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

Implementeren van een versnelling van de CONTACT-programmatuur met behulp van de Graphics Processing Unit.

(Engelse titel: Implementataion of a speeding-up of 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

ONNO LEON MEIJERS

Delft, Nederland
AUGUST 2012

BSc verslag TECHNISCHE WISKUNDE

"Implementeren van een versnelling van de CONTACT-programmatuur met behulp van de Graphics Processing Unit."

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

ONNO LEON MEIJERS

Technische Universiteit Delft

Begeleider

Prof.dr.ir. C. Vuik

Dr.ir. E.A.H. Vollebregt

Overige commissieleden

Dr. J.G. Spandaw

Prof.dr.ir. H.X. Lin

AUGUST, 2012

Delft

Contents

Chapter 1

Introduction

When a train runs on rails there is contact between the wheels of the train and the rails. This contact can lead to wear and resistance. If you do not research this abrasion you can have mayor crashes. To simulate the contact there is a package called CONTACT[1]. This program is created to calculate deformations in a three-dimensional frictional contact problem. VORtech creates mathematical software for governments departments and commercial clients. Their software program CONTACT, which is used for several computations that arise when dealing with mechanical contact problem, originates from professor J.J. Kalker at Delft University of Technology. I would like to thank VORtech for making this project available.

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

The GPU(Graphics Processing Unit) is known for its parallel power, since its consist of a lot of separate processors, each able of performing floating point operations simultaneously. By the work of Pieter Loof, see [2], CONTACT now uses Fourier based convolutions. The calculation of a FFT is perfect to calculate in parallel, so when we implement these calculations on the GPU we should decrease the calculation time even more.

The goal of this Bachelor Project is to implement the Fast Fourier Transforms of CON-

[1] http://www.kalkersoftware.org/index.php?mid=home

TACT on the Graphics Processing Unit by using CUDA, instead of calculating the Fast Fourier Transforms on the CPU (Central Prossesing Unit).

Chapter 2

Problem Definition

2.1 The Normal Contact Problem

In the following we will explain the problem with an example. We take a simple stationary train wheel and a rail. We assume for this simple example that the wheel is a disc and the rail is a cylinder. When we look in the real world at the train we see that there is contact between wheel and the rail. And by the weight of the train, pressure will be created at the bottom of the wheel on the rails. Both objects are a little deformed by this force. The problem is to find the pressure and the corresponding deformation.

We call the contact area C, and the rest of the area that is not in contact, is called the exterior area E. We know the original shapes of the wheel and the rail when they are not in contact. So we know the difference in heights. The undeformed distance.

$$h(x,y) = h^{(\text{rail})}(x,y) - h^{(\text{wheel})}(x,y) \quad \text{for} \quad (x,y) \in C \cup E \tag{2.1}$$

We call this $h(x,y)$. In general some points of $h(x,y)$ will be negative, in this situation all $h(x,y) \in C$ will be non-positive. This means that the wheel and rail penetrate each other, but this can not happen in the real world. We will call $\underline{x} = (x,y)$ We know that the materials will deform a bit. This will cause a vertical displacement of the points in the contact area in the wheel and the rail. We call $u(x)$ the displacement difference through deformation and we will call this the deformation from now on. It is given by

$$u(\underline{x}) = u^{(\text{rail})}(\underline{x}) - u^{(\text{wheel})}(\underline{x}). \tag{2.2}$$

Then we have a total difference $e(\underline{x})$, that we call the deformed difference. It is given by

$$e(\underline{x}) = h(\underline{x}) + u(\underline{x}). \tag{2.3}$$

When we take a look in the real world we see that there is no penetration so the total distance must be larger or equal to zero.

$$e(\underline{x}) \geq 0 \tag{2.4}$$

And we see that for the points that are in contact the distance is zero.

$$e(\underline{x}) = 0 \quad \text{for} \quad \underline{x} \in C \tag{2.5}$$

Because we know $h(\underline{x})$ from (2.3), we now have a formula for $u(\underline{x})$:

$$u(\underline{x}) = -h(\underline{x}) \quad \text{for} \quad \underline{x} \in C. \tag{2.6}$$

Then we have $p(\underline{x})$, which is the pressure at point $\underline{x}$. We assume that the pressure is compressive, which means that there is no stickiness between the wheel and the rail.

$$p(\underline{x}) \geq 0 \tag{2.7}$$

Furthermore, we assume that there is no pressure in the points that are not in the contact area.

$$p(\underline{x}) = 0 \quad \text{for} \quad \underline{x} \in E \tag{2.8}$$

Finally, we know the relationship between $p(\underline{x})$ and $u(\underline{x})$ for each point in the contact area:

$$u(\underline{x}) = \int_{\underline{y} \in C} A(\underline{x}, \underline{y}) p(\underline{y}) d\underline{y}, \tag{2.9}$$

Here $A(\underline{x}, \underline{y})$ is the influence coefficient of the pressure at point $\underline{y}$ on deformation at point $\underline{x}$. $A(\underline{x}, \underline{y})$ is analytically determined and thus given. We calculate $A(\underline{x}, \underline{y})$ as follows. $B(\underline{x})$ is the function of the deformation in each point $\underline{x}$ caused by 1 unit of force at $B(\underline{0})$. So we calculate B by putting 1 unit of force at the center of our material and then we measure the difference of each point with its original position. Then we have the expression $A(\underline{x}, \underline{y}) = B(\underline{x} - \underline{y})$ And note that we have a convolution here:

$$u(\underline{x}) = \int_{\underline{y} \in C} B(\underline{x} - \underline{y}) p(\underline{y}) d\underline{y} \tag{2.10}$$

See [2] for more detailed derivation and more formulas.

2.2 Discretization and Calculation

We can not solve equations (2.3)-(2.5),(2.7)-(2.9) analytically, so we discretize these equations. In CONTACT we choose a potential contact area and choose a discretization grid of $mx \times my$ rectangles of size $\delta x \times \delta y$. We assume that the elements have a constant pressure over the whole element, so $p = constant$. The deformation u has its value at the center of the element and the same holds for h and e. A_{ij} is the influence coefficient of the pressure at the center of element j to the center of element i. In Figure 2.1 we see a potential contact area with a discretization grid.

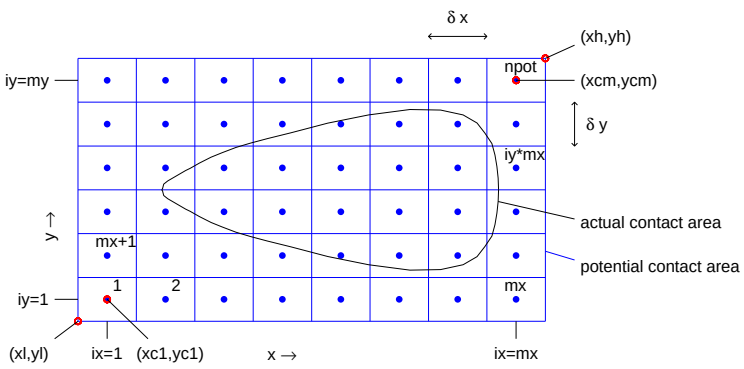

Figure 2.1: Example of a discretized contact area

So all the equations change to its discrete counterpart and we change the notation. We call $h_{ix,iy} = h(xci, yci)$ and also for the other functions, and $A_{(ix,iy),(jx,jy)} = A((xci, yci), (xcj, ycj))$

$$\begin{aligned}
h_{(ix,iy)} &= h^{(\text{rail})}_{(ix,iy)} - h^{(\text{wheel})}_{(ix,iy)} \\
u_{(ix,iy)} &= u^{(\text{rail})}_{(ix,iy)} - u^{(\text{wheel})}_{(ix,iy)} \\
e_{(ix,iy)} &= h_{(ix,iy)} + u_{(ix,iy)} \geq 0 \\
e_{(ix,iy)} &= 0 \quad \text{for} \quad x \in C \\
u_{(ix,iy)} &= -h_{(ix,iy)} \\
p_{(ix,iy)} &\geq 0 \\
p_{(ix,iy)} &= 0 \quad \text{for} \quad x \in E \\
u_{(ix,iy)} &= \textstyle\sum_{(jx,jy)} A_{(ix,iy),(jx,jy)} p_{(jx,jy)} \\
A_{(ix,iy),(jx,jy)} &= B_{(ix-jx),(iy-jy)}
\end{aligned}$$

The matrix A is $mx \cdot my \times mx \cdot my$. Since we take $p_{jx,jy} = 0$ outside the contact area, we have

$$Ap = u.$$

u and p are here vectors and A is the matrix.

In CONTACT everything is stored in 1d arrays and as we can see in Figure 2.1 the elements (ix, iy) are labeled by the following formula: $I = (iy - 1) \cdot mx + ix$. So we get for a point $\underline{x}$ in the center of an element:

$$\underline{x} = (xci, yci) = (ix, iy) = \underline{x}_I.$$

Note that we use both u,U and p,P. These are equivalent by U is the grid function of size $mx \times my$ and u is the vector with horizontal numbering as we can see in Figure 2.1. u is $mx \cdot my$ large. The same holds for p and P.

We choose the whole potential contact area the contact area C_0 for the first iteration and then we can compute u within the contact area, p outside the contact area and their relationships inside the contact area with the matrix-vector product. We use the CG (Conjugate Gradient) method to find the solution for p so that we can solve the equations. We will explain some more about CG in the next subsection. After we have found this p, CONTACT determines on the basis of certain criteria which elements are for the next iteration $i = 1$ in the contact area and which are not. For this project it is not needed to define these criteria in much detail. Next we start all over again and set $p = 0$ outside the new contact area C_1 and $u = -h$ within. And we use the CG method again to find for this new contact area C_1 the best possible p. So we solve the system that there is $p = 0$ outside the contact area and $Ap = u$ within. We go on until the absolute error is less than a certain number, for example 10^{-5}. In our example this means that we move the wheel up and down until we have an equilibrium.

2.3 CG method in CONTACT

We do not need to know a lot of the CG method in general, except that it is an iterative solution method searching in the space $\text{span}\{u, Au, A^2u, \ldots\}$ to find p at the given conditions, such that the error is minimal in a certain norm.

In the CG method we need to calculate u with the matrix-vector product Ap. Because u is the vector representation of U, these are the same. We can calculate U or u in three ways. We have already seen that we can calculate u with the matrix-vector product Ap. Second we can calculate U by superposition. The elements of grid P contains the values times the unit of force that work on the contact area. So U is a superposition of B times the constant of P for all elements of P. See Chapter 4 for an example. Third U can be calculated by replacing A with B and then the matrix-vector multiplication is written as a 2D convolution. The formula for this becomes as follows:

$$U(ix, iy) = \sum_{jy} \sum_{jx} B(ix - jx, iy - jy) \cdot P(jx, jy)$$

We can calculate the convolution directly or we can calculate this 2D convolution within the Fourier domain. To calculate the convolution directly we will explain the geometric interpretation. We lay P over B and we lay the right-top point of P at the center of B. We flip P and the left-bottom element of U is the sum of this sub block of B multiplicated per element with the flipped P. For the other elements of U we shift the flipped P over B as many steps as we take from the left-bottom element in U. Note that B only needs to be of the size $(2nx-1)\times(2ny-1)$. See Chapter 4 for an example. Note that this way of shifting the flipped P over B can also be done when we take the flipped B and place P over the right- bottom element and shift P to the left and up. We will take flipped P and this resembles in the Figure 2.2.

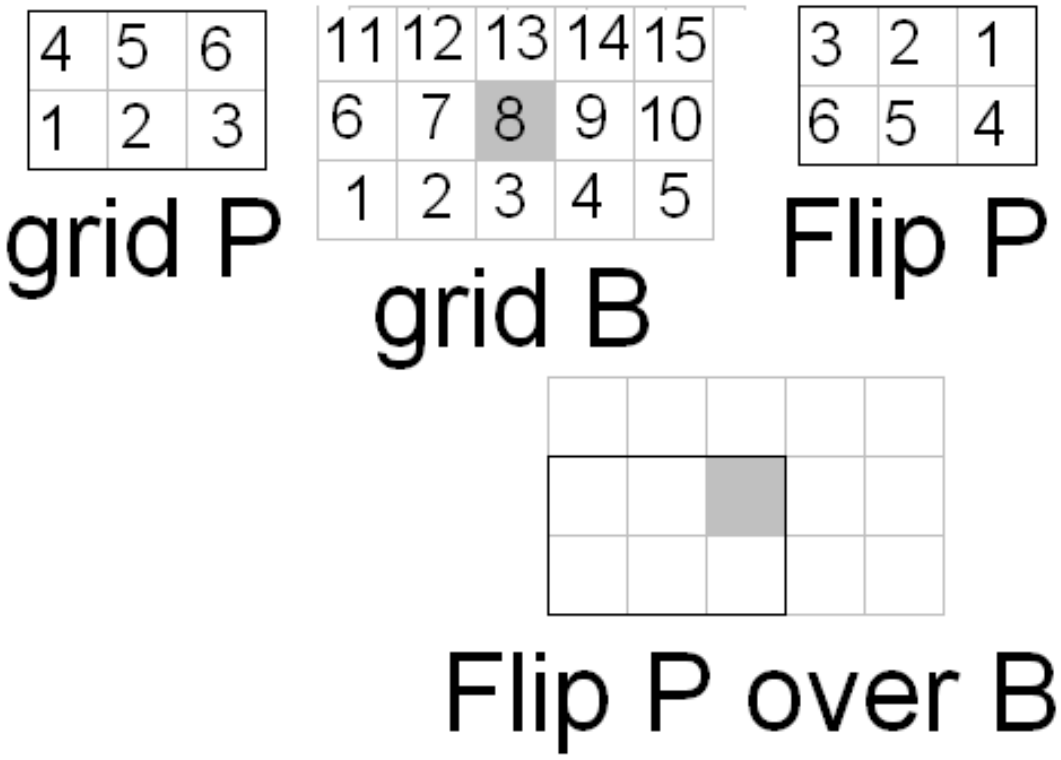

Figure 2.2: P is shifted over B

To calculate a 2D convolution in a Fourier domain, it is necessary to first have to compute the 2D Fourier transforms of the grids. Then the transformed grids must be multiplied element wise, so the grids must be of equal size. This means that the input grid P has to be enlarged to the same size as the influence grid B, by adding zeros. This results in P'. After the element wise multiplication of B^f and P'^f the result U'^f has to be transformed back to the normal domain. But because we have added zeros to P, the result U' contains to much information. The grid U can be found in the bottom right corner of U'. For an example see Chapter 4.

The calculation within the Fourier domain will be faster for larger grids. Because if you calculate the convolution directly it will cost $O((mx \cdot my)^2)$ operation this is about the same as the matrix-vector product. And if you calculate the convolution in the Fourier domain it will cost $O(mx \cdot my \log(mx \cdot my))$ if you use a fast algorithm, see Section 3.2 for this. That is why CONTACT is using this method with Fourier transformations. This is the reason that the CG method of CONTACT is faster. The normal CG method calculates for each iteration k the

matrix-vector product Ap^k. So this is $O((mx \cdot my)^2)$. The CG method of CONTACT uses a convolution in a Fourier domain as explained above and is $O(mx \cdot my \log(mx \cdot my))$. So we need to calculate Fourier transformations of the matrices. In Section 3.1 we give a reminder of the definitions of Fourier transforms. We observe that CONTACT uses for each iteration k 3 FFTs to calculate the deformation.

Because the calculation of an FFT can be perfectly done in parallel, we can use the Graphics Processing Unit (GPU). The GPU is known for its parallel power, since it consist of a lot of separate processors, each capable of performing floating point operations simultaneously. So if we implement these calculations in an efficient way on the GPU we should decrease the calculation time. This will be clarified in Chapter 5.

2.4 Research Questions

The goal of this Bachelor Project is to implement the FFTs of CONTACT on the Graphics Processing Unit, instead of calculate the Fast Fourier Transforms on the Central Prossesing Unit. After we have implemented this, we try to answer the following research questions:

- How much faster is it?
- Do we need to speed this process up even further?
- What is better: CUDA or Portland Fortran?
- What becomes the new bottleneck?

Chapter 3

Fourier based Convolutions

CONTACT is using the Fourier domain, therefore we first give a short reminder of Fourier transformation and convolutions and its properties.

3.1 Fourier transforms

3.1.1 Continuous

A continuous Fourier transformation is a transformation that transforms a function $f(x)$ in a new function $\hat{f}(\omega)$:$\mathbb{R} \to \mathbb{C}$. The function $\hat{f}(\omega)$ is the frequency spectrum of $f(x)$. It uses an integral of the function times complex e-powers which are the frequencies to calculate the other function. Now we give the definition:

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \omega} \mathrm{d}x$$

with i the imaginary unit. The inverse transformation is under suitable conditions given by:

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\omega) e^{2\pi i \omega x} \mathrm{d}\omega$$

with i the imaginary unit. Next we give a definition of a convolution. A convolution of a function f and g is defined as:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(y) g(x-y) \mathrm{d}y = \int_{-\infty}^{\infty} f(x-y) g(y) \mathrm{d}y$$

Now the Fourier transform has the property that if you have to calculate a convolution in the normal domain, it is the same as multiplying it in the Fourier domain. So if we have the functions f and g and the convolution $h(z)$=$f * g$ then $\hat{h} = \hat{f} \cdot \hat{g}$.

3.1.2 Discrete 1D

In this application we have a discrete system, so we need the discrete Fourier transform. The discrete Fourier transform makes from a vector $\mathbf{v} \in \mathbb{C}^n$ a vector $\hat{\mathbf{v}} \in \mathbb{C}^n$, where $\hat{\mathbf{v}}$ is the frequency spectrum of $\mathbf{v}$. The discrete 1D Fourier transform $\hat{\mathbf{v}}_l$ of a vector $\mathbf{v}$ with length n is defined as:

$$\hat{v}_l = \sum_{k=1}^{n} v_k e^{\frac{-i2\pi}{n}(k-1)l} \quad \text{for} \quad l = 1..n,$$

with i the imaginary unit. To transform the Fourier transform back to the normal domain we need the inverse Fourier transform. So next we give the definition of the 1D discrete inverse Fourier transform. The inverse Fourier transform is comparable to the Fourier transform. There are two differences. The first one is the minus sign before the imaginary unit and the second one is that it is scaled by $\frac{1}{n}$ so we get our vector $\mathbf{v}$ back if we do the following:

$$v_j = \frac{1}{n}\sum_{k=1}^{n} \hat{v}_k e^{i\frac{2\pi}{n}(k-1)j} \quad \text{for} \quad j = 1..n$$

A discrete 1D convolution of vector $\mathbf{f}$ of length nf and vector $\mathbf{g}$ of length ng with $ng > nf$ is defined as:

$$(\mathbf{f} * \mathbf{g})_k = \sum_{m=1}^{nf} f_m g_{k-m} \quad \text{for} \quad k = 1..(ng - nf),$$

A convolution in the normal domain can be seen as the next element of f is multiplied with the previous element of g, because when m is increased with 1, $k-m$ goes down by 1. This is about the same as we shift P over B.

3.1.3 Discrete 2D

For a discrete matrix V of size $n \times m$ the 2D Fourier transform $\hat{V}$ is defined as:

$$\hat{V}_{x,y} = \sum_{k=0}^{n-1}\sum_{l=0}^{m-1} V_{k,l} e^{i2\pi\left(\frac{kx}{n}+\frac{ly}{m}\right)} \quad \text{for} \quad x = 1..n,\ y = 1..m,$$

with i the imaginary unit. In the same way as the 1D inverse transform, the formula for the 2D inverse transform can be found out of the normal 2D transformations. So there is no minus sign before the imaginary unit and we use a scaling-factor of $\frac{1}{nm}$ to get the normal matrix back:

$$V_{j,q} = \frac{1}{nm}\sum_{k=0}^{n-1}\sum_{l=0}^{m-1} F_{k,l} e^{i2\pi\left(\frac{kj}{n}+\frac{lq}{m}\right)} \quad \text{for} \quad j = 1..n,\ q = 1..m$$

The 2D convolution is just a simple expansion of the 1D convolution. For matrices F of $mx \times my$ and G of $mk \times ml$, $mk > mx$ and $ml > my$ a convolution is defined as:

$$(F * G)[k,l] = \sum_{n=1}^{my}\sum_{m=1}^{mx} F_{m,n} G_{k-m,l-n} \quad \text{for} \quad k = 1,2,..,(mk - mx),\ l = 1,2,..,(ml - my)$$

Also in the discrete case we can calculate a convolution of F and G by multiplying $\hat{F}$ and $\hat{G}$.

Now we have the definitions but there are faster ways to calculate the DFTs of a matrix, see Section 3.2 for some more information about this. When we use the faster algorithms we have to perform $O(n\log(n))$ operations instead of $O(n^2)$ operations.

3.2 Fast Fourier Transform

FFT stands for Fast Fourier Transform. This is not a special way of transforming. It is just a method to speed up the calculations of the Fourier transforms. The idea is that a transform over n elements can be split into two transforms over $\frac{n}{2}$. Instead of approximately n^2 multiplications and additions, only $2\left(\frac{n}{2}\right)^2 = \frac{n^2}{2}$ calculations have to be done. But these parts can be split up again. This splitting can be continued, making the needed amount of calculations smaller and

smaller. By using the FFT, a Fourier transform of a vector of length n is only $O(n \log n)$ instead of $O\left(n^2\right)$. This method works best when n is a power of 2, because then the successive splitting in two parts is nice and neat. If n is not a power of 2, FFT is still faster. Pieter Loof has shown in [1, Section 9.5, p. 56-59] that you can choose your grid size smart with small prime divisors, so that the calculation time is optimal.

Chapter 4

Example

In this chapter we will show all the calculations of U with a simple example. We take the following problem.

$$B = \begin{array}{|c|c|c|c|c|}\hline 0&0&0&0&0\\\hline 0&2&3&-1&0\\\hline 0&0&1&0&0\\\hline\end{array} \quad P = \begin{array}{|c|c|c|}\hline 1&0&0\\\hline 0&-1&0\\\hline\end{array}$$

superposition ofBP

First we will calculate U by superposition. So U is the sum of the value of P times B laid over the positions of P and taken the same grid size as P. So U is B times 1 in the top left corner plus B times -1 in the south center position.

$$U = 1 * \begin{array}{|c|c|c|}\hline 3&-1&0\\\hline 1&0&0\\\hline\end{array} - 1 * \begin{array}{|c|c|c|}\hline 0&0&0\\\hline 2&3&-1\\\hline\end{array} = \begin{array}{|c|c|c|}\hline 3&-1&0\\\hline -1&-3&1\\\hline\end{array}$$

matrix-vector product

Second we write $u = Ap$ by the formulas given in Section 2.1. We have $I = (iy-1) * nx + ix$ to write grids as vectors. This means that we have the following numbering:

$$\begin{array}{|c|c|c|}\hline 4&5&6\\\hline 1&2&3\\\hline\end{array}$$

For instance this gives

$$p = \begin{pmatrix} 0\\-1\\0\\1\\0\\0 \end{pmatrix}$$

$$u = \begin{pmatrix} 3&2&0&1&0&0\\ -1&3&2&0&1&0\\ 0&-1&3&0&0&1\\ 0&0&0&3&2&0\\ 0&0&0&-1&3&2\\ 0&0&0&0&-1&3 \end{pmatrix} \begin{pmatrix} 0\\-1\\0\\1\\0\\0 \end{pmatrix} = \begin{pmatrix} -1\\-3\\1\\3\\-1\\0 \end{pmatrix}.$$

convolution directly

Third we will calculate U by multiply the grids. So we take flip(P)=$\begin{array}{|c|c|c|}\hline 0&-1&0\\\hline 0&0&1\\\hline\end{array}$ and we lay it

on top of a sub grid of B and we multiply the elements on top of each other and take the sum of this. So for example

$$U(1) \text{ is } \begin{array}{|c|c|c|}\hline 0 & -1 & 0 \\ \hline 0 & 0 & 1 \\ \hline \end{array} \text{ on top of } \begin{array}{|c|c|c|}\hline 0 & 2 & 3 \\ \hline 0 & 0 & 1 \\ \hline \end{array} = 0*0+0*0+1*1+0*0+-1*2+0*3 = -1$$

and

$$U(5) \text{ is } \begin{array}{|c|c|c|}\hline 0 & -1 & 0 \\ \hline 0 & 0 & 1 \\ \hline \end{array} \text{ on top of } \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 2 & 3 & -1 \\ \hline \end{array} = 0*2+0*3+1*-1+0*0+-1*0+0*0 = -1$$

convolution in Fourier domain

Finally we will calculate U by the convolution in the Fourier domain. First we have to enlarge P so it has the same size as B.

$$P' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Then we will calculate the Fourier transform of B and P'. i is the imaginary unit and the numbers are rounded.

$$\hat{B} = \begin{pmatrix} 5 & -1.81-4.84i & -0.69+3.58i & -0.69-3.58i & -1.81+4.84i \\ -2.50-2.60i & -2.27+2.59i & 1.80-0.66i & -1.11+2.92i & 4.08-2.26i \\ -2.5+2.60i & 4.08+2.26i & -1.11-2.92i & 1.80+0.66i & -2.27-2.59i \end{pmatrix}$$

$$\hat{P'} = \begin{pmatrix} 0 & 0.69+0.95i & 1.81+0.59i & 1.81-0.59i & 0.69-0.95i \\ 1.5+0.87i & 1.98-0.21i & 1.10-0.99i & 0.09-0.41i & 0.33+0.74i \\ 1.5-0.87i & 0.33-0.74i & 0.09+0.41i & 1.10+0.99i & 1.98+0.21i \end{pmatrix}$$

Then if we element wise multiply $\hat{B}$ and $\hat{P'}$ we get:

$$\hat{U'} = \begin{pmatrix} 0 & 3.35-5.07i & -3.35+6.07i & -3.35-6.07 & 3.35+5.07i \\ -1.50-6.06i & -3.95+5.59i & 1.33-2.51i & 1.09+0.70i & 3.03+2.29i \\ -1.50+6.06i & 3.03-2.29i & 1.09-0.70i & 1.33+2.51i & -3.95-5.59i \end{pmatrix}$$

Then we transform $\hat{U'}$ back with the inverse transform and we get U'.

$$U' = \begin{pmatrix} 0 & 0 & 0 & -1 & 0 \\ 0 & 2 & 3 & -1 & 0 \\ 0 & 0 & -1 & -3 & 1 \end{pmatrix}$$

Now we take the right-bottom sub grid to retrieve U

$$U = \begin{array}{|c|c|c|}\hline 3 & -1 & 0 \\ \hline -1 & -3 & 1 \\ \hline \end{array}$$

We see that we can compute the deformation in different ways. We can use vectors and grids, matrix-vector products, convolutions or FFT's. And we see that all these methods give the same answers. Therefor we choose the fastest method. The fastest method is the method using FFT's. Next will implement this on the GPU to solve the problem even faster.

Chapter 5

Implementation

5.1 Options

For this project we have to implement the calculations of FFT's of CONTACT onto the GPU. Because CONTACT is in Fortran we have the following options:

- We could use CUDA Fortran by The Portland Group.
- We could use CUDA and use the interoperability of Fortran and C.

If we use CUDA Fortran then we use software made by The Portland Group. They have made a package that will do the second option for you. So we will ask in Fortran to calculate the FFT of something and The Portland Group has made a set of function calls that does this. But you do not know that it will be as fast as option 2 when you make the CUDA calls yourself in C. So first we choose to write our own CUDA program that calculates the FFT. Then we have to call the C function by Fortran and look out for problems that may occur when transferring data from Fortran to C and back.

5.2 Implementation path

Now we will explain how we got to the implementation of the 2D FFT's of CONTACT using the GPU. To implement the 2D FFT's on the GPU we have followed the following steps. Note that we use the words "GPU" and "device", these words denotes the same processor.

5.2.1 Fortran 1D FFT on CPU

We started with a small program that calculates a 1D FFT of a test influence grid using the FFTW library [1]. To calculate a FFT using the FFTW library we need to make a "plan" with `dfftw_plan_dft_r2c_1d`. When we have called for a plan, the computer allocates memory for the process and chooses the fastest way to calculate the FFT of this grid size. Then we need to execute the plan with `dfftw_execute_dft_r2c`. Finally we need to destroy the plan with `dfftw_destroy_plan`. If we use the CUFFT library [2] we do about the same. For the 1D FFT we need to tell the compiler where the FFTW library is and use the `-fftw3 -lm` options. Below you find the most important lines of this program.

[1] FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data. It also has a Fortran interface, for more information see www.fftw.org

[2] The CUFFT library provides a simple interface for computing FFTs. The CUFFT library uses CUDA and the GPU to calculate FFTs faster. For more information see, developer.nvidia.com/cuda/cufft

```
ifort 1Dfftw.f -L$(FFTW)/lib -fftw3 -lm
program fft
include 'fftw3.f'

call dfftw_plan_dft_1d(integer*8 plan,integer N,double complex in, &
          double complex out,FFTW_FORWARD, FFTW_ESTIMATE )
call dfftw_execute_dft(integer*8 plan, double complex in, double complex out)
call dfftw_destroy_plan(plan)
```

Where `$(FFTW)` the path is to where the library is located at your computer.

5.2.2 Fortran & C on CPU

After we have made the FFTW file, we write a simple C file that calculates the square of an integer and call this with a simple Fortran program that will call this function by sending an Integer. We do this to find out how the C and Fortran work together with a simple program, like the square function. Now we have to watch out for interoperability of C and Fortran so we have to add an _ (underline) behind the function name because Fortran does this with its own function calls. We send an `integer*4` in Fortran to an int in C because these are equivalent and we need to tell the function that it can be called from outside using extern[4]. Now we create a `.o` file out of the C file.

```
gcc -c TestC.c
ifort simple.f TestC.o

extern "C" void square_(int *n){
*n=(*n)*(*n)
}

program simple
call square(integer n)
```

5.2.3 Fortran & C, 1D FFT on GPU

Implementation with 1 C function
Next we write a larger function in CUDA, than the square function, that calculates a 1D FFT and write a small Fortran program that makes a test influence grid and calls the function in C to calculate its 1D FFT. Now we have to tell the compiler where to find the CUDA library and the CUFFT function.

```
nvcc  -L$(cuda)/lib64 -lcufft -c myfft.cu
ifort -L$(cuda)/lib64 -I$(cuda)/include -lcufft -lcuda main.f myfft.o

#include <cufft.h>
#include "cuda.h"

extern "C" void myfft(int *n, cufftComplex *data){

cudaMalloc((void**) &d_data, size);
cudaMemcpy(d_data, data, size,cudaMemcpyHostToDevice);
cufftPlan1d(&plan,*n,CUFFT_C2C)
```

```
cudaExecC2C(plan, (cufftComplex *) d_data, (cufftComplex *)d_data, &
            CUFFT_FORWARD);
cudaMemcpy(data, d_data, size,cudaMemcpyDeviceToHost);
cufftDestroy(plan);
cudaFree(d_data);
}

program main
call myfft(integer N, complex in)
```

Implementation with 3 C functions

Then we write a C program that contained 3 separate functions: a function that makes the FFT-plan, a function that executes the plan, and a function that destroys the plan. Finally we write a Fortran program that makes the test and calls the separate functions.

```
extern "C" void
cufftplan1d_(cufftHandle *plan, int *n){
cufftPlan1d(plan, *n, CUFFT_C2C,1);
return;}

extern "C" void cufftexecc2c_(cufftHandle *plan,int *n,cufftComplex *idata, &
                              cufftComplex *odata){
cudaMalloc(&d_data, size);
cudaMemcpy(d_data, idata, size, cudaMemcpyHostToDevice);
cufftExecC2C(*plan, d_data, d_data,CUFFT_FORWARD);
cudaMemcpy(odata, d_data, size, cudaMemcpyDeviceToHost);
cudaFree(d_data);
}

extern "C" void destroy_plan_(cufftHandle *plan){
cufftDestroy(*plan);
return;
}

program main
call cufftPlan1d(integer plan,integer N)
call cufftExecC2C(intger plan,integer N,complex in[N],complex out[N])
call destroy_plan(integer plan)
```

We have written here the most important lines of this code with 3 separate `extern "C"` functions and a Fortran code that calls those 3 functions. We have written a simple Fortran with the FFTW to get used to the programming as it is in the original implementation. Next we have written a simple function to learn the interoperability of C and Fortran. Then we have written a 1D implementation of a FFT on the device. Finally we have split the different functions into different subroutines. We have split the functions to have an easy implementation of the FFT on the GPU. Now we only have to implement the 2D FFT to run the program CONTACT partially on the GPU.

5.2.4 Fortran & C 2D FFT on GPU

In this section we start with the 2D FFT. These are the functions that CONTACT uses and so the functions we have to implement on the GPU. We start with single precision while CONTACT

uses double precision. So we start again with a pure Fortran test using the fftw library. Then we write a C program that contains the 3 separate functions as in the previous subsection. These are the functions needed to calculate a 2D FFT of a grid. For this grid we use a test grid similar as in Chapter 4 but then 4×6. We now use the same function-call name as CONTACT and the FFTW library uses, so we can replace them later. Also because we use C-functions with Fortran the arrays are inversed. So the row dimension in Fortran is the column dimension in C.[4].

```
extern "C" void dfftw_plan_dft_r2c_2d_(cufftHandle *plan, int *n1,int *n2){
cufftPlan2d(plan, *n2,*n1,CUFFT_C2R)
return;
}
```

Note here that because row and column are changed that n2 and n1 also are changed. Then we have the following functions in a C file.

```
extern "C" void dfftw_plan_dft_r2c_2d_(cufftHandle *plan, int *n1,int *n2)
extern "C" void dfftw_plan_dft_c2r_2d_(cufftHandle *plan, int *n1,int *n2)
extern "C" void dfftw_execute_dft_r2c_(cufftHandle *plan, int *n1, int *n2, &
                                       cufftReal *idata, cufftComplex *odata)
extern "C" void dfftw_execute_dft_c2r_(cufftHandle *plan, int *n1, int *n2, &
                                       cufftReal *idata, cufftComplex *odata)
extern "C" void dfftw_destroy_plan_(cufftHandle *plan)

program main2
call dfftw_plan_dft_r2c_2d(intger plan,int n1,int n2)
call dfftw_execute_dft_r2c(intger plan,int n1,int n2,real in,complex out)
call dfftw_destroy_plan(plan)
```

We have given the function names and its input and note that there is no line with "include fftw3.f" in program main, because we have replaced these function calls with the same name for function that we have written in C. Also we have written the complex to a real function which we will use in CONTACT.

In order to change the variables of CONTACT to single precision we can run CONTACT by replacing

```
with fftw3.f
```

for nothing and tell CONTACT to run with the CUDA libraries and with the file `fftcuda.o` created from `fftcuda.cu` with

```
nvcc  -L$(cuda)/lib64 -lcufft -c fftcuda.cu
```

If we now call for example `dfftw_plan_dft_r2c_2d` we do not call the function written in the FFTW library but the function written in the `.cu` file. Now we run some tests. We use tests provided by CONTACT, see Chapter 6 for an explanation. In Table 6.2 we see that that for a large problem it is a bit faster but that it takes a lot of extra time to convert doubles to singles and back and also our answer is slightly different with the original implementation of CONTACT, that is using the FFTW library on the CPU.

So we replace every single for a double and run tests but for this we need a graphic device with capability 1.3 or higher. In Subsection 6.1.2 we will explain more about the capability of the device. Also we need to tell the device to run in double precision and we do this with -arch sm_13.[3] See Chapter 6 for the results.

5.2.5 Faster implementation of CONTACT

We now have implemented the FFT's on the GPU. But we have done it in the following way: Every time CONTACT uses an call to a FFTW subroutine we have replaced it with a C-function call. So with every call we have to convert the variables from Fortran to C and back and also we have to copy information from the CPU to the GPU and back that could be left on the device. A memory copy from host to device or vice versa costs time so we can improve this if we do this less and also we can do the multiplication per element on the device which can also give an improvement. The adapted code is given in the Appendix. We explain the required adaptations in this Section.

We will improve the calculation of the deformation u, so we have an influence grid B called c and have chosen a traction p and will calculate u with the Fourier transformations, see Section 2.3. So we need to calculate the Fourier transformation of c and p, we need to multiply these per element with each other and we take the inverse transformation of this to reclaim u. In CONTACT there are some optimizations, like if the size of c does not change then you do not need to recalculate the Fourier transformation of c and if the size of p does not change then we will not make a new plan.

In the Fortran-file the c and p are filled if needed and the the C-file is called. Now we will do the following in a C-file, see Table 5.1: First we will calculate the FFTs of c and p and we will leave the results on the device. Then we will calculate the multiplication per element on the device and then we will calculate the inverse transformation of the result to reclaim u'. But because u' is too large (2.3) the results need to be reclaimed out of this grid and this is still done in the Fortran-file. Next there are some things to take into account. First, when the CPU calls a GPU-function, the device calculates the function but the CPU runs further so while the GPU is still running it could send a new function to the device. So first we will give the multiplication per element kernel and then the other functions:

```
__global__ void convolutie(cufftDoubleComplex *A, cufftDoubleComplex *B, &
                           cufftDoubleComplex *C, &int mx, int my){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  double scale = 1./((mx)*(my));
  if(i<(mx/2+1)*(my)){
    C[i].x = ((A[i].x * B[i].x)-(A[i].y*B[i].y))*scale;
    C[i].y = ((A[i].x * B[i].y)+(A[i].y*B[i].x))*scale;
  }
}
```

See for the results of this improvement Chapter 6.

Action	**Functions**
We make space on the GPU for ps,pf, us and uf	cudaMalloc()
We send cf to the device	cudaMemcpy()
Or we calculate cf from cs	cudaMalloc() cufftPlan2d() cudaMemcpy() cudaExecD2Z() cudaMemcpy() cudaFree() cufftDestroy()
We send ps to device	cudaMemcpy()
We calculate pf	cudaExecD2Z
We calculate uf with the multiplication per element on the device	convolutie()
We calculate us	cudaExecZ2D
We send us to host	cudaMemcpy()
free the variables ps,pf,us and uf	cudaFree()

Table 5.1: CUDA process of the faster implementation

Chapter 6

Results

6.1 Build-up of results

6.1.1 Test problem

The test problems concern an investigation of the shape of the contact patch for a wheel with S1002 profile on a rail with UIC60 pro9th. The wheel set is displaced over 6.2mm such that the contact is just moving from the tread to the flange. No yaw is included in this case[5]. For the results we shall compare for every implementation, 3 of these tests provided by CONTACT. The first test is called norm_problem 1 and is a 71×81 grid, the second problem is a 143×163 grid and the third is a 287×323 but is called norm_problem 4 because it is about 4 times larger then the first one. We shall show the test problem by its grid size.

6.1.2 Computer used

It is important to explain what computer is used because all these results are dependent on the computer it has run.

CPU	Intel Core 2 Duo E8500 (@3.16 GHz)
Theoretical Peak	21.5 Gflops
Memory	4GB RAM
CUDA release	4.1
Screen dedicated GPU	NVIDIA Quadro NVS 290
CUDA dedicated GPU	NVIDIA Tesla C1060
Memory	4GB
Number of SMs	30
Number of cores	240
Compute capability	1.3
Theoretical Peak	933 Gflops

Table 6.1:

We would like to point out the compute capability because most calculations are done with the GPU. Every CUDA compatible GPU is tagged with a 2 digit number: the compute capability. The number expresses to what degree the GPU can perform different tasks and operations and indicates the availability and amount of different resources on the device, e.g. whether the device

can perform double precision arithmetic, the amount of registers available, the maximal number of threads allowed per block, the throughput of arithmetic instructions for different variable formats, etc. More precisely: the compute capability number consists of a major revision number and a minor revision number, notation: c.x. The major revision number (c = 1 or 2) indicates the core architecture, and the minor revision number indicates smaller improvements with respect to core architecture and available resources and newly added features.[6] It does not mean that when you have an implementation with an older device and you run it on a newer device it is always faster. But the new devices can be much faster. When you want to use double precision you need at least an device of 1.3.

6.1.3 Presentation results

We give the results in the following matter:
We give the time that CONTACT and `m_aijpj` needs to do the test calculation. We give all our results rounded to 2 decimals.

- In the first column is the size of the grid chosen.
- Then we give the total time the original implementation needs to calculate the size of the grid it needs to calculate. It calculates the size needed to solve the problem in optimal time. So the grid is enlarged till the grid has only prime factors of 2,3,5 and 7. We would like to have only these prime factors of our grid because the FFTW library but also the CUFFT library are optimized for numbers with only prime factors 2,3,5 and 7. Then CONTACT calculates u with the FFTW library on the CPU. This is done in the CG loop so it is done many times and this total time is given in the second column.
- In the third column is given the same time as the second column but then the calculation of u is done with the CUFFT library on the GPU. Every new table contains the results of a new implementation.
- In the fourth column is given the Speedup, so the time of the CPU divided by the time of the implementation on the GPU.
- In the fifth column the overhead is given. So the time that is extra in the FFT or in the time the program extra costs with the original implementation.
- Then we give the time for the whole program CONTACT extra costs.
- In the last column the Speedup factor of the whole program is given.

6.2 Results of implementations

6.2.1 Single Precision

Now we give in Table 6.2 the results of our first implementation. We give the double CPU implementation as reference time and in the first implementation we change every variable to single in `m_aijpj` and calculate on the device also in single.

Grid size Test problem	Old MVP FFT on CPU	New MVP FFT on GPU	Speedup	Overhead	Rest	Overall Speedup
71×81	0.20	0.22	0.91×	0	0.08	0.93×
143×163	1.40	0.71	1.97×	0.18	0.55	1.35×
287×323	10.94	3.97	2.76×	2.14	6.08	1.40×

Table 6.2: Test problem with the first single implementation.

We see that the smallest test-problem costs slightly more time and we see that the calculation of the FFTs for the largest problem is a lot faster, 11 seconds vs 4 seconds. But we also gain an extra 2 seconds for other processes in the largest test. These processes are most likely conversions from doubles to singles and back. This is because the single precision is bad implemented. We have changed the variables in our file but in the main program the variables are still stored in double precision. This creates that extra time. Also what we can't see in these numbers is that our answers are slightly different than those computations on the CPU. This is because the GPU has slightly different routines of truncation. The difference do not result in a different contact area, but in more elements with a very small deformation, but not large enough to include it into the contact area.

6.2.2 Double Precision

So next we will give in Table 6.3 the results when we calculate everything in double precision and try to get rid of the extra 2 seconds. We change all variables to doubles and also the functions. For double-variable calls we change for example `cufftComplex` for `cufftDoubleComplex`.

Grid size Test problem	Old MVP FFT on CPU	New MVP FFT on GPU	Speedup	Overhead	Rest	Overall Speedup
71×81	0.20	0.35	0.57×	-0.01	0.08	0.66×
143×163	1.40	1.20	1.17×	0.01	0.55	1.11×
287×323	10.94	5.63	1.94×	-0.19	6.08	1.48×

Table 6.3: Test problem with the Double implementation.

Now we see that all the tests costs extra time compared to Table 6.2 except overall in the largest test. This was to be expected because double precision takes more time than single precision. We see that for the largest problem we are still a lot faster and that it does not take extra time for the double to single conversion which results in better results. Note that we have for tests 1 and 2 a small overhead, these are most likely rounding errors. For the largest test we have a negative overhead, this is because due to the slightly different answers we have a different calculation path. In the fifth iteration an element is placed different than in the CPU calculation, this results in less iterations after the whole program is finished.

6.2.3 Improved implementation

Then we will show in Table 6.4 the next results of the improved C-function. We do this as explained in subsection 5.2.5. So we do less memory copies and we calculate the multiplication per element on the device, so we expect an improvement:

Grid size Test problem	Old MVP FFT on CPU	New MVP FFT on GPU	Speedup	Overhead	Rest	Overall Speedup
71×81	0.20	0.27	0.74×	-0.01	0.08	0.82×
143×163	1.40	1.00	1.40×	-0.02	0.55	1.27×
287×323	10.94	4.64	2.36×	-0.09	6.08	1.60×

Table 6.4: Test problem with the improved implementation.

We see that we are faster with every test as expected but not close to our expectations so next we are going to test why the results are so slow.

6.3 Expectations of results

Because our results are disappointing we next try to look what takes more time than expected and what speedup we can expect. The first idea about why our results are disappointing is that, because we are switching between Fortran and C, the conversion of variables from Fortran to C takes time. But we could not create a test sufficient to test this. But there is no need to test this because there a package called "c_binding". When allocating the data with this package you tell the computer that it is actually a C variable. Thus it is already stored as a C variable and it takes no time to convert. Note that this is not yet done in this project, so we must asume that it costs some extra time to convert these data.

6.3.1 FFT with a "plan"

In Table 6.6 and 6.7 we will give an answer about the question; what is faster, the FFTW function on the CPU or the cuFFT function on the GPU? In Tables 6.6 and 6.7 we see the results of the following test. We make a FFTWplan on the CPU, we execute the plan and we destroy the plan, these results are in the second column. We shall explain again what this plan is. The plan is a subroutine of the FFTW library and also of the CUFFT library and it allocates space for the calculation and it checks what the fastest way is to calculate this FFT. With the fastest way we mean that it can be divided by its prime factors or it has to use an other algorithm to calculate the FFT of this size.

In the third column you can find the results of a Cuda file that does the process of calculating a FFT as the second column but than on the GPU, this is also what we have done in the implementations of double. This process is shown in 6.5.

Action	**Functions**
We make a plan like on the CPU but now on the GPU	cufftPlan2d()
then we allocate space on the device	cudaMalloc()
copy the data from the host to the device	cudaMemcpy()
execute the plan	cufftExecD2Z
copy the data from the device to the host	cudaMemcpy()
destroy the plan and free the variables	cufftDestroy() cudaFree()

Table 6.5: CUDA process.

These processes are run a 100.000 times and the total time is divided by the same number. This has been done to get the time it takes to run it once, because the device startup time is divided out of the equation.

In the fourth column the speedup factor is given of the GPU time with the CPU time as reference time.

In the fifth and sixth column the same is given as in the third and fourth column except that everything is run in single precision. Note that the cufftExecD2Z changes to cufftExecR2C.

We present the results in the following way: In Table 6.6 we have given the powers of 2, because they should be perfect for calculating in parallel. In Table 6.7 we give the same results but then for some special numbers: 129 because it is close to 128 but has a prime factor of 43, 201 because it has a prime factor of 67 and 243 because it is a power of 3. These numbers should be slower for the process, but are they faster on the GPU or is the CPU faster?

	FFT on CPU double precision	FFT on GPU double precision	Speedup	FFT on GPU single precision	Speedup
32×32	$19\mu s$	$322\mu s$	$0.06\times$	$94\mu s$	$0.04\times$
64×64	$40\mu s$	$358\mu s$	$0.11\times$	$509\mu s$	$0.08\times$
128×128	$203\mu s$	$478\mu s$	$0.42\times$	$570\mu s$	$0.36\times$
256×256	$760\mu s$	$1399\mu s$	$0.54\times$	$812\mu s$	$0.94\times$
512×512	$4480\mu s$	$3300\mu s$	$1.35\times$	$2016\mu s$	$2.22\times$

Table 6.6: Calculate a FFT for powers of 2, with plan every time

We see that for small powers of two the process is much slower.

	FFT on CPU double precision	FFT on GPU double precision	Speedup	FFT on GPU single precision	Speedup
129×129	$1500\mu s$	$2121\mu s$	$0.71\times$	$715\mu s$	$2.10\times$
201×201	$3480\mu s$	$2618\mu s$	$1.33\times$	$1047\mu s$	$3.32\times$
243×243	$1171\mu s$	$1815\mu s$	$0.65\times$	$1175\mu s$	$1.00\times$

Table 6.7: Calculate a FFT- for special numbers, with plan every time

We see that the FFTW function in Fortran on the CPU can still be faster for problems in double precision. For single precision we can achieve a speedup for large numbers but not for small powers of 2. But the calculation of a FFT should be much faster on the GPU then on the CPU? In Section 6.3.2 we will answer this question.

6.3.2 Execute function

So the next test in Table 6.8 and 6.9 we time 100.000 times only the calculations of an FFT and the time divided by the same number. We do this to time a calculation of only one FFT but we can not measure only one FFT. The next test is to observe if there is an opportunity to speedup the process. In the second column are the results of the FFTW-function on the CPU in double precision and in the third column are the results of the calculations on the device with the CUFFT function in double precision. Again we give in the same for single precision in the last two columns and we use the same numbers as in Table 6.6 and 6.7.

	FFTexecute double precision on CPU	FFTexecute double precision on GPU	Speedup	FFTexecute single precision on GPU	Speedup
32×32	5μs	59μs	0.08×	40μs	0.13×
64×64	24μs	67μs	0.36×	45μs	0.36×
128×128	160μs	78μs	2.05×	49μs	3.26×
256×256	700μs	184μs	3.80×	74μs	9.46×
512×512	4380μs	629μs	6.96×	208μs	21.06×

Table 6.8: Calculate a FFT-execute for powers of 2

	FFTexecute double precision on CPU	FFTexecute double precision on GPU	Speedup	FFTexecute single precision on GPU	Speedup
129×129	1316μs	703μs	1.87×	135μs	9.74×
201×201	3228μs	974μs	3.31×	360μs	8.97×
243×243	1060μs	442μs	2.40×	247μs	4.29×

Table 6.9: Calculate a FFT-execute for special numbers

If we combine these two results we conclude that the calculation is indeed much faster on the GPU than on the CPU. This could be explained for big problems but not for problems like 64×64 or smaller. Also you have to transfer the data to the GPU so as a result we have not much improvement in total for large problems and a slowdown for small problems.

To discus our answer we check the flop rate. We take an estimate of the number of flops ($O(nm^2 \log nm)$) and calculate the flop rate. We do this only for powers of two because then we have a better estimate. In Table 6.10 we show that the flop rate is fine [6, Section 11.6, p. 73]. If we calculate the flop rate for the special numbers it would be less than 1 Gflops. This because the grid cannot be split nicely in half again and again.

	FFTexecute double precision on CPU	flops	floprate in Gflops/s on CPU
32×32	5μs	$1.05 * 10^4$	2.05
64×64	24μs	$4.92 * 10^4$	2.05
128×128	160μs	$2.29 * 10^5$	1.43
256×256	700μs	$1.05 * 10^6$	1.50
512×512	4380μs	$4.72 * 10^6$	1.08

Table 6.10: Flop rate for powers of 2

6.3.3 FFT Execute in a Figure

When we make a graph for the execute function of the different implementations for grid size to 128 we compare Figure 6.1 and 6.2

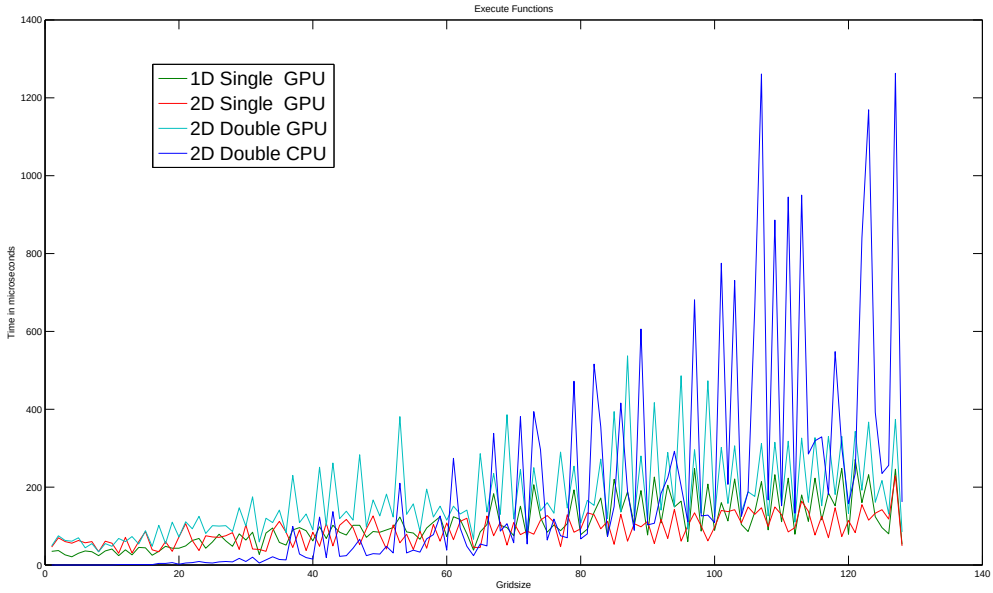

Figure 6.1: Execute functions

We want to point out two things. First that if a grid size has a large prime factor then it takes a lot of time to calculate its Fourier transformed. Second we see that the CPU implementation begins a lot faster and is slower for grid sizes larger than 80. In CONTACT we use an optimized algorithm. This algorithm calculates the first number with only prime divisors 2,3,5 and 7 larger then the grid you optimize. Figure 6.2 shows these results.

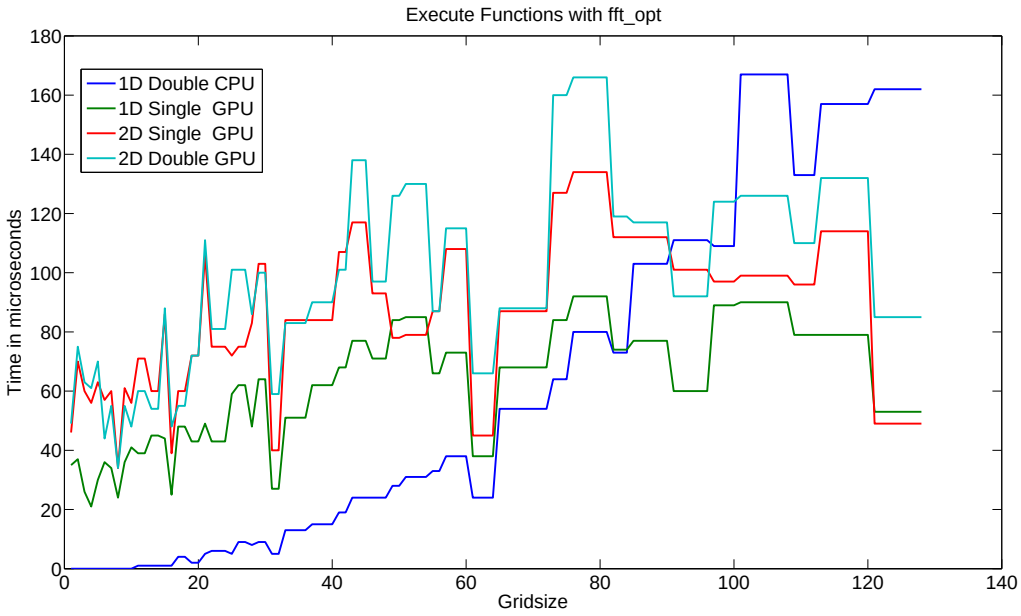

Figure 6.2: Execute functions

6.3.4 1D FFT

Because we are using Fourier transformations as a trick to calculate the convolution we could also use the one dimensional FFT. In Tables 6.11 and 6.12 we give the results of the 1D FFT in single precision with the 2D FFT in single precision as the reference time. We give the executes functions only in the same manner as in the previous subsections.

	FFTexecute 2D single precision on GPU	FFTexecute 1D single precision on GPU	Speedup
32×32	40μs	27μs	1.48×
64×64	45μs	38μs	1.18×
128×128	49μs	53μs	0.92×
256×256	74μs	78μs	0.95×
512×512	208μs	221μs	0.94×

Table 6.11: Calculate a 1D FFT-execute for powers of 2

	FFTexecute 2D single precision on GPU	FFTexecute 1D single precision on GPU	Speedup
129×129	135μs	282μs	0.48×
201×201	360μs	396μs	0.91×
243×243	247μs	167μs	1.48×

Table 6.12: Calculate a 1D FFT-execute for special numbers

We see in the results for the 1D FFT that for powers of 2 the 1D algorithm is almost the same. But for the special numbers there are large differences, for 243 which is a power of 3 this is faster but for numbers with large prime numbers the 1D algorithm does not recognize the fastest algorithm and it is much slower. So we can conclude that we do not achieve a speedup if we change to the 1 dimensional FFT.

6.3.5 Pinned Memory

In our problem we transfer a lot of data back and forth and this takes a lot of time. We can speed this up if we use Pinned Memory (or Page-locked memory). As opposed to ordinary pageable host memory the runtime environment also offers the possibility to allocate so-called page-locked. Page-locked means that the memory pages are locked, thus that physical addresses remain unchanged and that the memory will not being swapped out by the operating system. The main benefit is that page-locked memory is about twice as fast as ordinary host memory. However, page-locked memory is very scarce and should used sparingly since the host may slow down or even hang.[6]

In Table 6.13 and 6.14 we compare for the same numbers as in previous subsections the time it costs to do the plan but with the pinned memory.

	FFT on GPU single precision	FFT on GPU single precision Pinned	Speedup
32×32	494μs	481μs	1.03×
64×64	509μs	499μs	1.02×
128×128	570μs	551μs	1.03×
256×256	812μs	707μs	1.14×
512×512	2016μs	1543μs	1.31×

Table 6.13: Calculate a 1D FFT-execute for powers of 2, with make plan every time

	FFT on GPU single precision	FFT on GPU single precision Pinned	Speedup
129×129	715μs	686μs	1.04×
201×201	1047μs	971μs	1.08×
243×243	1175μs	1085μs	1.08×

Table 6.14: Calculate a 1D FFT-execute for special numbers, with make plan every time

In Table 6.13 and 6.14 we see that Pinned-memory is indeed faster than normal allocated memory. But because these processes are bigger than just a memory copy we do not see a great speedup factor. Also what is not in these data is that the allocation of Pinned Memory costs more time than a normal allocation. So if we have to transfer very small data packages than the speedup achieved with the faster copying is less then the extra time for the allocation. If we do this for small grids there is a slowdown instead of a speedup. But we conclude that with Pinned Memory we can speedup the process.

Chapter 7

Conclusions

7.1 Problems to small

The problems that are normally solved with CONTACT are not larger than 90000 grid points but usually much smaller like 6000 grid points. Because these problems are so small the CPU with the FFT implementation can solve the problem very fast. When using a GPU implementation we get an overhead like the startup time of the device. The overhead of the GPU implementation is hard to gain back with the better compute capabilities especially for small problems with 6000 grid points or less. We have seen that the actual calculation time with the overhead of the GPU is slower than the CPU for problems of 6400 grid points or less.

7.2 Pinned memory is faster

CONTACT can achieve a speedup when using pinned memory. But pinning memory costs time so it does not achieve a speedup when the data that is pinned only is used once. When using pinned memory you can also hang your computer. There is now even so called zero-copy memory. When we would use this there will be, like the name says no copy, you create an environment where the CPU and the GPU both can use the data directly. It is probably worth researching.

7.3 1D FFT is not faster than 2D FFT

Because CONTACT is using the Fourier domain as a trick to calculate a convolution we could choose to do the calculations not with a 2D FFT but with a 1D FFT. This did not result in a clear answer because for different values these implementations are faster. Because we do not know the implementation of the FFTW library we cannot conclude that the 2D FFT is faster but it seems like there is reason to expect an improvement with the 1D FFT.

7.4 CUDA better than Portland Fortran

Although we have done no research, we have learned that if you use Portland Fortran it is most likely easier to implement. But you loose a lot of control like where and how to store data and when to send the data. So CUDA is harder to build but better for your calculation time.

7.5 Bottleneck

Because our implementation is not always faster, the bottleneck is still calculating the FFTs in the CG method.

7.6 Recommendations

7.6.1 Research where double precision is needed

We have seen that using double precision takes a lot more time than using single precision. Also you cannot achieve double precision on every GPU. So when you want to achieve a speedup with the GPU you must research properly if you need the double precision and if you do where.

7.6.2 Implement the whole CG method

CONTACT uses the CG method and this method is not particularly a method that can easily be parallelized. Because the CG method chooses an answer, calculates the result and the chooses a better answer. We have tried to improve the calculation time of the calculation of the answer. We have done this on the GPU, but then we have to give the result back to the CPU, then the CPU chooses the new answer and send it to the GPU so that it can calculate the new answer. But these processes can not be done in parallel so the GPU is a lot of time idle. So the speedup achieved with the faster calculation time is lost with all this sending and this time the GPU or the CPU is waiting on the other. CONTACT can achieve a speedup when the whole CG method is implemented on the GPU. Than you get the speedup achieved with the faster calculation time but you do not loose it in the sending and waiting. I expect that you will not achieve a large speedup for small problems of 6000 grid points. But for large problems you will get a big improvement, although I can not give a number.

Appendix A

Execution stack

Filename	**subroutine**	**call**
contact.f		nonhz
nonhz.f	nonhz	Contac
scontc.f	Contac	PanPrc
	Panprc	snorm
snorm.f	snorm	solvpn
solvpn.f	solvpn	normcg
	normcg	VecAijPj
m_aijpj.f	gf3_VecAijPj	fft_VecAijPj
	fft_VecAijPj	

Appendix B

Code

B.1 fftfunction.cu

```
#include <cuda.h>
#include <cufft.h>
#include <stdbool.h>
#include <complex.h>
//
//calculation of convolution
//
__global__ void convolutie(cufftDoubleComplex *A, cufftDoubleComplex *B,
cufftDoubleComplex *C,  int mx, int my)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  double scale = 1./((mx)*(my));
  if(i<(mx/2+1)*(my)){
    C[i].x = ((A[i].x * B[i].x)-(A[i].y*B[i].y))*scale;
    C[i].y = ((A[i].x * B[i].y)+(A[i].y*B[i].x))*scale;
  }
  __syncthreads();
}

extern "C" void dfftw_plan_dft_r2c_2d_(cufftHandle *plan, int *mx,int *my)
{
  if(cufftPlan2d(plan, *my, *mx, CUFFT_D2Z ) != CUFFT_SUCCESS)
    printf("plan gefaald real to complex; waarschijnlijk GPU niet voldoende\n"
    );
  //printf("cufftplanr2c: %d\n", plan);
  return;
}

extern "C" void dfftw_plan_dft_c2r_2d_(cufftHandle *plan, int *mx,int *my)
{
  if(cufftPlan2d(plan, *my, *mx, CUFFT_Z2D) != CUFFT_SUCCESS)
    printf("plan gefaald\n");
  //printf("cufftplanc2r: %d\n", plan);
  return;
```

```
}

extern "C" void dfftw_destroy_plan_(cufftHandle *plan)
{
  //printf("cufftdestroy: %d\n", plan);
  cufftDestroy(*plan);
  return;
}

void foutmelding(cufftResult error,char* st){
    if(error == CUFFT_SETUP_FAILED)
      printf("calculating %s has failed; setup\n",st);
    if(error == CUFFT_INVALID_PLAN)
      printf("calculating %s has failed; plan\n",st);
    if(error == CUFFT_INVALID_VALUE)
      printf("calculating %s has failed; value\n",st);
    if(error == CUFFT_INTERNAL_ERROR)
      printf("calculating %s has failed; intern\n",st);
    if(error == CUFFT_EXEC_FAILED)
      printf("calculating %s has failed; execute\n",st);
    if(error == CUFFT_SETUP_FAILED)
      printf("calculating %s has failed; setup\n",st);
    if(error == CUFFT_UNALIGNED_DATA)
      printf("calculating %s has failed; unaligned\n",st);
}
extern "C" void fftfunction_(int *bol,cufftDoubleReal *cs,cufftDoubleComplex
*cf, cufftHandle *plan_ps,cufftHandle *plan_uf,cufftDoubleReal *ps,
cufftDoubleReal *us,int *mx,int *my,int *idebug)
{
  int N= (*mx)*(*my);
  int i;
  cufftResult error;
  cudaError_t error2;
  size_t size1 =  sizeof(cufftDoubleReal)*N;
  size_t size2 =  sizeof(cufftDoubleComplex)*(*mx/2+1)*(*my);
  cufftHandle plan_cs;
  cufftDoubleReal *d_ps, *d_us, *d_cs;

  cufftDoubleComplex *d_cf,*d_pf,*d_uf;

  cudaMalloc((void**) &d_cf, size2);
  cudaMalloc((void**) &d_pf, size2);
  cudaMalloc((void**) &d_uf, size2);
  //
  //calculating cf if needed
  //
  if(*bol==1){
    cudaMemcpy(d_cf, cf, size2, cudaMemcpyHostToDevice);
    error2 =cudaGetLastError();
```

```
    if(error2!=cudaSuccess)
      printf( "Cuda error memcpy cf 1: %s.\n", cudaGetErrorString(error2));
  }
  else{
    cudaMalloc((void**) &d_cs, size1);
    cudaMemcpy(d_cs, cs, size1, cudaMemcpyHostToDevice);
    error2 =cudaGetLastError();
    if(error2!=cudaSuccess)
      printf( "Cuda error memcpy cs: %s.\n", cudaGetErrorString(error2));

    if(cufftPlan2d(&plan_cs, *my, *mx, CUFFT_D2Z ) != CUFFT_SUCCESS)
      printf("plan gefaald; misschien GPU niet voldoende\n");

    error =cufftExecD2Z(plan_cs, (cufftDoubleReal *)d_cs,
     (cufftDoubleComplex *)d_cf);
    if(error!= CUFFT_SUCCESS)
      foutmelding(error,"cs");
    cudaMemcpy(cf, d_cf, size2, cudaMemcpyDeviceToHost);
    error2 =cudaGetLastError();
    if(error2!=cudaSuccess)
      printf( "Cuda error memcpy cf 2: %s.\n", cudaGetErrorString(error2));
    if(*idebug>5){
      for(i=0;i<(*mx/2+1)*(*my);i++){
printf("cf = %f\n",cf[i]);
      }
    }

    cufftDestroy(plan_cs);
    cudaFree(d_cs);
  }
  //
  //calculating pf
  //
  cudaMalloc((void**) &d_ps, size1);

  cudaMemcpy(d_ps, ps, size1, cudaMemcpyHostToDevice);
  error2 =cudaGetLastError();
  if(error2!=cudaSuccess)
    printf( "Cuda error memcpy ps: %s.\n", cudaGetErrorString(error2));
  error = cufftExecD2Z(*plan_ps, (cufftDoubleReal *)d_ps,
   (cufftDoubleComplex *)d_pf);
  if(error!= CUFFT_SUCCESS)
    foutmelding(error,"pf");
  cudaThreadSynchronize();
  cudaFree(d_ps);
  //
  //calculating uf
  //
  int threadsPerBlock = 32;
```

```
  int blocksPerGrid = ((*mx/2+1)*(*my)+threadsPerBlock -1) / threadsPerBlock;

  convolutie<<<blocksPerGrid, threadsPerBlock>>>(d_cf,d_pf,d_uf,*mx,*my);
  cudaFree(d_cf);
  cudaFree(d_pf);
  //
  //calculating us
  //
  cudaMalloc((void**) &d_us, size1);

  error=cufftExecZ2D(*plan_uf, (cufftDoubleComplex *)d_uf,
   (cufftDoubleReal *)d_us);
  if(error!= CUFFT_SUCCESS)
    foutmelding(error,"pf");

  cudaMemcpy(us, d_us, size1, cudaMemcpyDeviceToHost);
  error2 =cudaGetLastError();
  if(error2!=cudaSuccess)
    printf( "Cuda error memcpy us: %s.\n", cudaGetErrorString(error2));

  cudaFree(d_us);
  cudaFree(d_uf);
return;
}
```

B.2 m_aijpj.f90

```
#ifdef WITH_FFTW

subroutine fft_VecAijPj (Igs, ladd, iigs, U, ik, P, jk, C)
!
! Purpose: Compute the displacements U (which is a gf3) in elements "iigs"
!          of Igs in direction(s) "ik" due to the tractions P (which is
!          a gf3 and has its own Igs) in direction(s) "jk", with influence
!          coefficients C, using the Fast Fourier Transform.
!
!            ladd == flag that indicates whether the displacements must
!                    overwrite (.false.) or be added to (.true.) the
!                    initial contents of U.
!
!            iigs == indication of the set of elements where U is required:
!                     AllElm, AllExt, AllInt, Exter, Adhes, Slip.
!
!            ik and jk must be a single coordinate direction 1, 2 or 3.
!
   use m_timers_contact
   use m_hierarch_data
   implicit none
```

```
!
! subroutine parameters:
!
   type(t_eldiv)               :: igs
   type(t_inflcf),  target     :: c
   type(t_gridfnc3)            :: u, p
   logical,         intent(in) :: ladd
   integer,         intent(in) :: iigs, ik, jk
!
! local variables:

   integer, parameter     :: idebug = 0
   logical, parameter     :: time_fft = .true.
   integer                :: fft_mx, fft_my, ix0, ix1, iy0, iy1, iarea
   integer   :: bol
   integer                :: ix, iy, ii, igs0, igs1
   integer                :: len_cf, iof1, iof2, arrsiz(fft_ndim)
   type(t_grid),  pointer :: grid
   integer           :: fftw_plan_cs
!dir$attributes align : 16 :: cs, cf
   real(kind=8),    dimension(:), allocatable :: cs
   complex(kind=8), dimension(:), allocatable :: cf

   if (time_fft) call timer_start(itimer_ffttot)
   if (idebug.ge.1) then
      if (iigs.eq.AllElm) write(*,*) 'fft_vecaijpj: starting for AllElm ...'
      if (iigs.eq.AllInt) write(*,*) 'fft_vecaijpj: starting for AllInt ...'
   endif
!
!  Check whether the use of FFT's is allowed in this version (license)
!
   if (.not.allow_fft) then
      write(lout,11)
      write(   *,11)
 11   format(/,' ERROR: The use of FFTs is disabled in this version. ', &
               'Please contact us via',/,'        www.kalkersoftware.org ', &
               'for an extended license file.',/)
      stop
   endif
!
!  Note: ik and jk must be 1, 2 or 3 (single matrix block).
!        iigs must be AllElm or AllInt.
!
   if (ik.le.0 .or. ik.gt.3 .or. jk.le.0 .or. jk.gt.3) then
      write(*,*) 'ERROR: FFT allowed for single coord.direction only: &
      ik,jk=', ik,jk
      stop
   endif
   if (iigs.ne.AllElm .and. iigs.ne.AllInt) then
```

```
      write(*,*) 'ERROR: FFT allowed only for AllElm or AllInt (',iigs,')'
      stop
   endif

   grid  => p%grid
!
!  Get the selection of the input grid to be used in the Fourier transform
!  Note: include one column to the left of the actual contact area (ixmin-1)
!
   if (iigs.eq.AllInt) then
      ix0    = max(1, min(igs%ixmin-1, p%eldiv%ixmin-1))
      ix1    = max(igs%ixmax, p%eldiv%ixmax)
      iy0    = min(igs%iymin, p%eldiv%iymin)
      iy1    = max(igs%iymax, p%eldiv%iymax)
      iarea = (ix1-ix0+1)*(iy1-iy0+1)
   else
      iarea = 0
   endif
!
!  Use full potential contact area when requested (AllElm) or if the
!     encompassing rectangle is not much smaller (factor 1/2.5).
!     Note: Norm+CG is faster with encompassing rectangle, NormGPCG
!           with always using fullbox
!
   if (iigs.ne.AllInt .or. 1.1*iarea.gt.grid%mx*grid%my .or. .false.) then
      ix0    = 1
      ix1    = grid%mx
      iy0    = 1
      iy1    = grid%my
   endif

   if (ix1.lt.ix0 .or. iy1.lt.iy0) then
      write(*,*) 'ERROR: empty grid [',ix0,':',ix1,'] x [',iy0,':',iy1,']'
      stop
   endif
!
!  Compute favourable data sizes for use in the FFT
!
   fft_mx = ix1 - ix0 + 1
   fft_my = iy1 - iy0 + 1
   if (.true.) then
      fft_mx = opt_fft_size(fft_mx)
      fft_my = opt_fft_size(fft_my)
   else
      fft_mx=2**ceiling( log(1d0*fft_mx) / log(2d0) )
      fft_my=2**ceiling( log(1d0*fft_my) / log(2d0) )
   endif

!  do iof1 = 100, 128
```

```
!     iof2 = opt_fft_size( iof1 )
!  enddo
!
   arrsiz       = (/ 2*fft_mx, 2*fft_my /)
   if (idebug.ge.2) write(*,'(4(a,i5))') ' fft_vecaijpj: grid size=', &
    ix1-ix0+1,' x', &
        iy1-iy0+1,', array size=',2*fft_mx,' x',2*fft_my
!
!  Check validity of Fourier transform data of the influence coefficients
!    - make sure that the fft_cf array is present and of the correct length
!    - invalidate all sections of fft_cf when the grid sizes have changed
!
   len_cf = (fft_mx+1) * 2*fft_my
   if (.not.allocated(c%fft_cf)) then
      allocate(c%fft_cf(len_cf,3,3))
      c%fft_ok = .false.
   endif
   if (size(c%fft_cf,1).ne.len_cf) then
      deallocate(c%fft_cf)
      allocate(c%fft_cf(len_cf,3,3))
      c%fft_ok = .false.
   endif
   if (c%fft_mx.ne.fft_mx .or. c%fft_my.ne.fft_my) then
      c%fft_ok = .false.
      c%fft_mx = fft_mx
      c%fft_my = fft_my
   endif
!
!  Compute the Fourier transform of the influence coefficients for coordinate
!  directions (ik,jk) when not yet available.
!
   allocate(cf((fft_mx+1) * 2*fft_my))

   if (.not.c%fft_ok(ik,jk)) then
!
!     Copy input data C to work array cs.
!     Note that c%cf is (-mx+1:mx,-my+1:my), and can both be smaller and
!     larger than cs.
!
      allocate(cs( 2*fft_mx  * 2*fft_my))


      cs = 0d0
      do iy = -min(fft_my,grid%my)+1, min(fft_my,grid%my)
         iof1 = (iy+fft_my-1)*2*fft_mx + fft_mx
         do ix = -min(fft_mx,grid%mx)+1, min(fft_mx,grid%mx)
            cs(iof1+ix) = c%cf(ix,iy,ik,jk)
         enddo
      enddo
```

```
      if (idebug.ge.2) write(*,*) 'fft_vecaijpj: done copying c...'
      if (idebug.ge.3) write(*,'(4(6f8.2,/))') cs
!
!     Perform the transformation of space domain cs to Fourier domain fft_cf
!
   endif
   if (c%fft_ok(ik,jk)) then
      cf=c%fft_cf(:,ik,jk)
   endif
!
!  Create work-arrays and FFTW descriptors when needed
!
   if (.not.plan_present .or. any(arrsiz.ne.plan_arrsiz)) then
!
!     Destroy work-arrays and descriptors that were created before
!
      if (plan_present) then
         deallocate(ps, us, pf, uf)
         call dfftw_destroy_plan(fftw_plan_ps)
         call dfftw_destroy_plan(fftw_plan_uf)
      endif
!
!     Allocate 1D work-arrays for spatial and Fourier transformed P and U
!
      allocate(ps( 2*fft_mx  * 2*fft_my))
      allocate(us( 2*fft_mx  * 2*fft_my))
      allocate(pf((fft_mx+1) * 2*fft_my))
      allocate(uf((fft_mx+1) * 2*fft_my))
      if (idebug.ge.10) write(*,*) 'fft_vecaijpj: allocate ok...'
!
!     Create descriptors for ps->pf and uf->us for current grid sizes
!        and arrays
!
      call dfftw_plan_dft_r2c_2d(fftw_plan_ps, arrsiz(1), arrsiz(2), ps, pf)
      call dfftw_plan_dft_c2r_2d(fftw_plan_uf, arrsiz(1), arrsiz(2), uf, us)
      plan_arrsiz  = arrsiz
      plan_present = .true.
   endif
!
!  Copy input data P to work array, padded with zeros
!
   ps = 0d0
   do iy = iy0, iy1
      iof1 = (iy-iy0)*2*fft_mx - ix0 + 1
      iof2 = (iy-1)*grid%mx
      do ix = ix0, ix1
         ps(iof1+ix) = p%val(iof2+ix,jk)
      enddo
   enddo
```

```
   if (idebug.ge.2) write(*,*) 'fft_vecaijpj: done copying p...'
   if (idebug.ge.3) write(*,'(4(6f8.2,/))') ps

  if(c%fft_ok(ik,jk)) bol=1
  if(.not.c%fft_ok(ik,jk)) bol=0
  call fftfunction(bol,cs,cf,fftw_plan_ps,fftw_plan_uf,ps,us,arrsiz(1),arrsiz(2)
   ,idebug)

!
!     Destroy work-array, mark availability of column (ik,jk) in fft_cf.
!
  if (.not.c%fft_ok(ik,jk)) then
    c%fft_cf(:,ik,jk) = cf
    deallocate(cs)
    c%fft_ok(ik,jk) = .true.
    if (idebug.ge.4) write(*,*) 'fft_vecaijpj: FFT(c)=...'
    if (idebug.ge.7) then
      do iy=1,2*fft_my
        write(*,'(4f8.2,5x,4f8.2)')                                          &
          (real(cf((iy-1)*(fft_mx+1)+ix)), ix=1,fft_mx+1),                   &
          (imag(cf((iy-1)*(fft_mx+1)+ix)), ix=1,fft_mx+1)
      enddo
    endif
  endif

  deallocate(cf)


   if (idebug.ge.2) write(*,*) 'fft_vecaijpj: IFFT(u)=...'
   if (idebug.ge.3) write(*,'(4(6f8.2,/))') us
!
!  Determine range for element division codes igs
!
   call IgsRange(iigs, igs0, igs1)
!
   if (.not.ladd) then
!
!     Copy the relevant output data from the work array to array U
!     uf(f_mx-1+[1:f_mx], f_my-1+[1:f_my]), no flipping required (?!)
!
      do iy = iy0, iy1
         iof1 = (fft_my+iy-iy0-1)*2*fft_mx + fft_mx-ix0
         iof2 =    (iy-1)*  grid%mx
         if (idebug.ge.8) write(*,*) 'iy=',iy,': iof2=',iof2,', iof1=',iof1
         do ix = ix0, ix1
            ii = iof2 + ix
            if (idebug.ge.8) write(*,*) '   ix=',ix,': adr2=',ii,&
               ', adr1=', iof1+ix
            if (igs%el(ii).ge.igs0 .and. igs%el(ii).le.igs1) then
```

```
               u%val(ii,ik) = us(iof1+ix)
            endif
         enddo
      enddo
   else
!
!     Add the relevant output data from the work array to array U
!
      do iy = iy0, iy1
         iof1 = (fft_my+iy-iy0-1)*2*fft_mx + fft_mx-ix0
         iof2 =    (iy-1)*  grid%mx
         do ix = ix0, ix1
            ii = iof2 + ix
            if (igs%el(ii).ge.igs0 .and. igs%el(ii).le.igs1) then
               u%val(ii,ik) = u%val(ii,ik) + us(iof1+ix)
            endif
         enddo
      enddo
   endif
   if (idebug.ge.5) write(*,*) 'fft_vecaijpj: done copying u...'

   if (time_fft) call timer_stop(itimer_ffttot)
   if (idebug.ge.1) write(*,*) 'fft_vecaijpj: done, returning...'
end subroutine fft_VecAijPj

! end of version using FFTW
#endif
```

Bibliography

[1] Pieter Loof, Speeding-up the CONTACT package by means of the Fourier Transforms, TU Delft, Netherlands, June 2011, Bachelor Thesis, http://repository.tudelft.nl/assets/uuid:05883ddc-1319-41be-97a6-73ee114d6425/TUD_bachelorverslag.pdf

[2] Michiel de Reus, Speeding-up the CONTACT-package by means of the Graphics Processing Unit, TU Delft, Netherlands, August 2010, Bachelor Thesis, http://repository.tudelft.nl/assets/uuid:cd76bcaf-ced8-4ecb-88e8-1f938d2afe47/verslag.pdf

[3] http://www.sdsc.edu/us/training/assets/docs/NVIDIA-03-Toolkit.pdf

[4] www.computationalmathematics.org/topics/files/calling_cuda_from_fortran.html

[5] Edwin A.H. Vollebregt,100-FOLD SPEED-UP OF THE NORMAL CONTACT PROBLEM AND OTHER RECENT DEVELOPMENTS IN CONTACT, 9th International Conference on Contact Mechanics and Wear of Rail/Wheel Systems (CM2012), Chengdu, China, August 27-30, 2012

[6] MARTIJN DE JONG ,Speeding-up the CONTACT software via the GPU NormalCG: matrix-vector products and ConvexGS: inner products, TU Delft, Netherlands, February 2011, Internship