Comput Visual Sci (2011) 14:249–256
DOI 10.1007/s00791-012-0180-1

A 2589 line topology optimization code written for the graphics card

Stephan Schmidt · Volker Schulz

Received: 20 May 2009 / Accepted: 31 March 2012 / Published online: 14 August 2012

Abstract We investigate topology optimization based on the solid isotropic material with penalization approach on compute unified device architecture enabled graphics cards in three dimensions. Linear elasticity is solved entirely on the GPU by a matrix-free conjugate gradient method using finite elements. Due to the unique requirements of the single instruction, multiple data stream processors, special attention is given to the procedural generation of matrix–vector products entirely on the graphics card. The GPU code is found to be extremely efficient, being faster than a 48 core shared memory CPU system. CPU and GPU implementations show different performance bottlenecks. The sources are available at http://www.mathematik.uni-trier.de/~schmidt/gputop.

1 Introduction

Few areas are developing as quickly as the computational power of microprocessors. However, the means to sustain the exponential increase in processing speed as predicted by Moore's law appears to be shifting from making a single processing unit faster to making it wider, i.e. increasing the parallelism by switching to many processing cores and highly parallel architectures. The gap between the processor's computational power and the memory's ability to deliver data is also further deteriorating. These developments can lead to future systems which are very fast, highly parallel, but also highly heterogeneous, requiring numerical algorithms which are able to scale well to such systems.

One such system, which is designed for highly parallel throughput, is the commodity graphics card. Originally highly adapted to rendering three-dimensional polygonal data, these graphics adapters now have reached such a flexibility that they can well be used in scientific computing. Although nowadays they offer a quite fine-grained programming control, best performance still is achieved by a stream processing approach with many processing cores of which each can process many threads in a single instruction, multiple data (SIMD) fashion. Presently, NVIDIA's Compute Unified Device Architecture (CUDA) appears to be the most widely used solution for using graphics cards in scientific computing. Due to the ease of availability of commodity graphics hardware and CUDA being an extension of the C/C++ programming language, we chose to use CUDA for the studies presented in this article, with other alternatives being for example the OpenCL extension of C/C++.

The aim of this work is to study the applicability of this novel hardware in the field of PDE constrained optimization. With respect to solving problems involving PDEs, most literature on stream computing is either focused on linear algebra [7] or simulation alone [12,15], but not on optimization. It is finite differences [10] that are almost always used. The interest in computational fluid dynamics and finite volume methods is also particularly strong [6,9,13,18]. This work focuses on topology optimization based on the power law or "Solid Isotropic Material with Penalization (SIMP)" approach using finite elements on a structured mesh in three dimensions [2,21]. The SIMP approach for topology optimization models the distribution of material and voids by assuming constant material properties in each finite element. The optimization variable is the material density in each

Communicated by Gabriel Wittum.

S. Schmidt
Imperial College London, London, UK
e-mail: s.schmidt@imperial.ac.uk

V. Schulz (✉)
University of Trier, Trier, Germany
e-mail: Volker.Schulz@uni-trier.de

element raised to some power times the actual properties of the solid material. This concept is important from an applicational point of view and has been considered previously for GPU implementation for a two dimensional heat conduction problem in [20] with good results.

The problem size and computational time needed to solve topology optimization problems is usually dominated by the computational requirements of solving the linear elasticity state equation. The optimization step, which is presented in [19] and usually consists of the gradient computation, the mesh independency filter, and the mass preserving density update, is usually of limited computational complexity compared to solving the state equation. Depending on the domain size and the various input parameters, it may happen that one design update follows thousand conjugate gradient iterations for the state equation. In accordance with Amdahl's law, this work focuses more on the GPU acceleration of the finite element linear elasticity state equation solver than on the acceleration of the optimization overhead. GPU acceleration of solid mechanics solvers and linear elasticity has been studied recently in [11], where the GPU is used as a coprocessor. Here, however, we focus on using the GPU as the complete state equation solver, and not as a mere accelerator or coprocessor. This is done by solving the state equation matrix-free, i.e. the matrix–vector products for the residuals needed in the CG iteration are generated procedurally without storing the matrix anywhere. In the literature, the graphics card is mostly used as a coprocessor that computes matrix–vector products by using a discretization matrix that is pre-assembled by the CPU and explicitly stored on the graphics card, which requires much more total memory and needs many more costly accesses to the device memory. Because some older GPUs do not have the ability of atomic memory operations and because the standard finite element assembly by looping over elements produces a lot more memory accesses than necessary, we present a special "nodal" computation of the finite element matrix–vector product necessary to achieve this procedural residual computation.

Unfortunately, optimization problems hardly allow overlapping GPU with CPU computations. A CPU-based design update requires knowledge of the PDE state computed on the graphics card and vice versa. Theoretically, a domain decomposition approach could be used for the solution of the state equation on CPU and GPU in a heterogenous parallel fashion, but it can be quite difficult to find a correct partition of the domain which accounts for the significant differences in computational speed. The additional memory synchronization steps between the CPU main memory and the graphics card also greatly limit the expected increase in speed from this approach.

The work is structured as follows: Sect. 2 briefly recapitulates linear elasticity and topology optimization with the SIMP approach. In the following Sect. 3, the programming model of the GPU is discussed. Special attention is given to the procedural matrix–vector product. In Sect. 4, we compare the time needed by the GPU solver with different CPU implementations and study the performance of the most important subroutines on both architectures. The GPU is found to perform better than a 48 core shared memory system.

2 Linear elasticity and topology optimization based on the SIMP approach

2.1 Linear elasticity

We consider a three-dimensional body occupying a domain $\Omega \subset \mathbb{R}^3$. The deformation of the body under body forces $f : \Omega \to \mathbb{R}^3$ and boundary tractions $t : \Omega \to \mathbb{R}$ is modeled by linear elasticity in the displacement formulation. Let $u : \Omega \to \mathbb{R}^3$ be the displacements of the material under load. The linearized strain is given by

$$\epsilon_{ij} := \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right), \tag{2.1}$$

and the load linear form by

$$L(v) := \int_\Omega f \cdot v \, dA + \int_{\partial\Omega} t \cdot v \, dS.$$

The displacements u can be computed as the solution of the energy bilinear form

$$a(u, v) := \int_\Omega E_{ijkl}\epsilon(u)_{ij}\epsilon(v)_{kl} \, dS = L(u) \quad \forall v \in V, \tag{2.2}$$

where E_{ijkl} is an a priori given constant elasticity tensor. In homogeneous isotropic media, symmetry allows to reduce the order of the elasticity tensor by using the Voigt notation. Due to symmetry of (2.1), one can write

$$\tilde{\epsilon} := (\epsilon_{11}, \epsilon_{22}, \epsilon_{33}, \epsilon_{12}, \epsilon_{13}, \epsilon_{23})^T =: Bu$$

and the Cauchy Stress Tensor σ is given by

$$\sigma = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & & & \\ \nu & 1-\nu & \nu & & & \\ \nu & \nu & 1-\nu & & & \\ & & & 1-2\nu & & \\ & & & & 1-2\nu & \\ & & & & & 1-2\nu \end{bmatrix} \tilde{\epsilon}$$
$$=: CBu,$$

where E is Young's modulus and ν is Poisson's ratio. Hence, (2.2) can also be expressed as

$$a(u, v) = \int_\Omega (Bv)^T C Bu \, dS = L(u) \quad \forall v \in V. \tag{2.3}$$

In the following, we discretize (2.3) by using finite elements with linear test and trail functions on 8-node cubes in three dimensions, which results in a symmetric, positive definite linear system

$$Ku = f.$$

For more details on finite elements and elasticity theory, see [5]. The compliance c of the structure is given by

$$c(u) = u^T f = u^T K u,$$

which is the objective function to be minimized.

2.2 SIMP approach

The SIMP approach in topology optimization models the distribution of solid material and voids by introducing a pseudo density $\rho \in [0, 1]$ into (2.3):

$$a(u, v) = \sum_{e=1}^{N} \int_{\Omega_e} (Bv)^T \rho_e{}^p C Bu \, dS = L(u) \quad \forall v \in V. \tag{2.4}$$

The exponent p is used as a penalty factor enforcing a 0/1 density distribution as intermediate values are greatly reduced in effectiveness. Here, N is the number of mesh elements and ρ is considered constant for each element. The topology optimization problem is now given by

$$\min_{(u,\rho)} J(u, \rho) := u^T K(\rho) u \tag{2.5}$$

subject to

$$K(\rho)u = f \tag{2.6}$$

$$\sum_{e=1}^{N} \rho_e = v_0 \tag{2.7}$$

$$\rho_e \in \{0, 1\}. \tag{2.8}$$

Equation (2.7) is a volume constraint, as otherwise the solution will be a solid body. Condition (2.8) is usually relaxed to $\rho_e \in [\rho_0, 1]$, preventing the state equation from becoming singular. More details on topology optimization in general and the SIMP approach in particular can be found in [4]. Each optimization step follows [19]: According to [3], a scheme for updating the design variables is given by:

$$\rho_e = \begin{cases} \max(\rho_0, \rho_e - m) & \text{if } \rho_e B_e^\eta \le \max(\rho_0, \rho_e - m) \\ \rho_e B_e^\eta & \text{if } \max(\rho_0, \rho_e - m) < \rho_e B_e^\eta \\ & \quad < \min(1, \rho_e + m) \\ \min(1, \rho_e + m) & \text{if } \min(1, \rho_e + m) \le \rho_e B_e^\eta \end{cases}, \tag{2.9}$$

where m is a positive move-limit, $\eta = 0.5$ is a numerical damping coefficient and B_e is found from the optimality condition as

$$B_e = \frac{-\frac{\partial c}{\partial \rho_e}}{\lambda \frac{\partial V}{\partial \rho_e}},$$

where λ is a Lagrangian multiplier that can be found by a bisectioning algorithm. Alternatively, this scheme can also be interpreted as a certain projected gradient descent method [1]. The sensitivity of the objective function is found as

$$\frac{\partial c}{\partial \rho_e} = -p \rho_e^{p-1} u_e^T K_e u_e,$$

where K_e is the element stiffness matrix of which K is assembled. In order to prevent checker boarding and arrive at a mesh-independent structure, there is an additional mesh independence filter. However, we would like to refer to [4,19] instead of repeating the mesh filter here, too.

3 Implementation on the graphics processing unit

3.1 Overview

The domain Ω is discretized by using a cartesian mesh of cubic mesh elements. We employ a matrix-free conjugate gradient method to solve the state equation. The conjugate gradient method can be interpreted as an iterative solver for the linear equation $Ku = f$, where K is a symmetric and positive-definite matrix. For more details see [14,17]; however, the algorithm can be summarized briefly as follows:

$r_0 \leftarrow f - K u_0$
$p_0 \leftarrow r_0$
$k \leftarrow 0$
loop
$\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T K p_k}$
$u_{k+1} \leftarrow u_k + \alpha_k p_k$
$r_{k+1} \leftarrow r_k - \alpha_k K p_k$
if $r_{k+1} \le \epsilon$ **then**
exit
end if
$\beta_k \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
$p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$

$k \leftarrow k + 1$
end loop

For the discussions later on, it is worth noting that each step of the CG method necessitates one matrix–vector product for the computation of Kp_k, and two scalar products for the computation of α_k and β_k. Fascinatingly, the computation of the matrix–vector product will be the bottleneck of the CPU implementation, while the computation of the scalar products will be the bottleneck for the GPU implementation.

In order to further speed up the optimization, a design update is conducted before the state is fully converged. Especially for the first steps of the optimization scheme, it is often unnecessary to compute the "exact" gradient based on a fully converged state equation residual. As the optimization progresses, the design updates become smaller and likewise the defect in the state equation diminishes. In this sense the two-loop approach, i.e. an outer optimization loop and an inner loop for the state equation solver, is weakened and both the state equation residual as well as the norm of the gradient are driven to zero simultaneously. The system matrix K is never created explicitly; instead, we generate the product Ku needed for the conjugate gradient method only procedurally, which is—due to the cartesian mesh—an ideal task for the SIMD graphics processor. Due to the effectiveness of the optimality criteria update, only very few optimization iterations are needed, compared to the number of CG iterations. The mesh independency filter from [19] also requires an additional halo layer. Thus, these two steps are performed by the CPU, as—in accordance with Amdahl's law—their acceleration would improve the total computational speed-up only by an insignificant amount. However, porting these steps to the GPU is canonical. Additionally, a flexible filter radius during the mesh filter step of the OC update would require an equally flexible caching strategy on the GPU.

The graphics card acts like an autonomous compute device, meaning that after the data have been copied into the device memory, the actual computation is conducted by the GPU autonomously. It is therefore important to perform the entire CG method on the graphics card, as copying data from the system's RAM to the device memory is rather slow. The technical details of GPU performance tuning can be found in [8]. This especially includes managing the GPU's heterogeneous memory hierarchy of device, shared, and constant memory, and fulfilling the necessities for memory coalescing as well as avoiding shared memory bank conflicts.

Due to the alignment requirements for coalesced device memory access, storing three displacements for each node is highly unfavorable. Therefore, we store the density ρ_e for each element as part of the three nodal displacements components u_i in one "double4" variable. The overhead of storing $N_x \cdot N_y \cdot N_z$ displacements instead of $(N_x-1)(N_y-1)(N_x-1)$ is negligible. Similarly, this also means that for a two dimensional cartesian mesh of size $N_x \times N_y$, where N_x and N_y are not a multiple of the GPU block size, the vector must be padded with zeros, such that threads (0, 1), (0, 2), ... can access aligned vector components v_ℓ where $\ell = i + N_x \cdot j$.

3.2 Finite element matrix–vector product

3.2.1 Element based

The standard approach to compute Ku is given by

$$u_{\text{new}} = Ku = \sum_{e=1}^{N} K_e u_e, \tag{3.1}$$

where u_e refers to the restriction of u to the respective element e. An outer loop over all the elements with two inner loops testing each vertex of the element with each other results in incrementing one value of the state vector for each vertex. Processing one element results in 3 memory accesses per vertex: loading u_e^ℓ, the value of u_e at vertex ℓ of node e, loading u_{new}^ℓ, and writing back the incremented value. The element based matrix–vector product computation is tempting, because no special treatment of nodes using the natural boundary condition is necessary, which is very inline with the SIMD requirement.

Unfortunately, we found the standard approach to be unfavorable for the graphics processing unit for several reasons. The maximum number of threads allowed per block is 512. Thus, it is in general impossible to compute (3.1) by using only one block, and there will be nodes belonging to several blocks. Some GPUs lack atomic write operations to the device memory and the correct value at these nodes belonging to more than one block is not guaranteed. Besides, if one thread per element is used, the update of only one vertex fulfills the alignment requirement for a coalesced device memory access, whereas the remaining seven other vertices do not. Additionally, incrementing in the device memory this often is highly undesirable. Alternatively, a possible storage of intermediate values of u_{new} in the shared memory cuts the number of useable threads in half.

3.2.2 Nodal based

Instead of element-based finite element matrix–vector product computation, we assume that there is one thread per test function, i.e. per node. We use the built-in data type "float4" or "double4" to store three displacements and the density in each node, satisfying the GPU memory alignment. In order to minimize device memory access, each thread loads the state at its node into the shared memory, such that all threads of the block can operate on shared memory when

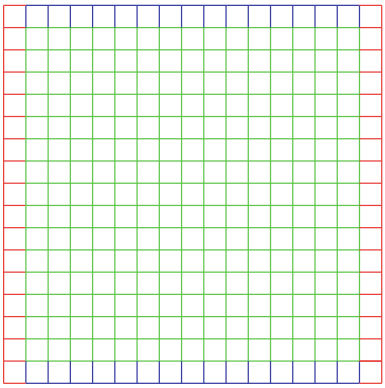

Fig. 1 16 × 16 patch with one halo-layer in shared memory per block. *Green* shows inner 16 × 16 nodes loaded coalescently. *Blue* shows halo nodes loaded coalescently. *Red* shows halo nodes loaded uncoalescently

accessing the values of neighbor nodes, thus minimizing device memory access. A node generates a non-zero entry in the finite element matrix for itself and each other node sharing the same mesh element. For linear test and trial functions, the nodal finite element matrix–vector product computation therefore requires knowledge of one neighbor node. Hence, some threads must also load halo values, which cannot be done in an entirely coalesced way and which may also interfere with the SIMD execution if the number of halo values is not a multiple of the half-warp size.

Due to the limited amount of shared memory, we follow the common strategy to extend a 2D computation into the third dimension by loading three successive 2D slices into the shared memory. Each thread then loops or "streams" slice-wise into the third dimension by discarding the last plane from the shared memory and loading one new plane. This is similar to the single-pass approach discussed in [16]. However, we do not store the data for the slowest varying dimension in the registers, but in the shared memory instead, because this data is also needed multiple times due to the larger support of our nodal finite element stencil. One such plane is sketched in Fig. 1. The strategy of feeding the shared memory is adapted from [10]. The central part of the code without feeding the shared memory is shown in Listing 1 below. The complete sources are available at http://www.mathematik.uni-trier.de/~schmidt/gputop.

Listing 1 Central Part of Computing Ku Procedurally

```
1  for(int ek1=0;ek1<2;ek1++)
2  {
3    const int EIDK = k-ek1;
4    if(EIDK >= 0 && EIDK < NZ-1)
5    {
6      for(int ej1=0;ej1<2;ej1++)
7      {
8        const int EIDJ = j-ej1;
9        if(EIDJ >= 0 && EIDJ < NY-1)
10       {
11         for(int ei1=0;ei1<2;ei1++)
12         {
13           const int EIDI = i-ei1;
14           if(EIDI >= 0 && EIDI < NX-1)
15           {
16             const REAL Dens =
17       pow(s_u[ind-ei1*IOFF-ej1*JOFF-ek1*KOFF].w,
18             gpupexp);
19             const int LID1 = ei1+ej1*2+ek1*4;
20             for(int ek2=0;ek2<2;ek2++)
21             {
22               for(int ej2=0;ej2<2;ej2++)
23               {
24                 for(int ei2=0;ei2<2;ei2++)
25                 {
26                   const int LID2 = ei2+ej2*2+ek2*4;
27                   const int idiff = ei2-ei1;
28                   const int jdiff = ej2-ej1;
29                   const int kdiff = ek2-ek1;
30                   const REAL4 MyU =
31                   s_u[ind+idiff*IOFF+jdiff*JOFF
32                       +kdiff*KOFF];
33                 MyRes.x += Dens*(GPU_EleStiff[LID1]
34                             [LID2]*MyU.x
35                 + GPU_EleStiff[LID1][LID2+8]*MyU.y
36                 + GPU_EleStiff[LID1][LID2+16]*MyU.z);
37                 MyRes.y += Dens*(GPU_EleStiff[LID1+8]
38                             [LID2]*MyU.x
39                 + GPU_EleStiff[LID1+8][LID2+8]*MyU.y
40                 + GPU_EleStiff[LID1+8][LID2+16]*MyU.z);
41                 MyRes.z += Dens*(GPU_EleStiff[LID1+16]
42                             [LID2]*MyU.x
43                 + GPU_EleStiff[LID1+16][LID2+8]*MyU.y
44                 + GPU_EleStiff[LID1+16][LID2+16]*MyU.z);
45                   }
46                 }
47               }//end e2 loops
48             }//if i okay
49           }//i-loop
50         }//if j okay
51       }//j-loop
52     }//if k okay
53   }//k-loop
54   }//end if not dirichlet
55   //store results
56   res[indg_cur] = MyRes;
57   }//end if active
58   __syncthreads();
```

Lines 1–11 loop over all the elements that have node (i,j,k) associated with this thread as a vertex. The "if"-statements

are needed for boundary nodes. The variables EID∗ hold the global index of the element being processed. Line 16 loads the constant density for the element in computation. The node/vector component with global index (i,j,k) has different local indices depending on the element it is a vertex of. The loops starting in lines 19–23 iterate over local indices which node (i,j,k) has in adjacent elements. Line 25 computes the index of the trial function being tested with, and needed to access the element stiffness stored in GPU_Ele\−Stiff[24][24], residing in the constant memory. Lines 26–28 compute the same indices for accessing the states in the shared memory. The following computation for the matrix–vector product is straight forward and the kernel ends with a single store to the device memory per component of the vector u. Consequently, this is also one perfectly coalesced write operation per thread, completely avoiding the problems with atomic increment instructions.

3.3 Single precision

Although nowadays GPUs are competitive by using double precision floating point numbers, the desire to remain backwards compatible with 1.x compute capability devices means that we must also address some issues raising from the use of a single precision number representation. Consequently, this will also enable us to conduct performance studies between single and double precision GPU and CPU computations later on.

Apparently, the optimality criteria update is sensitive towards round-off errors. In order to find the Lagrange multiplier λ of the volume constraint, we employ the same bi-sectioning algorithm as in [19]. However, for the single precision test runs, the termination criteria

$$\sum_{e=1}^{N} \rho_e \in [v_0 - \epsilon, v_0 + \epsilon]$$

has to be relaxed to $\epsilon = 10^{-3}$, as the original termination criterion is below single precision accuracy. This does not lead to any problems with the volume fraction being noticeably violated. We also have to modify the actual optimality criteria update scheme. In single precision, components v_i of the gradient vector v with $|v_i| < 10^{-5}$ can have an unreliable sign. Although the relative error is insignificant, the updating scheme (2.9) is quite sensitive to components of the gradient being "−0" instead of "+0" due to the discontinuous nature of max and min operators. Since the optimal solution without constraints is completely filled and without voids, the sign of the gradient can be corrected manually.

The single precision limit is especially problematic when the problem size increases, as the condition number of the system matrix becomes very large for problems with 10^6–10^7 unknowns. The result usually is a poor convergence behavior of the CG method for large domains, and highly uneven material distributions.

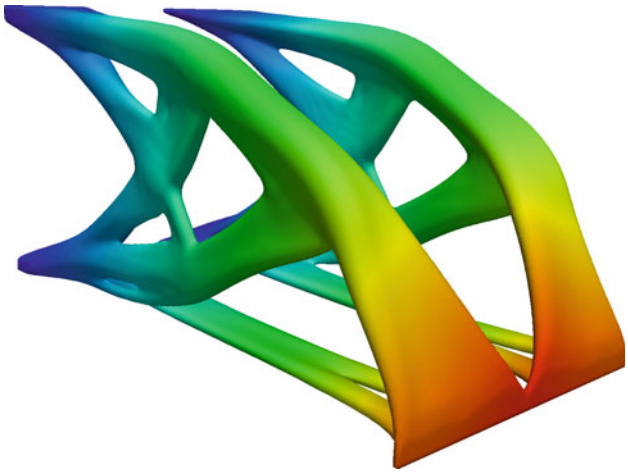

Fig. 2 Final cantilever design on a $100 \times 100 \times 200$ mesh

4 Results and performance studies

4.1 Extruded cantilever beam

We test the GPU topology optimizer on a mesh with $100 \times 100 \times 200$ points in three space dimensions by using a setup such that a cantilever beam can be expected as the optimal solution. This results in a problem with a moderate number of unknowns: 6,000,000 nodal displacements and 1,950,399 cell densities, a total of 7,950,399 unknowns. The optimization must find a design that fills at most 20 % of the mass. The other constants are chosen as follows: Young's modulus $E = 1$, Poisson's ratio $\mu = 3 \times 10^{-1}$, penalty exponent $p = 3.0$, filter radius 2.8, and minimum density $\rho_0 = 0.15$. The final cantilever is shown in Fig. 2, together with obvious boundary conditions. The convergence rates for the single and double precision implementations of the CG method for the first optimization iteration can be seen in Fig. 3; they show inferior performance as could be expected, when single precision is used. It should be noted that we often need less time to solve the linear elasticity equation when using double precision over single precision, because—although more computational time is needed per iteration—the faster convergence rate more than compensates for this.

4.2 CPU implementation

Before discussing the GPU implementation, we will study the performance of the CPU version first, such that the GPU code is measured against a well-tuned alternative. For this

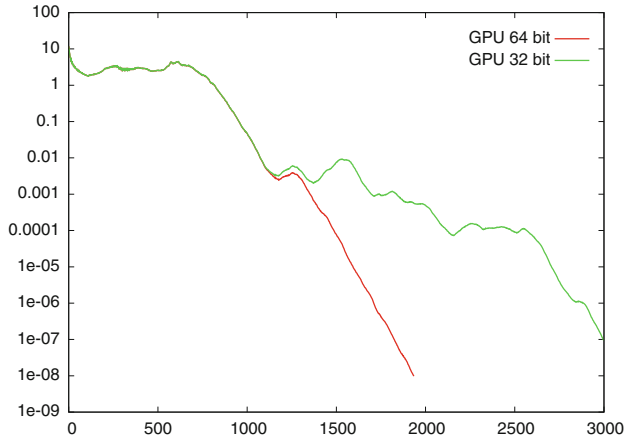

Fig. 3 Convergence rate of the CG method in single and double precision for the first optimization iteration

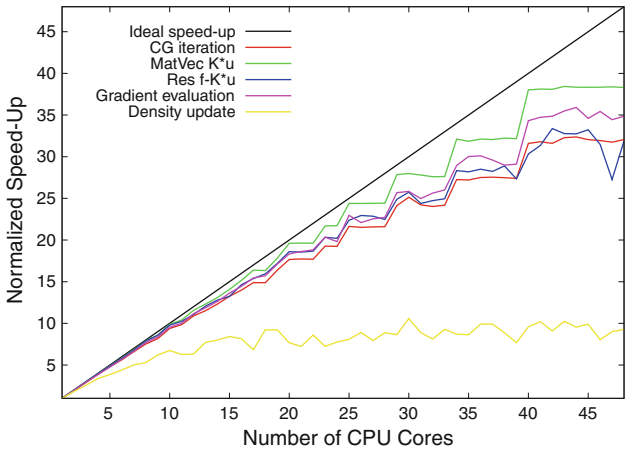

Fig. 4 CPU scaling behavior of different subroutines during topology optimization

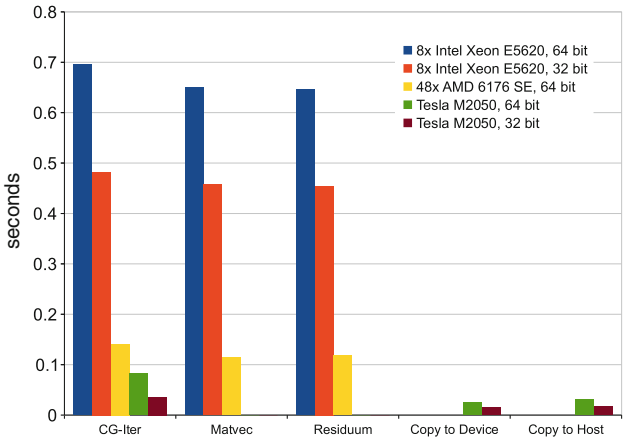

Fig. 5 CPU and GPU time needed for different subroutines. Time is averaged for each subroutine over one complete topology optimization

purpose, we study the parallel performance and scaling, especially of the nodal finite element matrix–vector product, on an AMD compute server consisting of AMD 6176 SE "Magny" cores with 512 KiB L2 cache each. The server consists of four CPUs with 12 cores each. Furthermore, there is also 12 MiB of L3 cache per CPU. Thus, the machine has a total of 48 shared memory cores communicating by AMD HyperTransport at 1.8 GHz. The speed up of the different routines of the topology optimization code is shown in Fig. 4. Due to the nature of the HyperTransport system and the preferred memory areas of each core, the scaling shows a step-wise increase in parallel efficiency. This phenomenon has been observed for a variety of other benchmarks before and can be considered normal for this architecture, unless more effort is invested by making the operating system manually pin threads to cores with matching memory preference.

The scaling of the nodal matrix–vector product seems to be very promising for the GPU implementation. However, note that the CG iteration, the residual computation, and the gradient evaluation both include the computation of norms and inner products, which is implemented by using the OpenMP tree reduction command. Unfortunately, the computation of scalar values has a rather bad parallel scaling; this aspect will be of considerable importance for the GPU implementation later on. The density and actual optimization update shows very little parallel scaling and has therefore not been implemented to the GPU.

4.3 GPU implementation

With four floating point numbers per mesh node, the problem of three dimensional linear elasticity is very data intensive, especially when one considers that the actual multiplication with the element stiffness K_e is of very low algorithmic intensity. Nevertheless, the GPU implementation running on an NVIDA Tesla M2050 device can outperform the 48 core AMD 6176 SE shared memory system. A detailed comparison of different architectures and floating point number representations can be seen in Fig. 5. Considering the time spent in each subroutine, the CPU and GPU have completely different bottlenecks. Note that for the CPU architectures, the time needed for one CG iteration is almost completely spent in either computing the matrix–vector product $K(\rho)u$ or the residuum $f - K(\rho)u$, not including the computation of any vector norms. For the CPUs, the overhead of the CG method is thus almost negligible compared to the computation of the matrix–vector products.

For the graphics card, the computation of the matrix–vector product requires on average only 5.34×10^{-6} s in double precision and 4.95×10^{-6} s in single precision, which is negligible compared to the 0.0817 s of one CG iteration in double precision or 0.0362 s in single precision. In direct comparison, the 48 core AMD 6176 SE needs 0.1410 s per CG iteration, which is slower by a factor of 1.726. Thus, the graphics card spends a disproportional time on the additional

calculations of inner products and norms needed by the CG method, while still outperforming the shared memory system. This suggests that the Richardson iteration would be much better suited, but the very poor convergence behavior makes using that method infeasible. However, a multi-grid method with a smoothing step based on the Richardson iteration could be very promising for future research.

The updating of the material density is not done on the graphics card due to the bad scaling; therefore, each optimization step on the GPU also has to include copying the data through the PCIe bus to the main memory, which in double precision requires on average 0.0308 s for one upload and 0.0256 s for one download. This is less than a single CG iteration and has therefore only very little impact on the overall performance.

Summarizing the above, the GPU implementation can outperform a 48 core shared memory system, making the Tesla device very cost efficient. The finite element matrix–vector product on a structured mesh is very well suited for a GPU implementation, but the overhead of the CG method should be addressed as part of further research by using more iterations of a Richardson type, perhaps as a multi-grid smoother.

5 Conclusions and outlook

We have investigated topology optimization on CUDA enabled graphics cards. An optimization procedure very similar to [19] in three dimensions has been conducted on the graphics card and found to be able to outperform a CPU implementation on a 48 core shared memory system. The CPU code is found to perform and scale well on all 48 cores. Special attention is given to conduct a matrix-free CG iteration on the device with procedural generation of the matrix–vector product. Both CPU and GPU implementation show entirely different computational bottlenecks when we measure the performance: The CPU implementation is dominated by the time needed to compute a matrix–vector product, whereas the GPU is dominated by inner products and norms needed by the CG method.

References

1. Ananiev, S.: On equivalence between optimality criteria and projected gradient methods with application to topology optimization problem. Multibody Syst. Dyn. **13**(1), 25–38 (2003)
2. Bendsøe, M.P.: Optimal shape design as a material distribution problem. Struct. Optim. **1**, 193–202 (1989)
3. Bendsøe, M.P.: Methods for Optimization of Structural Topology, Shape and Material. Springer, Berlin (1995)
4. Bendsøe, M.P., Sigmund, O.: Topology Optimization—Theory, Methods and Applications, 2nd edn. Springer, Berlin (2004)
5. Braess, D.: Finite Elemente, Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie, 2nd edn. Springer, Berlin (1997)
6. Brandvik, T., Pullan, G.: Acceleration of a 3d Euler solver using commodity graphics hardware. In: Proceedings of the 46th AIAA Aerospace Sciences Meeting, vol. AIAA 2008-607. AIAA (2008)
7. Buatois, L., Caumon, G., Lévy, B.: Concurrent number cruncher: an efficient sparse linear solver on the GPU. In: Lecture Notes in Computer Science, vol. 4782, pp. 358–371 (2007)
8. NVIDIA Coorperation.: NVIDIA CUDA C programming guide 4.0, May 2011
9. Elsen, E., LeGresley, P., Darve, E.: Large calculation of the flow over a hypersonic vehicle using a GPU. J. Comput. Phys. **227**, 10148–10161 (2008)
10. Giles, M.B.: Using NVIDIA GPUs for computational finance. http://people.maths.ox.ac.uk/~gilesm/hpc/
11. Göddeke, D., Wobker, H., Strzodka, R., Mohd-Yusof, J., McCormick, P., Turek, S.: Co-processor acceleration of an unmodified parallel solid mechanics code with FEASTGPU. Int. J. Comput. Sci. Eng. **4**(4), 254–269 (2009)
12. Goodnight, N., Woolley, C., Lewin, G., Luebke, D., Humphreys, G.: A multigrid solver for boundary value problems using programmable graphics hardware. In: HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 102–111, Aire-la-Ville, Switzerland, Switzerland, Eurographics Association (2003)
13. Hagen, T.R., Lie, K.A., Natvig, J.R.: Solving the Euler equations on graphics processing units. In: Lecture Notes in Computer Science, vol. 3994, pp. 220–227 (2006)
14. Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. J. Res. Natl. Bureau Stand. **49**(6), 409–436 (1952)
15. Komatitsch, D., Michéa, D., Erlebacher, G.: Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. J. Parallel Distrib. Comput. **69**(5), 451–460 (2009)
16. Micikevicius, P.: 3D finite-difference computation on GPUs using CUDA. In: GPGPU-2 Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, pp. 79–84 (2009)
17. Nocedal, J., Wright, S.J.: Numerical Optimization. Springer Series in Operations Research. Springer, Berlin (1999)
18. Phillips, E.H., Zhang, Y., Davis, R.L., Owens, J.D.: Rapid aerodynamic performance prediction on a cluster of graphics processing units. In: Proceedings of the 47th AIAA Aerospace Sciences Meeting, vol. AIAA 2009-565. AIAA (2009)
19. Sigmund, O.: A 99 line topology optimization code written in matlab. Struct. Multidiscip. Optim. **21**(2), 120–127 (2001)
20. Wadbro, E., Berggren, M.: Megapixel topology optimization on a graphics processing unit. SIAM Rev. **51**(4), 707–721 (2009)
21. Zhou, M., Rozvany, G.I.N.: The COC algorithm, part II: topological, geometry and generalized shape optimization. Comput. Methods Appl. Mech. Eng. **89**, 197–224 (1991)