Accelerating crystal plasticity simulations using GPU multiprocessors

Mellbin, Ylva; Hallberg, Håkan; Ristinmaa, Matti

Published in:
International Journal for Numerical Methods in Engineering

DOI:
10.1002/nme.4724

2014

International Journal for Numerical Methods in Engineering 2014, 100(2), 111-135

Accelerating crystal plasticity simulations using GPU multiprocessors

Ylva Mellbin, Håkan Hallberg, Matti Ristinmaa

Division of Solid Mechanics
Lund University, Box 118, S-221 00 Lund, Sweden
*ylva.mellbin@solid.lth.se

Abstract

Crystal plasticity models are often used to model the deformation behavior of polycrystalline materials. One major drawback with such models is that they are computationally very demanding. Adopting the common Taylor assumption requires calculation of the response of several hundreds of individual grains to obtain the stress in a single integration point in the overlying FEM structure. However, a large part of the operations can be executed in parallel to reduce the computation time. One emerging technology for running massively parallel computations without having to rely on the availability of large computer clusters is to port the parallel parts of the calculations to a graphical processing unit (GPU). GPUs are designed to handle vast numbers of floating point operations in parallel. In the present work, different strategies for the numerical implementation of crystal plasticity are investigated as well as a number of approaches to parallelization of the program execution. It is identified that a major concern is the limited amount of memory available on the GPU. However, significant reductions in computational time – up to 100 times speedup – are achieved in the present study, and possible also on a standard desktop computer equipped with a GPU.

Keywords: Crystal plasticity, Graphics processing unit, CUDA, GPGPU, Parallelization

1 Introduction

The material properties of metals are dependent on the microstructure of the material. To be able to accurately model plastic deformations it is therefore necessary for the model

DOI: 10.1002/nme.4724

to be able to describe the concurring changes in the microstructure of the metal. Plastic deformations result in increased dislocation density, which in turn makes the material harder, and at the same time the grain orientations evolve, producing a deformation texture in the material. One way of modeling this process is by adopting a crystal plasticity model, which describes plastic deformations through modeling of the slip in the crystal lattice. Early works on crystal plasticity are found in e.g. [1, 2, 3, 4, 5, 6]. The models may be either rate-independent or rate-dependent. For the rate-independent models, there is the problem of resolving the active slip system, which in the worst case lacks a unique solution. For rate-dependent models, formulated as creep models, the basic approach is that all slip systems are considered active at all times. In order to reduce the problem size somewhat and achieve a real elastic domain, i.e a viscoplastic model, it is possible to introduce a threshold into the equations governing the slip, at the cost of introducing conditionals into the code. Whichever approach is chosen, the problem remains ill-conditioned. In this implementation a rate-dependent crystal plasticity model without threshold will be considered.

For polycrystalline materials the stress resulting from a given deformation is found by modeling slip in a number of crystalline grains with different orientations. This provides a model of polycrystal plasticity, also capable of predicting the evolution of crystallographic texture. Although crystal plasticity modeling has become a standard tool in computational mechanics, it is hampered by the significant computational cost related to the method. This is because the model requires the slip to be calculated in a large number of grains in order for it to be statistically relevant. Considering a rate-dependent crystal plasticity model the equations used for computing the slip rates are stiff ordinary differential equations, leading to a high computational cost of the method that has so far put restrictions on the usage of it.

Different approaches to calculating the slip have been used, both implicit and explicit. A common method is to use an implicit Euler method with Newton-Raphson iterations for finding the solution, see [7]. Another approach is the rate tangent scheme discussed in [5], which is explicit but requires the inversion of a matrix with the size of the number of active slip systems. In [8, 9] an explicit Runge-Kutta algorithm is used for crystal plasticity simulations. Another method is introduced in [10], reformulating the system of equations into an optimization problem. Usually the explicit methods for calculation of the constitutive response are used together with dynamic calculations where an explicit time stepping algorithm is used.

With the recent advances in the use of graphical processing units (GPUs) for parallelization of non-graphical applications, it is possible to get considerable parallel throughput on an ordinary desktop computer without having to rely on the availability of large computer clusters. The availability and low cost of General Purpose Graphical Processing Units (GPGPUs) make them attractive compared to more traditional, CPU-based, cluster solu-

tions. Current applications include e.g. finite element-based phase field simulations [11], molecular dynamics models [12, 13], fluid dynamics simulations [14, 15, 16], and general finite element calculations [17, 18].

Since crystal plasticity models, using the Taylor assumption, do not result in any coupling between the grains, it is well suited for parallelization. This has been utilized in [19, 20, 21] for parallelization on CPUs. Those works show that there is much to be gained from parallelization of the calculations, but the implementations rely on the availability of large computer clusters. GPU-implementation presents a way of making it feasible to run crystal plasticity models on a desktop computer, achieving significant speedup without having to depend on external resources.

The largest challenge when porting crystal plasticity simulations to the GPU is in the present work found to be the limited amount of on-chip memory available on the GPU, which becomes an issue since the equations governing the description of plastic slip are stiff and requires an implicit solver to obtain optimal computational efficiency. Still, the present GPU-implementation of crystal plasticity shows that significant reductions in computation time can be achieved. Reduction of computation time by a factor of up to 100 is achieved.

The layout of the present paper is as follows: Section 2 discusses aspects of GPU hardware and CUDA programming. Section 3 details the crystal plasticity model and Section 4 describes different numerical implementations of the model. GPU implementation strategies are discussed in Section 5. Illustrative simulation examples are shown in Section 6 and a concluding discussion on the results is given in Section 7.

2 GPU programming

Since GPUs were originally produced to satisfy the needs of the gaming industry they are small, cheap and mass produced. A high degree of parallelism can be obtained on an ordinary desktop, or even on a laptop computer. GPUs have been developed to be able to simultaneously perform the same floating point operation quickly on thousands of graphical pixels, and thus rely heavily on parallel execution. They are optimized for floating point operations and for working on large data sets [22, 23]. Originally, GPUs where only marginally programmable through different graphics tools, but with the introduction of GPGPUs and CUDA (Compute Unified Device Architecture), a greater flexibility in GPU programming has been achieved. Some aspects of the GPU architecture, especially of the memory layout, is discussed in the following sections.

2.1 Processor architecture

The parallelism in GPUs differs from "ordinary" parallel processors since on the GPU processor cores are grouped together into what is called streaming multiprocessors (SM). The cores in one SM are slaved so that they all run the same instruction at any given time,

but operating on different data. In graphics applications this hardware support for SIMD (Single Instruction Multiple Data) instructions is a useful feature for performing the same operation on all pixels on the screen. The side effect of this is that conditional statements in the code should be avoided, since the threads that are not active inside a conditional statement will still have to wait for those threads which fulfill the conditional to finish. This differs from ordinary CPU parallelization where the processors work independently of each other.

2.2 Memory architecture

The memory on the GPU is also originally tailored to suit graphics applications. This means that the amount of writable on-chip memory is limited. More memory-demanding calculations have to use the slower off-chip memory.

The memory available on the GPU is split into global, local, constant, texture, register and shared memory, illustrated in Figure 1. The global, local, texture and constant memories are situated off-chip. The global memory is accessible from the CPU, while the local memory is only accessible from the GPU, but both use the same cache to speed up access, which is otherwise slow. The texture memory is similar to the global, but is a constant memory i.e. its content is uploaded from the CPU and can not be changed from the GPU. On newer generations of GPU cards the off-chip memory is cached to make access quicker. Local and global memory shares the same cache. On some GPUs there is also the possibility for fetching constants in the global memory using the texture cache. For the global memory there is the possibility to use *coalesced memory access*, meaning that if the memory is properly aligned then threads running in a SIMD fashion can fetch their respective memory in one single transaction.

The constant memory is also situated off-chip, but with a fast on-chip cache which is able to broadcast a value to all threads in a warp in one single clock cycle. Warps are further discussed in section 2.3. On the chip there is also a number of registers for each single processor and a shared memory, accessible for all processors in the same SM. The shared memory is physically the same memory as the cache for the local memory. It is therefore possible to decide how large a part should be used for each purpose.

Local variables are, as far as possible, placed in the registers. When the program uses more memory than available in the registers, the remainder will use the local memory. The compiler tries to determine which variables are most advantageous to place in the registers and which can be placed in local memory with the smallest possible loss of efficiency. But it is also possible to force certain variable to be placed in the registers.

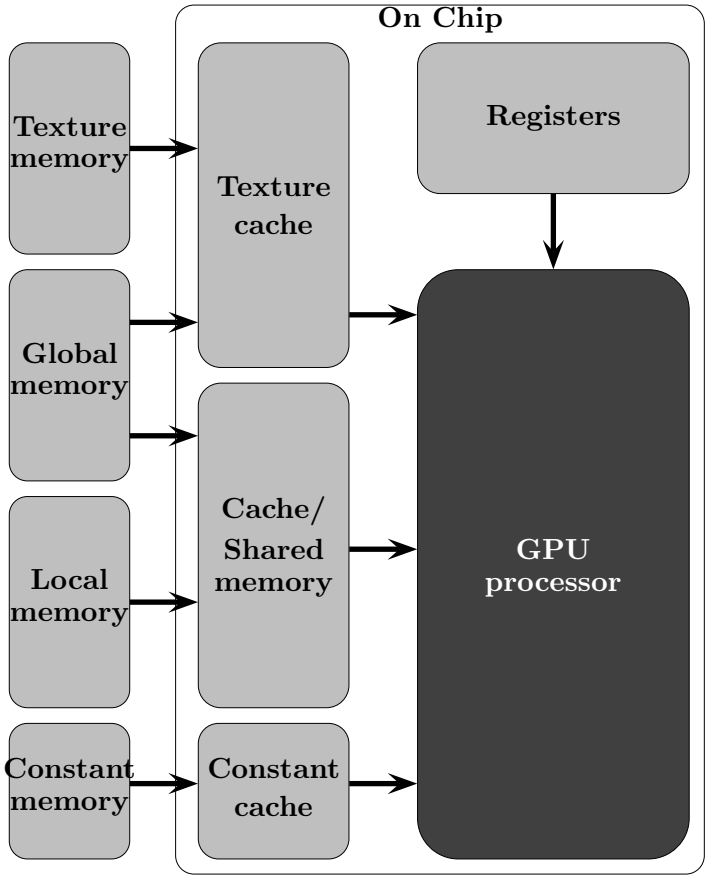

Figure 1: Memory architecture on the GPU.

DOI: 10.1002/nme.4724

2.3 Programming model

A GPU application typically runs its main program on the CPU, transfers data to the GPU, runs computationally intensive parts on the GPU and then transfers the results back to the CPU. Since the bus speed between CPU and GPU is a possible bottleneck it is important to limit data transfer to a minimum [24].

The functions implemented on the GPU are called *kernels*, and contain large numbers of virtual threads executing in parallel. All the threads in a CUDA kernel make up a *grid*, which is divided into *blocks*. The blocks are in turn subdivided into *warps* of 32 threads, where the treads in a warp are synchronized. The GPU has a scheduler that can switch which warp is running on each SM, allowing one warp to run if another has to wait e.g. while variables have to be fetched from local memory, thus hiding memory latency. In order to get optimal performance it is therefore important that the total number of threads in a kernel is considerably larger than the number of threads that can be run in parallel, allowing the scheduler to perform optimally [23]. The number of registers used by each thread determines how many warps can coexist on the same SM. Therefore it may sometimes be favorable to limit the number of registers allowed to each thread even though this results in increased spill to local memory, since it at the same time makes it easier for the scheduler to hide memory latency [24].

3 Crystal plasticity

This section details the crystal plasticity model employed in the present work. At first the single crystal description is given, which is then generalized to polycrystals by homogenization based on the Taylor assumption.

Let the motion of a body be described by a function $\boldsymbol{\varphi}$ which maps the position $\mathbf{X}$ of a particle in the reference configuration to position $\mathbf{x} = \boldsymbol{\varphi}(\mathbf{X}, t)$ of the same particle in the current configuration at time t. The deformation gradient $\mathbf{F}$ is defined as $\mathbf{F} = \partial_X \boldsymbol{\varphi}$. The volume change between the reference and the current configuration then becomes $J = \det(\mathbf{F})$, where $\det(\cdot)$ is the determinant of a tensorial quantity. The right Cauchy-Green deformation tensor is then defined as

$$\mathbf{C} = \mathbf{F}^T\mathbf{F} \tag{1}$$

where $(\cdot)^T$ denotes the transpose of a tensorial quantity.

The deformation of crystalline solids is based on two mechanisms. One part is the elastic component due to distortion of the crystal lattice. The other part is the plastic slip deformation that occurs as dislocations move in their slip systems. Here the notion plastic is used although rate dependent evolution of the inelastic deformation is considered. The slip systems generally comprises the close-packed planes and directions in the lattice.

For face-centered cubic materials, considered here, this provides 12 slip systems of the type $\{111\}\langle 110\rangle$, see [25]. To take elastic and plastic contributions to the deformation into account, the deformation gradient is split into an elastic and a plastic part using a multiplicative split, cf. [26, 27], which provides

$$\mathbf{F} = \mathbf{F}^e\mathbf{F}^p \tag{2}$$

where superscripts e and p denote elastic and plastic quantities, respectively. The elastic right Cauchy-Green deformation tensor can then be defined as

$$\mathbf{C}^e = \mathbf{F}^{eT}\mathbf{F}^e \tag{3}$$

Letting a superposed dot denote the material time derivative, the evolution of the plastic part of the deformation gradient is defined by

$$\dot{\mathbf{F}}^p = \mathbf{l}^p\mathbf{F}^p \tag{4}$$

where the plastic velocity gradient $\mathbf{l}^p$ is calculated through superposition of all crystallographic slip rates according to

$$\mathbf{l}^p = \sum_\alpha \dot{\gamma}^\alpha \mathbf{M}^\alpha \otimes \mathbf{N}^\alpha \tag{5}$$

where $\dot{\gamma}^\alpha$ is the slip rate in slip system α, cf. [3]. Each slip system $\alpha = 1, 2...12$ is represented in the reference configuration by the orthonormal vectors $\mathbf{M}^\alpha$ and $\mathbf{N}^\alpha$, representing the slip direction and the normal to the slip plane of system α. Since $\mathrm{tr}(\mathbf{l}^p) = 0$ this gives $\det(\mathbf{F}^p) = 1$, i.e. the volume change is purely elastic, $J = J^e = \det(\mathbf{F}^e)$, and the plastic deformation is an isochoric process. Here $\mathrm{tr}(\cdot)$ denotes the trace of a tensor. The slip rate in a certain slip system α depends on the resolved shear stress τ^α on that slip system, providing

$$\dot{\gamma}^\alpha = \dot{\gamma}^\alpha(\tau^\alpha, [\cdot]) \tag{6}$$

where $[\cdot]$ indicates dependencies on other variables.

Considering isothermal processes, and denoting the density in the reference configuration by ρ_0 and the Helmholtz free energy by ψ, the dissipation inequality takes the form

$$D = \frac{1}{2}\mathbf{S} : \dot{\mathbf{C}} - \rho_0\dot{\psi} \geq 0 \tag{7}$$

where $\mathbf{S}$ is the second Piola-Kirchhoff stress tensor and is related to the Cauchy stress tensor by

$$\boldsymbol{\sigma} = \frac{1}{J}\mathbf{F}\mathbf{S}\mathbf{F}^T \tag{8}$$

A tensorial contraction over two indices is denoted by $(\cdot) : (\cdot)$.

Assuming the Helmholtz free energy to be a function of the elastic right Cauchy-Green deformation tensor, $\mathbf{C}^e$, and quantities related to the slip resistance, g^α ($\alpha = 1, ..., n$ where n is the number of slip systems), allows the dissipation inequality to be written as

$$D = (\mathbf{S}^e - 2\rho_0 \frac{\partial \psi}{\partial \mathbf{C}^e}) : \frac{1}{2}\dot{\mathbf{C}}^e + \mathbf{\Sigma}^e : \mathbf{l}^p - \sum_{\alpha=1}^{n} G^\alpha \dot{g}^\alpha \geq 0 \tag{9}$$

where use was made of eqs. (1) - (4), and where the second Piola-Kirchhoff stress tensor in the intermediate configuration and the Mandel stress tensor where introduced as

$$\mathbf{S}^e = \mathbf{F}^p \mathbf{S} \mathbf{F}^{pT}, \qquad \mathbf{\Sigma}^e = \mathbf{C}^e \mathbf{S}^e \tag{10}$$

Furthermore, in eq. (9) the slip resistance was identified as

$$G^\alpha = \rho_0 \frac{\partial \psi}{\partial g^\alpha} \tag{11}$$

Requiring that no dissipation should occur during purely elastic processes requires that

$$\mathbf{S}^e = 2\rho_0 \frac{\partial \psi}{\partial \mathbf{C}^e}, \qquad \boldsymbol{\sigma} = \frac{2}{J}\mathbf{F}^e \rho_0 \frac{\partial \psi}{\partial \mathbf{C}^e} \mathbf{F}^{eT} \tag{12}$$

Finally, taking advantage of eq. (5) in eq. (9) leads to that the dissipation inequality can be written as

$$D = \sum_{\alpha=1}^{n} (\tau^\alpha \dot{\gamma}^\alpha - G^\alpha \dot{g}^\alpha) \geq 0 \tag{13}$$

where the resolved shear stress was introduced as

$$\tau^\alpha = \mathbf{M}^\alpha \mathbf{\Sigma}^e \mathbf{N}^\alpha \tag{14}$$

For the specific model it is assumed that the Helmholtz free energy can be split into an elastic and a plastic part, where the elastic part depends on J and the elastic right Cauchy-Green deformation tensor $\mathbf{C}^e$. The plastic part of the Helmholtz energy is a function of the slip parameters g^α which are related to the slip resistance. The Helmholtz free energy can then be stated as

$$\rho_0 \psi(J, \mathbf{C}^e, g^\alpha) = \rho_0 \psi^e(J, \mathbf{C}^e) + \rho_0 \psi^p(g^\alpha) \tag{15}$$

The elastic part is assumed to have the Neo-Hookean form

$$\rho_0 \psi^e = \frac{\kappa}{2}\left[\frac{1}{2}(J^2 - 1) - \ln(J)\right] + \frac{\mu}{2}(J^{-2/3}\mathrm{tr}(\mathbf{C}^e) - 3) \tag{16}$$

where κ and μ are the bulk and shear moduli, respectively. The second Piola-Kirchhoff stress tensor in the intermediate configuration then becomes

$$\mathbf{S}^e = 2\rho_0 \frac{\partial \psi}{\partial \mathbf{C}^e} = \frac{\kappa}{2}(J^2 - 1)\mathbf{C}^{e-1} + \mu J^{-2/3}(\mathbf{I} - \frac{\mathrm{tr}(\mathbf{C}^e)}{3}\mathbf{C}^{e-1}) \tag{17}$$

where $\mathbf{I}$ is the second order identity tensor. Taking advantage of the fact that $\mathbf{M}^\alpha$ and $\mathbf{N}^\alpha$ are orthogonal, the resolved shear stress τ^α on slip system α can now be calculated as

$$\tau^\alpha = \mu \mathbf{M}^\alpha \hat{\mathbf{C}}^e \mathbf{N}^\alpha \tag{18}$$

where $\hat{\mathbf{C}}^e = J^{-2/3}\mathbf{C}^e$ is the isochoric part of the elastic right Cauchy-Green deformation tensor.

In order to describe the cross-hardening that occurs between different slip systems we let the plastic part of the Helmholtz energy assume a quadratic form, as also used in [28, 29], according to

$$\rho_0 \psi^p = \frac{1}{2} Q \sum_\alpha \sum_\beta h_{\alpha\beta} g^\alpha g^\beta \tag{19}$$

with $h_{\alpha\beta} = \delta_{\alpha\beta} + q(1 - \delta_{\alpha\beta})$. The parameter q controls the ratio between self-hardening and cross-hardening. The slip resistance then becomes

$$G^\alpha = \rho_0 \frac{\partial \psi}{\partial g^\alpha} = Q \sum_\beta h_{\alpha\beta} g^\beta \tag{20}$$

The remaining evolution laws for the slip rate $\dot{\gamma}^\alpha$ are given by the power law

$$\dot{\gamma}^\alpha = \dot{\gamma}_0 \left(\frac{|\tau^\alpha|}{G_r^\alpha} \right)^m \mathrm{sign}(\tau^\alpha) \tag{21}$$

where G_r^α is the total slip resistance on the system. This slip resistance is given by the sum

$$G_r^\alpha = G_0 + G^\alpha \tag{22}$$

where G_0 is a constant resulting from lattice friction while G^α is due to dislocation interactions, individual for each slip system. The evolution laws for the slip parameters g^α are finally given by

$$\dot{g}^\alpha = (1 - B g^\alpha) \frac{|\tau^\alpha|}{G_r^\alpha} |\dot{\gamma}^\alpha| \tag{23}$$

With the above the dissipation inequality can be written as

$$D = \sum_{\alpha=1}^{n} \left[|\tau^\alpha| \left(1 - \frac{G^\alpha}{G_r^\alpha} \right) + B g^\alpha |\tau^\alpha| \frac{G^\alpha}{G_r^\alpha} \right] |\dot{\gamma}^\alpha| \geq 0 \tag{24}$$

As $G_r^\alpha \geq G^\alpha$ it can be concluded that the mechanical dissipation is always positive. The specific model considered here is an isothermal variant of the model given in [30] where also kinematic hardening is considered, here only isotropic hardening is considered. In [30] it was also concluded that the mechanical dissipation given by eq. (24) provides a realistic heat generation due to the plastic deformation process.

4 Numerical method

The crystal plasticity model described above is employed in a nonlinear finite element framework. In the integration points within each element, the stresses are calculated by considering the deformations in a large number of crystals. Assuming all crystals in one integration point to be subjected to the same deformation gradient, i.e. adopting a Taylor assumption, an efficient approach is obtained. Whereas explicit dynamic calculations only require the stress to be calculated, implicit dynamic and static calculations also require that the algorithmic tangent stiffness is evaluated.

The next section describes the generalization to polycrystal plasticity, and introduces the tangent stiffness. The following sections discuss different strategies for the numerical implementation. Aiming at efficient GPU implementation, parallelization and memory consumption issues related to the different schemes are discussed.

4.1 Polycrystal plasticity

When the displacements have been calculated, the strains are subsequently used for calculating the evolution of γ^α and g^α. The new values of γ^α and g^α are in turn used for calculating the plastic deformation gradient, stress, and tangent stiffness in each grain. Finally the stress and tangent stiffness for each integration point is calculated as the averages over all grains at that point.

When using the single crystal constitutive model described above for modeling of a polycrystalline material there are several different homogenization schemes that can be used. In the present work the homogenization introduced by Taylor in [1] is employed, whereby the grains are assumed to be subjected to the same deformation gradient. This approach has the advantage that no coupling between the grains exists, which makes the model suitable for parallelization. In order to obtain the global stress, assuming all grains to be of equal size, the stresses in each grain are averaged according to

$$\mathbf{S} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{S}_i \tag{25}$$

where $\mathbf{S}_i$ is the stress in grain i and n is the number of grains. Homogenization based on the Taylor assumption has previously been used, for example, in the crystal plasticity models in [31, 6, 32, 33, 34]. Since the formula can be interpreted as assuming all grains

to be of the same size, the volume fraction of grains with a specific orientation can be calculated as the number of grains with this orientation divided by the total number of grains.

The algorithmic tangent stiffness $\mathbf{D}$, necessary to obtain proper quadratic convergence in the Newton-Raphson scheme in an implicit finite element program, is defined as

$$\mathbf{D} = 2\frac{d\mathbf{S}}{d\mathbf{C}} \tag{26}$$

and is averaged according to the scheme in eq. (25), providing

$$\mathbf{D} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{D}_i = \frac{1}{n}\sum_{i=1}^{n}2\frac{d\mathbf{S}_i}{d\mathbf{C}} \tag{27}$$

Since the second Piola-Kirchhoff stress tensor can be obtained from

$$\mathbf{S} = \mathbf{F}^{p-1}\mathbf{S}^e\mathbf{F}^{p-T} \tag{28}$$

the tangent stiffness can be expressed as

$$\mathbf{D} = 2\left(\frac{d\mathbf{F}^{p-1}}{d\mathbf{C}}\mathbf{S}^e\mathbf{F}^{p-T} + \mathbf{F}^{p-1}\frac{d\mathbf{S}^e}{d\mathbf{C}^e}\frac{d\mathbf{C}^e}{d\mathbf{C}}\mathbf{F}^{p-T} + \mathbf{F}^{p-1}\mathbf{S}^e\frac{d\mathbf{F}^{p-T}}{d\mathbf{C}}\right) \tag{29}$$

Establishing the components of eq. (29), differentiation of eq. (17) provides

$$\frac{d\mathbf{S}^e}{d\mathbf{C}^e} = a_1\mathbf{C}^{e-1}\otimes\mathbf{C}^{e-1} - a_2(\mathbf{I}\otimes\mathbf{C}^{e-1} + \mathbf{C}^{e-1}\otimes\mathbf{I}) - a_3\frac{d\mathbf{C}^{e-1}}{d\mathbf{C}^e} \tag{30}$$

with

$$a_1 = \frac{\kappa}{2}J^2 + \frac{\mu}{9}(J^{-2/3})\mathrm{tr}(\mathbf{C}^e) \tag{31}$$

$$a_2 = \frac{\mu}{3}J^{-2/3} \tag{32}$$

$$a_3 = \frac{\mu}{3}J^{-2/3}\mathrm{tr}(\mathbf{C}^e) - \frac{\kappa}{2}(J^2-1) \tag{33}$$

and using $\mathbf{C}^e = \mathbf{F}^{p-T}\mathbf{C}\mathbf{F}^{p-1}$ provides

$$\frac{d\mathbf{C}^e}{d\mathbf{C}} = \frac{d\mathbf{F}^{p-T}}{d\mathbf{C}}\mathbf{C}\mathbf{F}^{p-1} + \mathbf{F}^{p-T}\frac{d\mathbf{C}}{d\mathbf{C}}\mathbf{F}^{p-1} + \mathbf{F}^{p-T}\mathbf{C}\frac{d\mathbf{F}^{p-1}}{d\mathbf{C}} \tag{34}$$

Thus it can be concluded that calculation of the algorithmic tangent stiffness only requires the calculation of

$$\frac{d\mathbf{F}^{p-1}}{d\mathbf{C}} \tag{35}$$

or alternatively

$$\frac{d\mathbf{F}^{p}}{d\mathbf{C}} = \frac{d\mathbf{F}^{p}}{d\mathbf{F}^{p-1}}\frac{d\mathbf{F}^{p-1}}{d\mathbf{C}} \tag{36}$$

in order to be completed.

4.2 Backward Euler

A glance at eq. (21), describing the evolution of γ^α, reveals that its behavior will be controlled by the term raised to a power m. Since the exponent m is usually taken as a large number, the term will become close to zero when the resolved shear stress is less than the slip resistance, and one when they are equal. Should the resolved shear stress become larger than the slip resistance the value of the term $(\cdot)^m$ will grow rapidly, making the system very sensitive to the value of the ratio $|\tau^\alpha|/G_r^\alpha$. This sensitivity makes explicit methods less than optimal. A common approach in crystal plasticity implementations is to use a backward Euler method, and to solve the resulting equations using Newton-Raphson iterations. Applying this method on the equations for $\dot{\gamma}^\alpha$ and $\dot{g}^\alpha$ provides the residual equations

$$R_1^\alpha = \Delta\gamma^\alpha - \Delta\gamma_0 \left(\frac{|\tau^\alpha|}{G_r^\alpha}\right)^m \text{sign}(\tau^\alpha) \tag{37}$$

$$R_2^\alpha = g_{n+1}^\alpha - \frac{1}{B}\left[1 - (1 - Bg_n^\alpha)\exp\left(-B\frac{|\tau^\alpha|}{G_r^\alpha}|\Delta\gamma^\alpha|\right)\right] \tag{38}$$

where $\Delta\gamma_0 = \gamma_0\Delta t$, with Δt denoting the time increment. It turns out that one of the equation sets can be eliminated through use of eq. (37) in eq. (38), providing

$$g_{n+1}^\alpha = \frac{1}{B}\left[1 - (1 - Bg_n^\alpha)\exp(-B\left(\frac{|\Delta\gamma^\alpha|}{\Delta\gamma_0}\right)^{1/m}|\Delta\gamma^\alpha|)\right] \tag{39}$$

resulting in a simple form of the residual function which appears as

$$R^\alpha(\Delta\gamma^\alpha) = \Delta\gamma^\alpha - \Delta\gamma_0 \left(\frac{|\tau^\alpha|}{G_r^\alpha}\right)^m \text{sign}(\tau^\alpha) \tag{40}$$

The resulting residual function in eq. (40) is then iteratively solved.

An exponential update is used for the plastic deformation gradient to update its value from state n to $n+1$ according to

$$\mathbf{F}_{n+1}^p = \mathbf{A}^p\mathbf{F}_n^p \tag{41}$$

with $\mathbf{A}^p$ being defined as

$$\mathbf{A}^p = \exp(\Delta\mathbf{l}^p) \tag{42}$$

The increment in the plastic velocity $\Delta\mathbf{l}^p$ is defined from

$$\Delta\mathbf{l}^p = \sum_\alpha \Delta\gamma^\alpha \mathbf{M}^\alpha \otimes \mathbf{N}^\alpha \tag{43}$$

By this approach, the new elastic right Cauchy-Green deformation tensor can be found as

$$\mathbf{C}^e_{n+1} = \mathbf{A}^{p-T}\mathbf{F}^{p-T}_n\mathbf{C}_{n+1}\mathbf{F}^{p-1}_n\mathbf{A}^{p-1} \tag{44}$$

which by introduction of a trial elastic deformation tensor as $\hat{\mathbf{C}}^{e,trial}_{n+1} = \mathbf{F}^{p-T}_n\hat{\mathbf{C}}_{n+1}\mathbf{F}^{p-1}_n$ gives the updated quantity

$$\hat{\mathbf{C}}^e_{n+1} = \mathbf{A}^{p-T}\hat{\mathbf{C}}^{e,trial}_{n+1}\mathbf{A}^{p-1} \tag{45}$$

To promote computational efficiency, the exponential $\mathbf{A}^p$ is calculated using a Pade approximation on the form

$$\mathbf{A}^p = (\mathbf{I} - \frac{1}{2}\Delta\mathbf{l}^p)^{-1}(\mathbf{I} + \frac{1}{2}\Delta\mathbf{l}^p) \tag{46}$$

As concluded above the algorithmic tangent stiffness requires the calculation of

$$\frac{d\mathbf{F}^{p-1}_{n+1}}{d\mathbf{C}} = \mathbf{F}^{p-1}_n\frac{d\mathbf{A}^{p-1}}{d\mathbf{C}} = \mathbf{F}^{p-1}_n\frac{d\mathbf{A}^{p-1}}{d\gamma^\alpha}\frac{d\gamma^\alpha}{d\mathbf{C}} \tag{47}$$

The term $d\gamma^\alpha/d\mathbf{C}$, appearing last in eq. (47), can be calculated from the system

$$0 = \frac{dR^\alpha}{d\mathbf{C}} = \frac{\partial R^\alpha}{\partial\gamma^\beta}\frac{d\gamma^\beta}{d\mathbf{C}} + \frac{\partial R^\alpha}{\partial\mathbf{C}} \tag{48}$$

which makes all the necessary components of the algorithmic tangent stiffness available.

4.3 Runge-Kutta

Since implicit methods include matrix inversions, implying significant memory requirements, they are not optimal for use on a GPU. In order to reduce the memory requirements an explicit Runge-Kutta method (a third order Kutta method) has been implemented, which should be more suited for parallelization on the graphics card. However, the drawback is that they are less suited for stiff ordinary differential equations (ODEs). Expressing the system in $\dot{\mathbf{F}}^p$ and $\dot{g}^\alpha$ instead of $\dot{\gamma}^\alpha$ and $\dot{g}^\alpha$ reduces the size of the problem from 24 to 21 ODEs. Because of the lack of stability of explicit methods, the update from state n to $n+1$ is performed in multiple steps, where the value of $\mathbf{C}$ in step i is calculated as

$$\mathbf{C}_i = (1-\lambda)\mathbf{C}_n + \lambda\mathbf{C}_{n+1} \tag{49}$$

where

$$\lambda = \frac{t_i - t_n}{\Delta t} \tag{50}$$

Defining a vector y from $\mathbf{F}^p$ and g^α and defining $\dot{y} = f(t, y)$, the update is performed as

$$k_1 = f(t_i, y_i) \tag{51}$$

$$k_2 = f(t_i + \frac{1}{2}\Delta t, y_i + \frac{1}{2}\Delta t k_1) \tag{52}$$

$$k_3 = f(t_i + \frac{3}{4}\Delta t, y_i + \frac{3}{4}\Delta t k_2) \tag{53}$$

$$y_{i+1} = y_i + \Delta t(\frac{2}{9}k_1 + \frac{1}{3}k_2 + \frac{4}{9}k_3) \tag{54}$$

In order to calculate the tangent stiffness corresponding to $\mathbf{F}^p$ the component $d\mathbf{F}^p/d\mathbf{C}_{n+1}$ is required. Noting that

$$\frac{\overline{\dot{dy}}}{d\mathbf{C}_{n+1}} = \frac{d\dot{y}}{d\mathbf{C}_{n+1}} = \frac{df}{d\mathbf{C}_{n+1}} = \frac{\partial f}{\partial y}\frac{dy}{d\mathbf{C}_{n+1}} + \frac{\partial f}{\partial \mathbf{C}_{n+1}} \tag{55}$$

it is possible to introduce $Y = (y, dy/d\mathbf{C}_{n+1})$ and $\dot{Y} = F(t, f, df/d\mathbf{C}_{n+1})$. Using the Runge-Kutta scheme above we now get

$$k_1 = F(t_i, Y_i) \tag{56}$$

$$k_2 = F(t_i + \frac{1}{2}\Delta t, Y_i + \frac{1}{2}\Delta t k_1) \tag{57}$$

$$k_3 = F(t_i + \frac{3}{4}\Delta t, Y_i + \frac{3}{4}\Delta t k_2) \tag{58}$$

$$Y_{i+1} = Y_i + \Delta t(\frac{2}{9}k_1 + \frac{1}{3}k_2 + \frac{4}{9}k_3) \tag{59}$$

and are thus able to calculate the values of the derivatives $d\mathbf{F}^p/d\mathbf{C}_{n+1}$ in state $n + 1$. This approach corresponds to what is introduced in [35], and later used in [8] for crystal plasticity modeling.

Since $\mathbf{F}^p$ and g^α are coupled, the analytic expressions for calculating the derivatives $d\mathbf{F}^p/d\mathbf{C}$ and $dg^\alpha/d\mathbf{C}$ yield a set of 147 coupled ODEs, requiring a large number of matrix multiplications for each function evaluation. This makes the calculations unsuitable for implementation on a GPU. It was found in the present work that even on a CPU the computational cost for the calculation of the analytic derivatives is comparable to that found when using numerical differentiation. Therefore the derivatives are calculated through numerical differentiation, with the system of 21 ODEs being solved seven times for each grain: once to obtain the values of $\mathbf{F}^p$ and g^α, and six times with different perturbations added to $\mathbf{C}$ in order to be able to calculate the derivatives. This approach reduces the size of the calculations and increases the parallelism since the calculations in each grain can now be easily split over seven threads.

Runge-Kutta schemes with adaptive step-size are not used in this implementation. This is in part because the resulting branch divergence and increased overhead would make

the methods less suited for implementing on a GPU. The main reason, however, is that different step size in the perturbed solutions may disturb the tangent stiffness, such that the quadratic convergence of the Newton-Raphson iterations is influenced.

4.4 Operator split

In order to get optimal performance on the GPU, the solution method should optimally be even less memory demanding than the third order Runge-Kutta scheme, while at the same time at least as stable. In order to reduce the size of the problem an operator split method was introduced, solving $\mathbf{F}^p$ and g^α separately in each time step. Since the memory constraints rules out implicit methods, $\mathbf{F}^p$ is solved with a semi-implicit Euler method according to

$$\mathbf{F}^p_{i+1} = \mathbf{F}^p_i + \Delta t \mathbf{l}^p_i \mathbf{F}^p_{i+1} \tag{60}$$

where $\mathbf{F}^p_{n+1}$ can be calculated directly as

$$\mathbf{F}^p_{i+1} = (\mathbf{I} - \Delta t \mathbf{l}^p_i)^{-1} \mathbf{F}^p_i \tag{61}$$

The slip parameters g^α are then updated through a one-step Newton iteration by evaluating

$$g^\alpha_{i+1} = g^\alpha_i + \Delta t (I - \Delta t \frac{\partial \dot{g}^\alpha}{\partial g^\beta})^{-1} \dot{g}^\alpha_i \tag{62}$$

In order to get rid of the matrix inversion, only the diagonal elements of the Jacobian are used, leaving

$$g^\alpha_{i+1} = g^\alpha_i + \frac{\Delta t \dot{g}^\alpha_i}{(1 - \Delta t \frac{\partial \dot{g}^\alpha}{\partial g^\alpha})} \tag{63}$$

This method yields about the same stability as the explicit Runge-Kutta method, but since it removes the need to store values of the derivatives at several stages the memory requirements are reduced. As for the Runge-Kutta case, the value of $d\mathbf{F}^p/d\mathbf{C}_{n+1}$, necessary for the calculation of the tangent stiffness, is obtained numerically.

5 GPU implementation

For the GPU implementation of the crystal plasticity model, the large number of uncoupled grains provides a significant potential for parallelization. The calculations of slip, stresses and tangent stiffness are performed on the GPU using one thread per grain, or seven threads per grain in the cases with a numeric tangent stiffness. The program execution is illustrated in Figure 2. The CPU parts of the code are written in Fortran and the GPU part in CUDA C.

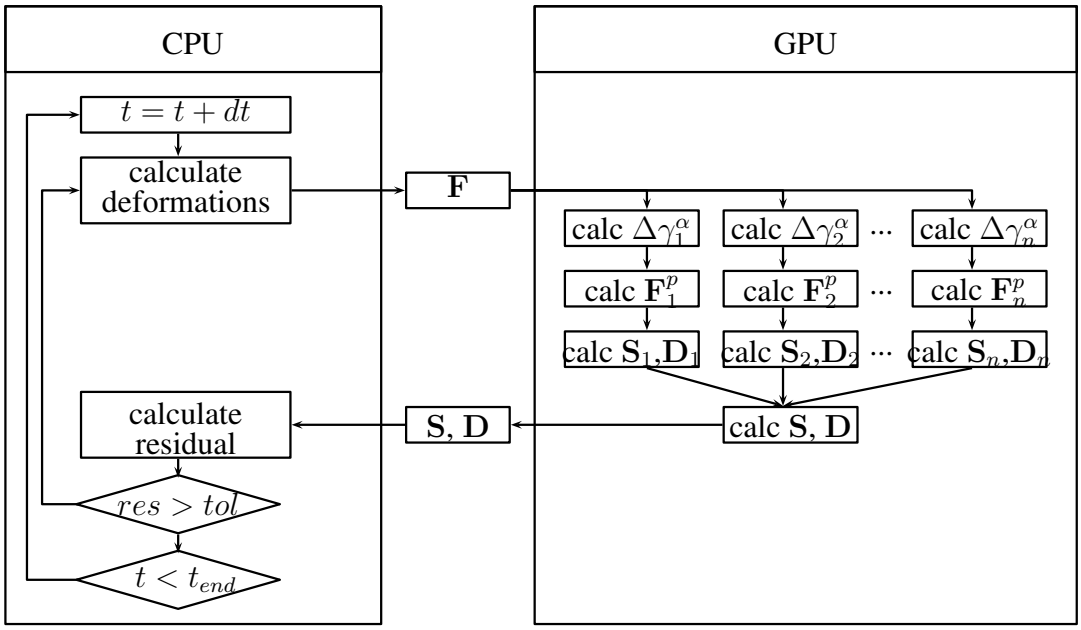

Figure 2: Sketch of the program flow when running on the GPU.

For the backward Euler approach the Newton-Raphson scheme, used in the calculation of $\Delta\gamma^\alpha$ to minimize the system residual, contains a loop that runs until a condition on the maximum residual is fulfilled. This seemingly breaks the rule about avoiding conditional statements. However, all grains in the same element are usually subject to similar deformations. Therefore the threads are likely to require approximately the same number of loop iterations to converge. Data from profiling the program execution supports this assumption.

Since the calculations involve a large number of matrices, the memory limitations of the GPU is the main bottleneck. Therefore some effort has been put into limiting the demand on local memory and trying to minimize the number of cache misses. Since there is no coupling between the grains, there is no obvious use for shared memory. Instead the largest possible cache for the local memory is used. All material parameters are put in the constant memory. Since $\mathbf{M}^\alpha$ and $\mathbf{N}^\alpha$ are constant but individual for each grain, resulting in matrices too large for the constant memory, they have been placed in the texture memory.

All floating point calculations are performed with double precision since the high exponential m makes the equations too sensitive to round-off errors for permitting use of single precision.

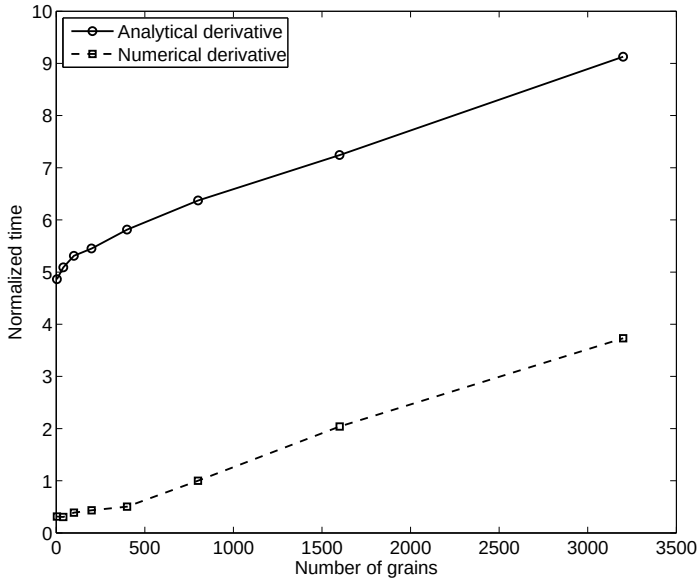

Figure 3: Comparison between Runge-Kutta implementations using analytical and numerical derivatives respectively, for different numbers of grains per iteration point.

5.1 Optimization strategies

Since memory transactions are the main bottleneck, one focus of the present work has been on minimizing memory usage and making it more effective. The array structure of the global matrices has been rearranged so that memory access can be performed in a coalesced manner. For the backward Euler implementation the GPU part of the program has also been split into two kernels, one for the Newton-Raphson iteration used for finding $\Delta\gamma^\alpha$ and one for computing stresses and tangent stiffness. While this split results in some extra overhead it has the benefits of reducing the total amount of memory used in each kernel and allowing values calculated in the first kernel to be treated as constants in the second kernel, making it possible for them to be fetched through the texture cache.

As mentioned in section 4.3, analytical calculations of the tangent stiffness for the explicit methods results in a large system not very well suited for GPU implementation. By using numerical differentiation the system size is reduced and the degree of parallelism increased. Figure 3 shows the decrease in runtime achieved by switching differentiation scheme. A forward difference scheme is used for calculating the numerical derivative.

Variables which are used by multiple threads, e.g. $\mathbf{C}$ and $J^{-2/3}$ which has the same value for all grains in the same integration point, are precomputed on the CPU. This both saves the need for performing the same computation hundreds of times, and allows them to

be uploaded into the constant memory. The use of constant memory results in fast access while at the same time reducing the load on the registers. This in turn results in less usage of local memory and in fewer cache misses.

Listing 1: Array reduction for the stress tensor

```
__global__ void reducePK(double *g_idata, double *g_odata, int n){
    extern __shared__ double sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*n + tid;
    sdata[tid] = 0;

    while(i < (blockIdx.x+1)*n){
        sdata[tid] += g_idata[i];
        i += BLOCK_SIZE;
    }
    __syncthreads();

    if(BLOCK_SIZE>=512){
        if(tid<256) sdata[tid]+=sdata[tid+256];
        __syncthreads();
    }
    if(BLOCK_SIZE>=256){
        if(tid<128) sdata[tid]+=sdata[tid+128];
        __syncthreads();
    }
    if(BLOCK_SIZE>=128){
        if(tid< 64) sdata[tid]+=sdata[tid+ 64];
        __syncthreads();
    }
    if(tid<32) {
        if(BLOCK_SIZE>=64) sdata[tid]+=sdata[tid+32];
        if(tid<16)
                if(BLOCK_SIZE>=32) sdata[tid]+=sdata[tid+16];
        if(tid<8)
                if(BLOCK_SIZE>=16) sdata[tid]+=sdata[tid+ 8];
    }
    if (tid < NUM_GP) {
        g_odata[blockIdx.x+tid*9] = sdata[tid]*NUM_GP/n;
    }
}
```

In order to reduce memory transfers and eliminate the possible bottleneck caused by the memory bus between CPU and GPU, as much data as possible is kept in the global memory on the GPU. Memory transfers are further reduced by calculating the average stress and tangent stiffness in each integration point by an array reduction implemented on the GPU. The code for averaging of the stress tensor is given in Listing 1. This is a modification of the array reduction presented in [36]; the reduction is stopped when eight elements remain in the array, corresponding to the eight integration points present in each

brick element in the FE model, and the number of blocks in the kernel is chosen equal to the size of the tensor so that each block is responsible for one component of the stress tensor.

In order to get enough threads for the scheduler to be able to hide latency by switching between warps, all grains belonging to the same element are considered at the same time. It would be possible, for large structures, to combine grains from more than one element in the same kernel, but once the scheduler has enough threads to properly fill the GPU, increasing the number of threads even further would not increase efficiency significantly, while the overhead costs would increase.

One way of reducing the work and memory load for each thread would be to split the calculations for each grain over several threads, using one thread for calculation of each of the nine elements in the many 3-by-3 matrices, e.g. $\mathbf{F}^p$ and $\mathbf{l}^p$, and of the values for each of the twelve slip system, g^α, γ^α, and τ^α. This approach reduces register pressure and memory spills, but of course also reduces the number of grains that can be treated in parallel as well as introducing the issue of sharing data between threads. This requires the use of shared memory and makes it necessary to keep the threads synchronized. Synchronization can be achieved either explicitly, which reduces the performance of the kernel since it limits the options for the scheduler, or by making sure that all threads related to the same grain reside in the same warp. The latter approach means that 16 threads (a half-warp) have to be launched for each grain, instead of just 12, increasing the total number of threads required by a third. Initial efforts revealed that neither approach results in an overall speedup of the program, and since the readability of the code suffers heavily the subject has not been pursued further.

6 Results

In order to be able to evaluate the speedup, the same model has been implemented in both a single CPU version and in a version where calculation of the stresses and the tangent stiffness are performed on a GPU. The code has also been run using two different hardware setups. The first one with an Intel Xeon E5-2650 @2.0 GHz CPU with 64 GB memory and a Nvidia Tesla K20 GPU with 5 GB @2.6 GHz GDDR5 memory. The second system is a laptop with an Intel core i3-2310M @2.1 GHz CPU with 4 GB memory and a Nvidia GeForce GT 540M GPU with 2 GB dedicated DDR3 memory working at 900 MHz. Tesla K20 is at present time a top of the line card and is especially suited for scientific calculations. The GeForce is an ordinary graphics card such as the one that might be found in an average desktop or laptop computer. On both setups the CPU-part of the code has been compiled using Intels Fortran compiler version 13.1.1 and -O3 optimization. The CUDA code was compiled with Nvidias nvcc with -O3 optimization; the first setup with nvcc version 5.5 and compiling for compute capacity 3.5, and the second setup with

nvcc version 5.0 compiling for compute capacity 2.1.

Most of the simulations have been conducted on a single brick element, using a trilinear isoparametric formulation with eight integration points, subject to pure tension. This example has been investigated using both an explicit and an implicit finite element formulation, i.e. when only the stresses are needed and when both stresses and tangent stiffness need to be calculated. In order to show that the scaling remains the same for a larger structure some tests have also been performed on a simple geometry, consisting of a plate with a hole, shown in Figure 15. The geometry is discretized using 370 brick elements. Material parameters, pertaining to pure Cu, are taken from [30] and are summarized in Table 1.

6.1 Simulations based on a single element

Since the code is parallelized so that for each element the calculations for all grains are performed in parallel, the characteristics of the implementation can be demonstrated with an example containing only one single element. Therefore a series of tests has been performed where one 3D brick element with eight integration points has been subjected to pure tension, see Figure 4. Different numbers of grains per integration point are used to evaluate the global material behavior under loading. The individual grains are initiated with random orientations in terms of the three Bunge-Euler angles (φ_1,Φ,φ_2). These orientations are initiated as $\varphi_1 = 2\pi r_1$, $\Phi = \text{acos}(1 - 2r_2)$ and $\varphi_2 = 2\pi r_3$ where $r_{1,2,3}$ are random numbers, obtained from a uniform distribution in the interval [0,1]. The initial orientation distribution is shown in the pole figure in Figure 6a. The material is loaded in the rolling direction (RD).

Figure 5 shows a stress/strain plot obtained from the simulations. The large deformations also affect the texture of the microstructure, as can be seen from Figure 6b where the completely random orientation distribution evolves into the texture shown in Figure 6b at the end of the deformation process. The pole figures are of the equal-area mapping type and the texture plots are {111}-pole figures. It is concluded that the initially randomly distributed {111}-poles move in the RD-direction on the lower and the upper side of the pole figure and towards two equally distributed horizontal bands.

6.1.1 Explicit finite element formulation

In the explicit test program a constant displacement rate of 0.4 mm/s is used with a time step of 2.5×10^{-4} s. The simulation is continued for 10000 steps, which results in 100% strain. With such small strain increments, which is usual when running explicit dynamics, the Runge-Kutta and operator split solvers only requires 2 steps for the integration over one time step, which gives them an advantage when compared to the backward Euler method which requires 2-3 iterations to converge.

Table 1: Material parameters

Parameter	Value	Description
μ	80 GPa	Shear modulus
κ	164 GPa	Bulk modulus
G_0	55 MPa	Lattice friction
Q	200 MPa	Hardening parameter
$\dot{\gamma}_0$	0.001	Reference slip rate
g_0	0.007	Initial value of g^α
q	1.4	Ratio between self and cross hardening
B	8	Parameter controlling saturation of g^α
m	26	Rate sensitivity

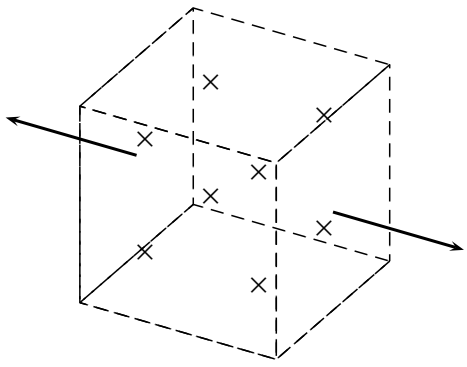

Figure 4: The one-element test case. Positions of the integration points are indicated by "x" and the loading direction is shown by arrows.

DOI: 10.1002/nme.4724

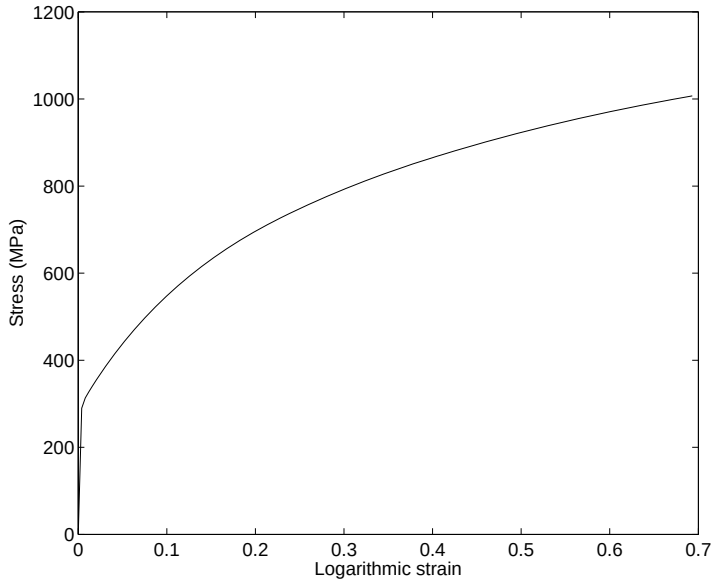

Figure 5: Stress/strain curve showing σ_{11} versus the logarithmic strain from straining one element to twice its length. The flow stress behaviour is representative for polycrystalline copper.

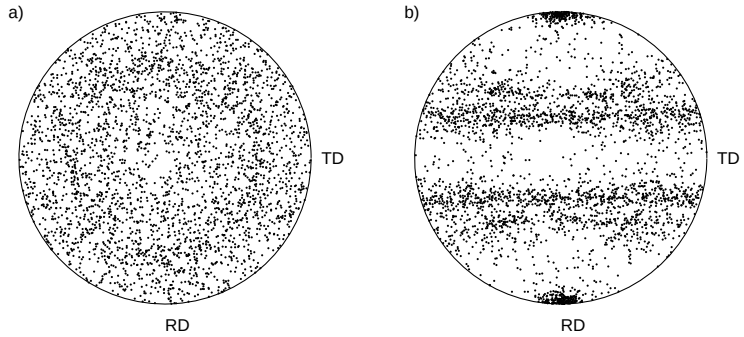

Figure 6: Pole figures showing the {111} poles of the 1000 grains in one of the integration points a) before and b) after deformation, at logarithmic strain of 0.7.

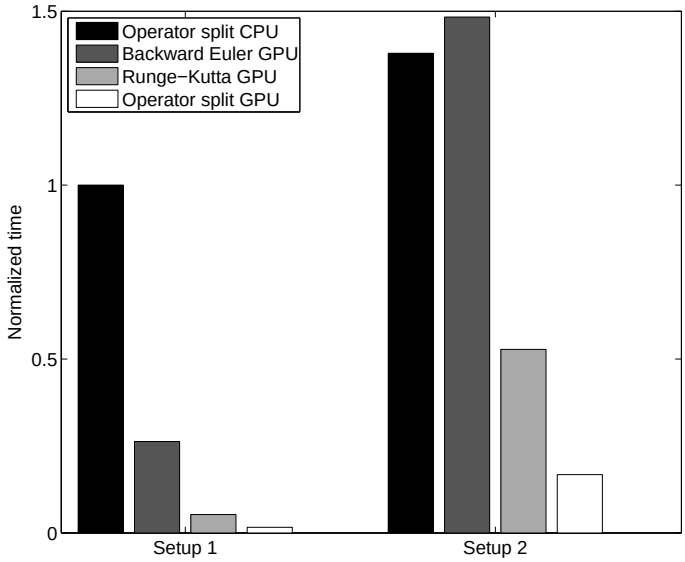

Figure 7: Comparison between the GPU and CPU implementations on different hardware with 1000 grains per integration point. Setup 1: Intel Xeon E5-2650 @2.0 GHz and Nvidia Tesla K20. Setup 2: Intel core i3-2310M @2.1 GHz and Nvidia GeForce GT 540M.

Figure 7 shows the computational time for running one brick element with 1000 grains per integration point on both of the considered hardware setups, comparing the best serial implementation, in this case the operator split, with the different GPU implementations. The time scale in this and the following figures in this section has been normalized with the time required for running 1000 grains on the CPU on the first setup. On the first setup the GPU implementation of the operator split method is more than 60 times faster than the CPU implementation, and even on the laptop an almost ten times speedup can be seen.

A comparison between the runtimes on hardware setup 1 for the different implementations when the number of grains per integration point is varied is shown in Figure 8. It clearly shows that for this testcase the backward Euler method is inferior to the other two, even though all three methods show that a considerable speedup can be achieved on the GPU. Figure 9 shows only the GPU implementations. Here it can be noted that the graphs do not exhibit a linear scaling, but rather displays a steplike behaviour. This has to do with the properties of the GPU; running one thread takes approximately the same time as running one thread per processor core. Thus performance will be optimal when the number of threads is evenly divisible with the number of available cores, while adding just one extra thread to this number will drastically increase the required time. This phenomenon is clearly seen in Figure 10 as well, where clear dips in the speedup appear when

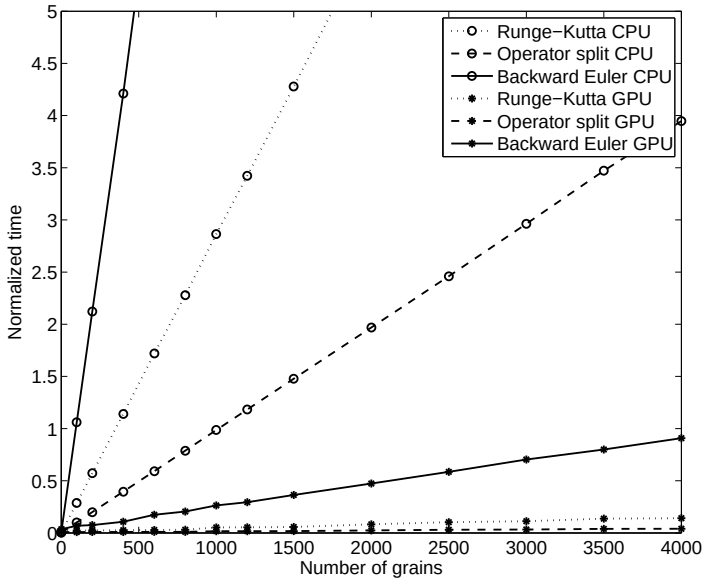

Figure 8: Comparison between GPU and CPU implementation of the different numerical schemes for different numbers of grains per iteration point.

the number of grains used is not optimal considering the hardware. Since the backward Euler method uses twice as many registers as the other methods, and consequently has a lower maximal occupancy, the dips are not in the same places for this implementation. For optimal numbers of grains, operator split and Runge-Kutta methods give a peak speedup of 100 and 80 times respectively, while the more memory intensive backward Euler comes close to 50 times speedup.

6.1.2 Implicit finite element formulation

The implicit test program uses the same constant displacement rate of 0.4 mm/s, but with the longer time step 0.01 s. With those settings the plastic material response is initiated in the first step. The response is identical to that obtained for the explicit calculations, the difference is in that now the tangent stiffness is calculated.

The Newton-Raphson scheme used for finding $\Delta\gamma^{\alpha}$ usually requires 5-6 iterations in the plastic region. The Runge-Kutta and operator split methods both require around 80 steps for the integration of one time step. Figure 11 shows the computational time for running one brick element with 1000 grains per integration point to an elongation of 100% on both of the considered hardware setups. The time scale in this and the following figures has been normalized with the time required for running 1000 grains on the CPU using the backward Euler scheme on the first setup, chosen as reference. The performance of

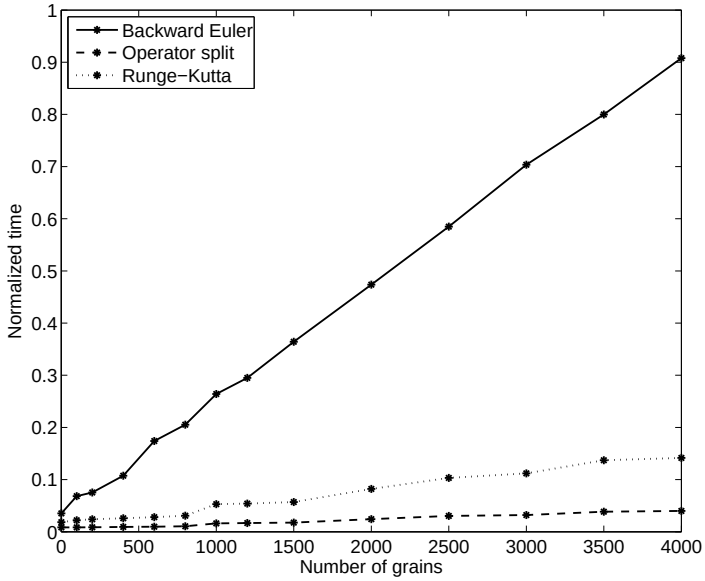

Figure 9: Comparison between the different numerical schemes implemented on the GPU for different numbers of grains per iteration point.

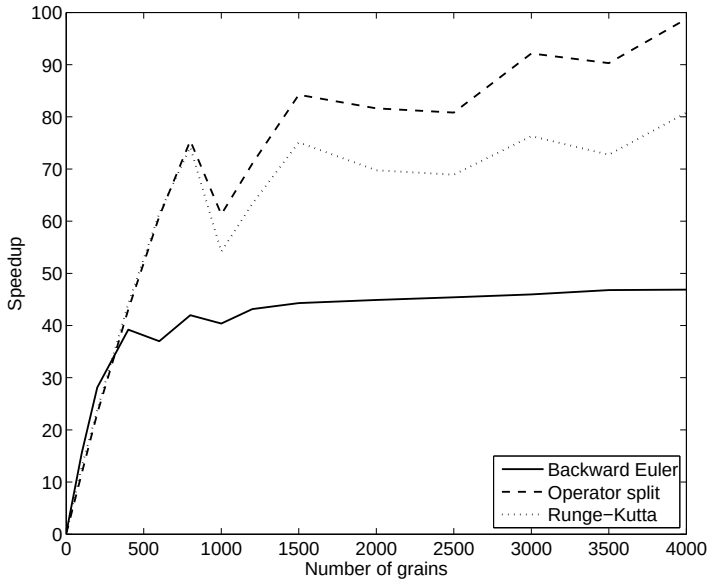

Figure 10: Speedup achieved with the GPU implementation compared to the CPU implementation for each of the three different numerical schemes, for different numbers of grains per iteration point.

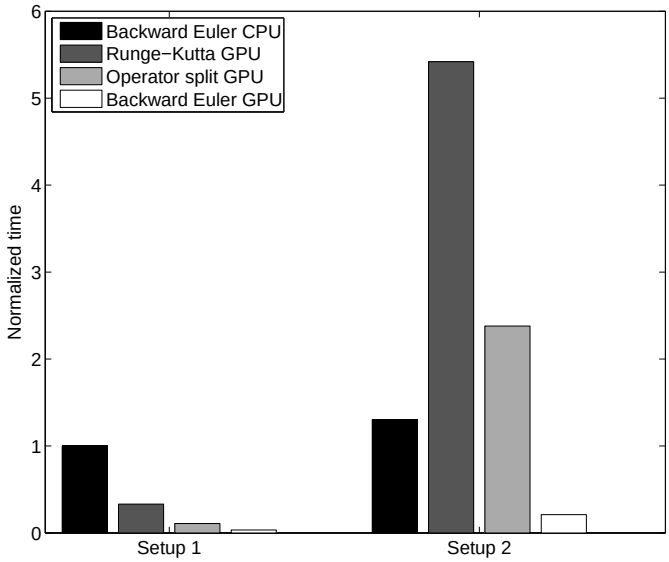

Figure 11: Comparison between the GPU and CPU implementations on different hardware with 1000 grains per integration point. Setup 1: Intel Xeon E5-2650 @2.0 GHz and Nvidia Tesla K20. Setup 2: Intel core i3-2310M @2.1 GHz and Nvidia GeForce GT 540M.

the CUDA implementations of the different numerical schemes is in Figure 11 compared to that of the serial CPU code for the backward Euler approach. With the first setup all GPU versions get enough speedup to yield improvements compared to the CPU performance, but on the laptop the explicit methods are unable to compete with the more stable method. Still, for the backward Euler it is possible, even on this modest hardware setup, to achieve a six times speedup.

On hardware setup 1, a typical integration step for one element, i.e. 8 integration points, with 1000 grains per integration point using the backward Euler approach takes about 19 ms on the GPU, divided in 14 ms for finding $\Delta\gamma^\alpha$ and 5 ms for calculating stresses and tangent stiffness. Using the CPU implementation the same step takes 670 ms, with 560 ms required for finding $\Delta\gamma^\alpha$ and 110 ms for calculating stresses and tangent stiffness. The comparatively longer time required for calculating the tangent stiffness on the GPU is due to the larger memory requirements for this part of the algorithm which makes it less suited for GPU implementation.

A comparison of how the computation time changes when changing the number of grains per integration point for the different CPU and GPU implementations is shown in Figure 12. As expected the time required for the CPU implementations scales linearly. The GPU implementation shows a step-like behavior, as in the case with explicit dynamics, which

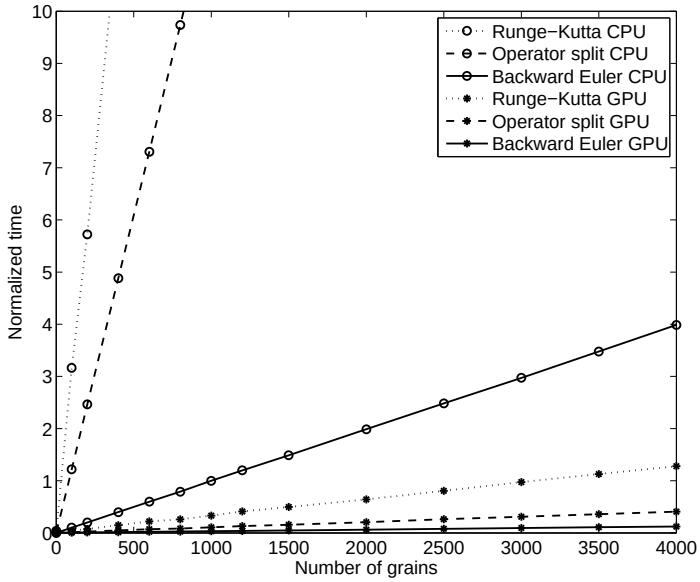

Figure 12: Comparison between GPU and CPU implementation of the different numerical schemes for different numbers of grains per iteration point.

can be seen more clearly in Figure 13. Figure 14 shows the speedup achieved from running each of the numerical methods on the GPU as compared to using the same scheme on the CPU. The graph shows that although the implicit method does result in a speedup factor of more than 30, it is less suited for GPU implementation than the other method where the operator split method, specially tailored for GPU implementation, gives a 120 times speedup. However, the stiff problem makes the more stable implicit method the overall winner, as is obvious from Figure 13, in spite of it not being the most optimal one for GPU implementation. In Figure 14 it can also be noted that the Runge-Kutta and operator split methods reach maximal efficiency for lower numbers of grains than the backward Euler method. This is due to the fact that those methods use seven threads per grain in order to calculate the numerical derivatives necessary for the tangent stiffness, which means that fewer grains are needed to get enough threads to saturate the GPU.

6.2 Simulations based on a 3D geometry

Since it is important to confirm that the speedup due to parallelization remains the same also for a larger FE structure, this has been tested as well, using the backward Euler method which was evaluated to be the most favorable for implicit finite element calculations. The geometry used consists of a standard plate with a circular hole at the center, shown in Figure 15. Due to symmetry only one eight of the structure is modeled and it is discretized

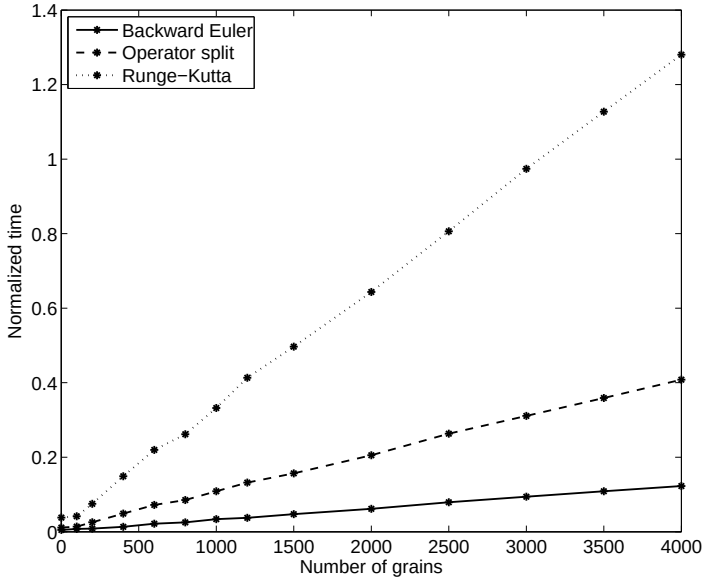

Figure 13: Comparison between the different numerical schemes implemented on the GPU for different numbers of grains per iteration point.

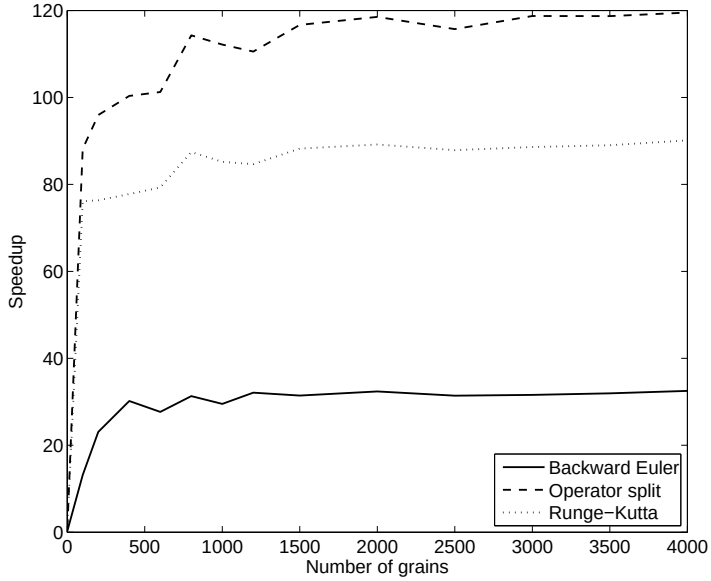

Figure 14: Speedup achieved with the GPU implementation compared to the CPU implementation for each of the three different numerical schemes, for different numbers of grains per iteration point.

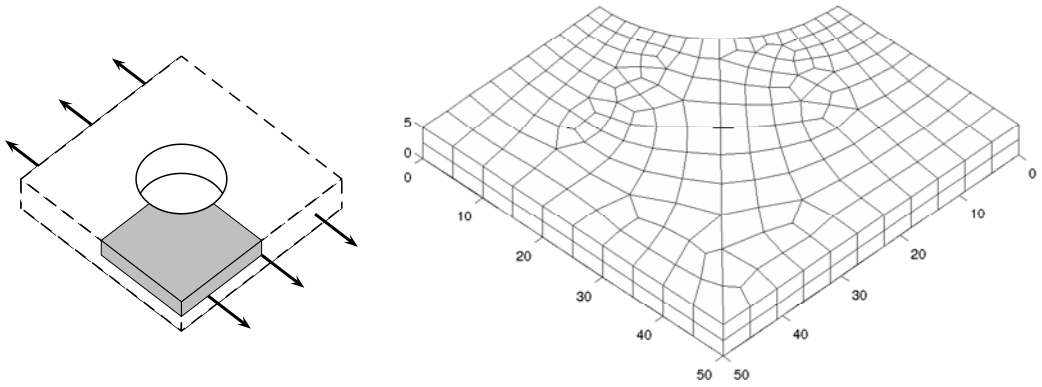

Figure 15: Plot of the geometry used for testing a larger structure, with loading direction indicated by arrows. Taking advantage of symmetries, only one eight of the structure is actually modeled, indicated by the shaded region in the left figure and detailed in the right figure. The numbers are in millimeters.

using 370 brick elements. While the GPU solution makes it possible to run even larger structures, such an example would be unfeasible to use for comparison, since the time requirements for the CPU version would become too exhaustive. The chosen problem size is also large enough that all data can not be fitted into the GPU memory at the same time, which means that the costs related to memory transfers are visible already for a problem of this size. The structure is subjected to tensile deformation at a constant displacement rate of 2.5 mm/s using a time step of 0.01 s and for a sequence of 1000 loadsteps. Figure 16 shows the resulting stress distribution, plotted in the deformed geometry.

Figure 17 shows the computational time for running calculations on the structure with different numbers of grains per integration point. Due to the long run times required on the CPU, that implementation has only been tested for up to 800 grains per integration point. The corresponding execution time has been used for normalizing the time-scale on the vertical axis in Figure 17. The results show that the scaling behavior of the GPU implementation when the number of grains is increased remains the same as in the one-element case. The speedup is slightly less for the larger structure, as can be seen in Figure 18. This is because the problem becomes too large to keep all the required data in the memory on the GPU, which means that more time has to be spent on data transfer. However, the results confirm that even for a larger problem, the time required by the FEM part of the program remains negligible compared to the time required for the crystal plasticity part of the calculations.

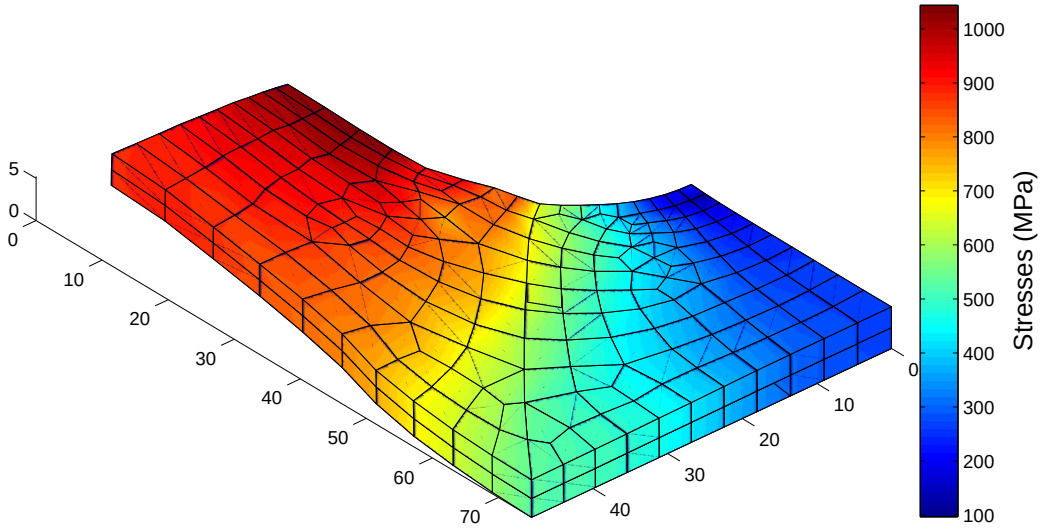

Figure 16: Resulting effective stress (von Mises) from running the plate with hole structure of 370 elements with 1000 grains per integration point. The dimensions of the geometry are given in millimeters. The stress distribution is shown in the deformed geometry.

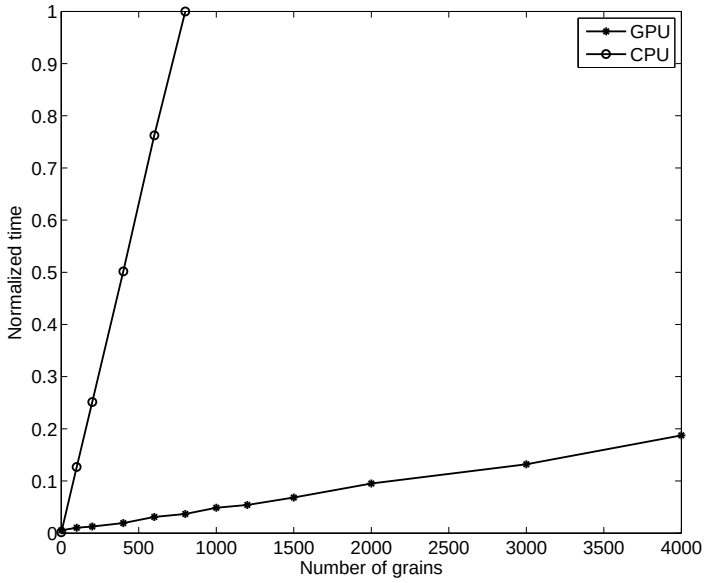

Figure 17: Comparison between GPU and CPU implementation using the backward Euler scheme when running a plate with hole structure of 370 elements.

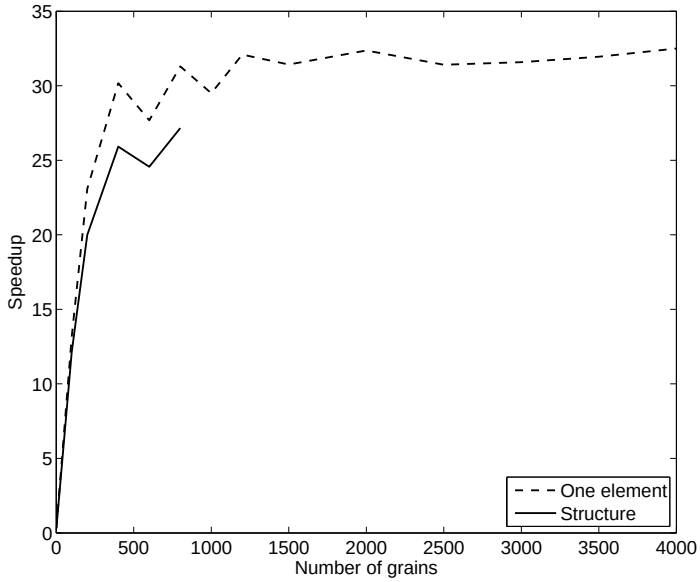

Figure 18: Speedup achieved with the GPU implementation using the backward Euler scheme for different numbers of grains per iteration point, for both the one element case and the larger structure.

7 Discussion/Conclusion

The present work shows the possibility for obtaining significant speedup of crystal plasticity simulations by taking advantage of the rapidly developing computational capacity of GPUs. Advantage has been taken of the large degree of parallelism inherent to the polycrystal plasticity model. Different numerical methods and strategies for their parallelization have been considered.

All comparisons in the present work have been made between a serial CPU and the GPU solution. The Xeon processor in hardware setup 1 has 8 processor cores and the i3 processor in hardware setup 2 has 2 processor cores, and both are capable of running 2 threads per core. This capacity could be exploited by using e.g. openMP parallelization on the CPUs. Although such CPU parallelization is not targeted here, it can be noted that the speedups of the GPU compared to the CPUs would theoretically be reduced by factors between 8 and 16 and between 2 and 4 respectively, depending on the hardware setup and assuming ideal CPU parallelization. Results from CPU parallelization of crystal plasticity implementations can be found e.g. in [20, 37, 38, 39, 19].

It can be noted that an optimal implementation of the code should take full advantage of both GPU and CPU parallelization. This strategy has, however, not been pursued further

here since the GPU implementation is the primary focus. An optimal implementation should also exploit the possibility of running on multiple GPUs.

The results show that both the Runge-Kutta and the operator split methods that are introduced are well suited for GPU implementation, giving a large speedup compared to the CPU implementation of the same methods. The computation time was reduced by factors as large as 120. Comparing the Runge-Kutta implementation to other works on GPUs it can be noted that a speedup of about 20 for double precision calculations on a Tesla C1060 is reported in [40], while less than ten times speedup is achieved using a Fermi GPU in [41]; both speedups are calculated in comparison to a serial CPU implementation. Considering that the Tesla K20 is a better card than the ones used in [40, 41], but also that the crystal plasticity model yields a much larger set of ODEs than that investigated in [40], the results in the present work are found to be reasonable.

The more memory demanding backward Euler method yields a smaller, but still significant, speedup. However, since the problem is ill-conditioned, the stability of the implicit method still makes it the most efficient method when running an implicit finite element program with long time steps. For an explicit approach with shorter time steps the operator split approach introduced in this work is proven to be both the most efficient method and the one best suited for GPU implementation.

While the limited amount of on-chip memory remains the dominant bottleneck in the calculations, a large number of threads allows the scheduler to hide much of the memory latency by switching between warps. The parallelization strategy has been chosen such that maximal speedup is achieved for the number of grains usually used in crystal plasticity simulations.

Considering the availability of GPUs, parallelization on GPUs makes it feasible to run crystal plasticity simulations on a desktop computer, rather than requiring large CPU-based clusters. Although it is preferable to use a graphics card especially suited for scientific calculation, the suggested approach reduces the computation time significantly even when running the simulations on an ordinary laptop.

Code

Source code for the CUDA implementation of the backward Euler solution has been made public at [42].

Acknowledgement

The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at Lunarc. Ola Olsson, Viktor Kämpe, and Markus Billeter, at the Computer Graphics Research Group at Chalmers University of Technology, are

acknowledged for assistance concerning optimization of the code.

Y. Mellbin and H. Hallberg gratefully acknowledges funding from the Crafoord foundation under grant number 20130667.

References

[1] G.I. Taylor. Plastic Strain in Metals. *Journal of the Institute of Metals*, 62:307–324, 1938.

[2] R. Hill. Continuum micro-mechanics of elastoplastic polycrystals. *Journal of Mechanics Physics of Solids*, 13:89–101, April 1965.

[3] J. R. Rice. Inelastic constitutive relations for solids: an internal-varible theory and its applications to metal plasticity. *Journal of the Mechanics and Physics of Solids*, 19:433–455, 1971.

[4] J.W. Hutchinson. Bounds and self-consistent estimates for creep of polycrystalline materials. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 348(1652):101–127, 1976.

[5] D. Peirce, R.J. Asaro, and A. Needleman. Material rate dependence and localized deformation in crystalline solids. *Acta Metallurgica*, 31(12):1951–1976, 1983.

[6] R.J. Asaro and A. Needleman. Texture development and strain hardening in rate dependent polycrystals. *Acta Metallurgica et Materialia*, 33(6):923–953, 1985.

[7] 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.

[8] J.L. Raphanel, G. Ravichandran, and Y.M. Leroy. Three-dimensional rate-dependent crystal plasticity based on runge-kutta algorithms for update and consistent linearization. *International Journal of Solids and Structures*, 41(22-23):5995–6021, 2004.

[9] X. Ling, M.F. Horstemeyer, and G.P. Potirniche. On the numerical implementation of 3d rate-dependent single crystal plasticity formulations. *International Journal for Numerical Methods in Engineering*, 63(4):548–568, 2005.

[10] M. Schmidt-Baldassari. Numerical concepts for rate-independent single crystal plasticity. *Computer Methods in Applied Mechanics and Engineering*, 192(11–12):1261–1280, 2003.

[11] A. Yamanaka, T. Takaki, and Y. Tomita. Simulation of Austenite-to-ferrite Transformation in Deformed Austenite by Crystal Plasticity Finite Element Method and Multi-phase-field Method. *ISIJ International*, 52(4, SI):659–668, 2012.

[12] J.A. Anderson, C.D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

[13] J.A. Baker and J.D. Hirst. Molecular dynamics simulations using graphics processing units. *Molecular Informatics*, 30(6-7):498–504, 2011.

[14] M. Liebmann, C.C. Douglas, G. Haase, and Z. HorvaÌĄth. Large scale simulations of the euler equations on gpu clusters. In *2010 Ninth International Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES)*, pages 50–54, 2010.

[15] A. Corrigan, F.F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids*, 66:221–229, May 2011.

[16] F. Mossaiby, R. Rossi, P. Dadvand, and S. Idelsohn. OpenCL-based implementation of an unstructured edge-based finite element convection-diffusion solver on graphics hardware. *International Journal for Numerical Methods in Engineering*, 89(13):1635–1651, 2012.
[17] C. Cecka, A.J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85(5):640–669, 2011.
[18] A. Dziekonski, P. Sypek, A. Lamecki, and M. Mrozowski. Generation of large finite-element matrices on multiple graphics processors. *International Journal for Numerical Methods in Engineering*, 94(2):204–220, 2013.
[19] G. Cailletaud, O. Diard, F. Feyel, and S. Forest. Computational crystal plastiticy : From single crystal to homogenized polycrystals. *Technische Mechanik*, 23:130–145, 2003.
[20] K. Inal, K. W. Neale, and P. D. Wu. Parallel computing techniques for metal plasticity applications. In *Parallel Computing Techniques for Metal Plasticity Applications*. The 17th annual international symposium on high performance computing systems and applications, 2003.
[21] G. Sarma, T. Zacharia, and D. Miles. Using high performance fortran for parallel programming. *Computers & Mathematics with Applications*, 35(12):41 – 57, 1998.
[22] NVIDIA Corporation. *CUDA C Programming Guide 6.0*, 2014.
[23] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.
[24] NVIDIA Corporation. *CUDA C Best Practice Guide 6.0*, 2014.
[25] L. Anand. Single-crystal elasto-viscoplasticity: applications to texture evolution in polycrystalline metals at large strain. *Computer Methods in Applied Mechanics and Engineering*, 193:5359–5383, 2004.
[26] E. Kröner. Allgemeine Kontinuumstheorie der Versetzungen und Eigenspannungen. *Archive for Rational Mechanics and Analysis*, 4(1):273–334, January 1959.
[27] E.H. Lee. Elastic-plastic deformation at finite strains. *Journal of Applied Mechanics*, 36:1–6, 1969.
[28] L. Méric, P. Poubanne, and G. Cailletaud. Single crystal modeling for structural calculations: Part 1 - Model presentation. *Journal of Engineering Materials and Technolog*, 113(1):162–1701, 1991.
[29] E.P. Busso and G. Cailletaud. On the selection of active slip systems in crystal plasticity. *International Journal of Plasticity*, 21(11):2212 – 2231, 2005.
[30] P. Håkansson, M. Wallin, and M. Ristinmaa. Prediction of stored energy in polycrystalline materials during cyclic loading. *International Journal of Solids and Structures*, 45(6):1570–1586, 2008.
[31] L. Anand and M. Kothari. A computational procedure for rate-independent crystal plastcity. *Journal of the Mechanics and Physics of Solids*, 44(4):525–558, 1996.
[32] S.R. Kalidindi. Incorporation of deformation twinning in crystal plasticity models. *Journal of the Mechanics and Physics of Solids*, 46(2):267–290, 1998.
[33] P.D. Wu, K.W. Neale, E. van der Giessen, M. Jain, A. Makinde, and S.R. MacEwen. Crystal plasticity forming limit diagram analysis of rolled aluminum sheets. *Metallurgical and Materials Transactions*, 29A:527–535, 1998.
[34] Z. Zhao, W. Mao, F. Roters, and D. Raabe. A texture optimization study for minimum earing in aluminum by use of a texture component crystal plasticity finite element method. *Acta Materialia*, 52:1003–1012, 2004.

[35] M. Wallin and M. Ristinmaa. Accurate stress updating algorithm based on constant strain rate assumption. *Computer Methods in Applied Mechanics and Engineering*, 190(42):5583–5601, 2001.

[36] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[37] K. Inal, P.D. Wu, and K.W. Neale. Instability and localized deformation in polycrystalline solids under plane-strain tension. *International Journal of Solids and Structures*, 39(4):983–1002, 2002.

[38] N.J. Sørensen and B.S. Andersen. A parallel finite element method for the analysis of crystalline solids. *Computer Methods in Applied Mechanics and Engineering*, 132(3–4):345–357, 1996.

[39] H. Li and H. Yang. An efficient parallel-operational explicit algorithm for taylor-type model of rate dependent crystal plasticity. *Computational Materials Science*, 54(0):255–265, 2012.

[40] L. Murray. GPU acceleration of Runge-Kutta integrators. *Parallel and Distributed Systems, IEEE Transactions on*, 23(1):94–101, Jan 2012.

[41] V.M. Garcia, A. Liberos, A. M. Climent, A. Vidal, J. Millet, and A. Gonzalez. An adaptive step size gpu ode solver for simulating the electric cardiac activity. In *Computing in Cardiology, 2011*, pages 233–236, Sept 2011.

[42] Source code. http://www.solid.lth.se/resources/code/. Accessed: 2014-08-12.