

Department of Construction Sciences

Solid Mechanics

ISRN LUTFD2/TFHF-10/5160-SE(1-34)

GPU parallelization of crystal plasticity FE analyses

Master Thesis by

Per Johansson

SUPERVISOR

HÅKAN HALLBERG AND YLVA MELLBIN, DIV. OF SOLID MECHANICS

EXAMINER

PROF. MATTI RISTINMAA, DIV. OF SOLID MECHANICS

Copyright © 2011 by Div. of Solid Mechanics and Per Johansson

Printed by Media-Tryck, Lund, Sweden

For information, address:

Division of Solid Mechanics, Lund University, Box 118, SE-221 00 Lund, Sweden.

Homepage: <http://www.solid.lth.se>

Preface

This master thesis has been carried out at Lund Institute of Technology at the division of Solid Mechanics as the last part of my Master in Mechanical Engineering.

I would like to address thanks to my two supervisors Håkan Hallberg and Ylva Mellbin for their support during this work.

Malmö, March 2011

Per Johansson

Abstract

Polycrystalline materials are defined as materials that contain several grains oriented in different directions. The material model used for polycrystalline materials can for example be crystal plasticity where each grain, on a microscopic level is observed in the aspects of stresses and stiffness.

What many do not realize is that a computer's graphics card also include a processing hardware, GPU (Graphics Processing Unit) that have previously only been used in calculations related to computer graphics. This type of hardware consist of a multi-core processing unit to do massively parallel calculations. This technology can have a big impact on the computationally demanding analysis as crystal plasticity.

In this thesis a FE analysis, where the material behavior is described by a crystal plasticity model, has been changed to be executed on the GPU, this resulted in a speed up dependant of the number of grains considered. The result have only speed up impact when the number of grains are over 400, around 1200 the total speed up are around 2 times the original CPU program. Although the results show a positive increase in speed, this is not a programming structure that is suitable for GPU calculation, due to many constraints of syntax and a memory management that is not suitable for the GPU.

Contents

1	Introduction	1
1.1	Objectives	1
1.2	Disposition	1
2	Background - CUDA and GPU programming	2
2.1	CUDA	2
2.1.1	CUDA Overview	3
2.1.2	CUDA Memory Overview	4
2.1.3	Specification TESLA C2050	5
2.2	PGI CUDA Fortran	6
2.2.1	SUBROUTINE and FUNCTIONS	7
2.2.2	Variable attributes	8
2.2.3	Compute compatibility	9
2.2.4	Examples	10
2.3	CULA	14
3	Background - Crystal plasticity	15
3.1	Polycrystalline materials	15
3.1.1	Crystal plasticity model	16
3.1.2	Numeric implementation	19
4	Implementation	21
4.1	CUDA implementation of crystal plasticity	21
5	Results and Conclusion	23
5.1	Speed increase	23
5.2	Conclusion	25
5.2.1	Recommendation	25

Chapter 1

Introduction

In the material model crystal plasticity for polycrystalline materials each grain, on a microscopic level is observed in the aspects of stresses and stiffness. In recent years the development of graphics cards has made it possible to use a new programming technology that uses the graphics card's multi-core processing unit to do massively parallel calculations. This technology can have a big impact on the computationally demanding analysis of polycrystalline materials.

GPU programming is evaluated by implementing crystal plasticity FE analysis on a Cook's membrane where grain stiffness and tension are calculated on the graphic card. The influence the number of grain has on the calculation speed is also examined by changing the number of grains in the analysis.

1.1 Objectives

The purpose of this thesis is to study the use of parallel programming when solving FE equations using polycrystalline materials. The programming architecture evaluated is CUDA in the language FORTRAN through Portland Groups compiler CUDA FORTRAN.

1.2 Disposition

The report begins with a short background on CUDA and also on modeling of polycrystalline materials. The next chapter starts with the numerical implementation followed by the implementation and the changes due to the implementation of CUDA architecture. The last chapter consists of results and conclusions.

Chapter 2

Background - CUDA and GPU programming

2.1 CUDA

Regular programs typically uses the computer's CPU (Central Processing Unit) to solving calculation problems. What many do not realize is that a computer's graphics card also include a processing hardware, GPU (Graphics Processing Unit) that have previously only been used in calculations related to computer graphics. In recent years, a massive performance development has been taking place on the graphics cards due to the demands of the gaming industry.

In 2007 the graphics card developer NVIDIA introduced a new technology called CUDA (Compute Unified Device Architecture) that is an architecture that have made it possible to handle calculations on both the CPU and the GPU using the programming language C. This have made it possible to use the big advantage of parallel calculation on the multi core GPU.

After NVIDIA's launch several software companies processed the technology, such as the Portland Group, which has developed a FORTRAN compiler see section 2.2. Also EM Photonics, which has developed a library corresponding to LAPACK performing linear algebra calculations in parallel on the GPU for more info see section 2.3.

2.1.1 CUDA Overview

Some of NVIDIA's CUDA architecture terminology is listed below and will be used throughout the thesis.

Host Application or data handled by the CPU.

Device Application or data handled by the GPU.

Kernel Application written to be executed on the GPU.

Thread An execution of a kernel with a given index.

Block A group of threads.

Grid A group of blocks. The grid is handled and executed on a single device (GPU chip)

MP The GPU chips are organized in a collection of multiprocessors (MP) where one MP is responsible of handling one or more blocks in a grid. When the number of blocks are bigger than number of MPs a schedule will determine which block will be executed.

SP Each MP is divided into a number of stream processors (SP). Where each SP is handling one or more threads in a block.

The total amount of CUDA cores are defined by $MP \times SP$.

In this context an application can be a main-, sub- program or function.

2.1.2 CUDA Memory Overview

CUDA memory spaces differs from the usual host memory, first because it is a hardware card that comes with its own synchronous dynamic random access memory (SDRAM), corresponding to an ordinary RAM memory which every computer has. In order to execute a program on the GPU, this requires that memory transfers from the host memory to the memory allocated on the device. Then, to examine the results it is necessary to transfer it back to the host from the device.

- Device code can:
 - R/W per-thread registers
 - R/W per-thread local memory
 - R/W per-block shared memory
 - R/W per-grid global memory
 - Read only per-grid constant memory
- Host code can
 - R/W per grid global and constant memories

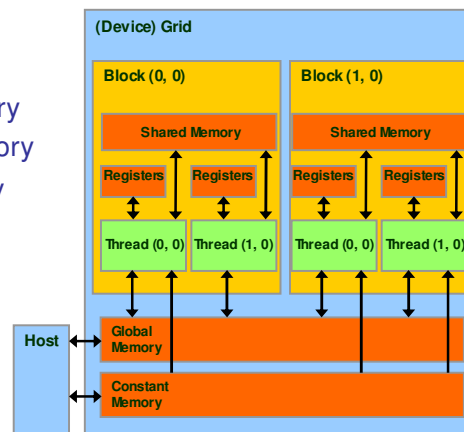


Figure 2.1: CUDA memory overview [2]

Global Memory

As seen in figure 2.1 the global memory is accessible during read or write from the host and also from the unique threads in the device. This is a big memory stored off chip in a bank of SDRAM chips, which makes it very slow to access. The reason for this is that the location on SDRAM tends to have long access latencies and also a finite access bandwidth. This can result in a poor execution time with too many threads with global memory requests. This memory type can also be called device memory.

Shared Memory

A small memory located on the chip, divided between the blocks. This memory is only accessible for threads within the block that are executed by the current MP. Due to the fact that it is stored on the chip contribute to high access performance.

Constant Memory

A read-only memory type for the device but read or write accessible for the host. Constant memory is also located on the chip which makes the access performance very high.

Registers

A scalar memory type that is assigned to each thread. It is handled by, and located on, the MP which makes the access performance very good. When registers runs out each thread gets assigned local memory which is located in the global memory.

Local Memory

A read or write memory for each thread located on the global memory which makes it also very slow to access.

2.1.3 Specification TESLA C2050

The card used in this thesis is NVIDIA TESLA C2050, the specification is shown below.

CUDA Capability version:	2.0
Number of cores:	14 (MP) x 32 (SP) = 448
Clock rate:	1.15 GHz
Total amount of global memory:	2817982464 bytes
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	32768
Maximum size of each dimension of a block:	1024 x 1024 x 64
Maximum size of each dimension of a grid:	65535 x 65535 x 1

2.2 PGI CUDA Fortran

The Portland Group Inc have developed a compiler called PGI CUDA Fortran which has a built-in support for using CUDA architecture in addition to standard FORTRAN.

CUDA C and also CUDA Fortran are lower-level explicit programming models together with a library of runtime components that gives the programmer a direct control of most aspects of GPU programming.

When NVIDIA launched CUDA the main area was computational parallel programming, this through various libraries and a C compiler, CUDA C. CUDA fortran as well as CUDA C is a lower-level explicit programming model with many runtime library components.

CUDA Fortran makes it possible to use the following operations in a fortran program

- Declaring variables that are allocated in the GPU device memory
- Allocating dynamic memory in the GPU device memory
- Copying data from the host memory to the GPU memory , and back
- Writing subroutines and functions to execute on the GPU
- Invoking GPU subroutines from the host

CUDA Fortran is built by using the PGI Fortran compiler with the filename extension *.cuf* which is a free-format CUDA Fortran. Although the flag *-Mcuda* is required by the PGI Fortran compiler.

2.2.1 SUBROUTINE and FUNCTIONS

To designate where a subroutine or function will be executed is defined through additional attributes as:

Attributes(host)

This type of declaration is the same as in standard FORTRAN, and is also the default when no attribute is declared and will just mean that the subroutine is executed on the host.

Attributes(global)

A subroutine with the attribute *global* is a kernel subroutine or also called device kernel, because it is executed on the device. Invoking of a device kernel is done by the use of special chevron, which determines the number of thread blocks that will be used and the number of threads containing in each thread block.

Parallelization is done according to the number of threads and blocks that are specified when invoking the device kernel. The device kernel will then be executed in parallel with the only difference that the internal variables *threadidx* and *blockidx* will vary between 1 and the number of threads and blocks that you define at the call of the device kernel. By calling a device kernel with 1, 1 within chevrons makes it a non parallel execution. This can be useful if you want to call a subroutine that will be execution on the GPU as serial code.

Example:

```
attributes(global) subroutine kernel(...)  
  ...
```

To invoke a device kernel:

```
call kernel<<<dimGrid,dimBlock>>>(...)
```

Attributes(device)

A subroutine or function with the attribute *device* will be compiled for execution on the device. This type of subroutine or function can only be invoked by another subroutine with the attribute *device* or *global* and must also appear in the same MODULE as the subroutine that invoked it.

Example:

```
attributes(device) subroutine deviceSub(...)  
  ...  
attributes(device) function deviceFunc(...)  
  ...
```

Restrictions on the device subprogram

A subroutine or function with the attribute *global* or *device* is a device subprogram and has to satisfy some restrictions. It can't call a host subroutine or function, neither can it be recursive. There are also restriction in standard Fortran intrinsic functions, for example MATMUL is not allowed. Input and output statements as for example READ,WRITE and PRINT are not allowed either. Objects with the attribute *pointer* or *allocatable* are not allowed in subprograms and neither are automatic arrays without fixed size. Device subprogram can't handle a temporary array definition for example

```
A = (/1,1,1/)
```

which is the standard way to assign an array, instead it have to be done element wise.

2.2.2 Variable attributes

Device data

The attribute *device* define the array to be in the global memory space. The declaration of device data is made according to:

```
double precision, device :: a(10)
```

When calling a device subprograms with dummy arguments, to get the corresponding size a integer with the attribute *value* have to be declared. Integer with the attribute *value* don't have to be located on the device memory which can be usefull in other cases as well.

```
attributes(global) subroutine dev_kernel(a,b,n)
  double precision, dimension(n,n) :: a,b
  integer, value :: n
```

here are *a* and *b* dummy arrays and *n* dummy argument.

Shared data

The attribute *shared* defines the array to be located at the chip (in the shared memory) and must follow the same criterion as the device data type.

```
double precision, shared :: a(10)
```

Constant data

The attribute *constant* defines the array to be of the type constant data and can be define according to

```
double precision, constant :: a(10)
```

As defined in section 2.1.2, constant data can only be written by the host. Constant data arrays have to be fixed size and can not be allocatable.

Data transfer

Data transfers by using standard F90 assignment are shown below, here the variable a and b are host arrays while a_{dev} and b_{dev} are device arrays located on the global memory slot.

```
a = adev
adev = a
b = a + adev
```

The assignment shown below are not legal because of where the actual computation will be performed is not explicit defined

```
a = adev + bdev
adev = adev + a
```

As shown in figure 2.1 on page 4 data transfer between shared memory and host is not allowed.

2.2.3 Compute compatibility

To let the compiler know what kind of hardware that is used for the calculation PGI have a flag that determine the kind of compute compatibility of the hardware. The compute compatibility is defined by NVIDIA and will just determine which kind of CUDA version that is used on the graphic card. The version determine the amount of memory in the different slots, how many threads and block that is available and even what kind of functions that are allowed. One import function that is only available in version 1.3 or higher is double-precision operation.

According to section 2.1.3, TESLA C2050 is of compute compatibility 2.0, then the flag looks like `-ta=nvidia,cc20`

2.2.4 Examples

To describe the benefits of using GPU parallelization and the differences compared to serial CPU programs follows here an example of a matrix multiplication that is well suited to parallelize using CUDA.

This program parallelizes using sub matrix multiplication and to avoid global memory request, the sub matrices are copied to the shared memory. As described in section 2.2.3 the hardware that is used is of compute compatibility 2.0 that makes it possible to use maximum sub matrix sizes of 32x32 otherwise the shared memory block will be too small.

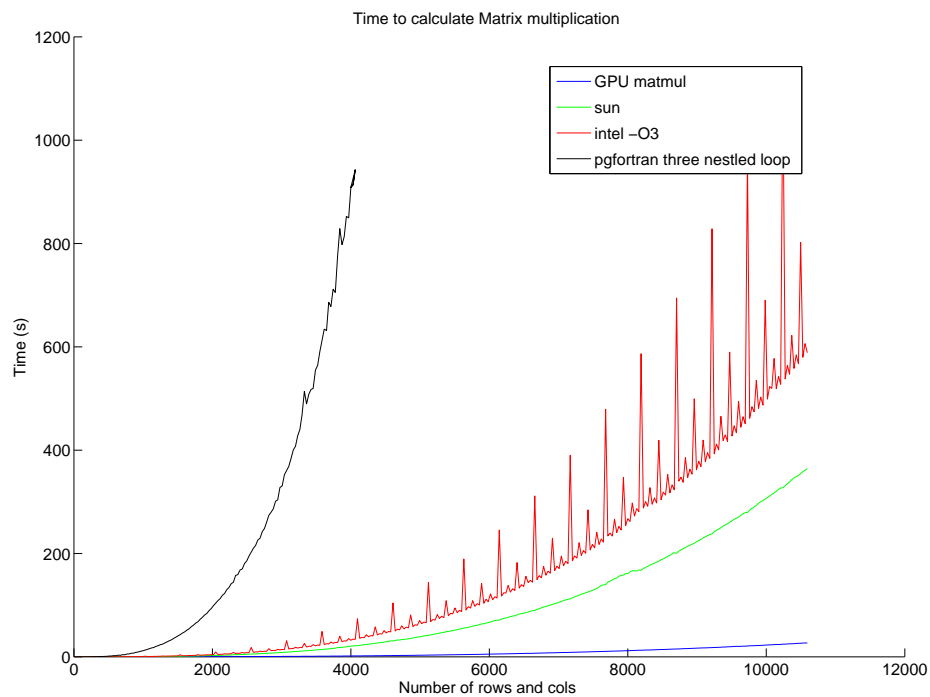


Figure 2.2: Example comparing MATMUL

A possible reason for the unusual appearance of intel-O3 can be that the curve is created by ALLOCATE and DEALLOCATE a certain matrix size that can be divided by 32 in a do loop, because this is the best way to use the GPU code. This may have contributed to the irregular appearance of the graphs in figure 2.2.

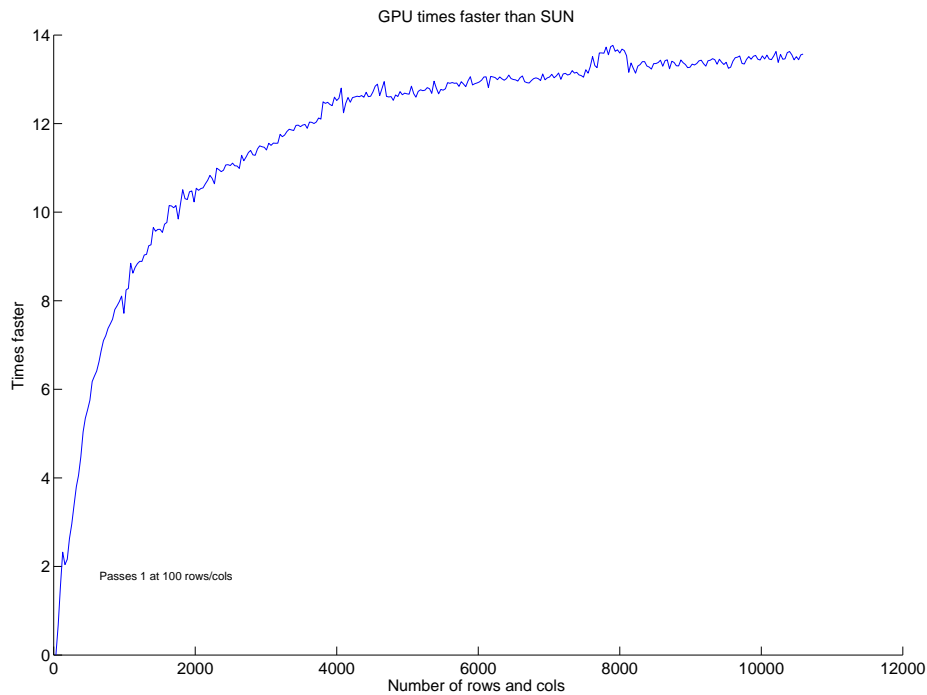


Figure 2.3: Example comparing MATMUL

From figure 2.2 it is clearly seen the advantage of using the GPU. Often you can find test results that show the result where the GPU code is compared to a not optimized CPU application as the test with PGFORTRAN in order to make the GPU version look better. Figure 2.2 shows also how big the difference is between compilers and the different kind of optimizations. The non-optimized PGFORTRAN took 7 hours to calculate a matrix multiplication of 10592 rows and cols compared to the fastest GPU which took 26 seconds. As figure 2.3 shows the benefits start already when the size is 100 x 100 and have dramatically big impact on speed over that.

The code used for GPU *matmul* can be seen on next page and illustrates many different kinds of syntax differences to ordinary FORTRAN.

```

module mmul_mod
  use cudafor
contains

  !!! mmul_kernel computes A*B into C where A is NxM, B is MxL, C is then NxL

  attributes(global) subroutine mmul_kernel( A, B, C, N, M, L )
    double precision, device :: A(N,M), B(M,L), C(N,L)
    integer, value :: N, M, L
    integer :: i, j, kb, k, tx, ty

  !!! submatrices are declared to be in CUDA shared memory
    double precision, shared :: Asub(32,32), Bsub(32,32)

  !!! the value of C(i,j) being computed, a temporary scalar
    double precision :: Cij

  !!! Start execution, first get my thread indices
    tx = threadidx%x
    ty = threadidx%y

  !!! This thread computes C(i,j) = sum(A(i,:) * B(:,j))
    i = (blockidx%x-1) * 32 + tx
    j = (blockidx%y-1) * 32 + ty

    Cij = 0D0

  !!! Do the k loop in chunks of 32, the block size
    do kb = 1, M, 32

  !!! Fill the submatrices; each of 32x32 threads in the thread block loads
  !!! one element of Asub and Bsub
    Asub(tx,ty) = A(i,kb+ty-1)
    Bsub(tx,ty) = B(kb+tx-1,j)

  !!! Wait until all elements are filled
    call syncthreads()

  !!! Multiply the two submatrices; ! Each of the 32x32 threads accumulates the
  !!! dot product for its element of C(i,j)
    do k = 1,32
      Cij = Cij + Asub(tx,k) * Bsub(k,ty)
    enddo

  !!! Synchronize to make sure all threads are done reading the submatrices before
  !!! overwriting them in the next iteration of the kb loop
    call syncthreads()

  enddo

```

```

!!! Each of the 32x32 threads stores its element to the global C array
    C(i,j) = Cij

    end subroutine mmul_kernel

!!! The host routine to drive the matrix multiplication
subroutine mmul( A, B, C )

!!! assumed shape input arrays
    double precision, dimension(:,:) :: A, B, C

!!! Array dimensions
    integer :: N, M, L

!!! allocatable device arrays
    double precision, device, allocatable, dimension(:,:) :: Adev,Bdev,Cdev

!!! dim3 variables to define the grid and block shapes, predefined in cudafor
    type(dim3) :: dimGrid, dimBlock
    integer :: r

!!! Begin execution, first determine the sizes of the input arrays
    N = size( A, 1 )
    M = size( A, 2 )
    L = size( B, 2 )

!!! Allocate the device arrays using F90 ALLOCATE
    allocate( Adev(N,M), Bdev(M,L), Cdev(N,L) )

!!! Copy A and B to the device using F90 array assignments
    Adev = A(1:N,1:M)
    Bdev = B(1:M,1:L)

!!! Create the grid and block dimensions
    dimGrid = dim3( N/32, L/32, 1 )
    dimBlock = dim3( 32, 32, 1 )

!!! Launch the GPU kernel, wait for completion
    call mmul_kernel<<<dimGrid,dimBlock>>>( Adev, Bdev, Cdev, N, M, L )
    r = cudathreadsynchronize()

!!! Copy the results back
    C(1:N,1:L) = Cdev

!!! Deallocate device arrays and exit
    deallocate( Adev, Bdev, Cdev )

    end subroutine mmul
end module mmul_mod

```

2.3 CULA

CULA is a linear algebra library that utilizes CUDA architecture to improve computational speed. This is a way to use GPU parallel computing without any experience of CUDA. CULA contains libraries familiar to LAPACK interface, which contains for examples of system solvers, eigenvalues routines, singular value decomposition. This can be used in C, C++, FORTRAN, Python and even MATLAB through compiling a mex file.

The only requirements for using CULA is that an NVIDIA GPU with CUDA support is installed, to use double-precision operation the graphic card must be of at least compute capability 1.3 see section 2.2.3.

To illustrate CULA, a program solving the linear equation system $Ka = f$ is shown below.

```
subroutine EQ_SOLVE(K,F,INFO)
  implicit none

  external CULA_INITIALIZE
  external CULA_DGESV
  external CULA_SHUTDOWN

  double precision,intent(in)    :: K(:, :)
  double precision,intent(inout) :: F(:)
  integer,intent(inout)          :: INFO

  integer                        :: NRHS=1, ndof
  integer, allocatable           :: IPIV(:)

  integer :: CULA_INITIALIZE, CULA_DGESV, STATUS

  ndof=size(K,1)
  allocate(IPIV(ndof),STAT=status)

  STATUS = CULA_INITIALIZE()

  STATUS = CULA_DGESV( ndof, NRHS,K, ndof, IPIV, F, ndof, INFO )

  call CULA_SHUTDOWN

  deallocate(IPIV)

end subroutine eq_solve
```

Chapter 3

Background - Crystal plasticity

3.1 Polycrystalline materials

Polycrystalline materials are defined as materials that contain several grains oriented in different directions. In large deformation, crystal plasticity models is often used and reflect the dislocations that occur in the material as slip within a discrete slip system. Dislocations are defects in crystal structures, caused by movement of atoms in the crystal. The movement of dislocations is the mechanism behind plastic deformation.

This makes the model more accurate when studying deformation in polycrystalline materials, because the internal physics are considered in a microscopic level.

In this thesis the crystal structure FCC is considered which contains three slip directions on four closed-packed planes, austenitic steel is a material of this crystal structure.

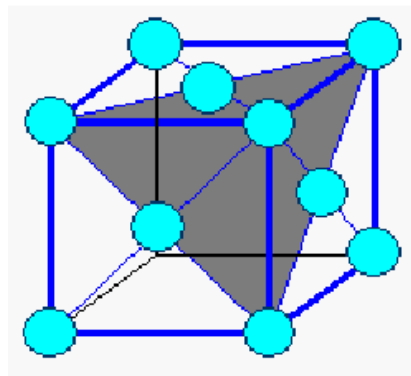


Figure 3.1: FCC crystal structure with the plane (1,1,1) highlighted

3.1.1 Crystal plasticity model

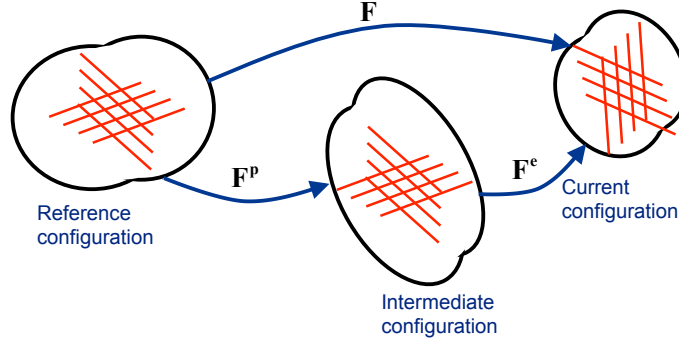


Figure 3.2: Multiplicative split of the deformation gradient

The motion of particles in a body can be described by a particle changing position from one to another. To observe this movement the deformation gradient \mathbf{F} is introduced that maps the line segments from the reference configuration to the current configuration. To separate the reversible and the irreversible part of the deformation gradient a new stress-free intermediate configuration is introduced. This results in a multiplicative split of the deformation gradient, cf. [3], [4],

$$\mathbf{F} = \mathbf{F}^e \mathbf{F}^p \quad (3.1)$$

The time rate of the irreversible part of the deformation gradient is found by introducing the plastic velocity gradient, \mathbf{l}^p

$$\dot{\mathbf{F}}^p = \mathbf{l}^p \mathbf{F}^p \quad (3.2)$$

The plastic deformation takes place on specific slip planes in a crystal plasticity model. For FCC, the slip occurs in $\{111\}\langle\bar{1}01\rangle$ and the plastic evolution is governed by \mathbf{l}^p on a macroscopic scale, here formed as the sum of the shear rate in all slip systems, cf. [8]. Here $\{\}$ specify a family of planes and $\langle\rangle$ a family of directions where slip occurs. The superscript α in equations (3.3) denotes an index which ranges from 1 to n , where n is the number of slip systems.

$$\mathbf{l}^p = \sum_{\alpha=1}^n \dot{\gamma}^\alpha \mathbf{M}^\alpha \otimes \mathbf{N}^\alpha \quad (3.3)$$

Here is \mathbf{N}^α the normal direction to the slip plane and \mathbf{M}^α the slip direction both defined in the intermediate configuration. Since the material directions in the intermediate configuration are chosen to be coincident with the material directions in the reference configuration the orientation of the slip system is not updated. This can be viewed in figure 3.2.

The slip rate $\dot{\gamma}^\alpha$ is determined by a function containing the resolved shear stresses,

$$\dot{\gamma}^\alpha = g(\boldsymbol{\tau}^\alpha, \dots) \quad (3.4)$$

The resolved shear stress can be seen as the shear stress acting on the slip plane in the slip direction. Using the slip direction and the vector normal to the slip plane described above, the resolved shear stress in intermediate configuration is introduced as

$$\boldsymbol{\tau}^\alpha = \mathbf{M}^\alpha \boldsymbol{\Sigma} \mathbf{N}^\alpha \quad (3.5)$$

where $\boldsymbol{\Sigma}$ is the Mandel stress $\boldsymbol{\Sigma} = \mathbf{R}^e \boldsymbol{\tau} \mathbf{R}^{eT}$ ($\mathbf{F}^e = \mathbf{R}^e \mathbf{U}^e$) here \mathbf{R}^e and \mathbf{U}^e are the rotation and stretch of the elastic part of the deformation.

Due to that the intermediate configurations slip directions and slip planes are assumed to be coherent with the orientation of the reference configuration, this implies that the intermediate configuration is an isoclinic configuration, cf. [5].

Specific model, plastic part

The slip rate is then modeled by a power law, according to equation (3.6)

$$\dot{\gamma}^\alpha = \dot{\gamma}_0 \left(\frac{|\boldsymbol{\tau}^\alpha - b^\alpha|}{G_0 + G^\alpha} \right)^m \text{sgn}(\boldsymbol{\tau}^\alpha - b^\alpha) \quad (3.6)$$

Here the parameters $\dot{\gamma}_0$ and m are introduced as reference slip rate and rate sensitivity. The resolved back stress b^α is introduced due to the directional resistance of dislocation movements on slip system level. This results in a kinematic hardening on macroscopic level.

The first part of the slip resistance is G_0 and reflects the lattice friction and is a constant material parameter, cf. [6]. The second part G^α manifest dislocation movements due to short-range interaction between dislocations.

$$G^\alpha = Q \sum_{\beta=1}^n h_{\alpha\beta} g^\beta \quad (3.7)$$

where the cross hardening is defined as $h_{\alpha\beta} = \delta_{\alpha\beta} + q(1 - \delta_{\alpha\beta})$

Back stresses are local on each slip direction and are taken as $b^\alpha = Hv^\alpha$

To determine v^α an evolution law similar to Armstrong - Frederick, cf. [1] is used according to equation (3.8). The evolution law for g^α is used in similar way to equation (3.8) according to [9] see equation (3.9)

$$\dot{v}^\alpha = \dot{\gamma}^\alpha - Rv^\alpha|\dot{\gamma}^\alpha| \quad (3.8)$$

$$\dot{g}^\alpha = (1 - Bg^\alpha)\frac{|\tau^\alpha - b^\alpha|}{G_0 + G^\alpha}|\dot{\gamma}^\alpha| \quad (3.9)$$

In the plastic part following constants are material parameters, G_0 slip resistance, m rate sensitivity, Q and H hardening, q latent-hardening, R the saturation of v^α , B the saturation of g^α

Specific model, elastic part

The second Piola-Kirchhoff stress tensor is defined as

$$\mathbf{S} = (\mathbf{F}^p)^{-1}(\mathbf{C}^e)^{-1}\boldsymbol{\Sigma}(\mathbf{F}^{pT})^{-1} \quad (3.10)$$

where the Mandel stress tensor can be calculated as

$$\boldsymbol{\Sigma} = 2\rho_0\mathbf{C}^e\frac{\partial\psi^e}{\partial\mathbf{C}^e}, \quad \mathbf{C}^e = \mathbf{F}^{eT}\mathbf{F}^e \quad (3.11)$$

and the thermodynamically properties are determined by the choice of the Helmholtz free energy function as

$$\rho_0\psi^e(\mathbf{C}_i^e, J) = K\frac{1}{2}((\ln J))^2 + G \cdot \text{tr}((\ln \mathbf{U}^e)^{dev}(\ln \mathbf{U}^e)^{dev}) \quad (3.12)$$

In the elastic part the material parameters are G the shear modulus and K the bulk modulus. ρ_0 is the mass density in the reference configuration. An index i was introduced in (3.12) to denote isochoric, i.e. volume preserving, quantities.

3.1.2 Numeric implementation

The crystal plasticity model is implemented in an ordinary Lagrangian formulated FE program. The difference is that the stresses and stiffness are calculated on a microscopic level in each grain. The algorithmic stiffness tensor is defined in each grain as $d\mathbf{S} = \mathbf{D}^{ATS} : d\mathbf{E}$ this can be calculated as

$$\mathbf{D}^{ATS} = 4\rho_0 \frac{d}{d\mathbf{C}} \left((\mathbf{F}^p)^{-1} \frac{\partial \psi}{\partial \mathbf{C}^r} (\mathbf{F}^p)^{-T} \right) \quad (3.13)$$

this is solved numerically by an Euler backward method where all the quantities above are calculated in state $n + 1$. A common number of grains can be about 400 in each integration point but can also be well over 1000. This makes it a very demanding analysis. For example an analysis with 372 four-node elements with 400 grains in each integration point will take about a week with 600 load steps if run as a serial program on a desktop computer.

In this thesis an example considering a cyclic loading of Cook's membrane is performed. The dimension of the geometry can be seen in figure 3.3 where $H_1 = 44\text{mm}$, $H_2 = 16\text{mm}$ and $L = 48\text{mm}$. The geometry is modeled by 372 four-node fully integrated plane strain elements. In figure 3.3 u is a prescribed displacement, because the right-hand side of the structure is clamped the displacement can be applied in a single point. The displacement is described according to $u = 2.5 \sin(\frac{2\pi}{5}t)$ where t is the time in seconds. The number of crystals in each integration point was then varied between 1 and 1200.

The result and the numerical implementation from the analyses are according to cf. [7].

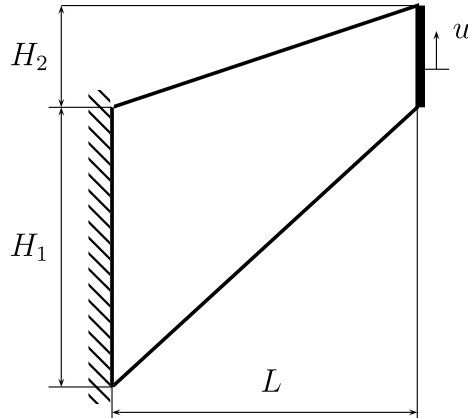


Figure 3.3: Geometry of Cook's membrane

The deformation gradient \mathbf{F} is assumed to be equal in all grain which is a result of the Taylor assumption, introduced by Taylor, cf. [10]. This assumption has the disadvantage of being kinematically over-constrain resulting in a too stiff response. According to the Taylor assumption the second Piola-Kirchhof stresses and the algorithmic stiffness tensor is then obtained through the average over all the grains.

The overall scheme can be viewed in figure 3.4

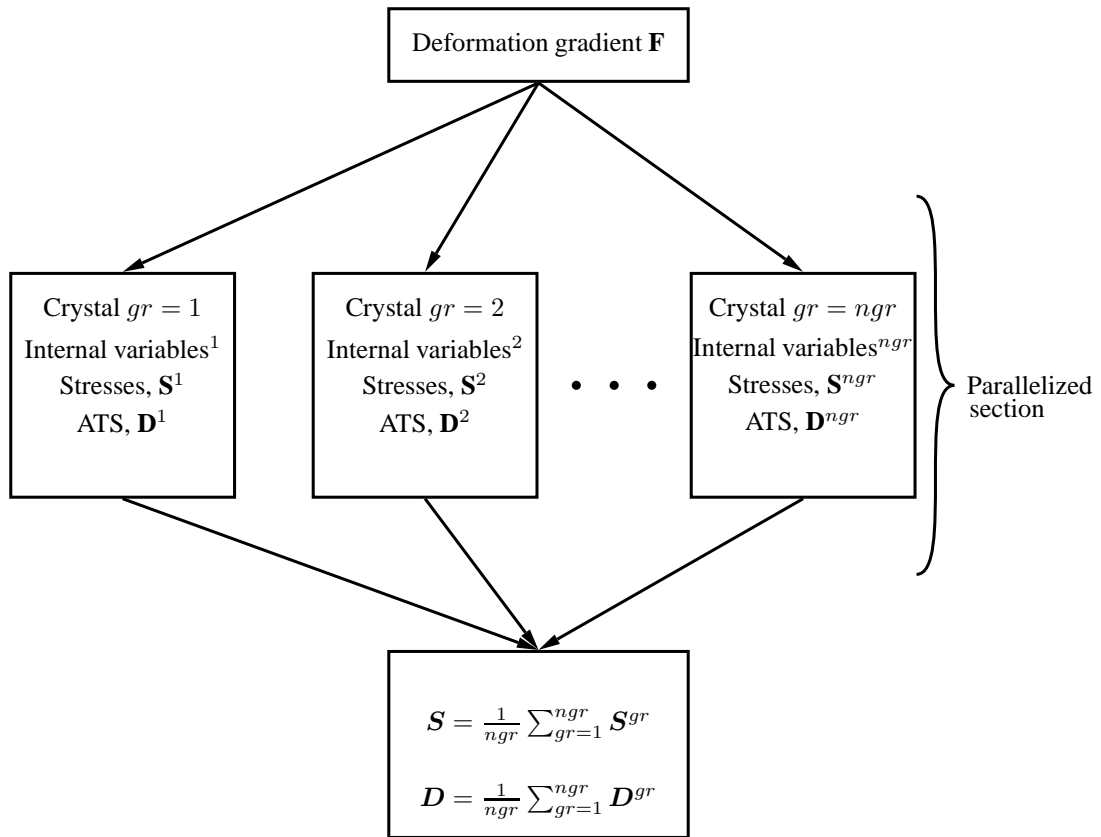


Figure 3.4: Structure of the grain calculations

Chapter 4

Implementation

4.1 CUDA implementation of crystal plasticity

After analyzing the structure of the program the advantage of parallel programming is in the calculation of the algorithmic stiffness tensor and the second Piola-Kirchhoff stress tensor for each grain. This is because every grain calculation is independent from other grains. This can be viewed in figure 3.4.

The calculation of the stiffness tensor and stress tensor must be performed within a device kernel according to 2.2.2 on page 8 and must also be placed in the same MODULE together with the other routines containing device code. To avoid data traffic between the host and the device the internal variables in the subroutine must be declared as device data. This results in that the only variables transferred between the host and device is the deformation gradient and the two output variables; the algorithmic stiffness tensor and the second Piola-Kirchhoff stress tensor.

In order to manage calculation of the average value of the algorithmic stiffness tensor and the second Piola-Kirchhoff stress tensor an another subroutine has to be created, that doesn't contain any parallelized parts. But it still has to be a device kernel because it has to be called from the host and the calculation contains device data. In this subroutine the calls to the memory of type global data has been reduce to improve speed.

As mentioned in 2.2.1 at page 7, subroutines and functions declared with the attribute *device* or *global* can't contain any calls of host code or use the FORTRAN command MATMUL. This criterion have made a big impact on the original code, because every subroutine and even the eigenvalue solver had to be rewritten and put in the same MODULE as the device kernel mentioned above.

```

module

contains

  subroutine main(...)

    call initY(...)

    do for all elements
      do for all intergration points

        copy F to device

        call polyUpd<<<dimGrid,dimBlock>>>(...)

        call sumD<<<1,1>>>(...)

        copy D and S to host
      end do
    end do

  end subroutine main

  attributes(global) subroutine polyUpd(...)

    Executes the amounts of times that is defined in dimGrid and dimBlock
    where gr defines the grain number as:
    gr = (blockIdx%x-1)*blockDim%x + threadIdx%x

  end subroutine polyUpd

  attribute(global) subroutine sumD(...)

    Anverage calculation of the algorithmic stiffness tensor and
    the second Piola-Kirchoff stress tensor

  end subroutine sumD

  ...

end module

```

Chapter 5

Results and Conclusion

5.1 Speed increase

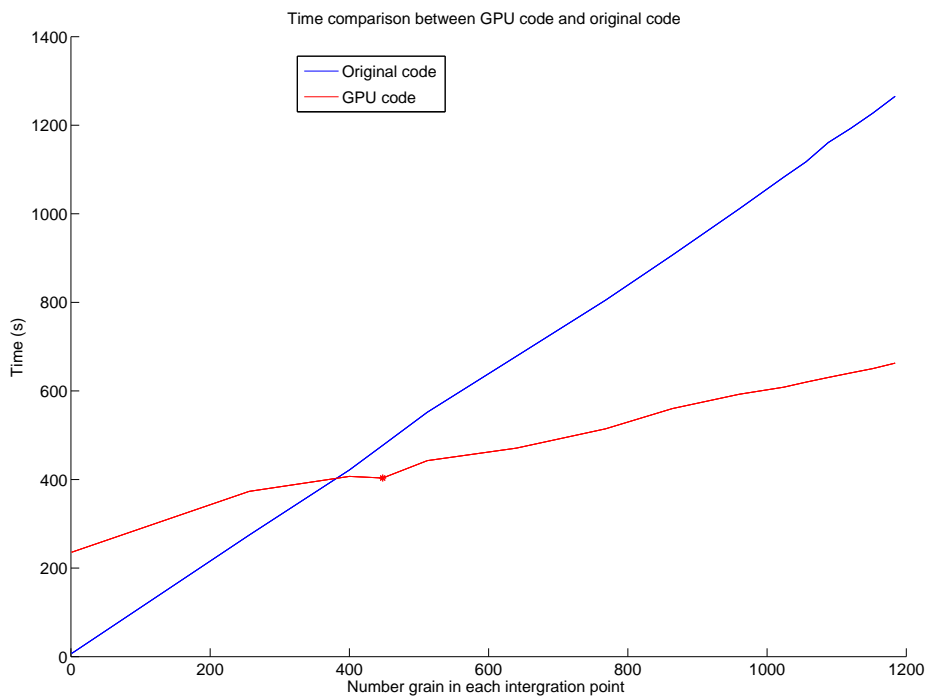


Figure 5.1: Shows the time improvement

The results in figure 5.1 shows one load step in a geometry containing 372 fully integrated four-node elements. As the figure shows, the original code is almost linear and takes 1 s per grain. The GPU code has a little different gradient first of all because of the big cost of data transfer but also because the clock rate is about half compared to the CPU. The benefits shown here are greater when using larger number of grains.

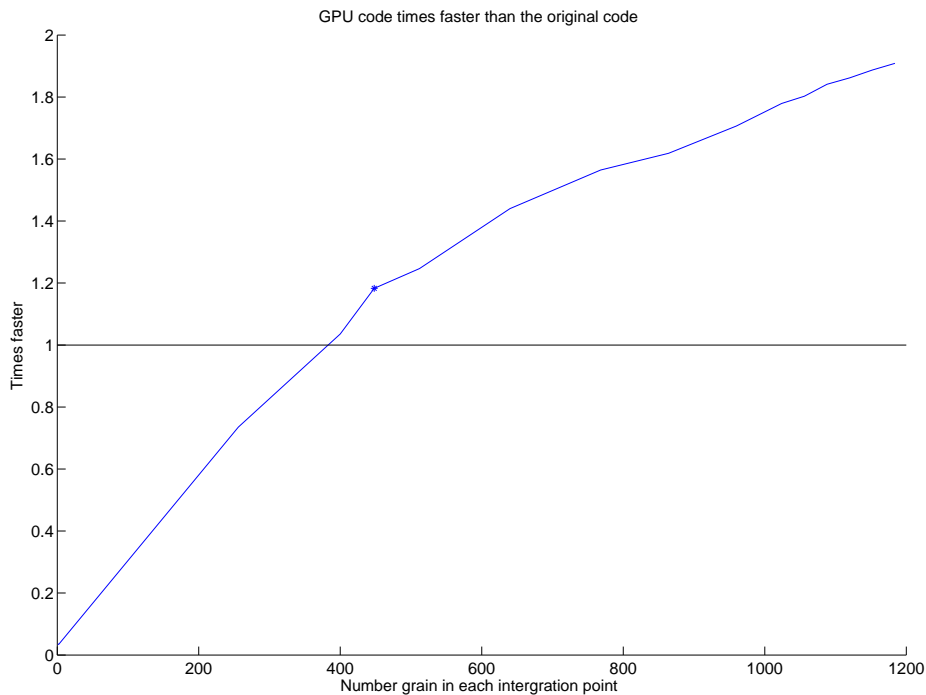


Figure 5.2: Shows the time improvement

In figures 5.1 and 5.2 the points are marked at 448 grains, this is a breaking point corresponding to the hardware, which has 448 cores. After this point a schedule manage the execution, which results in higher time development.

I have also tested to make some changes to the program that makes it very unstable in the meaning that all numbers of grains are not possible to calculate. This would increase the speed for 1024 grain to 372s which is almost 3 times faster then the original code, compared to figure 5.2 were the GPU code is not even 2 times faster.

Because CUDA FORTRAN demands large differences in syntax to get the code compiled. The same type of analys, 1024 grains takes about 3578 s in serial mode on the CPU. If you then compare the results then the parallelization is 5.9 times faster.

This is not an optimal program structure to run on the graphics card as shown in figure 5.2 and compared to figure 2.3 on page 11. The main reason is the large number of global memory requests as a result of the large number of matrix calculation.

5.2 Conclusion

The major part of the work in this thesis has been to get the code compiled using CUDA which is difficult work, because of the restriction in a device kernel and also the fact that all subroutines must be in the same module. This has resulted in a module with 1800 lines of device code, which is not preferable. Because it makes it very hard to compile and also it will probably involve a large number of global memory request.

The next problem was that the architecture memory handling and trying to remove the a big time consumer, the large request of global memory.

Although the structure of the program contained many elements that are not particularly attractive for GPU programming, the result was still positive to the extent that it went faster.

5.2.1 Recommendation

Here is a list that you should start to consider before beginning to implement CUDA.

- Is the serial code optimized, otherwise start there.
- Can the calculations be performed in parallel, for example does it contain a loop that consist of calculation independent from each other.
- Try to figure out a good way to use the different kind of memory slot to make it better. First of all through avoid transfers between the host and the device.
- Also to use sub calculations that can be located on the shared memory. This will lower the amount of global memory request.
- Don't use it on a program structure that contains more than 20 lines of code.

Bibliography

- [1] M.F. Horstemeyer, D.L. McDowell, and R.D. McGinty. Design of experiments for constitutive model selection: application to polycrystal elastoviscoplasticity. *Modelling and Simulation in Materials Science and Engineering*, 7(2):253, 1999.
- [2] D. Kirk and W.W. Hwu. *Programming massively parallel processors : a hands-on approach*. Morgan Kaufmann, San Francisco, Calif., 2010.
- [3] E. Kröner. Allgemeine kontinuumstheorie der versetzungen und eigenspannungen. *Archive for Rational Mechanics and Analysis*, 4:273–334, 1959. 10.1007/BF00281393.
- [4] E.H. Lee. Elastic-plastic deformation at finite strains. *Journal of Applied Mechanics*, 36:1–6, 1969.
- [5] J. Mandel. Plasticite classique et viscoplasticity. *CISM course no. 97*. Springer-Verlag, Udine., 1971.
- [6] T. Ohashi. Crystal plasticity analysis of dislocation emission from microvoids. *International Journal of Plasticity*, 21(11):2071 – 2088, 2005. Plasticity of Heterogeneous Materials.
- [7] M. Wallin P. Håkansson and M. Ristinmaa. Prediction of stored energy in polycrystalline materials during cyclic loading. *International Journal of Solids and Structures*, 45(6):1570 – 1586, 2008.
- [8] J.R. Rice. Inelastic constitutive relations for solids: An internal-variable theory and its application to metal plasticity. *Journal of the Mechanics and Physics of Solids*, 19(6):433 – 455, 1971.
- [9] P. Steinmann and E. Stein. On the numerical treatment and analysis of finite deformation ductile single crystal plasticity. *Computer Methods in Applied Mechanics and Engineering*, 129(3):235 – 254, 1996.
- [10] G.I. Taylor. Plastic strain in metals. *Journal of the Institute of Metals*, 62:307–324, 1938.