreads()` instruction for synchronization. This is particularly important when using shared memory.
This algorithm is implemented on both the CPU and the GPU, where on the CPU the tiling has no effect and we just iterate over ii and jj.
On the CPU we expect more or less equal speeds, regardless the size of the grid. There might be deviations when using small grids, due to memory latencies. For smaller grids, which allow the whole matrix A to be stored in the host memory, the matrix vector product is calculated using both A and B. This shows the effect of the extra addressing when using B.
On the GPU the size of the tiles is important. This is because of memory coalescing and thread divergence. If the number of grid points in a block is not a multiple of 32, we expect less performance because of warps that are not filled completely. On the other hand, we expect the number of memory transactions to be much higher than the number of arithmetic instructions, and for memory transactions only half-warps are important. So we expect the difference between tiles with $tx = 32$ and $tx = 16$ to be less then the difference between tiles with $tx = 16$ and $tx = 8$. If we chose tx to be a multiple of 32 none of these problems

arise.
As mentioned earlier, when threads in a half-warp access consecutive locations in the global memory, this will result in a single memory transaction. When using the tiling described earlier, for memory coalescing, threads in the same half-warp, with consecutive `threadIdx.x` indices have to access consecutive memory locations. Also the first thread should access a memory location on a 64-byte boundary. Assuming that in Algorithm 4.2 the arrays B, p and u are aligned on such a boundary, we can see that the write operations to u are always coalesced if `blockDim.x` is a multiple of 16, but the read operations from B and p are not, in general.
In Section 4.3 the results of these implementations will be discussed.

4.2 Using shared memory

In the previous section only the global memory was used. This resulted in a lot of non-coalesced memory transactions. We will now investigate the use of shared memory to gain higher performance. The total amount of shared memory per multiprocessor is 16 KiB, or 4096 4-byte floating point numbers. We must make sure not to exceed this amount per block, or else the kernel will fail to launch.
When using shared memory, a lot of addressing has to be performed. This means the threads need to perform a lot of integer arithmetic, not contributing to the overall floating-point performance. As we will see in Section 4.3, the size of the tiles in the y direction does not have much influence on the performance. For these reasons, we will use tiles with only one grid point in the y direction. The number of grid points in the x direction should be a multiple of 32 to make sure it can be divided in a particular number of warps. We might want to maximize the size in the x direction, for maximum use of the shared memory.

We define Tx as the number of tiles in the x direction, and tx as the size of the tiles in the x direction. Again we choose $kx = jx - ix + nx - 1$ and $ky = jy - iy + ny - 1$. Furthermore we have $jx = jTx \cdot tx + jtx$ and $ix = iTx \cdot tx + itx$. We can now rewrite Equation 4.1 to

$$u_{(ix,iy)} = \sum_{jy=0}^{ny-1} \sum_{jx=0}^{nx-1} B_{(kx,ky)} p_{(jx,jy)} = \sum_{jy=0}^{ny-1} \sum_{jTx=0}^{Tx-1} \left(\sum_{jtx=0}^{tx-1} B_{(kx,ky)} p_{(jx,jy)} \right). \quad (4.2)$$

Each block calculates one tile of $\mathbf{u}$ and each thread calculates one element of $\mathbf{u}$. This means we can use a register variable for this value of $\mathbf{u}$. For $\mathbf{p}$ and B we will use shared memory. Because shared memory is scarce we cannot load the complete arrays at once, and we have to load tiles of B and $\mathbf{p}$ during the kernel execution. We denote the regions of shared memory by B' and p'. We will focus on the inner-most loop of Equation 4.2.
For particular tiles of $\mathbf{u}$ and $\mathbf{p}$, that is, for particular values of iTx and jTx, we have

$$jx \in [jTx \cdot tx, (jTx + 1) \cdot tx - 1],$$
$$ix \in [iTx \cdot tx, (iTx + 1) \cdot tx - 1].$$

For kx we then have

$$kx \in [(jTx - iTx - 1) \cdot tx + nx, (jTx - iTx + 1) \cdot tx + nx - 2].$$

As expected, when calculating the contribution of a specific tile of $\mathbf{p}$ to a tile of $\mathbf{u}$ we need tx values of p and $2tx - 1$ values of B. We can now define new indices

$$\begin{cases} ix' = (iTx + 1) \cdot tx - ix - 1 \\ jx' = jx - jTx \cdot tx \\ kx' = kx - (jTx - iTx - 1) \cdot tx - nx, \end{cases}$$

so that we have

$$\begin{aligned} ix' &\in [0, tx - 1] \\ jx' &\in [0, tx - 1] \\ kx' &\in [0, 2tx - 2]. \end{aligned}$$

Furthermore, since $kx = jx - ix + nx - 1$ we have $kx' = ix' + jx'$ and $jx' = jx - jTx \cdot tx = jTx \cdot tx + jtx - jTx \cdot tx = jtx$. If we now define

$$\begin{cases} u' = u_{(ix,iy)} \\ p'_{jx'} = p_{(jx,jy)} \\ B'_{kx'} = B_{(kx,ky)}, \end{cases}$$

Equation 4.2 can be rewritten as

$$u' = \sum_{jy=0}^{ny-1} \sum_{jTx=0}^{Tx-1} \left(\sum_{jx'=0}^{tx-1} B'_{kx'} \, p'_{jx'} \right). \tag{4.3}$$

Algorithm 4.3 states how we can calculate the matrix-vector product using the shared memory. Remark that we perform one assignment for p', but two for B'. This is because we need to load one tile of $\mathbf{p}$ but two tiles of B.
When implementing this in CUDA, we will use a block to attain a particular value of iTx, and the threads of that block to attain the values of itx. Each thread will then loop over jy, jTx and jx'. The tiles of p and B are read from the global memory on the beginning of the loop over jTx, just before the values are needed. When reading from global memory, each thread will read one value. Of course after the read instructions, a synchronization is needed to make sure all threads read and stored their values. A synchronization is also needed before reading the new tiles from the global memory, to make sure all the previously read values are used.
When tx is a multiple of 16, the read operation of p will coalesce, but special care is needed when reading B. We will read $2tx$ values for each value of jTx, but because the size of B in the x-direction is $2nx-1$, for values of $iy \neq 0$, these read operations will in general be non-coalesced, since they do not start at a 64-byte boundary. Furthermore, when the very last tile of B is read, this will result in reading non-allocated memory location. This is of course always a bad idea. The common solution for these two problems, is to add a dummy value at the end of each row of B. thereby changing its dimensions to $2nx \times (2ny - 1)$

Algorithm 4.3 Algorithm when using shared memory.

$iy =$ `blockIdx.y`
$iTx =$ `blockIdx.x`
$ix' =$ `threadIdx.x`
$u' = 0$
for $jy = 0$ to $ny - 1$ **do**
  $ky = jy - iy + ny - 1.$
  for $jTx = 0$ to $Tx - 1$ **do**
    $kx = (jTx - iTx - 1)tx + nx +$ `threadIdx.x`
    $kx' =$ `threadIdx.x`
    $B'_{kx'} = B_{(kx,ky)}$
    $B'_{kx'+tx} = B_{(kx+tx,ky)}$
    $jx = jTx \cdot tx +$ `threadIdx.x`
    $jx' =$ `threadIdx.x`
    $p'_{jx'} = p_{(jx,jy)}$
    $ix' =$ `threadIdx.x`
    for $jx' = 0$ to $tx - 1$ **do**
      $kx' = jx' + ix'$
      $u' = u' + B'_{kx'} \cdot p'_{jx'}$
    end for
  end for
end for
$ix = (iTx + 1)tx - 1 -$ `threadIdx.x`
$ix' =$ `threadIdx.x`
$u_{(ix,iy)} = u'$

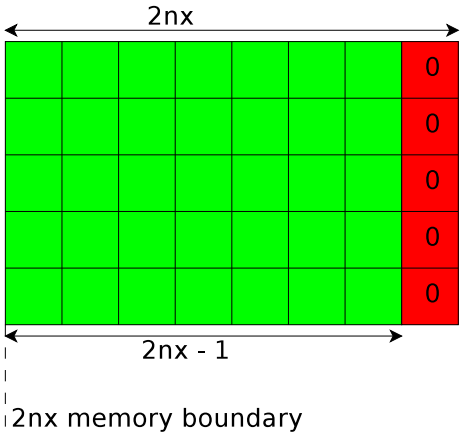


Figure 4.3: A dummy entry is padded to each row of B.

instead of $(2nx-1)\times(2ny-1)$. Of course when accessing a specific row of B, we should now stride by $2nx$ values instead of $2nx-1$ values. Figure 4.3 shows what changes when $nx=4$ and $ny=3$. The red entries are read, but never used for calculations. The dotted line shows where the $2nx$ boundaries are. We see that each tile will start on a tx boundary, so read transactions will coalesce if tx is chosen properly. Again the source code can be found in Appendix A.

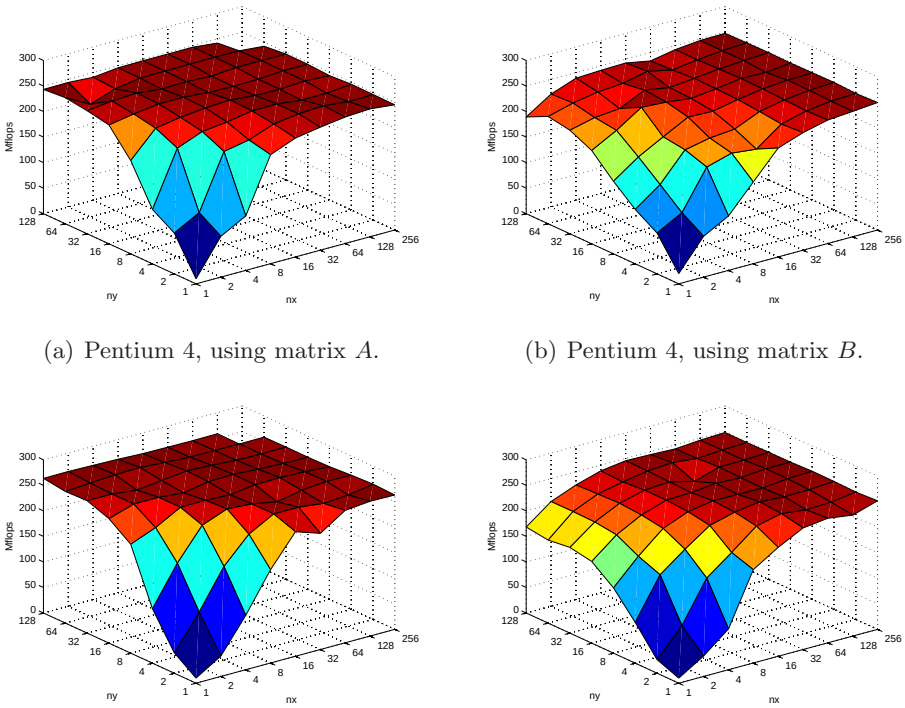
4.3 Results

For the results each implementation is run on the specified system. For each point in the dataset, the corresponding implementation is run 5 times, and the average runtime is used. The GPU on which all the implementations are tested is a NVIDIA Quadro FX 570M, consisting of 256 MiB global memory, and 4 multiprocessors. The clock rate is 0.95 GHz. The device has compute capability 1.1.[1] The system is running Ubuntu 9.04, and CUDA 3.0 is used.
The CPU's on which the direct implemenation is tested, are the Intel Core 2 Duo T7300 2.00 GHz, running Ubuntu 9.04, and the Intel Pentium 4 3.40 GHz, running Ubuntu 10.04.

4.3.1 Direct implementation

The direct implemtation is run on both the CPU and the GPU. Figure 4.4 shows the overall computation speed, expressed in Mflops for different grid sizes. We see that in both cases the performances is higher when using matrix A. This is because less integer arithmetic is needed for addressing. Both figures show that with growing grid sizes, the performance quickly converges to a maximum, as we expected. The maximum performance achieved is 252 Mflops.
When we use the GPU for the direct implementation, we can vary both the grid size and the tile size. Figure 4.5 shows the results for a grid of 256×256,

[1] See Section 3.5 for more information.



(a) Pentium 4, using matrix A.

(b) Pentium 4, using matrix B.

(c) Core 2 Duo T7300, using matrix A.

(d) Core 2 Duo T7300, using matrix B.

Figure 4.4: Performance of the direct implementation on the Pentium 4 and the Intel Core 2 Duo T7300.

for different tile sizes. Since a tile can consist of a maximum of 512 grid points, only values of tx and ty are valid with $tx \cdot ty \leq 512$. We see that when $tx \leq 8$ the performance is very low. This is because of the non-coalesced memory transactions. When $tx \geq 16$ the differences are small, but we see that the smaller tx and ty the higher the performance. This might be explained because when tiles are small, there are more blocks, and the hardware scheduler can assign more block per multiprocessor. This leads to better results, because during the time one active block is waiting for some memory transaction to be completed, the multiprocessor can perform work for another active block. The maximum performance achieved is 555 Mflops. This is about twice the speed when using the direct implementation on the CPU.

4.3.2 Shared Memory

In Figure 4.6 we see that using the shared memory implementation, the performance is much higher. The grid size in the x direction is 256 grid points, divided into tiles of size tx. The grid size in the y direction, ny, is varying from 1 to 32. We use tile size $ty = 1$ in the y direction. We see two distinct effects. For small values of ny the performance is low, but grows very quickly as ny grows, to a maximum. Furthermore, for small values of tx, the performance is very low, but grows quicly when tx gets bigger, and converges to a maximum

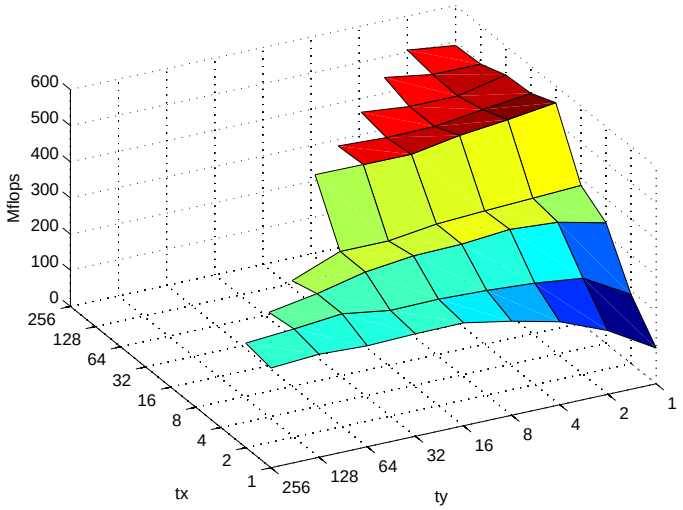
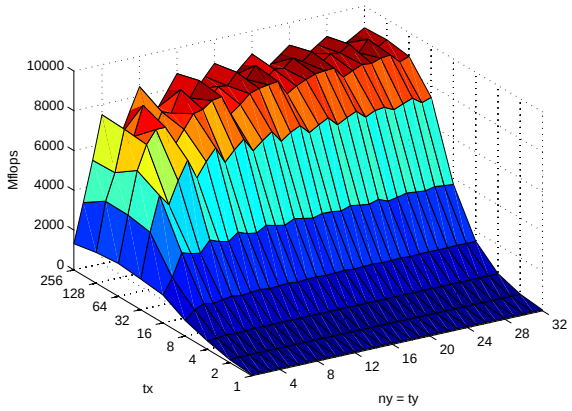
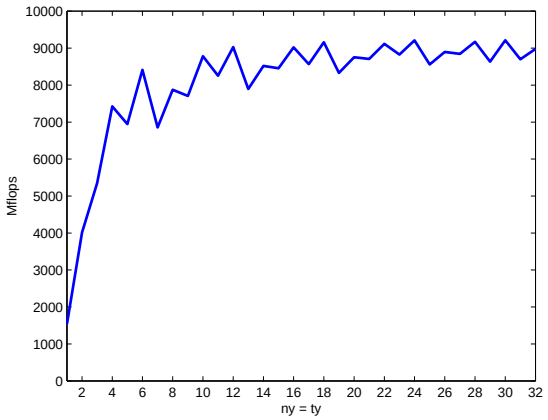


Figure 4.5: Results for grid size 256×256 and different tile sizes.

for $tx \geq 64$.
That the performance is low when ny is small, can be explained by the fact the when tx is small, the total number of tiles, and thus blocks, is small, and as we saw earlier, this limits the performance. Furthermore, when ty is odd, we see the performance drops a bit, which is probably caused because the total number of multiprocessors is even, so the last block has to run on a multiprocessor with the other multiprocessors idle. Here we assumed the multiprocessors are finished at the same time with the other blocks, which in general might not be true, but deviations are likely to be small compared to the time it takes to complete one block.
The rise in performance when tx gets bigger, can be explained by two reasons. First of all, For $tx < 32$ we have blocks consisting of less threads then 32, thus less then one warp. This decreases the performance tremendously. Furthermore, by looking at Algorithm 4.3 we see that the bigger tx, the more shared memory is used, and the less memory bandwidth is an issue. We see that for $tx \geq 64$ the performance is more or less stagnant.
Figure 4.7 shows that for $nx = 256$, ny has no significant influence on the performance if it is taken a multiple of 32. In the figure different grid sizes are shown, with ny multiples of 32. The tiles have size 1 in the y direction. The maximum performance achieved is 9305 Mflops, for $tx = 128$. This is a factor 17 higher then in the direct implementation on the GPU. We also see that for $tx \leq 32$ performance significanlty increases for growing tiles. This is the effect of warps being more and more filled, until we have full warps for $tx = 32$.
Finally, Figure 4.8 shows that the value of nx does not have a significant influence on the performance. This, together with the observation that ty does not have much influence either, makes us expect that the implementation of the algorithm performs well when the grid size is scaled up in either direction.



(a) Performance for ny varying from 1 to 32.

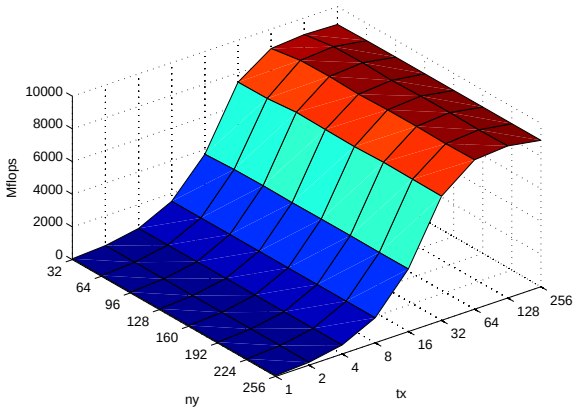


(b) Performance for different grid sizes in y direction, for $tx = 128$.

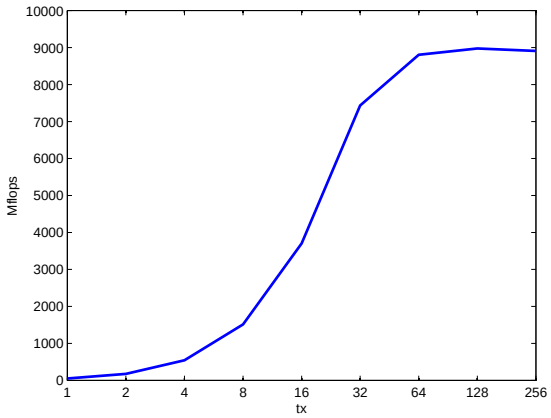
Figure 4.6: Results for shared memory implementation, using grid size $256 \times ny$ with $ty = 1$ and tx varying.

The tile size does have a significant influence, and tiles should be scaled up as much as posible.

We have seen that when only using the global memory, the tile sizes are not very important, as long as they are larger then the size of a warp. This is a consecuence of memory coalescence. When using shared memory the tile size is of great influence on the total performance. In general the tile size should be relatively large, so that the use of shared memory is maximized. This results in a performance gains of a factor 17 compared to the global memory implementation, and a gain of a factor 37 compared to the CPU implementation.



(a) Performance for different tile sizes $tx \times 1$ for growing grid in y direction.



(b) Performance for different tile sizes $tx \times 1$ when $ny = 32$.

Figure 4.7: Influence of different tile sizes $tx \times 1$ and grid sizes in the y direction, with ny varying from 32 to 256 and $nx = 256$.

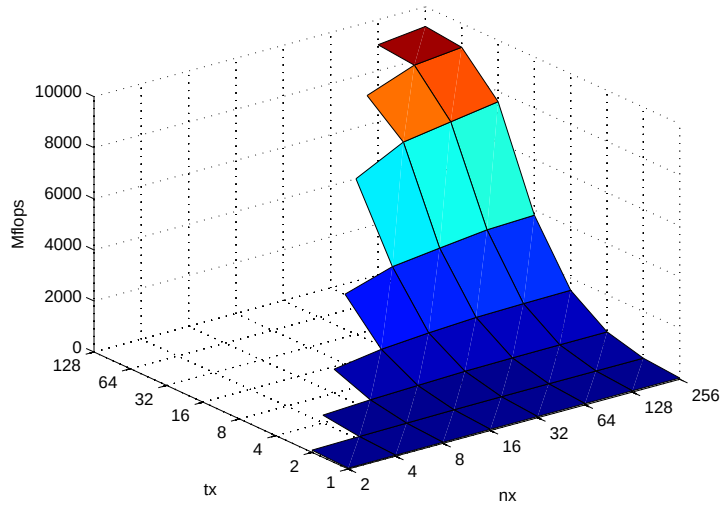


Figure 4.8: Influence of growing grid size in the x direction, for different tile sizes $tx \times 1$. $ny = 128$.

Chapter 5

Inner products

As stated in Section 2.4, during each iteration of the tangential algorithm, four inner products have to be calculated. In general there are four matrices of influence coefficients, A^{xx}, A^{xy}, A^{yx} and A^{yy}. During each iteration of the tangential algorithm, one value of s_x and one value of s_y are updated. With these new values of s_x and s_y, a system of equations is solved, resulting in new values of the corresponding elements of p_x and p_y. See Figure 5.1 for an illustration of the inner products calculated.

During a specific iteration, the inner products of the red rows of the A matrices with the $\mathbf{p}$ vectors are calculated, and the red values of the $\mathbf{W}^*$ vectors are added. This results in the two new blue elements of the $\mathbf{s}$ vectors. Then the second part of the algorithm calculates the new dark greens elements of the $\mathbf{p}$ vectors. All four A matrices have the translational invariance explained in Chapter 2, so we have four associated B matrices. Figure 5.2 visualizes the calculations using the B matrices. For simplicity the grids of the $\mathbf{W}^*$ vectors and $\mathbf{s}$ vectors are omitted.

Since after each iteration new values of $\mathbf{p}$ are calculated, the inner products of different iterations cannot be calculated parallel, but have to be calculated in a consecutive order. On the GPU this means a significant drawback of performance, since we can only perform four inner products simultaneously. In

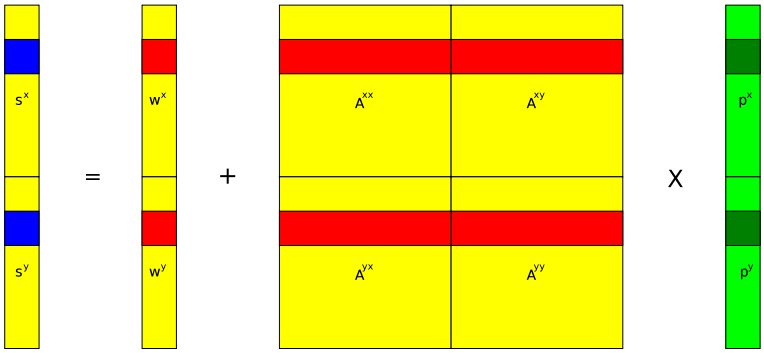


Figure 5.1: Inner products in terms of matrices A.

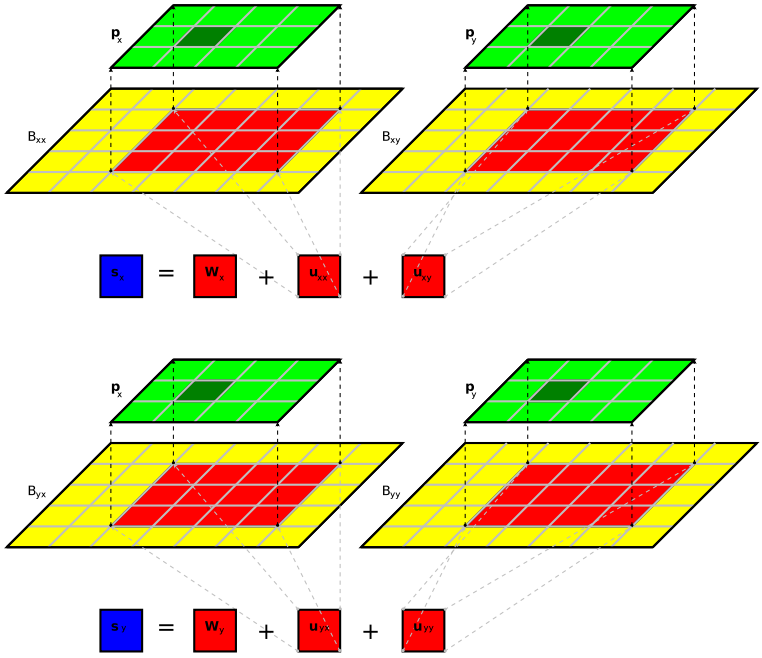


Figure 5.2: Inner products in terms of matrices B.

the next sections of this chapter we will consider implementations on the CPU and the GPU, and the results will be stated.
To determine the performance of a specific implementation, we need the total number of floating point operations executed.[1] Per inner product we have to perform $n = nx \cdot ny$ multiplications, and the same number of additions, so the total number of floating points operations per inner product is $2n$. Per iteration we calculate 4 inner products, and finally we add two inner products, and the corresponding element of $\mathbf{W}^*$ together for each direction x and y, so the total number of floating point operations, per iteration is given by $8n + 6$. During the experiments, we measure the time it takes to perform the inner products for each element, so the total number of iterations will be n, bringing the total number of floating point operations to $8n^2 + 6n$.
During the program execution, this whole process will be repeated a given number of times, to make sure background processes do not influence the outcome significantly. The actual number of times an experiment is run will be stated at the results.

5.1 CPU Implementation

To derive an algorithm to calculate a specific inner product, we use the same indexing as in Chapter 4, so we have ix and iy for elements of the $\mathbf{s}$ and $\mathbf{W}^*$ vectors, jx and jy for elements of the $\mathbf{p}$ vectors and kx and ky for elements of

[1] Remark that this is the theoretical number of operations. In reality more operations might be performed, but they do not count as 'useful' operations

the B matrices. Again we have $kx = jx - ix + nx - 1$ and $ky = jy - iy + ny - 1$, and all indices are zero-based, so for example $ix \in [0, nx - 1]$. From Equation 2.18 we have

$$\begin{cases} s_{Ix} = W^*_{Ix} + \sum_J B^{xx}_K \, p_{Jx} + \sum_J B^{xy}_K \, p_{Jy} \\ s_{Iy} = W^*_{Iy} + \sum_J B^{yx}_K \, p_{Jx} + \sum_J B^{yy}_K \, p_{Jy}, \end{cases}$$

If we concentrate on one inner product, we get the slip distance in one particular direction due to tractions in one particular direction. If we call this distance s' and omit the direction labels, we can write

$$s' = \sum_J B_K \, p_J = \sum_{jy=0}^{ny-1} \sum_{jx=0}^{nx-1} B_{(kx,ky)} \, p_{(jx,jy)}. \tag{5.1}$$

Using this equation we can derive an algorithm to perform the calculations on the CPU. Since the CPU works sequential, the four inner products are calculated independent of each other. Algorithm 5.1 shows the most direct method. In the experiments, this algorithm is implemented as a seperate function.
ix and iy are constant for a specific inner product. Of course during each iteration the algorithm is executed four times, with matrices B^{xx}, B^{xy}, B^{yx}, and B^{yy}, and the vectors $\mathbf{p}_x$ and $\mathbf{p}_y$. Finally, the contributions in a particular direction are added, and the appropriate value of $\mathbf{W}^*$ is added.

Algorithm 5.1 Direct algorithm to perform inner product.

```
s' = 0
for jy = 0 to ny − 1 do
  ky = jy − iy + ny − 1
  for jx = 0 to nx − 1 do
5:   kx = jx − ix + nx − 1
     s' = s' + B_(kx,ky) · p_(jx,jy)
  end for
end for
```

5.2 GPU Implementation

In the previous section it did not really matter that inner products of different elements cannot be performed parallel, since the CPU executes them sequentially, but on the GPU this means a major drawback. After the four inner products of a particular element are calculated, and the two new values of the **s** vectors are calculated, the second part of the tangential algorithm is executed on the CPU, calculating the new values of the **p** vectors. At this moment, CUDA does not offer a mechanism to synchronize between the kernel and the host thread. This means that our kernel can only perform four inner products and then has to terminate. Control is given back to the host thread, and the results of the kernel are used to calculate the new values of the **p** vectors. When the host thread is finished, the same kernel is launched again to calculate the

next four inner products. Since shared memory is not persistent between different kernel instances (in fact it is persistent during the live-time of one particular block only), we have to reinitialize it for each instance, making shared memory less useful as in the matrix-vector product implementations.

The global memory is persistent during different kernel instances, so we do not need to copy all the data to the device each time a kernel is launched, only the new values of $\mathbf{p}$ have to be updated.

When we port Algorithm 5.1 to the GPU, we have to chose a division of the elements in blocks and threads, and how to divide the four inner products over different blocks. As in Chapter 4, we will split the $\mathbf{p}$ vectors in rows, so the tile size in the x direction is equal to the grid size in the x direction. This way the tiles consist of a specific number of rows of $\mathbf{p}$. Remark that we implicitely assume that the grid is small enough in the x direction, that is, that $nx \leq 512$, otherwise we have to break the rows down in different tiles. This is necessary because the maximum number of threads per block is 512.

When calculating an inner product, we can perform the multiplications parallel, since they do not depend on each other. After these multiplications are performed, the results need to be added. This is called a 'sum reduction'. There are several strategies to perform such a reduction, with different performance. The fact that threads are executed in warps becomes very important when performing a sum reduction. When not all the threads in a warp perform a (useful) additions, performance is lost. In the next section we will consider different sum reduction methods and compare them to each other.

If the four inner products are calculated independent of each other, we need to perform four sum reductions. A better strategy is to perform the inner products corresponding to the same direction simultaneously, and add the individual elements along the way. This way we only need to reduce two sums, one for each direction.

During execution of the kernel, the results of the two multiplications are added and stored in shared memory. The resulting array of values is then reduced to a single value. In the algorithm the array of shared memory is called t and has nx elements. The final result of the reduction is stored in t_0. Since each block only processes a number of rows of the $\mathbf{p}$ vectors, we end up with an array r consisting of Ty values, where Ty is the number of tiles. This array needs to be reduced further to the final result. This last reduction can be performed on the GPU, with a kernel designated to just reduce an array, or it can be reduced by the host thread, thereby omiting the overhead of launching a kernel. Both possibilities are investigated.

When we have the final results of the inner products, we still need to add the appropriate value of $\mathbf{W}^*$. This is done by the host thread, since it consists of two single additions, one for each direction.

A final important difference with the full matrix-vector multiplication is that we now only need a specific part of the B matrices. The only elements we need are the red elements from Figure 5.2. This means we can change the indices kx and ky to $kx' = -ix + nx - 1$ and $ky' = -iy + nx - 1$, so that

$(kx', ky') = (0, 0)$ corresponds to the 'first' element needed.[2] This way the host thread can calculate this adress one time for each B matrix, and give it as an argument to the kernel. The fact that only a particular part of the B matrices is needed brings a problem with it. In general this sub-matrix will not be aligned on a 16-byte boundary, so memory read transactions will in general be non-coalesced. For devices of device capability 1.0 and 1.1 this will result in 16 memory transaction instead of one single transaction for each warp reading from the global memory. Devices of higher compute capability will perform only two memory transaction, so the problem is less significant. See Section 3.6 for more information on memory coalescence for different compute capability. In the experiments we will look at the effect of these non-coalesced memory transactions, and the difference in performance will be determined. To achieve coalesced memory transactions, we simply use the lower-left part of matrix B for each inner product. Since the matrix B is alligned at a 16-byte boundary, this will result in coalesced memory transactions. Of course these experiments will produce incorrect outcomes, but we are only interested in the performance. When we consider the x direction, Algorithm 5.2 gives an example how this can be implemented. Again, ix and iy are constant. Ty is the number of tiles and ty is the tile size in the y direction, so ty is the number of rows of $\mathbf{p}$ per tile. The results of this algorithm can be found in Section 5.3.

Algorithm 5.2 Algorithm when performing two inner products.

$Ty =$ `blockIdx.x`
$jy = Ty \cdot ty +$ `threadIdx.y`
$jx =$ `threadIdx.x`
$jj = jy \cdot nx + jx$
$kk' = jy \cdot stride + jx$
$t_{jx} = B^{xx}_{kk'} \cdot p_{jj,x} + B^{xy}_{kk'} \cdot p_{jj,y}$
`reduction(...)`
if $jx = 0$ **then**
  $r_{jy} = t_0$
end if

5.2.1 Sum reduction

The algorithm in the previous section is not complete, we still need to perform the sum reduction. There are several ways to perform a sum reduction, each with its own performance, in [2] two ways are considered.
The first method follows the pattern of Figure 5.3. The biggest drawback is that this pattern results in highly divergent warps, as explained in Section 3.3. During the first iteration, half the threads in a warp do not perform an addition, and after each iteration, this number doubles. This means a lot of potential performance is wasted. Algorithm 5.3 shows how to implement this reduction method. Here t is the thread index and n is the total number of threads.

[2] The elements in the lower left corner of the red rectangles in Figure 5.2.

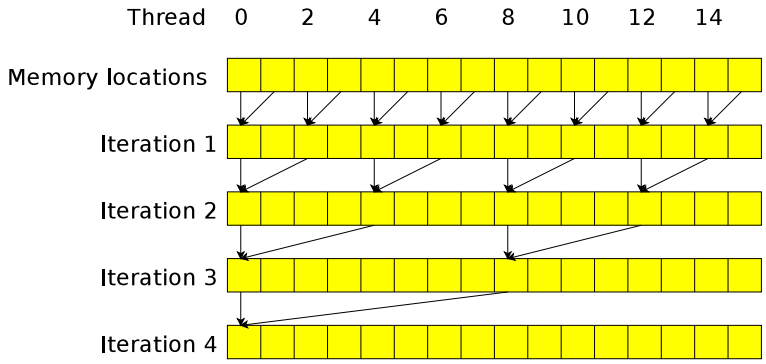


Figure 5.3: Sum reduction with divergent warps.

Algorithm 5.3 Algorithm for sum reduction, with divergent warps.

$s = 1$
while $s < n$ **do**
  if $t \bmod 2s = 0$ **then**
    $N_t = N_t + N_{t+s}$
  end if
  $s = 2s$
end while

One way to tackle this problem is to use threads with consecutive indices instead of just the even ones to perform the reduction, as in Figure 5.4. This way thread divergence is minimized. This introduces another problem, bank conflicts. For example during the first iteration we see that thread 0 uses memory bank 0 and memory bank 1, while thread 8 will also use memory bank 0 and memory bank 1. In Figure 5.4 the blue and red memory locations illustrate the bank conflict. Since a memory bank can only serve one specific value at a time, the memory transactions will be serialized, resulting in a performance drawback. Remark that the total number of threads needed to perform the sum reduction is half the number of elements needed to be add.

Algorithm 5.4 Algorithm for sum reduction, with memory bank conflicts.

$s = 1$
while $s < n$ **do**
  $i = 2st$
  if $i < n$ **then**
    $N_i = N_i + N_{i+s}$
  end if
  $s = 2s$
end while

The third method to perform the sum reduction does not has divergent warps, or memory bank conflicts. When using the pattern of Figure 5.5, the number of threads involved in the reduction halves each iteration, and the 'active' threads

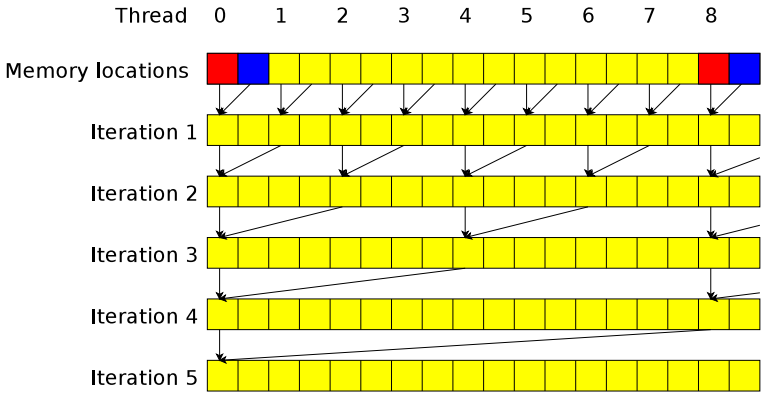


Figure 5.4: Sum reduction with bank conflicts.

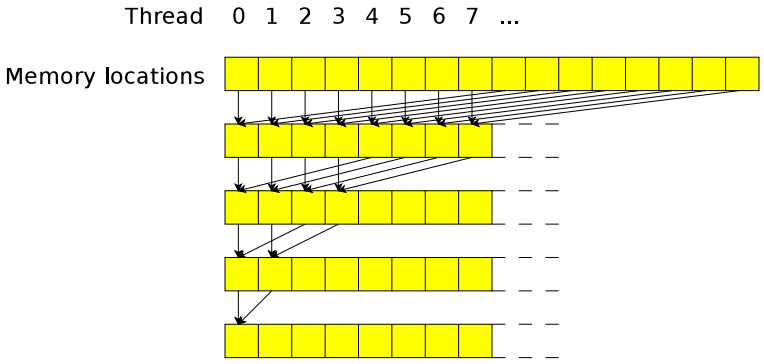


Figure 5.5: Sum reduction without warp divergence and bank conflicts.

are consecutive, belonging to the same warps. So only when the total number of 'active' threads comes under 32, this single warp will diverge. Using this method to reduce a sum, the total number of values needs to be a power of two. To perform a reduction with an arbitrary number of values, zeros can be padded until the total number of values is again a power of 2. Algorithm 5.5 gives a possible implementation of this method. n is the number of values to be added and N is an array containing these numbers. s is the number of 'active' threads, and t is the index of the current thread. The final answer will be N_0. As we see, all threads with $t \geq s$ will not perform an addition, but still all threads have to execute the while-loop and the conditional statement during each iteration. In the implementation there is also a `__syncthreads()` statement needed to make sure the additions are performed with the correct values. This results in a lot of instructions executed without any calculations performed. If we unroll the last iterations of the loop, one conditional statement can make sure threads belonging to warps that are no longer needed to perform calculations can be dismissed. Furthermore, when only one warp is active, the `__syncthreads()` statement is no longer needed. Algorithm 5.5 now changes to Algorithm 5.6. Remark that during these last 6 additions all threads in the

Algorithm 5.5 Algorithm for sum reduction without divergent warps and bank conflicts.

```
s = n/2
while s > 0 do
  if t < s then
    N_t = N_t + N_{t+s}
  end if
  s = s/2
end while
```

warp perform additions, but not all the results are used. It is just cheaper to perform useless calculations than to check whether or not to perform them. Remark that by unrolling the last 6 iterations, we assume $n \geq 64$, otherwise non allocated values of N are referenced. When the total number of values is smaller, we can just state all the additions explicitely, so for example for $n = 32$ this will result in just 5 additions.

Algorithm 5.6 Algorithm for sum reduction with enrolled iterations.

```
s = n/2
while s > 32 do
  if t < s then
    N_t = N_t + N_{t+s}
  end if
  s = s/2
end while
if t < 32 then
  N_t = N_t + N_{t+32}
  N_t = N_t + N_{t+16}
  N_t = N_t + N_{t+8}
  N_t = N_t + N_{t+4}
  N_t = N_t + N_{t+2}
  N_t = N_t + N_{t+1}
end if
```

Figure 5.6 shows the performance of the different reduction methods. Only 1 reduction is done per kernel launch, so only 1 multiprocessor is used. The time measurement is taken as the average over 10000 iterations. 'Fast' refers to the algorithm without warp divergence and memory bank conficts. Remark that when reducing n values, we perform n floating point operations. We see that the algorithm with the enrolled iterations is the fastest, and the performance is higher for greater values of n. For $n = 512$, the performance is 75.8 Mflops. Table 5.1 shows the different speeds and the relative performances of the different algorithms. In the experiments with the inner products we will use the fast reduction without the enrolled loop, to make sure it works correct with arrays of size $n < 64$.

As stated we only use 1 block, which is of course not very efficient. We will now

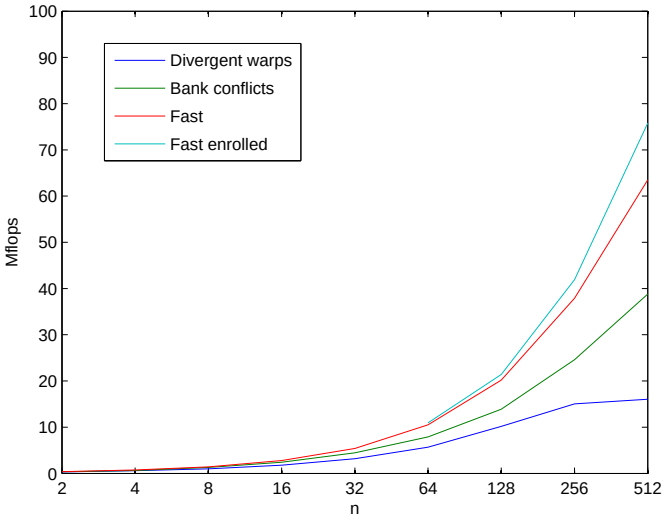


Figure 5.6: Performances of different reduction algorithms.

Algorithm	Speed (Mflops)	Relative performance
Divergent warps	16.0	1.00
Memory bank conflicts	38.8	2.42
Fast	63.5	3.96
Fast with enrolled loop	75.8	4.73

Table 5.1: Relative performance of different reduction algorithms.

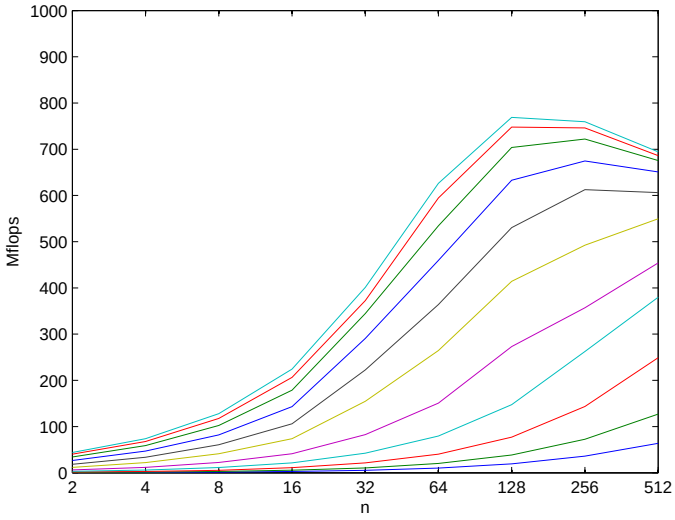


Figure 5.7: Performances of reduction algorithm for increasing number of blocks. Lowest performance for 1 block, highest for 1024 blocks.

investigate the fast unenrolled algorithm further to see the effect of using more blocks. An important question is whether or not $n = 512$ stays the fastest. Figure 5.7 shows the performance of the reduction algorithm for different number of blocks. The lowest performance corresponds to 1 block, to highest to 1024 blocks, and in between the number of blocks doubles. An important thing we see is that for 1024 blocks, we achieve maximum performance for $n = 128$. The exact reason for this is not clear. The performance depends on the resources per block, but also on the rate at which latency can be hide. The occupancy for $n = 512$ and $n = 256$ is 1. For $n = 128$, it is only $\frac{2}{3}$ since we have 64 threads, and a maximum of 8 active blocks per multiprocessor, so the number of active treads is only 512 per multiprocessor, yet we see that $n = 128$ is most efficient when having many blocks. For 64, 128 and 256 blocks $n = 256$ is most efficient and for less then 64 blocks, $n = 512$ becomes most efficient.

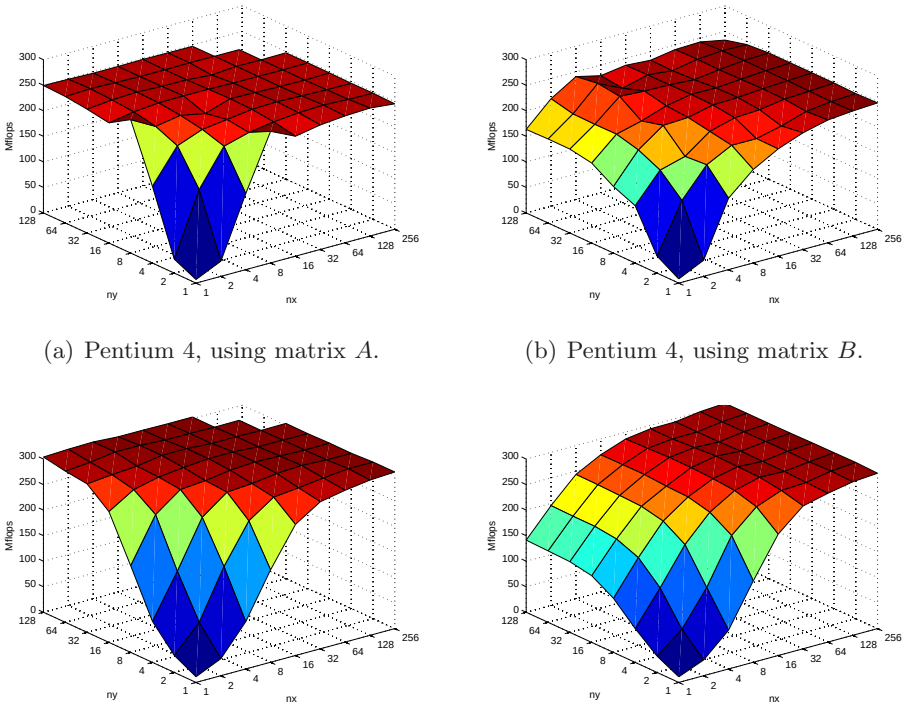
5.3 Results

In previous section the inner products needed during the Gauss-Seidel iterations are described in detail. This section gives the result for different experiments run. The same systems are used as in Section 4.3, so the GPU device is the NVIDIA Quadro FX 570M and the CPU devices are the Pentium 4 2.5 GHz and the Core 2 Duo T7300 2.0 GHz.

5.3.1 CPU implementation

On the CPU we again expect more or less constant performance once the grid size is big enough. Figure 5.8 shows the performance using both the matrix

A and B. The overall performance is taken as the average of 50 iterations. Again we see that the performances are nearly equal. Using the A matrix we converge faster to the maximum speed, because less addressing is necessary, so the performance is less dependent on the grid size. For example when using matrix A we only have to increase the jj counter, while when using matrix B, we have to increase both jj and kk. The maximum performance is 307 Mflops for the Intel Core 2 Duo T7300, and 261 Mflops for the Intel Pentium 4 3.40 GHz.



(a) Pentium 4, using matrix A.

(b) Pentium 4, using matrix B.

(c) Core 2 Duo T7300, using matrix A.

(d) Core 2 Duo T7300, using matrix B.

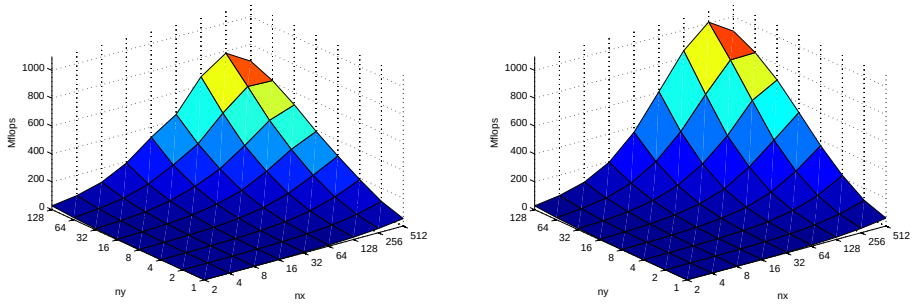
Figure 5.8: Performance of the inner products on the Pentium 4 and the Intel Core 2 Duo T7300.

5.3.2 GPU implementation

On the GPU we expect different behaviour. During execution of the algorithm to perform an inner product, each block performs two sum reductions. Each thread also reads 4 elements of the matrix B, and these transactions are in general non-coalesced. We will investigate what the consecuences of different divisions in blocks and threads are, and what the effect of memory coalescence is.

In Figure 5.7 we see that for sum reduction the performance increases with the number of blocks, and in Figure 5.6 we see that for increasing n, the tile size in this case, the performance also increases to a maximum for $n = 128$, $n = 256$ or $n = 512$ depending on the number of blocks. This suggests that for

larger problems, with a large grid size, performance is higher than for smaller problems. Figure 5.9 shows this effect. The left figure shows the results when performing non-coalesced global memory read transaction of matrix B. The right figure shows the performance when using coalesced transactions.



(a) Varying grid size, non-coalesced memory transactions.

(b) Varying grid size, coalesced memory transactions.

Figure 5.9: Performance of the inner product for different grid sizes.

When using non-coalesced global memory read transactions for B, we have a performance of 785 Mflops. When using coalesced transactions, the maximum performance is 1008 Mflops, which is about 28% higher. Both are achieved for $nx = 256$, so we have 256 threads per block. Beside this difference in performance the figures look very similar and we see that for growing grids the overal performances rises. This also suggests that it is only interesting to use the GPU for performing the inner products, when the grid size is large enough. For small grids the CPU implementation might perform better. In Figure 5.10 the performance can be seen as function of the total grid size. Here we used the results for the non-coalesced implementation, with $nx = 256$.
We see that for this particular GPU device, the NVIDIA Quadro FX 570M, it is only interesting to use the GPU for grids larger then $n \approx 5000$, since then the performance gets larger on the GPU than on the CPU. Of course different grid sizes can be interesting when using other GPU and CPU devices.

As a final experiment we will consider two particular grids, a smaller and a larger one and look at the effect of using different tile sizes in the y direction. For the small one we chose $nx = ny = 64$ and for the large one $nx = ny = 128$. This way we can see the consequence of varying both the number of blocks and the number of threads per block. Figure 5.9 suggests that the differences are not very large, since the diagonals, $n = nx \cdot ny$ for a certain n do not vary much. Figure 5.11 shows the results. In the figure both coalesced and non-coalesced global memory transaction are measured. Furthermore performances are measured when the final reduction is performed on the CPU and the GPU.
For the small grid we see that the performance is about 35 Mflops, and decreases slightly for larger ty. This suggests that it is more efficient to have more blocks than to have more threads per block. For the large grid this effect is even larger. $ty = 1$ is most efficient, $ty = 2$ is a little bit less efficient, but when $ty = 4$ we see a bigger decrease in performance. This can be explained because

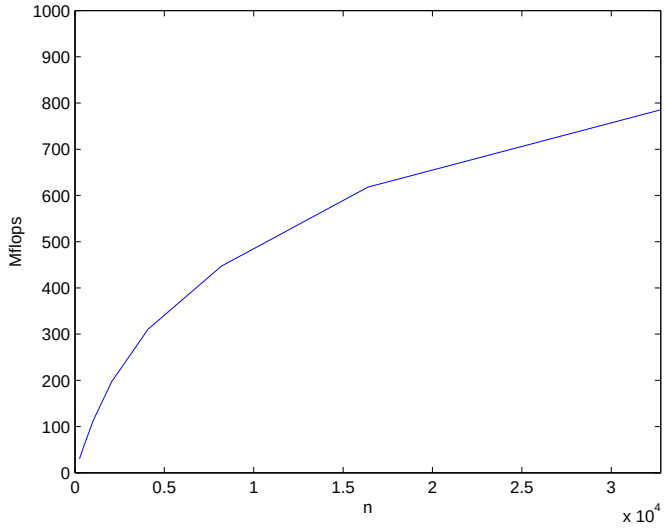
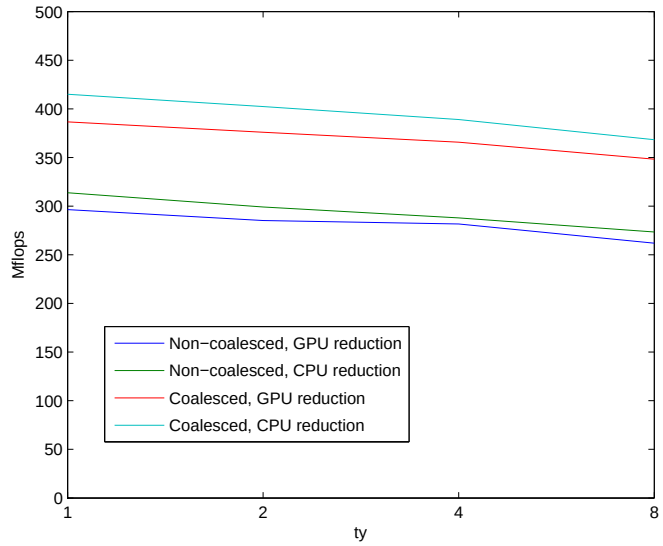


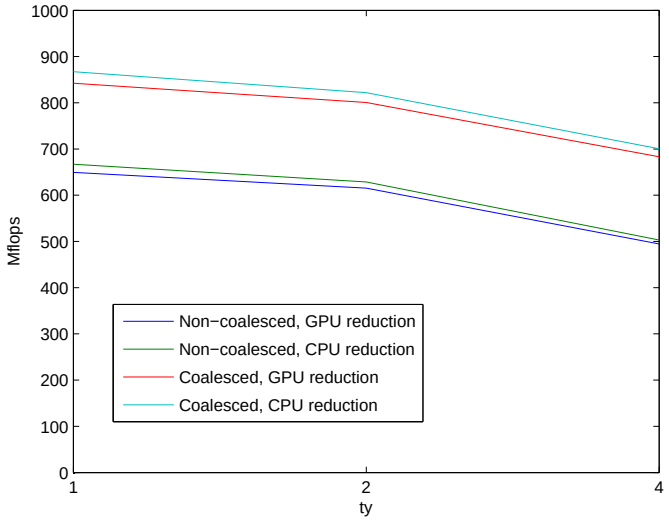
Figure 5.10: Performances as function of grid size n

the occupancy decreases. For $ty = 2$ we have 256 threads per block, resulting in 3 active blocks per multiprocessor, and an occupancy of 1. For $ty = 1$ we have 512 threads per block, resulting in 1 active block per multiprocessor, and an occupancy of only $\frac{2}{3}$.
Furthermore we see in both cases that the CPU reduction performs slighly better than the GPU reduction. This suggests that the final reduction is too small to be useful to be performed on the GPU. Again we see that coalescence results in a better performance. For the small grid the performance is about 35% higher, and for the large grid it varies between 30% and 40%.

To summarize, we have seen that only for relatively large grids the GPU is efficient at performing the inner products. The biggest drawback for small grids is that the total number of blocks is relative small, so latency hiding cannot be achieved very efficiently. Also the number of threads per block is small which results in a less effecient sum reduction.
The fact that the global memory transactions of the matrix B are non-coalesced decreases the performance by about 25%. On newer devices with compute capability 1.2 or higher this would probably be a smaller issue since only 2 memory transactions are performend instead of 16 in case of a non-coalesced acces pattern.



(a) Performance using different tile sizes for grid size 64×64.



(b) Performance using different tile sizes for grid size 128×128

Figure 5.11: Performance of grid sizes 64×64 and 128×128.

Chapter 6

Conclusions & recommendations

In this chapter conclusions will be made for the matrix-vector multiplication and the inner products. Furthermore recommendations will be stated and potential interesting further research is suggested.

6.1 Matrix-vector multiplication

When performing a full matrix-vector multiplication, we have seen that a significant speed-up can be achieved. In general, the larger the tiles, the higher the performance, until a tile size of about 128 elements is reached. It is very important to make sure the tile size is a multiple of 32. This ensures all warps are completely filled with threads.
Also the number of blocks is important for the performance. When $ny \leq 8$ the performance increases fast with the total number of blocks. On devices with more multiprocessors this effect might be even more significant.
A possible method to gain even more performance is to use seperate kernels for different tile sizes. For example if we have tiles with $ty = 1$, we do not need to iterate over the tiles in the y direction, and this instruction overhead can be overcome by creating a second kernel, for this special case, omiting the loop over the tiles in the y direction.
A final conclusion on the calculation of a full matrix-vector product is that when grid sizes become large, a dedicated device might be necessary, since when the primary device is used, time-outs might occur during kernel execution since the device is also needed for other purposes like video output.

6.2 Inner products

As we have seen the speed-ups for the inner products are less than for the full matrix-vector multiplication. For the used devices, the GPU only performed better when the grid size is larger than $n \approx 5000$. This is a consequence of how the Gauss-Seidel algorithm works. Between the calculation of the succesive inner products, the vector involved is updated, which means that we can only

calculate one inner product during each kernel invocation. This highly reduces the performance since all data need to be read from global memory again for the next inner product.
At this moment the non-linear part of the algorithm is executed on the CPU. One might suggest to port this to the GPU, but this introduces another problem. Since there is no way for blocks to synchronize with each other, the different blocks cannot determine whether another block is finished. As long as the whole algorithm is not divided in a per-block manner, we still need independent kernel launches, and are bounded by the high latency of the global memory.
The experiments run in this project are performed on a device with a relative small number of multiprocessors of 4. It might also be interesting to investigate the effect of more multiprocessors. For a given grid size, this will probably lead to a larger number of blocks each consisting of relatively few threads.
For smaller grids, it might be interesting to investigate performances when using the large A matrix structure, and use the highly optimized CUBLAS routines for performing the inner products. If for example a grid of size $n = 32 \cdot 64 = 2048$ is used, 64 MiB global memory is needed for 4 matrices[1]. This ensures memory coalescence during each global memory transaction. Also less adressing arithmetic is needed.

[1] Memory needed per matrix is $(32 \cdot 64)^2 \cdot 4\,\text{bytes} = 2^{24}\,\text{bytes} = 16\,\text{MiB}$.

Appendix A

Source code

The following sections contain the source code of the different implementations run, for the full matrix vector multiplication and the calculation of the inner products.

A.1 Matrix-vector multiplication

A.1.1 CPU implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

/* Direct matrix-vector product, in terms of matrix A */
void prod_A(float* u, float *A, float *p, unsigned int n) {

  unsigned int ii, jj, kk;

  kk = 0;

  for(ii = 0; ii < n; ii++) {

    u[ii] = 0;

    /* Calculate the inproduct corresponding to index jj */
    for(jj = 0; jj < n; jj++) {
      u[ii] += A[kk] * p[jj];
      kk++;
    }
  }

  return;

}

/* Direct matrix-vector product, in terms of matrix B */
void prod_B(float* u, float *B, float *p, unsigned int nx, unsigned int ny, unsigned int bx) {

  unsigned int ix, iy, jx, jy, ky, kx;
  unsigned int ii, jj, kk;

  ii = 0;

  unsigned int offset = bx - nx;

  for(iy = 0; iy < ny; iy++) {
    for(ix = 0; ix < nx; ix++) {
      u[ii] = 0;
      ky =  ny - 1 - iy;
      kx = nx - 1 - ix;
      jj = 0;
      kk = ky * bx + kx;
      for(jy = 0; jy < ny; jy++) {
        for(jx = 0; jx < nx; jx++) {
          u[ii] += B[kk] * p[jj];
          kk++;
          jj++;
        }
```

```
        kk += offset;
      }
      ii++;
    }
  }

  return;

}

int main( int argc, char **argv) {

  unsigned int nx, ny, n;
  unsigned int bx, by, m;
  unsigned int i;
  unsigned int ix, iy, jx, jy;
  unsigned int ii, jj;

  struct timeval start1, start2, stop1, stop2;
  float lapsed1, lapsed2;
  float speed1, speed2;
  float diff;

  /* default parameters */
  unsigned int iterations = 5;
  nx = 32;
  ny = 32;

  switch(argc) {
    case 4:
      sscanf(argv[2], "%d", &nx);
      sscanf(argv[3], "%d", &ny);
    case 2:
      sscanf(argv[1], "%d", &iterations);
      break;
    case 1:
      break;
    default:
      printf("Incorrect commandline options, exiting...\n");
      exit(1);
  }

  n = nx * ny;

  /* size of matrix B */
  bx = 2 * nx - 1;
  by = 2 * ny - 1;
  m = bx * by;

  /* declarations of matrices A and B and vector p */
  float *B;
  float *A;
  float *p;

  /* declarations of results of the direct and indirect matrix-vector product */
  float *u[2];

  /* random seed */
  srand( (unsigned)time( NULL ) );

  /* filling the grid B with random numbers */
  B = (float *)malloc(m * sizeof(float));

  for(ii = 0; ii < m; ii++) {
    B[ii] = (float) rand()/RAND_MAX;
  }

  /* create traction vector */
  p = (float *)malloc(n * sizeof(float));
  for(ii = 0; ii < n; ii++) {
    p[ii] = (float)rand() / RAND_MAX;
  }

  /* allocate space for the results */
  u[0] = (float *)malloc(n * sizeof(float));
  u[1] = (float *)malloc(n * sizeof(float));

  if(n <= 16384) {

    /* creating the big matrix A */
    A = (float *)malloc(n * n * sizeof(float));

    for(ii = 0; ii < n; ii++) {
      iy = ii / nx;
      ix = ii % nx;
      for(jj = 0; jj < n; jj++) {
        jy = jj / nx;
        jx = jj % nx;
        A[ii * n + jj] = B[(ny - 1 + jy - iy) * bx + (nx - 1  + jx - ix)];
      }
```

```
    }


    /* calculate the results by the direct function */
    gettimeofday (&start1, NULL);

    for(i = 0; i < iterations; i++) {
      prod_A(u[0], A, p, n);
    }
    gettimeofday(&stop1, NULL);
    lapsed1 = stop1.tv_sec - start1.tv_sec + 1e-6 * (stop1.tv_usec - start1.tv_usec);
    lapsed1 /= iterations;
    speed1 = mflops(lapsed1, n);

    free(A);
  }
  else {
    lapsed1 = -1;
  }

  gettimeofday(&start2, NULL);
  for(i = 0; i < iterations; i++) {
    prod_B(u[1], B, p, nx, ny, bx);
  }
  gettimeofday(&stop2, NULL);
  lapsed2 = stop2.tv_sec - start2.tv_sec + 1e-6 * (stop2.tv_usec - start2.tv_usec);
  lapsed2 /= iterations;
  speed2 = mflops(lapsed2, n);

  if(n <= 16384) {
    diff = normdiff(u[0], u[1], n);

    if(diff != 0) {
      printf("Different solutions, error in calculations. Exiting...\n");
    exit(1);
    }
  }
  else {
    diff = -1;
  }

  if(n <= 16384) {
    printf("%u %u %u %f %f %f %f\n", n, nx, ny, lapsed1, lapsed2, speed1, speed2);
  }
  else {
    printf("%u %u %u NaN %f NaN %f\n", n, nx, ny, lapsed2, speed2);
  }

  /* free all allocated memory space */
  free(B);

  free(p);
  free(u[0]);
  free(u[1]);

  return 0;
}
```

A.1.2 GPU implementation

Global memory implementation

```
#include <stdio.h>
#include <sys/time.h>

#include <cuda.h>

#include "util.h"

__global__ void Global_mem(float* dB, float* dp, float* du, unsigned int nx, unsigned int ny, unsigned int bx) {

  unsigned int ix, iy, ii, jx, jy, jj, kx, ky, kk;

  /* calculate absolute indices of u */
  iy = blockIdx.y * blockDim.y + threadIdx.y;
  ix = blockIdx.x * blockDim.x + threadIdx.x;
  ii = iy * nx + ix;

  /* initialize final answer to 0 */
  du[ii] = 0;

  /* loop over the y-direction of p */
  for(jy = 0; jy < ny; jy++) {

    /* determine which row of B is needed */
    ky = jy - iy + ny - 1;
```

```
    /* loop over the x-direction of p */
    for(jx = 0; jx < nx; jx++) {

      /* determine which elements of B and p are needed */
      kx = jx - ix + nx - 1;
      jj = jy * nx + jx;
      kk = ky * bx + kx;

      /* multiply the correct elements and add to u, remark that both read transaction are non-coalesced,
      since the order is reversed, the write action to u is coalesced */
      du[ii] += dB[kk] * dp[jj];
    }
  }
}

int main(int argc, char **argv) {

  unsigned int nx, ny;
  unsigned int bx, by;
  unsigned int n, m;
  unsigned int i, ii;
  unsigned int iterations;
  unsigned int Tx, Ty, tx, ty;

  float *hB;
  float *dB;
  float *hp;
  float *dp;
  float *du;

  float *u_cpu;
  float *u_gpu;

  struct timeval start, stop;
  float lapsed;
  float diff;

  cudaError_t err;

  /* default values */
  iterations = 5;
  nx = 64;
  ny = 16;
  tx = 64;
  ty = 1;

  /* simple commandline arguments, we have shared_memory <iterations> <nx> <ny> <tx> <ty> */
  switch(argc) {
    case 6:
      sscanf(argv[4],"%u", &tx);
      sscanf(argv[5],"%u", &ty);
    case 4:
      sscanf(argv[2], "%u", &nx);
      sscanf(argv[3], "%u", &ny);
    case 2:
      sscanf(argv[1], "%u", &iterations);
      break;
    case 1:
      break;
    default:
      printf("Incorrect commandline options, usage: %s <iterations> <nx> <ny> <tx> <ty>\n", argv[0]);
      exit(1);
  }

  bx = 2 * nx - 1;
  by = 2 * ny - 1;

  n = nx * ny;
  m = bx * by;

  Tx = nx / tx;
  Ty = ny / ty;

  if(tx == 0 || ty == 0 || nx % tx != 0 || ny % ty != 0) {
    printf("Tiles have an incorrect size, exiting...\n");
    exit(1);
  }

  if(tx * ty > 512) {
    /* of tiles are incorrect, output NaN's which can be dealed with by Matlab */
    printf("%u %u %u %u %u NaN NaN NaN\n", n, nx, ny, tx, ty);
    exit(1);
  }

  /* set dimensions of grid and threads */
  dim3 dimGrid(Tx, Ty);
  dim3 dimBlock(tx, ty);

  /* random seed */
  srand( 1337 );
```

```
/* filling the grid B with random numbers */
hB = (float *)malloc(m * sizeof(float));
for(ii = 0; ii < m; ii++) {
  hB[ii] = (float) rand()/RAND_MAX;
}

 /* create traction vector */
hp = (float *)malloc(n * sizeof(float));
for(ii = 0; ii < n; ii++) {
  hp[ii] = (float)rand() / RAND_MAX;
}


/* allocate memory for results*/
u_cpu = (float *)malloc(n * sizeof(float));
u_gpu = (float *)malloc(n * sizeof(float));

/* allocate memory on device, copy B matrix and p vector to device */
err = cudaMalloc((void**)&dB, m * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemcpy(dB, hB, m * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

err = cudaMalloc((void**)&dp, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemcpy(dp, hp, n * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMalloc((void**)&du, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

err = cudaMemset(du, 0, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* determine start time */
err = cudaThreadSynchronize();
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

gettimeofday(&start, NULL);
for(i = 0; i < iterations; i++) {

  /* invoke the kernel, we don't need any shared memory */
  Global_mem<<<dimGrid, dimBlock>>>(dB, dp, du, nx, ny, bx);

  /* check if an error occured when launching the kernel */
  err = cudaGetLastError();
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }

  /* copy result to host, part of the time measurement */
  err = cudaMemcpy(u_gpu, du, n * sizeof(float), cudaMemcpyDeviceToHost);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
}

/* determine stop time and calculate lapsed time per iterations */
err = cudaThreadSynchronize();
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
gettimeofday(&stop, NULL);
lapsed = stop.tv_sec - start.tv_sec + 1e-6 * (stop.tv_usec - start.tv_usec);
lapsed /= iterations;
```

```
  /* run on cpu too, for numerical comparison */
  prod_B(u_cpu, hB, hp, nx, ny, bx);

  /* calculate relative difference */
  diff = normdiff(u_cpu, u_gpu, n) / norm(u_cpu, n);

  /* print results */
  printf("%u %u %u %u %u %f %E %f\n", n, nx, ny, tx, ty, lapsed, diff, mflops(lapsed, n));

  /* free all allocated memory */
  free(u_cpu);
  free(u_gpu);
  free(hB);
  free(hp);

  err = cudaFree(dB);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(dp);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(du);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }

  return 0;
}
```

Shared memory implementation

```
#include <stdio.h>
#include <sys/time.h>

#include <cuda.h>

#include "util.h"

/* allocated shared memory array */
extern __shared__ float shared_mem[];

__global__ void Kernel_shared(float* dB, float* dp, float* du,
unsigned int nx, unsigned int ny, unsigned int stride) {

  unsigned int jTx, jx, jy, iy;
  unsigned int tx, Tx, iTx;
  unsigned int kx, ky;

  float *shared_p, *shared_B, *shared_u;
  float u;

  /* just for simple syntax later on */
  tx = blockDim.x;
  Tx = gridDim.x;
  iy = blockIdx.y;
  iTx = blockIdx.x;

  /* make sure shared memory has correct structure */
  shared_p = shared_mem;
  shared_u = &shared_p[tx];
  shared_B = &shared_u[tx];

  /* initialize result to zero */
  u = 0;

  /* loop over the different tiles of p in the y-direction */
  for(jy = 0; jy < ny; jy++) {

    ky = jy - iy + ny - 1;

    /* loop over the different tiles of p in the x-direction */
    for(jTx = 0; jTx < Tx; jTx++) {

      kx = (jTx - iTx - 1) * tx + nx + threadIdx.x;

      /* loading the two tiles of B and the one tile of p from the global memory */
      shared_B[threadIdx.x] = dB[ky * stride + kx];
      shared_B[threadIdx.x + tx] = dB[ky * stride + kx + tx];
      shared_p[threadIdx.x] = dp[jy * nx + jTx * tx + threadIdx.x];

      /* make sure all data is read */
      __syncthreads();
```

```
      /* loop over the different grid points in the current tile of p */
      for(jx = 0; jx < tx; jx++) {

        /* add the contribution of p to u */
        u += shared_B[jx + threadIdx.x] * shared_p[jx];

      }

     /* make sure all the contribution of the current tile of p are used, and can be overwritten by the next tile */
     __syncthreads();

    }

  }

  /* save the final answer to the global memory. This one is non-coalesced, but happens only once, so no big deal */
  du[iy * nx + (iTx + 1) * tx - 1 - threadIdx.x] = u;

}

int main(int argc, char **argv) {

  unsigned int nx, ny;
  unsigned int bx, by;
  unsigned int n, m;
  unsigned int i, ii;
  unsigned int iterations;
  unsigned int Tx, Ty, tx, ty;
  unsigned int stride;

  float *hB;
  float *dB;
  float *hp;
  float *dp;
  float *du;

  float *u_cpu;
  float *u_gpu;

  struct timeval start, stop;
  float lapsed;
  float diff;

  cudaError_t err;

  /* default values */
  iterations = 5;
  nx = 64;
  ny = 16;
  tx = 64;
  ty = 1;

  /* simple commandline arguments, we have shared_memory <iterations> <nx> <ny> <tx> <ty> */
  switch(argc) {
    case 6:
      sscanf(argv[4],"%u", &tx);
      sscanf(argv[5],"%u", &ty);
    case 4:
      sscanf(argv[2], "%u", &nx);
      sscanf(argv[3], "%u", &ny);
    case 2:
      sscanf(argv[1], "%u", &iterations);
      break;
    case 1:
      break;
    default:
      printf("Incorrect commandline options, usage: %s <iterations> <nx> <ny> <tx> <ty>\n", argv[0]);
      exit(1);
  }

  bx = 2 * nx - 1;
  by = 2 * ny - 1;
  stride = 2 * nx;

  n = nx * ny;
  m = stride * by;

  Tx = nx / tx;
  Ty = ny / ty;

  if(ty != 1 || nx % tx != 0) {
    /* of tiles are incorrect, output NaN's which can be dealed with by Matlab */
    printf("%u %u %u %u %u NaN NaN NaN\n", n, nx, ny, tx, ty);

    exit(1);
  }

  if(tx > 512) {
    printf("Tiles too big, exiting...\n");
    exit(1);
```

```
}

/* set dimensions of grid and threads */
dim3 dimGrid(Tx, Ty);
dim3 dimBlock(tx, ty);

/* random seed */
srand( 1337 );

/* filling the grid B with random numbers */
hB = (float *)malloc(m * sizeof(float));
for(ii = 0; ii < m; ii++) {
  if((ii + 1) % stride == 0) {
  hB[ii] = 0;
  }
  else {
    hB[ii] = (float) rand()/RAND_MAX;
  }
}

 /* create traction vector */
hp = (float *)malloc(n * sizeof(float));
for(ii = 0; ii < n; ii++) {
  hp[ii] = (float)rand() / RAND_MAX;
}


/* allocate memory for results*/
u_cpu = (float *)malloc(n * sizeof(float));
u_gpu = (float *)malloc(n * sizeof(float));

/* allocate memory on device, copy B matrix and p vector to device */
err = cudaMalloc((void**)&dB, m * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemcpy(dB, hB, m * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

err = cudaMalloc((void**)&dp, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemcpy(dp, hp, n * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMalloc((void**)&du, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

err = cudaMemset(du, 0, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* determine start time */
err = cudaThreadSynchronize();
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
gettimeofday(&start, NULL);

for(i = 0; i < iterations; i++) {
  /* invoke the kernel, we need 3 tiles of shared memory, one for p and two for B */
  Kernel_shared<<<dimGrid, dimBlock, 4 * tx * ty * sizeof(float)>>>(dB, dp, du, nx, ny, stride);

  /* check if an error occured when launching the kernel */
  err = cudaGetLastError();
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }

  /* copy result to host, part of the time measurement */
  err = cudaMemcpy(u_gpu, du, n * sizeof(float), cudaMemcpyDeviceToHost);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
```

```
  }

  /* determine stop time and calculate lapsed time per iterations */
  err = cudaThreadSynchronize();
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  gettimeofday(&stop, NULL);
  lapsed = stop.tv_sec - start.tv_sec + 1e-6 * (stop.tv_usec - start.tv_usec);
  lapsed /= iterations;

  /* run on cpu too, for numerical comparison */
  prod_B(u_cpu, hB, hp, nx, ny, stride);

  /* calculate relative difference */
  diff = normdiff(u_cpu, u_gpu, n) / norm(u_cpu, n);

  /* print results */
  printf("%u %u %u %u %u %f %E %f\n", n, nx, ny, tx, ty, lapsed, diff, mflops(lapsed, n));

  /* free all allocated memory */
  free(u_cpu);
  free(u_gpu);
  free(hB);
  free(hp);

  err = cudaFree(dB);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(dp);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(du);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }

  return 0;
}
```

A.1.3 Utility functions

```
#include <math.h>
#include <stdio.h>

/* Direct matrix-vector product, in terms of matrix B */
void prod_B(float* u, float *B, float *p, unsigned int nx, unsigned int ny, unsigned int bx) {

  unsigned int ix, iy, jx, jy, ky, kx;
  unsigned int ii, jj, kk;

  for(iy = 0; iy < ny; iy++) {
    for(ix = 0; ix < nx; ix++) {
      ii = iy * nx + ix;
      u[ii] = 0;
      for(jy = 0; jy < ny; jy++) {
        ky = jy - iy + ny - 1;
        for(jx = 0; jx < nx; jx++) {
          kx = jx - ix + nx - 1;
          jj = jy * nx + jx;
          kk = ky * bx + kx;
          u[ii] += B[kk] * p[jj];
        }
      }
    }
  }

  return;

}

float normdiff(float *u1, float *u2, unsigned int n) {
/* calculate the norm of the difference of u1 and u2 */

  unsigned int ii;
  float res, update;

  res = 0;

  for(ii = 0; ii < n; ii++) {
    update = u1[ii] - u2[ii];
    update *= update;
```

```
    res += update;
  }

  res = pow(res, 0.5);

  return res;

}

float norm(float *u, unsigned int n) {
/* calculate the norm of a vector */

  unsigned int ii;
  float res;

  res = 0;

  for(ii = 0; ii < n; ii++) {
    res += u[ii] * u[ii];
  }


  res = pow(res, 0.5);

  return res;

}


float mflops(float time, unsigned int n) {

  float size = n;

  return (time? 2 * size * size / time / 1e6 : 0);

}

bool ispowof2(unsigned int n) {

  if(n == 0)
    return false;

  while(n != 1) {
    if(n % 2 == 0)
      n /= 2;
    else
      return false;
  }

  return true;

}

void printu(float* u1, float* u2, int n) {

  int ii;

  for(ii = 0; ii < n; ii++) {
    printf("%5.4f", u1[ii]);
    (u2 ? printf(" %5.4f\n", u2[ii]) : printf("\n"));
  }

  printf("\n");

  return;

}
```

A.2 Inner product calculation

A.2.1 CPU implementation

Matrix A implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#include "util.h"

/* Direct inner product, in terms of matrix B */
float innerprod_B(float *B, float *p, unsigned int nx,
unsigned int ny, unsigned int bx, unsigned int iy, unsigned int ix) {
```

```
  unsigned int jx, jy, ky, kx;
  unsigned int jj, kk;

  float res;

  res = 0;

  for(jy = 0; jy < ny; jy++) {
    ky = jy - iy + ny - 1;
    kx = nx - 1 - ix;
    jj = jy * nx;
    kk = ky * bx + kx;
    for(jx = 0; jx < nx; jx++) {
      res += B[kk] * p[jj];
      jj++;
      kk++;
    }
  }

  return res;

}

/* Direct inner product, in terms of matrix A */
float innerprod_A(float *A, float *p, unsigned int n, unsigned int ii) {

  unsigned int jj;

  float res;

  res = 0;

  /* Calculate the inproduct corresponding to index jj */
  for(jj = 0; jj < n; jj++) {
    res += A[ii * n + jj] * p[jj];
  }

  return res;

}

int main( int argc, char **argv) {

  unsigned int nx, ny, n;
  unsigned int bx, by, m;
  unsigned int i, j, k;
  unsigned int ix, iy, jx, jy;
  unsigned int ii, jj;

  /* variables for timing and speed measuring */
  struct timeval *start, *stop;
  float *lapsed;
  float *speed;
  float averagespeed, averagelapsed;
  float varspeed, varlapsed;

  /* variable for numerical comparison */
  float diff;

  /* default parameters */
  unsigned int iterations = 5;
  nx = 32;
  ny = 32;

  switch(argc) {
    case 4:
      sscanf(argv[2], "%d", &nx);
      sscanf(argv[3], "%d", &ny);
    case 2:
      sscanf(argv[1], "%d", &iterations);
      break;
    case 1:
      break;
    default:
      printf("Incorrect commandline options, exiting...\n");
      exit(1);
  }

  start = (struct timeval *)malloc(iterations * sizeof(struct timeval));
  stop = (struct timeval *)malloc(iterations * sizeof(struct timeval));
  lapsed = (float *)malloc(iterations * sizeof(float));
  speed = (float *)malloc(iterations * sizeof(float));

  n = nx * ny;

  /* check if grid is small enough for four A matrices to fit in memory */
  if(n > 8192) {
    printf("%u %u %u NaN NaN\n", n , nx, ny);
    exit(1);
  }
```

```
/* size of matrix B */
bx = 2 * nx - 1;
by = 2 * ny - 1;
m = bx * by;

/* declarations of matrices A and B and vector p */
float *Bxx, *Bxy, *Byx, *Byy;
float *Axx, *Axy, *Ayx, *Ayy;
float *px, *py;

/* declarations of results of the direct and indirect matrix-vector product */
float *ux, *uy;
float *uxB, *uyB;

/* declarations of the initial deformation */
float *wx, *wy;

/* random seed */
srand( (unsigned)time( NULL ) );

/* filling the grids B with random numbers */
Bxx = (float *)malloc(m * sizeof(float));
Bxy = (float *)malloc(m * sizeof(float));
Byx = (float *)malloc(m * sizeof(float));
Byy = (float *)malloc(m * sizeof(float));

for(ii = 0; ii < m; ii++) {
  Bxx[ii] = (float) rand()/RAND_MAX;
  Bxy[ii] = (float) rand()/RAND_MAX;
  Byx[ii] = (float) rand()/RAND_MAX;
  Byy[ii] = (float) rand()/RAND_MAX;
}

/* creating A matrices */
Axx = (float *)malloc(n * n * sizeof(float));
Axy = (float *)malloc(n * n * sizeof(float));
Ayx = (float *)malloc(n * n * sizeof(float));
Ayy = (float *)malloc(n * n * sizeof(float));

for(ii = 0; ii < n; ii++) {
  iy = ii / nx;
  ix = ii % nx;
  for(jj = 0; jj < n; jj++) {
    jy = jj / nx;
    jx = jj % nx;
    Axx[ii * n + jj] = Bxx[(ny - 1 + jy - iy) * bx + (nx - 1  + jx - ix)];
    Axy[ii * n + jj] = Bxy[(ny - 1 + jy - iy) * bx + (nx - 1  + jx - ix)];
    Ayx[ii * n + jj] = Byx[(ny - 1 + jy - iy) * bx + (nx - 1  + jx - ix)];
    Ayy[ii * n + jj] = Byy[(ny - 1 + jy - iy) * bx + (nx - 1  + jx - ix)];
  }
}

/* create traction vectors */
px = (float *)malloc(n * sizeof(float));
py = (float *)malloc(n * sizeof(float));
for(ii = 0; ii < n; ii++) {
  px[ii] = (float)rand() / RAND_MAX;
  py[ii] = (float)rand() / RAND_MAX;
}

/* create initial deformation vectors */
wx = (float *)malloc(n * sizeof(float));
wy = (float *)malloc(n * sizeof(float));
for(ii = 0; ii < n; ii++) {
  wx[ii] = (float)rand() / RAND_MAX;
  wy[ii] = (float)rand() / RAND_MAX;
}

/* allocate space for the results */
ux = (float *)malloc(n * sizeof(float));
uy = (float *)malloc(n * sizeof(float));
uxB = (float *)malloc(n * sizeof(float));
uyB = (float *)malloc(n * sizeof(float));


for(i = 0; i < iterations; i++) {
  gettimeofday(&start[i], NULL);
  for(ii = 0; ii < n; ii++) {
    ux[ii] = wx[ii] + innerprod_A(Axx, px, n, ii) + innerprod_A(Axy, py, n, ii);
    uy[ii] = wy[ii] + innerprod_A(Ayx, px, n, ii) + innerprod_A(Ayy, py, n, ii);
  }
  gettimeofday(&stop[i], NULL);
}

timing(start, stop, lapsed, speed, &averagelapsed, &averagespeed, &varlapsed, &varspeed, iterations, n);

for(iy = 0; iy < ny; iy++) {
  ii = iy * nx;
  for(ix = 0; ix < nx; ix++) {
```

```
      uxB[ii] = wx[ii] + innerprod_B(Bxx, px, nx, ny, bx, iy, ix) + innerprod_B(Bxy, py, nx, ny, bx, iy, ix);
      uyB[ii] = wy[ii] + innerprod_B(Byx, px, nx, ny, bx, iy, ix) + innerprod_B(Byy, py, nx, ny, bx, iy, ix);
      ii++;
    }
  }

  diff = normdiff(uxB, ux, n) + normdiff(uyB, uy, n);
  if(diff != 0) {
    printf("Error in calculations, exiting...\n");
    exit(1);
  }

  printf("%u %u %u %f %f\n", n, nx, ny, averagelapsed, averagespeed);

  /* free all allocated memory space */
  free(Bxx);
  free(Bxy);
  free(Byx);
  free(Byy);
  free(Axx);
  free(Axy);
  free(Ayx);
  free(Ayy);

  free(px);
  free(py);

  free(wx);
  free(wy);

  free(ux);
  free(uy);
  free(uxB);
  free(uyB);

  free(start);
  free(stop);
  free(lapsed);
  free(speed);

  return 0;
}
```

Matrix B implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#include "util.h"

/* Direct inner product, in terms of matrix B */
float innerprod_B(float *B, float *p, unsigned int nx,
unsigned int ny, unsigned int bx, unsigned int iy, unsigned int ix) {

  unsigned int jx, jy, ky, kx;
  unsigned int jj, kk;

  float res;

  res = 0;

  for(jy = 0; jy < ny; jy++) {
    ky = jy - iy + ny - 1;
    kx = nx - 1 - ix;
    jj = jy * nx;
    kk = ky * bx + kx;
    for(jx = 0; jx < nx; jx++) {
      res += B[kk] * p[jj];
      jj++;
      kk++;
    }
  }

  return res;

}

int main( int argc, char **argv) {

  unsigned int nx, ny, n;
  unsigned int bx, by, m;
  unsigned int i, j, k;
  unsigned int ix, iy, jx, jy;
  unsigned int ii, jj;

  /* variables for timing and speed measurement */
```

```
struct timeval *start, *stop;
float *lapsed;
float *speed;
float averagespeed, averagelapsed;
float varspeed, varlapsed;

/* default parameters */
unsigned int iterations = 5;
nx = 32;
ny = 32;

switch(argc) {
  case 4:
    sscanf(argv[2], "%d", &nx);
    sscanf(argv[3], "%d", &ny);
  case 2:
    sscanf(argv[1], "%d", &iterations);
    break;
  case 1:
    break;
  default:
    printf("Incorrect commandline options, exiting...\n");
    exit(1);
}

start = (struct timeval *)malloc(iterations * sizeof(struct timeval));
stop = (struct timeval *)malloc(iterations * sizeof(struct timeval));
lapsed = (float *)malloc(iterations * sizeof(float));
speed = (float *)malloc(iterations * sizeof(float));

n = nx * ny;

/* size of matrix B */
bx = 2 * nx - 1;
by = 2 * ny - 1;
m = bx * by;

/* declarations of matrices A and B and vector p */
float *Bxx, *Bxy, *Byx, *Byy;
float *px, *py;

/* declarations of results of the direct and indirect matrix-vector product */
float *ux, *uy;

/* declarations of the initial deformation */
float *wx, *wy;

/* random seed */
srand( (unsigned)time( NULL ) );

/* filling the grids B with random numbers */
Bxx = (float *)malloc(m * sizeof(float));
Bxy = (float *)malloc(m * sizeof(float));
Byx = (float *)malloc(m * sizeof(float));
Byy = (float *)malloc(m * sizeof(float));

for(ii = 0; ii < m; ii++) {
  Bxx[ii] = (float) rand()/RAND_MAX;
  Bxy[ii] = (float) rand()/RAND_MAX;
  Byx[ii] = (float) rand()/RAND_MAX;
  Byy[ii] = (float) rand()/RAND_MAX;
}

/* create traction vectors */
px = (float *)malloc(n * sizeof(float));
py = (float *)malloc(n * sizeof(float));
for(ii = 0; ii < n; ii++) {
  px[ii] = (float)rand() / RAND_MAX;
  py[ii] = (float)rand() / RAND_MAX;
}

/* create initial deformation vectors */
wx = (float *)malloc(n * sizeof(float));
wy = (float *)malloc(n * sizeof(float));
for(ii = 0; ii < n; ii++) {
  wx[ii] = (float)rand() / RAND_MAX;
  wy[ii] = (float)rand() / RAND_MAX;
}

/* allocate space for the results */
ux = (float *)malloc(n * sizeof(float));
uy = (float *)malloc(n * sizeof(float));


for(i = 0; i < iterations; i++) {
  gettimeofday(&start[i], NULL);
  for(iy = 0; iy < ny; iy++) {
    for(ix = 0; ix < nx; ix++) {
      ii = iy * nx + ix;
      ux[ii] = wx[ii] + innerprod_B(Bxx, px, nx, ny, bx, iy, ix) + innerprod_B(Bxy, py, nx, ny, bx, iy, ix);
```

```
        uy[ii] = wy[ii] + innerprod_B(Byx, px, nx, ny, bx, iy, ix) + innerprod_B(Byy, py, nx, ny, bx, iy, ix);
      }
    }
    gettimeofday(&stop[i], NULL);
  }

  timing(start, stop, lapsed, speed, &averagelapsed, &averagespeed, &varlapsed, &varspeed, iterations, n);

  printf("%u %u %u %f %f\n", n, nx, ny, averagelapsed, averagespeed);

  /* free all allocated memory space */
  free(Bxx);
  free(Bxy);
  free(Byx);
  free(Byy);

  free(px);
  free(py);

  free(wx);
  free(wy);

  free(ux);
  free(uy);

  free(start);
  free(stop);
  free(lapsed);
  free(speed);

  return 0;
}
```

A.2.2 Sum reduction algorithms

For the first algorithm the complete program is stated. For the next three algorithms only the different kernels are stated. Remarkt that for the fast algorithms with and without enrolled loop, the number of threads per block is $\frac{n}{2}$ instead of n.

Divergent warps

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#include <cuda.h>

extern __shared__ float shared_mem[];

/* reduction with divergent warps */
__device__ void divergent_reduce(unsigned int n, unsigned int t, float *N) {

  unsigned int s;

  for(s = 1; s < n; s *= 2) {

    if((t % (2 * s)) == 0) {
      N[t] += N[t + s];
    }
  __syncthreads();
  }
}

__global__ void Kernel_reduce(float* N, float* Nresult, unsigned int n) {

  shared_mem[threadIdx.x] = N[threadIdx.x];

  __syncthreads();

  divergent_reduce(n, threadIdx.x, shared_mem);

  if(threadIdx.x == 0) {
    Nresult[blockIdx.x] = shared_mem[0];
  }
}

int main( int argc, char **argv) {

  unsigned int i, n;
  float *hN, *dN, hNresult, *dNresult, result;
```

```
struct timeval start, stop;
float lapsed;
float speed;
float diff;

/* default parameters */
unsigned int iterations = 1000;
n = 512;

switch(argc) {
  case 3:
    sscanf(argv[2], "%d", &n);
  case 2:
    sscanf(argv[1], "%d", &iterations);
    break;
  case 1:
    break;
  default:
    printf("Incorrect commandline options, exiting...\n");
    exit(1);
}

/* maximum of 512 elements */
if(n > 512) {
  printf("%u NaN NaN NaN\n", n);
  exit(1);
}

cudaError_t err;

/* set dimensions of grid and threads */
dim3 dimGrid(1000, 1, 1);
dim3 dimBlock(n, 1);

/* random seed */
srand( (unsigned)time( NULL ) );

/* filling the array */
hN = (float *)malloc(n * sizeof(float));
for(i = 0; i < n; i++) {
  hN[i] = (float) rand()/RAND_MAX;
}

/* allocate memory on device */
err = cudaMalloc((void**)&dN, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMalloc((void**)&dNresult, 1000 * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* copy data */
err = cudaMemcpy(dN, hN , n * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* determine start time */
err = cudaThreadSynchronize();
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
gettimeofday(&start, NULL);

/* perform kernel invocations */
for(i = 0; i < iterations; i++) {

  Kernel_reduce<<<dimGrid, dimBlock, n * sizeof(float)>>>(dN, dNresult, n);

  /* check if an error occured when launching the kernel */
  err = cudaGetLastError();
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
}

hNresult = 0;
/* copy result to host */
err = cudaMemcpy(&hNresult, dNresult, 1 * sizeof(float), cudaMemcpyDeviceToHost);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
```

```
  }

  /* determine stop time and calculate lapsed time per iterations */
  err = cudaThreadSynchronize();
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  gettimeofday(&stop, NULL);
  lapsed = stop.tv_sec - start.tv_sec + 1e-6 * (stop.tv_usec - start.tv_usec);
  lapsed /= iterations;
  speed = 1000 * n / lapsed * 1e-6;

  /* run on cpu for numerical comparison */
  result = 0;
  for(i = 0; i < n; i++) {
    result += hN[i];
  }

  diff = abs(result - hNresult);

  printf("%u %f %f %f\n", n, diff, lapsed, speed);

  /* free all allocated memory space */
  free(hN);

  err = cudaFree(dN);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(dNresult);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }

  return 0;
}
```

Shared memory bank conflicts

```
/* reduction with bank conflicts */
__device__ void conflicts_reduce(unsigned int n, unsigned int t, float *N) {

  unsigned int s, i;

  for(s = 1; s < n; s *= 2) {
    i = 2 * s * t;

    if(i < n) {
      N[i] += N[i + s];
    }
  __syncthreads();
  }
}

__global__ void Kernel_reduce(float* N, float* Nresult, unsigned int n) {

  shared_mem[threadIdx.x] = N[threadIdx.x];
  shared_mem[blockDim.x + threadIdx.x] = N[blockDim.x + threadIdx.x];

  __syncthreads();

  conflicts_reduce(n, threadIdx.x, shared_mem);

  if(threadIdx.x == 0) {
    Nresult[blockIdx.x] = shared_mem[0];
  }
}
```

Fast algorithm

```
/* reduction without divergent warps and bank conflicts */
__device__ void fast_reduce(unsigned int n, unsigned int t, float *N) {

  unsigned int i;

  for(i = n / 2; i > 0; i >>= 1) {

    if(t < i) {
      N[t] += N[t + i];
    }
  __syncthreads();
  }
}
```

```
__global__ void Kernel_reduce(float* N, float* Nresult, unsigned int n) {

  shared_mem[threadIdx.x] = N[threadIdx.x];
  shared_mem[blockDim.x + threadIdx.x] = N[blockDim.x + threadIdx.x];

  __syncthreads();

  fast_reduce(n, threadIdx.x, shared_mem);

  if(threadIdx.x == 0) {
    Nresult[blockIdx.x] = shared_mem[0];
  }
}
```

Fast algorithm, enrolled

```
/* reduction with last 6 steps enrolled, nx >= 64 */
__device__ void fast_reduce_enrolled(unsigned int n, unsigned int t, float *N) {

  unsigned int i;

  for(i = n / 2; i > 32; i >>= 1) {

    if(t < i) {
      N[t] += N[t + i];

    }
    __syncthreads();
  }

  /* enrolled iterations */
  if(t < 32) {
    N[t] += N[t + 32];
    N[t] += N[t + 16];
    N[t] += N[t + 8];
    N[t] += N[t + 4];
    N[t] += N[t + 2];
    N[t] += N[t + 1];

  }
}

__global__ void Kernel_reduce(float* N, float* Nresult, unsigned int n) {

  shared_mem[threadIdx.x] = N[threadIdx.x];
  shared_mem[blockDim.x + threadIdx.x] = N[blockDim.x + threadIdx.x];

  fast_reduce_enrolled(n, threadIdx.x, shared_mem);

  __syncthreads();

  if(threadIdx.x == 0) {
    Nresult[blockIdx.x] = shared_mem[0];
  }
}
```

A.2.3 GPU implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#include <cuda.h>

#include "util.h"

__device__ void fast_reduce(unsigned int nx, unsigned int thread, float *temp) {

  unsigned int i;

  for(i = nx / 2; i > 0; i >>= 1) {

    if(thread < i) {
      temp[thread] += temp[thread + i];
    }
  __syncthreads();
  }

}

extern __shared__ float shared_mem[];

__global__ void Kernel_reduce(float* dreductionx, float* dreductiony, unsigned int ny, float* dx, float* dy) {
```

```
  if(blockIdx.x == 0) {
    shared_mem[threadIdx.x] = dreductionx[threadIdx.x];
    __syncthreads();
    fast_reduce(ny, threadIdx.x, shared_mem);
    if(threadIdx.x == 0) {
      *dx = shared_mem[0];
    }
  }
  else {
    shared_mem[threadIdx.x] = dreductiony[threadIdx.x];
    __syncthreads();
    fast_reduce(ny, threadIdx.x, shared_mem);
    if(threadIdx.x == 0) {
      *dy = shared_mem[0];
    }
  }

}

__global__ void Kernel_inner_prod(float* dBxx, float* dBxy, float *dByx, float *dByy, float* dpy, float* dpx,
unsigned int nx, unsigned int ny, unsigned int stride,
unsigned int ix, unsigned int iy, unsigned int ty, float *dreductionx, float *dreductiony) {

  unsigned int jx, jy;
  unsigned int jj, kk;

  float temp;

  unsigned int index;

  float *prodx, *prody;

  prodx = shared_mem;
  prody = &prodx[ty * nx];

  index = threadIdx.x + threadIdx.y * nx;

  jy = ty * blockIdx.x + threadIdx.y;
  jx = threadIdx.x;

  jj = jy * nx + jx;
  kk = jy * stride + jx;
  temp = dpx[jj];
  prodx[index] = dBxx[kk] * temp;
  prody[index] = dByx[kk] * temp;
  temp = dpy[jj];
  prodx[index] += dBxy[kk] * temp;
  prody[index] += dByy[kk] * temp;

  __syncthreads();

  fast_reduce(ty * nx, index, prodx);
  fast_reduce(ty * nx, index, prody);

  if(index == 0) {
    dreductionx[blockIdx.x] = prodx[0];
    dreductiony[blockIdx.x] = prody[0];
  }

}

int main( int argc, char **argv) {

  unsigned int nx, ny, n;
  unsigned int ty;
  unsigned int bx, by, stride, m;
  unsigned int i;
  unsigned int ix, iy;
  unsigned int ii, kk;

  struct timeval start, stop;
  float lapsed;
  float speed;
  float diffx, diffy;

  /* default parameters */
  unsigned int iterations = 5;
  nx = 32;
  ny = 32;
  ty = 1;

  switch(argc) {
    case 5:
      sscanf(argv[4], "%d", &ty);
    case 4:
      sscanf(argv[2], "%d", &nx);
      sscanf(argv[3], "%d", &ny);
    case 2:
      sscanf(argv[1], "%d", &iterations);
      break;
```

```
  case 1:
    break;
  default:
    printf("Incorrect commandline options, exiting...\n");
    exit(1);
}

n = nx * ny;

/* size of matrix B */
bx = 2 * nx - 1;
stride = bx + 1;
by = 2 * ny - 1;
m = stride * by;

if(ty * nx > 1024) {
  printf("%u %u %u NaN NaN NaN NaN\n", n, nx, ny);
  exit(1);
}

cudaError_t err;

/* declarations of matrices B */
float *hBxx, *hBxy, *hByx, *hByy;
float *dBxx, *dBxy, *dByx, *dByy;

/* declarations of vectors p */
float *hpx, *hpy;
float *dpx, *dpy;

/* declarations of resulting vectors */
float *hux, *huy;
float *dux, *duy;
float *hux_gpu, *huy_gpu;

/* declarations of the initial deformation */
float *hwx, *hwy;

/* declaration of the device memory used to perform final sum reduction */
float *dreductionx, *dreductiony;

/* set dimensions of grid and threads */
dim3 dimGrid(ny / ty, 1);
dim3 dimBlock(nx, ty);

/* random seed */
srand( (unsigned)time( NULL ) );

/* filling the grids B with random numbers */
hBxx = (float *)malloc(m * sizeof(float));
hBxy = (float *)malloc(m * sizeof(float));
hByx = (float *)malloc(m * sizeof(float));
hByy = (float *)malloc(m * sizeof(float));
for(ii = 0; ii < m; ii++) {
  if((ii + 1) % stride == 0) {
    hBxx[ii] = 0;
    hBxy[ii] = 0;
    hByx[ii] = 0;
    hByy[ii] = 0;
  }

  hBxx[ii] = (float) rand()/RAND_MAX;
  hBxy[ii] = (float) rand()/RAND_MAX;
  hByx[ii] = (float) rand()/RAND_MAX;
  hByy[ii] = (float) rand()/RAND_MAX;
}

/* create traction vectors */
hpx = (float *)malloc(n * sizeof(float));
hpy = (float *)malloc(n * sizeof(float));
for(ii = 0; ii < n; ii++) {
  hpx[ii] = (float)rand() / RAND_MAX;
  hpy[ii] = (float)rand() / RAND_MAX;
}

/* create initial deformation vectors */
hwx = (float *)malloc(n * sizeof(float));
hwy = (float *)malloc(n * sizeof(float));
for(ii = 0; ii < n; ii++) {
  hwx[ii] = (float)rand() / RAND_MAX;
  hwy[ii] = (float)rand() / RAND_MAX;
}

/* allocate space for the results */
hux = (float *)malloc(n * sizeof(float));
huy = (float *)malloc(n * sizeof(float));
hux_gpu = (float *)malloc(n * sizeof(float));
huy_gpu = (float *)malloc(n * sizeof(float));
memset(hux, 0, n);
memset(huy, 0, n);
```

```
/* allocate memory on device, copy Bxx matrix to device */
err = cudaMalloc((void**)&dBxx, m * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemcpy(dBxx, hBxx, m * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* allocate memory on device, copy Bxy matrix to device */
err = cudaMalloc((void**)&dBxy, m * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemcpy(dBxy, hBxy, m * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* allocate memory on device, copy Byx matrix to device */
err = cudaMalloc((void**)&dByx, m * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemcpy(dByx, hByx, m * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* allocate memory on device, copy Byy matrix to device */
err = cudaMalloc((void**)&dByy, m * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemcpy(dByy, hByy, m * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* allocate memory on device, copy px vector to device */
err = cudaMalloc((void**)&dpx, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemcpy(dpx, hpx, n * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* allocate memory on device, copy py matrix to device */
err = cudaMalloc((void**)&dpy, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemcpy(dpy, hpy, n * sizeof(float), cudaMemcpyHostToDevice);
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* allocate resulting vectors on device and fill with zeros */
err = cudaMalloc((void**)&dux, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemset(dux, 0, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMalloc((void**)&duy, n * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMemset(duy, 0, n * sizeof(float));
```

```
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}

/* allocate resulting vectors on device and fill with zeros */
err = cudaMalloc((void**)&dreductionx, ny / ty * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
err = cudaMalloc((void**)&dreductiony, ny / ty * sizeof(float));
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}


/* determine start time */
err = cudaThreadSynchronize();
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
gettimeofday(&start, NULL);


kk = 0;
ii = 0;

/* perform kernel invocations */
for(i = 0; i < iterations; i++) {
  for(iy = 0; iy < ny; iy++) {
    kk = (ny - 1 - iy) * stride + (nx - 1);
    ii = iy * nx;
    for(ix = 0; ix < nx; ix++) {

      Kernel_inner_prod<<<dimGrid, dimBlock, ty * 2 * nx * sizeof(float)>>>
      (&dBxx[kk], &dBxy[kk], &dByx[kk], &dByy[kk],
      dpy, dpx,
      nx, ny, stride, ix, iy, ty, dreductionx, dreductiony);

      Kernel_reduce<<<2, ny / ty, ny / ty * sizeof(float)>>>
      (dreductionx, dreductiony, ny / ty, &dux[ii], &duy[ii]);

      /* check if an error occured when launching the kernel */
      err = cudaGetLastError();
      if(err != cudaSuccess) {
        printf("Cuda error: %s\n", cudaGetErrorString(err));
        exit(1);
      }

      /* copy result to host, part of the time measurement */
      err = cudaMemcpy(&hux_gpu[ii], &dux[ii], 1 * sizeof(float), cudaMemcpyDeviceToHost);
      if(err != cudaSuccess) {
        printf("Cuda error: %s\n", cudaGetErrorString(err));
        exit(1);
      }
      err = cudaMemcpy(&huy_gpu[ii], &duy[ii], 1 * sizeof(float), cudaMemcpyDeviceToHost);
      if(err != cudaSuccess) {
        printf("Cuda error: %s\n", cudaGetErrorString(err));
        exit(1);
      }

      hux_gpu[ii] += hwx[ii];
      huy_gpu[ii] += hwy[ii];

      kk--;
      ii++;
    }
  }
}

/* determine stop time and calculate lapsed time per iterations */
err = cudaThreadSynchronize();
if(err != cudaSuccess) {
  printf("Cuda error: %s\n", cudaGetErrorString(err));
  exit(1);
}
gettimeofday(&stop, NULL);
lapsed = stop.tv_sec - start.tv_sec + 1e-6 * (stop.tv_usec - start.tv_usec);
lapsed /= iterations;
speed = mflops(lapsed, n);

/* run on cpu for numerical comparison */
for(iy = 0; iy < ny; iy++) {
  for(ix = 0; ix < nx; ix++) {
    ii = iy * nx + ix;
    hux[ii] = hwx[ii] + innerprod_B(hBxx, hpx, nx, ny, stride, iy, ix) +
     innerprod_B(hBxy, hpy, nx, ny, stride, iy, ix);
```

```
      huy[ii] = hwy[ii] + innerprod_B(hByx, hpx, nx, ny, stride, iy, ix) +
       innerprod_B(hByy, hpy, nx, ny, stride, iy, ix);
    }
  }

  /* calculate relative difference */
  diffx = normdiff(hux, hux_gpu, n) / norm(hux, n);
  diffy = normdiff(huy, huy_gpu, n) / norm(huy, n);

  printf("%u %u %u %u %f %f %f %f\n", n, nx, ny, ty, lapsed, diffx, diffy, speed);

  /* free all allocated memory space */
  free(hBxx);
  free(hBxy);
  free(hByx);
  free(hByy);

  free(hpx);
  free(hpy);

  free(hwx);
  free(hwy);

  free(hux);
  free(huy);
  free(hux_gpu);
  free(huy_gpu);

  err = cudaFree(dBxx);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(dBxy);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(dByx);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(dByy);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(dpx);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(dpy);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(dux);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(duy);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
    err = cudaFree(dreductionx);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }
  err = cudaFree(dreductiony);
  if(err != cudaSuccess) {
    printf("Cuda error: %s\n", cudaGetErrorString(err));
    exit(1);
  }

  return 0;
}
```

Final CPU reduction

To perform the final reduction on the CPU instead of the GPU, the second kernel launch is replaced with the following code snippet.

```
/* copy values for final reduction to host */
        err = cudaMemcpy(hreductionx, dreductionx, ny / ty * sizeof(float), cudaMemcpyDeviceToHost);
        if(err != cudaSuccess) {
          printf("Cuda error: %s\n", cudaGetErrorString(err));
          exit(1);
        }
        err = cudaMemcpy(hreductiony, dreductiony, ny / ty * sizeof(float), cudaMemcpyDeviceToHost);
        if(err != cudaSuccess) {
          printf("Cuda error: %s\n", cudaGetErrorString(err));
          exit(1);
        }

        hux_gpu[ii] = hwx[ii];
        huy_gpu[ii] = hwy[ii];
        for(j = 0; j < ny / ty; j++) {
          hux_gpu[ii] += hreductionx[j];
          huy_gpu[ii] += hreductiony[j];
        }
```

Coalesced memory read transaction

The perform experiments with coalsced memory read transaction when reading matrix B from global memory, the kernel invocation is replaced by the following statement.

```
Kernel_inner_prod<<<dimGrid, dimBlock, ty * 2 * nx * sizeof(float)>>>
(dBxx, dBxy, dByx, dByy, dpy, dpx, nx, ny, stride, ix, iy, ty, dreductionx, dreductiony);
```

A.2.4 Utility functions

```
#include <math.h>
#include <stdio.h>

/* Direct inner product, in terms of matrix B */
float innerprod_B(float *B, float *p,
unsigned int nx, unsigned int ny, unsigned int bx, unsigned int iy, unsigned int ix) {

  unsigned int jx, jy, ky, kx;
  unsigned int jj, kk;

  float res;

  res = 0;

  for(jy = 0; jy < ny; jy++) {
    ky = jy - iy + ny - 1;
    kx = nx - 1 - ix;
    jj = jy * nx;
    kk = ky * bx + kx;
    for(jx = 0; jx < nx; jx++) {
      res += B[kk] * p[jj];
      jj++;
      kk++;
    }
  }

  return res;

}

float normdiff(float *u1, float *u2, unsigned int n) {
/* calculate the norm of the difference of u1 and u2 */

  unsigned int ii;
  float res, update;

  res = 0;

  for(ii = 0; ii < n; ii++) {
    update = u1[ii] - u2[ii];
    update *= update;
    res += update;
  }

  res = pow(res, 0.5);

  return res;

}

float norm(float *u, unsigned int n) {
/* calculate the norm of a vector */
```

```
  unsigned int ii;
  float res;

  res = 0;

  for(ii = 0; ii < n; ii++) {
    res += u[ii] * u[ii];
  }


  res = pow(res, 0.5);

  return res;

}

float mflops(float time, unsigned int n) {

  float size, speed;

  if(time == 0)
    return 0;

  size = n;

  speed = size * (8 * size + 6) / time / 1e6;

  return speed;

}

bool ispowof2(unsigned int n) {

  if(n == 0)
    return false;

  while(n != 1) {
    if(n % 2 == 0)
      n /= 2;
    else
      return false;
  }

  return true;

}

void printu(float* u1, float* u2, int n) {

  int ii;

  for(ii = 0; ii < n; ii++) {
    printf("%5.4f", u1[ii]);
    (u2 ? printf(" %5.4f\n", u2[ii]) : printf("\n"));
  }

  printf("\n");

  return;

}

float average(float* values, unsigned int n) {

  unsigned int i;
  float sum;

  sum = 0;

  for(i = 0; i < n; i++) {
    sum += values[i];
  }

  return sum / n;
}

void print(float *vector, unsigned int n) {

  unsigned int i;

  for(i = 0; i < n; i++) {
    printf("%f\n", vector[i]);
  }
  printf("\n");
}

void timing(struct timeval *start, struct timeval *stop,
float *lapsed, float *speed, float *averagelapsed, float *averagespeed,
float *varlapsed, float *varspeed, unsigned int iterations, unsigned int n) {
```

```
  int i;

  for(i = 0; i < iterations; i++) {
    lapsed[i] = stop[i].tv_sec - start[i].tv_sec + 1e-6 * (stop[i].tv_usec - start[i].tv_usec);
    speed[i] = mflops(lapsed[i], n);
  }

  *averagelapsed = average(lapsed, iterations);
  *averagespeed = average(speed, iterations);

  *varlapsed = variance(lapsed, iterations, *averagelapsed);
  *varspeed = variance(speed, iterations, *averagespeed);

  return;

}
```

Bibliography

[1] NVIDIA CUDA Programming Guide 2.2. Technical report, NVIDIA Corporation, 2009.

[2] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann Publishers, 2010.

[3] C.W. Oosterlee and C. Vuik. *Scientific Computing (wi4201)*. Delft University of Technology, 2009.