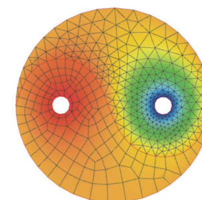
Publishing House
AKAPIT

COMPUTER METHODS IN MATERIALS SCIENCE

Informatyka w Technologii Materiałów

Vol. 16, 2016, No. 3



GPU IMPLEMENTATION OF KAMPMANN-WAGNER NUMERICAL PRECIPITATION MODELS

Øyvind Jensen*[1], Henrik Lam[2], Hallvard G. Fjær[1]

[1] Institute for Energy Technology, N-2027 Kjeller, Norway
[2] Impetus AFEA, SE-14160 Huddinge, Sweden
**Corresponding author: oyvind.jensen@ife.no*

Abstract

A Kampmann-Wagner type numerical precipitation model (KWN) has been implemented using NVIDIA's CUDA framework for numerical programming of the graphics processing unit (GPU). Different implementation strategies are discussed and subjected to performance measurements. We study two representative cases corresponding to a large and a small workload. The model is found to be well suited for a GPU implementation, provided that there is enough work to keep the device busy and the right parallelization strategy is chosen. For our hardware, we recommend a minimum work load of more than 2^{16} histogram bins (as the total of multiple histograms) which corresponds to 146 histogram bins per GPU core. When the KWN model is used in combination with other calculations that are processed by the CPU, the performance improvements can be such that the KWN model incurs only *negligible* additional execution time. Also if the KWN model is used standalone for a large case, the GPU implementation achieves good scalability and performance.

Key words: GPU, precipitation, numerical modeling, KWN-model

1. INTRODUCTION

This paper addresses the use of a computer's graphical processing unit (GPU) for efficient computer implementations of precipitation models that are based on Kampmann and Wagner's N-model (KWN) for diffusion controlled growth (Wagner & Kampmann, 1991). Using the CUDA-framework of NVIDIA, we implemented a GPU version of the KWN model for our simulation software, WeldSim (Myhr et al., 1998; Fjær et al., 2001; Myhr et al., 2002). A typical thermo-mechanical simulation of the heat treatment of aluminium parts, will often take several days using our optimized OpenMP implementation of the KWN model. By delegating the KWN model to the GPU, we experienced a four-fold speed up compared to the OpenMP implementation running with 4 threads. In this paper we describe how we achieved this speedup, and analyze the KWN model from the perspective of GPU-programming.

The KWN approach keeps track of the size distribution as well as the total number density of precipitates in the alloy. The model accounts for important features of age hardening materials, such as the dependence on the thermal history as well as the consumption of alloying elements that are stored in the precipitates during growth and released by dissipation. By specification of the relevant nucleation and growth kinetics, the model can, in principle, be used with any alloy that has some kind of precipitate. From the particle size distribution one derives macroscopic material properties of the material.

KWN-type models have been used successfully to model a wide range of age hardening materials. Of the aluminium alloys, the 6000 series (Myhr et al., 2002; Simar et al., 2007; Bahrami et al., 2012; Wu & Ferguson, 2009), the 7000 series (Nicolas &

 ISSN 1641-8581

Deschamps, 2003), and the 2000 series (Khan et al., 2008), as well as Al-Sc (Robson et al., 2003) and Al-Mg-Sc alloys (Fazeli et al., 2008) have been studied. Applications have also been reported for iron alloys, such as Fe-Cu (Yang & Enomoto, 2005), Fe-C-Mn (Öhlund et al., 2014) and Fe–C–Mn–Ti (Öhlund et al., 2015). Material specific characteristics of the growth mechanism can easily be added, such as the effect of excess vacancies on the diffusivity studied by Fazeli et al. (2008). The approach has also been adapted for multiple metastable phases and multiple alloy elements by den Ouden et al. (2013).

This paper is structured as follows: In section 2 we briefly outline the characteristics of Kampmann-Wagner type precipitation models with a focus on the computational structure. Section 3 provides a brief background for readers that are not familiar with GPU-programming, and in section 4, we analyze the KWN model and present strategies for an efficient parallelization. The numerical experiments are described in Section 5, and the performance results are presented and discussed in section 6.

2. THE SIZE CLASS APPROACH TO PRECIPITATION MODELING

Because we will analyze the KWN-type models from the perspective of parallel computations, it is useful to describe the central framework common to KWN-type precipitation models. We focus on the computational aspects, and do not aim to provide a description of the physics. On the contrary, we try to avoid material- and process-specific details and refer the interested reader to the relevant literature. (Wagner & Kampmann, 1991; Simar et al., 2012; Myhr & Grong, 2000; Myhr et al., 2001).

The central idea to KWN-type precipitation models is to keep track of a size distribution of precipitates in an alloy. The time evolution of this size distribution is calculated using assumptions about the stoichiometry of the precipitates, as well as thermodynamic properties and diffusion kinetics. The specific particle size distribution is then used to estimate macroscopic properties of the material, such as the yield stress, the hardness value (HV), and the work hardening rate.

2.1. The particle size distribution

If $r > 0$ denotes a characteristic size, the distribution $\phi(r) \geq 0$ denotes the number density per volume and size unit of precipitates of that size. The total number of precipitates per volume at a material point is then

$$N_{tot} = \int_0^\infty \phi(r)dr\,. \tag{1}$$

For spherical particles r corresponds to the radius, but other interpretations are possible. The mean size of the precipitates is given by the statistical expectation value:

$$\overline{R} = \frac{1}{N_{tot}} \int_0^\infty r\,\phi(r)dr\,. \tag{2}$$

Other quantities, such as the total precipitate-matrix interface area or the volume fraction of the secondary phase, are readily calculated. For example, the volume fraction of precipitates is given by

$$f = \int_0^\infty V(r)\phi(r)dr\,. \tag{3}$$

Here, $V(r)$ expresses the volume of a precipitate of size r, so that for spherical precipitates one would use $V(r) = 4\pi r^3/3$.

When some amount of solute element *i* is tied up in the precipitated phase, the remaining concentration in solid solution, C_{ss}^i, is reduced according to the expression:

$$C_{ss}^i = \frac{C_0^i - f C_p^i}{1-f}\,. \tag{4}$$

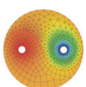
Here, C_0^i denotes the initial solute concentration in the matrix, and C_p^i is the concentration of solute *i* in the precipitate. In the following, we will drop the superscript (*i*) for aesthetic reasons, but urge the reader to keep in mind that the formalism is not restricted to alloys with only a single solute element.

The relations (1) – (4) all provide points of contact between the microscopic KWN-model and some macroscopic model. Information is aggregated from the precipitation model and made available as macroscopic quantities. Communication in the other direction is implemented in the equations that govern the time evolution of the particle size distribution.

2.2. Time evolution

As the precipitates grow or shrink, the number of particles is conserved, except for the particles that are dissolved at $r = r_{min}$ or nucleated slightly above a critical radius r^*. This is expressed with the number conserving equation,

$$\frac{\partial\phi}{\partial t} = -\frac{\partial(\phi v)}{\partial r} + j\,, \qquad r > r_{min}\,. \tag{5}$$



Here, $j = j(r,t)$ is the nucleation rate, and $v = v(r,t)$ is the growth rate of precipitates of size r. The product ϕv represents the flux of particles along the axis r, so that $\phi v|_{r=r'}$ would be the number of particles that pass through an imaginary boundary at r' per unit of time. The source term, j, is typically calculated using classical nucleation theory, and depends (at least) on the temperature T and the level of supersaturation of solute elements as macroscopic input parameters.

Let $v(r)$ denote the growth rate for particles of size r at a given time. In classical KWN approaches, we assume that the precipitate growth is limited by the diffusion of a particular solute element, and the instantaneous growth rate can be calculated as

$$v(r) = \frac{D}{r}\frac{C_{ss}-C_{int}}{C_p-C_{int}}. \tag{6}$$

Here, all quantities refer to the growth limiting solute element: D is the diffusion constant, and C_{int} is the concentration at the particle/matrix interface. We note that since $C_p > C_{int}$, the sign of v depends only on the difference $C_{ss} - C_{int}$.

In general, C_{int} must be determined from the Gibbs-Thomson equation (GT), which states how the solubility limit of the solute element at the particle/matrix boundary depends on the curvature of the particle. Although other approximations may sometimes be preferable (den Ouden et al., 2013; Perez, 2005), the GT correction for spherical particles in a binary or quasi-binary alloy is often expressed in the form

$$C_{int} = C_e \exp\left(\frac{\alpha}{r}\right). \tag{7}$$

Here, α encapsulates information about the solute, the solvent and the temperature. The planar equilibrium concentration (for $r = \infty$) is denoted by C_e. Returning focus to equation (6), we see that for small enough r, the growth rate will be negative because of the GT-effect. The size for which a precipitate neither grows nor shrinks, is precisely the critical radius r^*.

2.3. Discretization of the number conservation equation

The discretized form of the particle size distribution is a histogram. By partitioning the r-axis into several intervals we can define the number density of each "size class" i by,

$$N_i = \int_{r_i-\Delta r_i/2}^{r_i+\Delta r_i/2} \phi(r)dr. \tag{8}$$

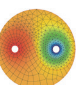
The height of the histogram bars is $\phi_i = N_i/\Delta r_i$, where Δr_i is the width of size class i.

A computationally efficient and numerically robust implementation of the KWN model was presented by Myhr and Grong (2000). They considered the size classes as control volumes, and solved the number conservation equation 5 for each control volume. All control volumes were coupled with the neighboring control volumes, corresponding to slightly larger/smaller precipitate sizes. This results in a tridiagonal system of equations. Numerical robustness and accuracy was achieved with an upwind scheme (Patankar, 1980), which adapts the numerical operations dynamically to the direction of information in the running simulation. Specifically, the sign of the growth rate determines if the equation for a particular size class include terms that depend on the neighboring size class, or if it is the other way around. The procedure is described thoroughly by Myhr and Grong (2000).

3. CHARACTERISTICS OF GPU PROGRAMMING

For readers that are unfamiliar with GPU programming, we provide a brief introduction to some of the central concepts. The reason that the GPU can outperform a contemporary CPU for certain computational workloads, is that the hardware is structured differently. A traditional CPU usually have a few cores, that can launch threads that are good at all-round instructions. These threads run fast, but due to the low number of cores, the speedup gained by using multiple threads is limited. Conversely, a GPU can have thousands of cores that launch light-weight threads. These threads are not as fast as the CPU counterpart, but they provide efficient parallelization by weight of numbers.

In order to utilize the GPU efficiently, the programmer needs to explicitly organize data into different categories of memory, and use each type of memory hardware in ways that benefit from the memory characteristics. In increasing order of speed, and decreasing order of capacity, the main memory categories are: global memory, shared memory, and on-chip registers. Threads running on the GPU are grouped into *blocks* that execute independently from each other. The threads within a block run logically in parallel, and are able to synchronize, and to communicate efficiently via shared memory. Of all threads in a block, batches of 32 threads (a *warp*) will execute physically in parallel.



The GPU contains several Streaming Multiprocessors (SM) that execute blocks from a job queue. Depending on the implemented algorithm, the code that runs on the GPU may need a low or a high number of on-chip registers for each thread. Hence, the amount of register memory of each SM will often limit the maximum number of threads in a block. The NVIDIA GPU architecture is based on the Single Instruction, Multiple Thread (SIMT) execution model. The advantage of SIMT is its capability to circumvent memory-access latency. Writing and reading from global memory can be costly and SIMT can hide the latency by letting the SM quickly switch to another block. However, SIMT cannot handle divergent branches within warps, and must resort to serialized execution of the different branches.

The efficiency of memory transfer to/from global memory depends strongly on the memory access pattern. To achieve good performance it may be necessary to organize data in a certain way. The fastest memory transfer to/from global memory is achieved when consecutive threads access consecutive memory locations (coalesced access). For overall performance, it is important to balance the time spent reading or writing to global memory with the time spent on calculations.

Finally, we mention that GPU hardware has evolved rapidly. Newer graphics cards have more functionality than older cards, and the relative performance of specific operations differ. The CUDA programmer can account for this by querying the Compute Capability (CC) of the installed GPU, and execute code that is optimized for the detected CC.

4. STRATEGIES FOR PARALLELIZATION

If the KWN precipitation model is used together with a macroscopic model, there may be an immediate, high level parallelization in that the CPU and the GPU can run simultaneously. This is only possible if the macroscopic model and the KWN model are coupled unidirectionally, that is, if only one of the models require input from the other. This is an important factor behind the four-fold speed improvement we experienced for our software, WeldSim.

In the thermo-mechanical welding calculations, the KWN model requires input from the thermal calculation, but the output from the KWN-model is only needed for the mechanical time steps. Since the mechanical time step can be set much larger than the thermal time step in most welding simulations, $\Delta t_{therm.} \ll \Delta t_{mech.}$, it is only occasionally that the CPU needs to wait for output from the GPU. If also the computations on the GPU are fast enough to complete before the CPU has finished processing the next macroscopic time step, there will be only *negligible additional execution time* for a simulation with or without the KWN micro structure model. However, to achieve this level of performance in practical applications, it was necessary to carefully optimize the KWN model for the GPU.

If the KWN model is used to simulate the local micro structure at several locations on a macroscopic geometry, for example at each of the nodes of a finite element mesh, the precipitation at one position does not depend on the particle distributions elsewhere. This provides a straightforward path to parallelization, by simply running one separate instance of the KWN model for each node.

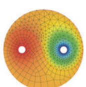
Parallelizing with OpenMP or MPI, we might expect good performance and scalability by this approach. However, for a GPU-implementation, this simple parallelization scheme does not work optimally, because of the limited resources available for each thread on GPU hardware. Fortunately, the KWN-type precipitation models are suitable for parallelization along several axes.

The time spent on the KWN-type precipitation model can be divided into a number of phases:

1. Nucleation and calculation of aggregate values that are necessary to determine the evolution of ϕ (preprocessing and postprocessing).
2. Setup of the tridiagonal matrix.
3. Solution of the tridiagonal system of equations.

All three phases are characterized by a computational complexity that scales linearly with the number of size classes and with the number of nodes. In our optimized OpenMP implementation (WeldSim), most of the execution time is spent in the phases 2 and 3. In phase 1 the time consuming task is the evaluation of integrals like equation 3. In phase 2 much computational resources are spent on the calculation of logarithms and exponential functions in order to determine the growth rate for each of the size classes. Phase 3 is solved using the Tri-Diagonal Matrix Algorithm (TDMA), which solves a three-diagonal system of equations by direct Gaussian elimination in two sweeps over the rows. It involves only elementary arithmetic operations on an array of floating point values.

Note that the aggregation phase could have been split up into pre- and post-processing phases, and this is how it is organized in our code. However, as



the pre- and post- calculations are very similar, we have chosen to simplify the description by combining the pre- and post-processing phases into a single aggregation phase. In the performance measurements, the aggregation phase is represented only by the preprocessing calculations.

The histograms and the equation systems must be stored in global memory on the GPU, but there is no need to copy this data back and forth to the host RAM during the simulation. The largest data transfers between the host computer and the device in each timestep are the small set of local input parameters and the aggregated output values for each node. Because of this, the time spent on data transfer between GPU and host RAM is negligible compared to the time spent processing the histograms.

Each phase of the KWN model is implemented in a separate *kernel* (function that execute on the GPU). We will now discuss to what extent the phases are suitable for parallelization on GPU hardware. The discussion will lead to a number of kernels that comprise three different parallelization strategies. Table 1 gives an overview of the strategies and the associated kernels. In section 5 these kernels will be used to measure and compare the performance of the parallelization strategies.

Table 1. *Overview of the kernels that implement the KWN phases for each of the parallelization strategies*

Phase	Node First	Class First	cuSparse
1	preprocess	preprocess	preprocess
2	setupNodeFirst	setupClassFirst	setupCuSparse
3	solveNodeFirst	solveClassFirst	cusparseSgtsvStrided Batch

4.1. The aggregation phase

The aggregation phase, which is represented by the *preprocess* kernel, consists essentially of elementary mathematical operations applied to consecutive memory arrays. In our OpenMP implementation, this does not take a significant portion of time. We implemented only the naive, node-based parallelization for this kernel, and we found that it was good enough for the needs of WeldSim. However, we don't expect very good performance of the preprocess kernel for cases with few nodes. On the other hand, the performance of the preprocess kernel relative to the other kernels provides useful insight on the merits of the naive parallelization scheme. The same kernel is used in all parallelization strategies.

For readers with a particular interest in cases with only a few simultaneous KWN models, we point out that a parallelized extraction of aggregate values from the particle size distribution can make use of well established parallel computing patterns. For example, the calculation of the total particle volume, equation 3, is readily formulated as a parallel transform-reduce operation acting on the particle size distribution histogram. As long as the histograms are large enough, we expect a high performance from such a GPU implementation.

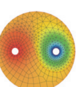
4.2. Setup of the tri-diagonal equation system

To define the equation system we must compute all coefficients of the tridiagonal matrix for each of the size classes and for each geometric node. This includes multiple independent evaluations of the growth rate as given by equations 6 and 7. To increase parallelism, we let each thread compute the coefficients for a single row in the matrix associated with a single mesh node.

Including the right hand side, each row of the tridiagonal equation system has four values, which fits into the structs *float4* or *double4* that are built-in language extensions in CUDA. The use of these types may result in faster memory transfer, compared to four separate float/double variables, due to the use of vectorized load/store instructions.

The calculation of coefficients for the equation system depends on values that are node specific, such as the concentration in solid solution and the local temperature. We reduce the number of load operations from global memory by letting several threads in a block compute coefficients for the same geometric node. The input values can then be loaded from global memory by a single thread, and made available to the block via shared memory. For NVIDIA devices with CC $\geq$ 2.0, reads from the same address in shared memory are broadcasted efficiently to all threads in the warp. This organization of work implies that consecutive threads will set up consecutive rows in the system of equations.

The phases 2 and 3 are tightly coupled, as the former creates the input to the latter. Since performance may depend strongly on the memory access pattern, the efficiency of one may come with the cost of a reduced efficiency of the other. It is therefore important to consider the performance of the two phases together. We investigated three different



kernels for the setup phase, and the differences between them will be clarified in the following sections.

4.3. Solution of the tridiagonal equation system

A sophisticated parallel algorithm for tridiagonal systems of equations is implemented in the cuSparse library. (Chang et al., 2012; Zhang et al., 2010) This solver is very fast, but requires significant amounts of temporary storage (NVIDIA Corporation, 2015). In comparison, the TDMA is unsuitable for a parallel implementation because the operations in one iteration depend on results from the previous iteration. On the other hand, TDMA does not require much working space memory. Moreover, for cases with a large number of geometric nodes, it is possible that the naive parallelization scheme applied to the TDMA can utilize the GPU sufficiently well, and thus achieve good performance.

In this work, we chose to investigate the cuSparse tridiagonal solver as well as two different implementations of the TDMA. Both TDMA kernels use, by necessity, the naive parallelization scheme where each thread applies the TDMA algorithm for a specific geometric node. This organization of work implies that consecutive threads will access rows of the equation system for consecutive geometry nodes.

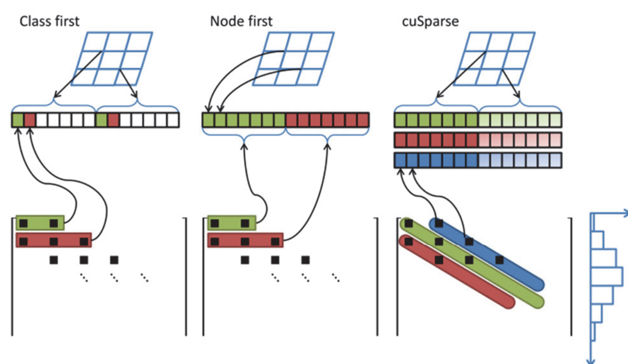


Fig. 1. Alternatives for the memory layout of the equation system. As depicted, the rows of the matrices can be associated with the size classes of the histograms.

4.4. Memory layout

To maximize the overall performance of the equation setup and the TDMA, two ways to organize the equation system data as a two-dimensional array was tested. As depicted in figure 1, the data can be stored with either consecutive size-classes (left) or consecutive node numbers (middle). With the "class first" layout the entire equation system for each node is stored in a continuous memory region. In an iteration over this data, the class index varies faster than the node index. Conversely, in the "node first" layout all equations that correspond to a certain size class are stored together, and the node index varies faster than the class index. For the cuSparse solver (right) each of the diagonals are supplied in a separate array, and multiple equation systems must be stored in the "class first" fashion. For each layout, the right hand sides of the equations are stored the same way as the matrix elements, but this is left out of the figure.

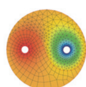
The faster memory access is achieved when consecutive threads access consecutive data addresses, as this allows coalesced memory operations. However, the chosen implementations of phases 2 and 3 imply that the organization of the equation system cannot be optimal for both phases simultaneously: The TDMA solver benefits from the "node first" memory layout, while the setup-kernel is more efficient with the "class first" layout. The optimal storage scheme for the TDMA approach must be determined based on empirical studies of the combined performance.

Table 1 summarize the parallelization strategies as a list of the specific kernel implementations that we will examine in the next section. For the setup and solution phases, we investigate three sets of kernels that comprise three different parallelization strategies:

1. Storing the equation system with consecutive node numbers. Kernels: *setupNodeFirst* and *solveNodeFirst*.
2. Storing the equation system with consecutive size classes. Kernels: *setupClassFirst* and *solveClassFirst*.
3. Storing the equation system in the format required by cuSparse. Kernels: *setupCuSparse* and *cusparseSgtsvStridedBatch*.

The cuSparse solver expects four vectors, one for each of the diagonals and one for the right hand side. The "strided batch" version of the solver accepts multiple equation systems. Using the terminology developed above, we may describe this storage format as a "class first" layout.

The histogram data and the tridiagonal equation system were stored as single precision floats (4 bytes), but intermediate calculations, like the growth rates for the size classes, were done using double precision (8 bytes). For all kernels we fixed the block size at 256 threads per block and used up to 56 blocks. These launch parameters imply four blocks per SM on our GPU, and maximizes the hardware



limit of 1024 threads per SM. Using the guided analysis system of NVIDIA Visual Profiler, we determined that these values provide a good balance with respect to latency, memory usage, and concurrency for large workloads.

5. PERFORMANCE MEASUREMENTS

The three parallelization strategies were tested and verified against our existing OpenMP-accelerated implementation. The performance testing was done on a computer with two Intel Xeon X5675 CPUs (3.07 GHz), 48 GB RAM, and an NVIDIA Tesla C2070 GPU with 6 GB memory. The GPU has 14 symmetric multiprocessors with 32 cores each, which makes a total of 448 cores. On the system, the CPU and the GPU are contemporaries. Error Correcting Codes (ECC) was enabled during performance measurements.

For each of the kernels listed in table 1, we perform the following steps:
1. Save the current time.
2. Launch multiple instances of the kernel asynchronously from within a for-loop. Each kernel will be executed as soon as the previous kernel finishes (in serial).
3. Wait for all the kernels to finish by calling "cudaDeviceSynchronize".
4. Query the current time and calculate the elapsed time.

We compiled an optimized executable that, for all kernels, performed this measurement procedure twice in rapid succession. This program was executed at least three times for each data point, and the minimum value of all measured execution times was collected as the result. This method provides an upper bound of the performance under ideal conditions.

To investigate how the performance of the kernels depend on the number of geometric nodes and the number of size classes, we varied both of these parameters and measured the execution time of the kernels. For a given number of nodes and size-classes, the number of kernel launches was adjusted so that the total computational work is held constant across all timing measurements. For the purpose of this analysis, we define the total computational work as

$$W = NCK. \tag{9}$$

Here, N is the number of nodes, C the number of size classes and K the number of kernel launches.

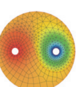
Keeping W constant allows a simple interpretation of the observed performance characteristics: A flexible kernel for general use should display a uniform, low execution time for a wide range of combinations of N, C and K. The performance of a given combination measures how much of the GPU's processing capability we are able to use, and how much is wasted due to latency and general overhead such as the instructions related to kernel launch and memory allocations.

The kernel launch and execution time can vary depending on the prior state of the GPU, and for small jobs this can lead to large variations in the measured execution time. Typically, the first in a series of rapid kernel launches will take slightly longer time, and this is commonly referred to as "warm up". In our measurements the warm up overhead is averaged out; however, in applications of the KWN model the kernels are typically executed without warm up. For this reason we checked how the performance depends on the number of kernel launches K, and we found that on our hardware, the warm up overhead is of the order 10-20 microseconds. Only for the tiniest configurations around $NC \approx 2^{10}$, is this overhead of any significance. (On the other hand, we will soon see that KWN models of that size are not relevant for GPU computations due to low performance.)

Two characteristic use cases of the KWN-model were considered: (1) a case with a large number of geometric nodes and (2) a case with few geometric nodes, including the special case with only a single node.

Case (1) could correspond to the post weld heat treatment of a work piece modeled with a fine FEM-mesh, or a welding simulation of a large work piece with long weld paths. A large portion of the geometry is in a relevant temperature range, and the simulation requires a large number of independent KWN-models. For this use case we measured the execution times of the total work load $W = 2^{29}$. The number of nodes varied from $N = 2^7$ to $N = 2^{17}$ and the number of size classes covered the range $C = 2^6$ to $C = 2^{17}$. Due to the memory limits of the GPU, the largest system we tested was 2 kernel launches with $NC = 2^{28}$. Our OpenMP implementation processes this work load in about one minute when running with a single thread.

Case (2) could arise in a tack welding process, where only a tiny part of the work piece reaches temperatures for which the KWN-model becomes relevant, or it could represent point models where



the simulation in a single point is assumed to be valid throughout a uniform material. For this case we fixed the total computational work at $W = 2^{23}$. N varied from 1 to 32, and C from 2^6 to 2^{18}. While this number of classes is extremely high for practical use of the KWN model, it is relevant in this context. It provides a reference point for the cuSparse tridiagonal solver, which is made for equation systems of even larger dimensions. The OpenMP implementation processes this work load in about one second when running with a single thread.

6. RESULTS

Figure 2 provides an overview of the combined performance of the kernels in each of the three parallelization strategies. For all strategies, we observe a high and relatively uniform utilization of the GPU for cases with $N \geq 2^{10}$ as long as the case is well within the memory constraints of the device. The "Class first" strategy has the worst performance in most of the configurations. For the cuSparse kernels,there is a visible dependence on the number of classes C, as the GPU utilization is better for higher C. This tendency applies even for the cases with low N, and this strategy performs reasonably well also in that region as long as C is large enough. The other two strategies do not exhibit any significant benefits from large C, and suffer severe penalties for all cases with $N < 2^{10}$.

Memory limits of the GPU prohibited the evaluation of the cuSparse solver for some cases with $NC > 2^{26}$. These configurations are colored black in the "cuSparse" pane of the figure. For work loads in this range there is no clear winner among the other two strategies. The "node first" strategy performs better in general, but receives a big penalty when N is large and we approach the memory limit of the device. Interestingly, the "class first" strategy displays a similar penalty at the memory limit when $C \geq 2^{16}$.

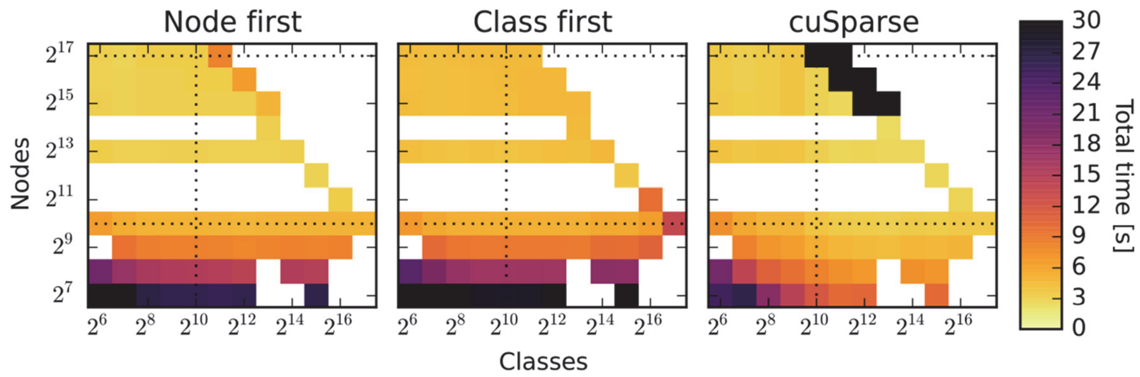


Fig. 2. Total execution time of case (1) for the strategies listed in table 1 with the color coded range capped at 30 s. The dotted lines indicate the combinations of nodes and classes that are examined more closely in figures 3–5.

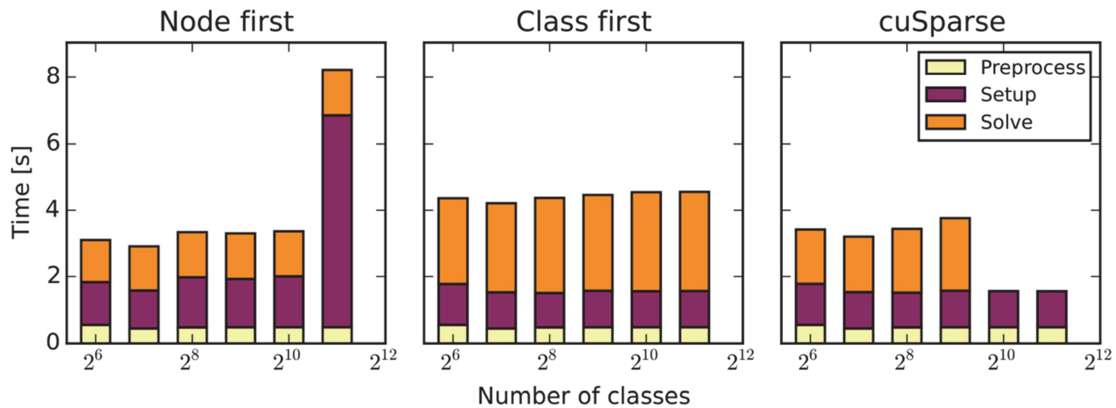
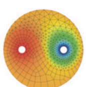


Fig. 3. Execution time of the individual kernels with $N = 2^{17}$ for case (1) with a total work load $W = 2^{29}$. This corresponds to the upper horizontal line in figure 2. The solver bars in the cuSparse pane are missing because the solver exited with an out of memory error.



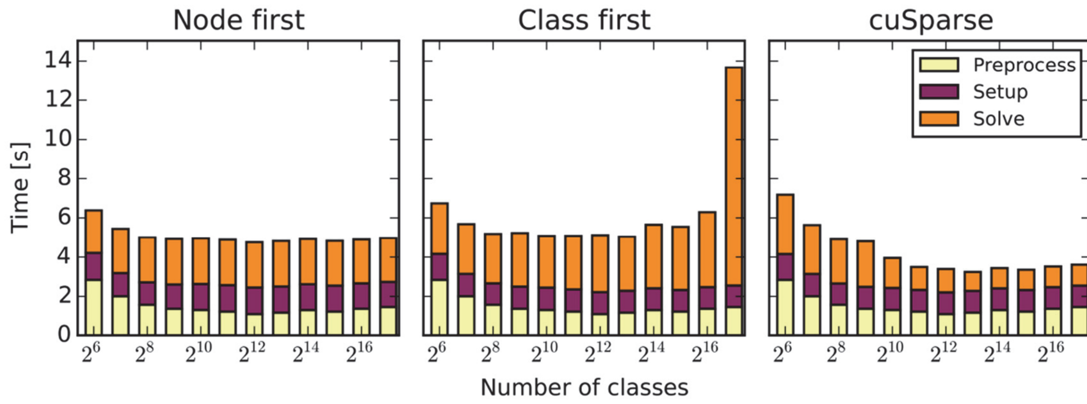


Fig. 4. Execution time of the individual kernels with $N = 2^{10}$ for case (1) with a total work load $W = 2^{29}$. This corresponds to the lower horizontal line in figure 2.

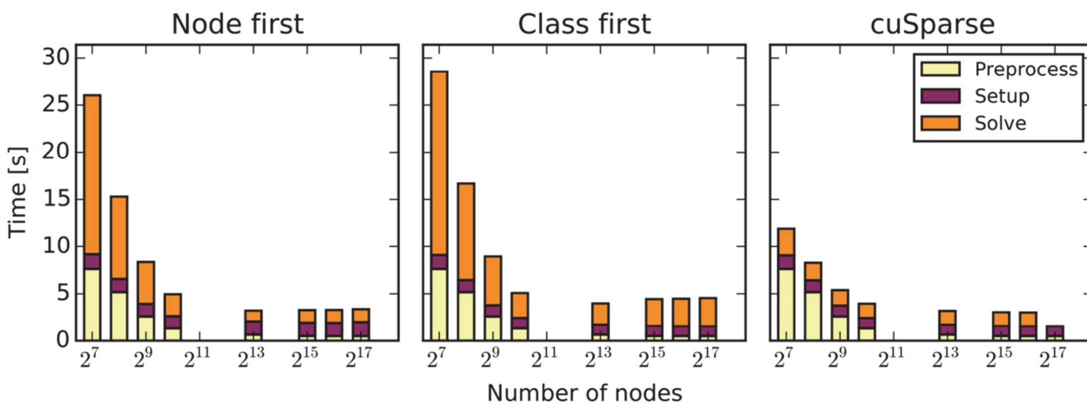


Fig. 5. Execution time of each kernel using a fixed number of classes, $C = 2^{10}$ for case (1) with a work load $W = 2^{29}$. This corresponds to the vertical line in figure 2. The high execution times for computations with few nodes reveal a low utilization rate of the GPU for the naive parallelization scheme.

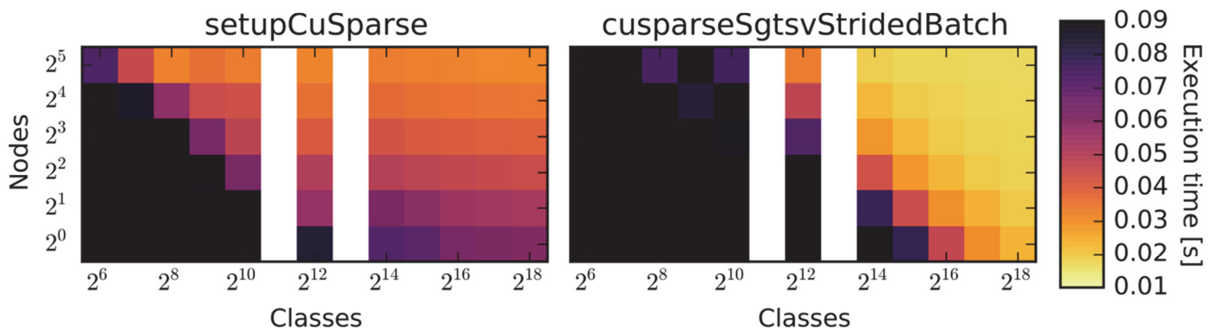
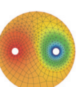


Fig. 6. Execution time of the kernels setupCuSparse and cusparseSgtsvStridedBatch for the small workload $W = 2^{23}$. The color coded range is capped at 0.09 s in order to better display the structure in the regions where the performance is within reasonable bounds.

In figures 3 and 4, we display the performance of the individual kernels for fixed numbers of nodes, corresponding to the horizontal lines in figure 2. We recognize the dramatic performance penalties for the "node first" strategy in figure 3 and for the "class first" strategy in figure 4. The penalties appear in different phases of the KWN-model for the two strategies: While the "node first" strategy has a bad performance in the setup kernel, the "class first" strategy suffers in the solver kernel.We have not been able to identify the exact mechanism behind the extreme performance drop in the kernels setupNodeFirst and solveClassFirst. However, a common trait of these kernels is that neighboring threads



need to access memory locations that are far apart, and the phenomenon seems to be triggered when the required amount of memory approaches the limitations of the device. For this reason, we believe that the issue is related to operations in global memory, and a possible explanation is that the simultaneous access of strongly diverging memory locations leads to more cache misses.

For the range of classes where the kernels have a stable, predictable performance, we see that the TDMA solvers benefit from the "node first" memory layout. Conversely, the setup kernels are more efficient with the "class first" layout. These observations are consistent with the discussion of coalesced memory access in section 4.

In figure 5 the number of size classes is kept constant and we see how the resource utilization of the individual kernels depend on the number of nodes. This corresponds to the vertical line in figure 2. Both the TDMA-based solvers and the preprocessing kernel perform poorly for the cases with few nodes. What these kernels have in common, is the naive parallelization scheme. When the hardware limits on maximum number of concurrent threads exceeds the number of nodes, parts of the GPU remain idle, and the throughput drops dramatically. The setup kernels and the cuSparse solver implement more sophisticated parallelization. These kernels are able to activate the entire GPU throughout the range of *N* displayed in the figure.

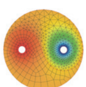
Having identified the limitations of the naively parallelized kernels, we expect them to perform poorly for case (2), where the number of nodes is even lower. For this work load, only the cuSparse based parallelization strategy seems viable, so we will now focus only on that strategy. As discussed in section 4, the implementation of a preprocessing kernel with better parallelization properties is straightforward. However, we will not follow that path in this work. Instead, we restrict the following discussion to the performance of the setup and solver kernels of the cuSparse based strategy.

As a reference value for the expected execution time of case (2), we use timing results corresponding to the best hardware utilization in case (1) and scale it down by a factor 2^{-6}. In figures 3 and 4 the fastest execution times for the cuSparse setup and solver are slightly more than 2 seconds. From this, we conclude that the total execution time both kernels will take at least 0.03125 s, and the GPU is utilized extremely well if the two kernels finish case (2) in such a short time. We choose the reference value 0.032 s for the following discussion, but do not expect to see this performance in practice. Even in the efficient regions of case (1) the utilization is sometimes lower than this reference value.

In figure 6, the execution time is displayed for case (2): simulations with few instances of the KWN model. For both kernels the triangular structure is striking. It reveals that for small workloads the performance of the cuSparse based strategy depends on the product *NC*, that is, on the total number of histogram bins. In the left pane the setup kernel displays an additional, almost linear improvement towards higher number of nodes. Since the axes of the plot are logarithmic, the linear trend in the plot is a logarithmic trend versus the node number *N*.

The best efficiency in figure 6 is found in the top right corner where the combined execution time is 0.048 s. This corresponds to an increase relative to the reference time by a factor of 1.5. The configurations with $NC = 2^{16}$ constitute an interesting diagonal in the right pane, for which the cuSparse solver alone exceeds the reference time by a factor of 1.5. For all workloads above this diagonal, $NC > 2^{16}$, the running time of the solver is less than the reference time. Here, most of the time is spent in the setup kernel, which, as displayed in the left pane, improves linearly with 2^N in the region. The running time as a multiple of the reference time varies approximately from 2 to 1 as *N* increases from 1 to 2^5.

In summary, for the triangle of configurations with $NC > 2^{16}$, $N \leq 2^5$, and $C \leq 2^{18}$, we see that the total running time of both kernels increases by a factor in the range of 1.5−3 relative to the reference time. A factor of 3 is a significant penalty, as it indicates a hardware utilization of only one third of the observed capacity in the larger case. Similarly, the factor of 1.5 means that the hardware delivers two thirds of the realistic performance. While this is a large performance drop, it is not unexpected, considering that the workload of case (2) is quite small compared to case (1). On the other hand, even with such reductions of performance, a GPU implementation of the KWN model may run faster than a corresponding CPU implementation, but this also depends on the available hardware. For $NC < 2^{16}$ the performance drop is dramatic, and the GPU implementation is unlikely to provide benefits over a CPU implementation. The diagonal at $NC = 2^{16}$ corresponds to about 4700 histogram bins per multiprocessor, or 146 per GPU core.



7. CONCLUSION

We have demonstrated that KWN-type models are well suited for parallelization on GPU hardware, provided that the total number of histogram bins is large enough, and that suitable algorithms are chosen. In particular, the benefits are great when the KWN microstructure model is coupled with a macroscopic model processed by the CPU that does not need the KWN output for every time step. In this situation, if the KWN model is efficient enough that it completes the time step equally fast as the macroscopic model, the microstructure can be included in the simulation with *negligible additional running time*.

To make the KWN fast enough, the cuSparse based strategy gives the best performance over a wide range of model configurations, including towards the limit of small models. The hardware utilization for small cases is reduced, and for our graphics card, we identified $NC > 2^{16}$ as a minimum model size if a GPU implementation is considered. This corresponds to about 146 histogram bins per GPU core. For smaller cases, the achieved performance deteriorates rapidly.

In the limit of large cases, memory limitations may prevent the use of the cuSparse solver, and the other two strategies are good alternatives. However, it may be important to examine their performance in configurations that approaches the memory limitations of the GPU. In our experiments the “node first” strategy performs better in most cases, but the setup kernel has a severe performance hit at the limit of the available device memory. The “class first” strategy has a more predictable performance in cases with a large number of nodes, but is generally slower due to the memory traffic pattern of the solver.

This work has been performed in the project AluCaW with support from the Norwegian Research Council, grant number 228466/O30, and with the following partners: Benteler Aluminium Systems Norway AS, Benteler Automotive Farsund AS, Impetus AFEA AS, Prediktor AS and the Institute for Energy Technology.

REFERENCES

Bahrami, A., Miroux, A., Sietsma, J., 2012, An age-hardening model for Al-Mg-Si alloys considering needle-shaped precipitates, *Metallurgical and Materials Transactions A*, 43, 4445-4453.

Chang, L.-W., Stratton, J. A., Kim, H.-S., Hwu, W.-M. W., 2012, A scalable, numerically stable, high-performance tridiagonal solver using GPUs, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Los Alamitos, CA, USA, 27:1-27:11.

Fazeli, F., Sinclair, C. W., Bastow, T., 2008, The role of excess vacancies on precipitation kinetics in an Al-Mg-Sc alloy, *Metallurgical and Materials Transactions* A, 39, 2297-2305.

Fjær, H. G., Myhr, O. R., Klokkehaug, S., Holm, E. J., 2001, Advances in aluminum weld simulations applying WeldSim, *Proceedings of the 11th Intl. Conf. on Computer Technology in Welding*, eds, Siewert, T. A., Pollock, C., Washington, 251-263.

Öhlund, C. E. I. C., den Ouden, D., Weidow, J., Thuvander, M., Offerman, S. E., 2015, Modelling the evolution of multiple hardening mechanisms during tempering of Fe–C–Mn–Ti martensite, *ISIJ International*, 55, 884-893.

Öhlund, C. E. I. C., Weidow, J., Thuvander, M., Offerman, S. E., 2014, Effect of Ti on evolution of microstructure and hardness of martensitic Fe-C-Mn steel during tempering, *ISIJ International*, 54, 2890-2899.

Khan, I. N., Starink, M. J., Yan, J. L., 2008, A model for precipitation kinetics and strengthening in Al–Cu–Mg alloys, *Materials Science and Engineering*: A, 472, 66-74.

Myhr, O. R., Grong, Ø., 2000, Modelling of non-isothermal transformations in alloys containing a particle distribution, *Acta Materialia*, 48, 1605-1615.

Myhr, O. R., Grong, Ø., Andersen, S. J., 2001, Modelling of the age hardening behaviour of Al-Mg-Si alloys, *Acta Materialia*, 49, 65-75.

Myhr, O. R., Grong, Ø., Klokkehaug, S., Fjær, H. G., 2002, Modelling of the microstructure and strength evolution during ageing and welding of Al-Mg-Si alloys, *Mathematical Modelling of Weld Phenomena* 6, ed, Cerjak, H., Graz, Austria, 337-364.

Myhr, O. R., Klokkehaug, S., Grong, Ø., Fjær, H. G., Kluken, A. O., 1998, Modeling of microstructure evolution, residual stresses and distortions in 6082-T6 aluminum weldments, *Welding Journal*, 77, 286S-292S.

Nicolas, M., Deschamps, A., 2003, Characterisation and modelling of precipitate evolution in an Al-Zn-Mg alloy during non-isothermal heat treatments, *Acta Materialia*, 51, 6077-6094.

Den Ouden, D., Zhao, L., Vuik, C., Sietsma, J., Vermolen, F. J., 2013, Modelling precipitate nucleation and growth with multiple precipitate species under isothermal conditions: Formulation and analysis, *Computational Materials Science*, 79, 933-943.

NVIDIA Corporation, 2015, *The API reference guide for cuSPARSE*, DU-06709-001_v7.5.

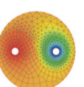
Patankar, S., 1980, *Numerical Heat Transfer and Fluid Flow*, Hemisphere Publ., Washington.

Perez, M., 2005, Gibbs-Thomson effects in phase transformations, *Scripta Materialia*, 52, 709-712.

Robson, J. D., Jones, M. J., Prangnell, P. B., 2003, Extension of the N-model to predict competing homogeneous and heterogeneous precipitation in Al-Sc alloys, *Acta Materialia*, 51, 1453-1468.

Simar, A., Bréchet, Y., de Meester, B., Denquin, A., Gallais, C., Pardoen, T., 2012, Integrated modeling of friction stir welding of 6xxx series Al alloys: Process, microstructure and properties, *Progress In Materials Science*, 57, 95-183.

Simar, A., Bréchet, Y., de Meester, B., Denquin, A., Pardoen, T., 2007, Sequential modeling of local precipitation,



strength and strain hardening in friction stir welds of an aluminum alloy 6005A-T6, *Acta Materialia*, 55, 6133-6143.

Wagner, R., Kampmann, R., 1991, Homogeneous second-phase precipitation, *Materials Science and Technology: Phase Transformations in Materials*, eds, Cahn, R. W., Haasen, P., Kramer, E. J., VCH, Weinheim, 213-302.

Wu, L., Ferguson, W. G., 2009, Modelling of precipitation hardening in casting aluminium alloys, *Materials Science Forum*, 618-619, 203-206.

Yang, J., Enomoto, M., 2005, Numerical simulation of copper precipitation during aging in deformed Fe-Cu alloys, *ISIJ International*, 45, 1335-1344.

Zhang, Y., Cohen, J., Owens, J. D., 2010, Fast tridiagonal solvers on the GPU, SIGPLAN Not., 45, 127-136.

IMPLEMENTACJA GPU NUMERYCZNEGO MODEL KAMPMANNA-WAGNERA DLA WYDZIELEŃ

Streszczenie

Model Kampmanna-Wagnera dla wydzieleń (KWN) został zaimplementowany za pomocą frameworku NVIDIA CUDA w numerycznym programie dla kart graficznych (GPU). W pracy przedyskutowano różne strategie implementacji i oceniono wydajność poszczególnych rozwiązań. Badano dwa reprezentatywne przypadki odpowiadające małemu i dużemu obciążeniu obliczeniowemu. Zauważono, że model jest odpowiedni dla implementacji GPU, zakładając że obciążenie jest wystarczające aby procesory były obciążone i że wybrana jest odpowiednia strategia zrównoleglenia. Dla urządzeń użytych w pracy zarekomendowano minimalne obciążenie 2^{16} histogramów czyli dyskretyzowanych cząstek (jako sumę wszystkich histogramów), co odpowiada 146 histogramów na jedną GPU. Kiedy model KWN jest połączony z innymi obliczeniami prowadzonymi na GPU, poprawa wydajności może być uzyskana dzięki temu że model KWN wykorzystuje tylko niewielką część czasu procesora. Ponadto, jeżeli model KWN jest wykorzystany oddzielnie dla dużego zadania, implementacja GPU osiąga dobrą skalowalność i wydajność.

Received: June 24, 2016
Received in a revised form: September 20, 2016
Accepted: November 9, 2016