Bone structure analysis on multiple GPGPUs

Bone structure analysis on multiple GPGPUs

Peter Arbenz*, Cyril Flaig, Daniel Kellenberger

ETH Zürich, Computer Science Department, Universitätsstrasse 6, 8092 Zürich, Switzerland

Abstract

Osteoporosis is a disease that affects a growing number of people by increasing the fragility of their bones. To improve the understanding of the bone quality, large scale computer simulations are applied. A fast, scalable and memory efficient solver for such problems is ParOSol. It uses the preconditioned conjugate gradient algorithm with a multigrid preconditioner. A modification of ParOSol is presented that profits from the exorbitant compute capabilities of recent general-purpose graphics processing units (GPGPUs). Adaptations of data structures for the GPGPU are discussed. The fastest implementation on a GPGPU achieves a speedup of more than five compared with the CPU implementation and scales from 1 to at least 256 GPGPUs.

Keywords: Micro finite element analysis, voxel-based computation, multigrid preconditioned conjugate gradient algorithm, multiple GPGPUs

1. Introduction

In recent years, high-resolution peripheral quantitative computed tomography (HR-pQCT) has become the gold standard for the accurate prediction of bone strength [29]. Anatomy-specific micro finite element (μFE) analysis can take into account geometry and internal architecture of the bone. The accurate prediction of bone strength, in particular in the human radius, is of major interest since fractures in the distal radius are amongst the most common in humans and their occurrence is increasing due to an aging population [18, 23].

In a μFE analysis a compression test is simulated numerically. A model of a bone specimen, composed of voxels, is restrained between two parallel plates one of which is pushed towards the other. The resolution in the range of 10–100 μm triggered the name of the method. This approach results in bone models with a huge number of voxels (cubes) entailing very demanding computations with enormous numbers of degrees of freedom.

In 1996, van Rietbergen et al. [24] proposed to directly translate the voxels generated by the CT scan into finite elements and apply a structural solid mechanics analysis. The matrix of the resulting linear system is never formed. The linear system is solved by the preconditioned conjugate gradient (PCG) algorithm whereby matrix-vector products are executed element-by-element (EBE). This algorithm is extremely memory efficient. However, it does not scale, i.e., the number of iteration

*Corresponding author

Email addresses: arbenz@inf.ethz.ch (Peter Arbenz), cyril@flaig.ch (Cyril Flaig), kellenberger@inf.ethz.ch (Daniel Kellenberger)

steps increases rapidly with the size of the system matrix. Nevertheless, this code was used for models with up to a couple of million degrees of freedom.

Starting with Adams' code Prometheus [2] the conjugate gradient algorithm preconditioned by smoothed aggregation-based algebraic multigrid was used for the μFE analysis. To construct the preconditioner, the system matrix had to be built. Adams et al. [2] could conduct a large deformation (nonlinear) finite element analysis of a solid mechanics problem with up to 537 million degrees of freedom using 4088 IBM Power3 processors.

Later we wrote a similar code, ParFE [3, 4, 20], based on the Trilinos frame work [14]. This code was modified to admit a matrix-free, i.e. EBE, matrix-vector product on the finest level reducing the memory consumption of the code by about a factor 3–4 [4]. The largest model solved with the latter matrix-free code consisted of 1.5 billion degrees of freedom. A minimum of 4800 cores of a Blue Gene/L were required to store the data [6].

In recent years we have devised an extremely memory efficient code for micro-structural μFE analysis [10–12]. The code, ParOSol [21], exploits the fact that the voxels are cubes embedded in a rectangular grid that are available as a 3D image from a stack of CT scans. The PCG solver is complemented by a *geometric* multigrid preconditioner. The storage of ParOSol is based on a Morton space-filling curve [27]. This recursively defined curve is equivalent to an octree. ParOSol requires about 90 Bytes of memory space per degree of freedom (dof) while matrix-free ParFE required about 1600 B/dof. This allowed us to solve a model with 94.7 billion dofs on just 8000 cores [12]. Here, each core is equipped with 1.33 GB of main memory. Typical problem sizes of 100 million degrees of freedom requires just 8 cores with ParOSol, compared to 128 with ParFE.

In a usual μFE analysis, a 3D image is first translated into a finite element mesh, then stiffness matrix and right-hand side are assembled according to the displacement model of linear elasticity, and finally the linear system is solved. In contrast, in ParOSol matrices are never formed, not even inside the multigrid preconditioner. All necessary information is gathered directly from the 3D image. The coarser grids can be interpreted as obtained from an image of lower resolution.

In this paper we present a version of ParOSol, CUDA-ParOSol, that is adapted to NVIDIA's GPUs [15]. The main difference is a more regular access of data in order to exploit the SIMD features of a GPU. This is detailed in section 3. The experiments in section 4 show the success of the revision. They also show that the modifications increase the speed of the CPU version of ParOSol.

2. Mathematical model

The weak form of the displacement formulation of linear elasticity theory is used to analyze the bone strength [8]: Find the displacement field $\boldsymbol{u} \in [H^1_E(\Omega)]^3 = \{v \in [H^1(\Omega)]^3 : \boldsymbol{v}_{|\Gamma_D} = \boldsymbol{u}_S\}$ with nonempty Dirichlet boundary Γ_D such that

$$\int_\Omega [2\mu\, \boldsymbol{\varepsilon}(\boldsymbol{u}) : \boldsymbol{\varepsilon}(\boldsymbol{v}) + \lambda \operatorname{div} \boldsymbol{u} \operatorname{div} \boldsymbol{v}]\, d\Omega = \int_\Omega \boldsymbol{f}^T \boldsymbol{v}\, d\Omega + \int_{\Gamma_N} \boldsymbol{g}_S^T \boldsymbol{v}\, d\Gamma \tag{1}$$

for all $\boldsymbol{v} \in [H^1_0(\Omega)]^3$ with the volume forces $\boldsymbol{f}$, the boundary traction $\boldsymbol{g}$ on the Neumann boundary Γ_N, the linearized symmetric strain tensor

$$\boldsymbol{\varepsilon}(\boldsymbol{u}) := \frac{1}{2}(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T),$$

and the Lamé constants

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \qquad \mu = \frac{E}{2(1+\nu)}.$$

Here, E is the Young's modulus and ν the Poisson ratio.

In our computations we use two different boundary conditions. The Neumann boundary Γ_N is traction free, $\boldsymbol{g}_S = \boldsymbol{0}$. On the top and bottom of the domain we impose Dirichlet boundary conditions with fixed displacements. The engineers look for regions with high stresses and strains to determine the quality of the bone [30].

The displacements $\boldsymbol{u}$ are discretized by trilinear hexahedral elements. These are converted one-to-one from the voxels of the CT image. Thus, all elements are cubes of the same size. In the present version of ParOSol only the Young's modulus can vary in the domain. The Poisson ratio ν must be constant. Bone mass typically has a Poisson ratio $\nu = 0.3$. Applying this finite element discretization to (1) results in a linear symmetric positive definite (SPD) system

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{f}. \tag{2}$$

The number of degrees of freedom can exceed 10^9. For symmetric positive definite linear systems of this size the preconditioned conjugate gradient (PCG) algorithm is the solver of choice [26]. We use a geometric multigrid preconditioner to make the solver scalable.

The matrix $\boldsymbol{A}$ can be written as

$$\boldsymbol{A} = \sum_{e \subset \Omega} \boldsymbol{T}_e \boldsymbol{A}_e \boldsymbol{T}_e^T, \tag{3}$$

the form that is used in EBE matrix-vector products. The 0-1-matrices $\boldsymbol{T}_e$ map the local to the global degrees of freedom. Since the Young's modulus E appears linearly in $\boldsymbol{A}_e$ we can write the matrix-vector product as

$$\boldsymbol{A}\boldsymbol{x} = \sum_{e \subset \Omega} E_e \, \boldsymbol{T}_e \boldsymbol{A}_{\text{ref}} \boldsymbol{T}_e^T \boldsymbol{x}, \tag{4}$$

where E_e is the Young's modulus of element e.

For the multigrid preconditioner we coarsen by aggregating $2 \times 2 \times 2$ voxels. A voxel of the coarser level $\ell+1$ gets its Young's modulus by averaging the Young's moduli of the eight aggregated smaller voxels of level ℓ,

$$E_e^{\ell+1} = \frac{1}{8} \sum_{e' \subset e}^{1} E_{e'}^{\ell}, \tag{5}$$

where the Young's modulus of non-existing child elements is zero. Let the union of voxels on level ℓ be denoted by Ω_ℓ, $\ell = 0, \ldots, L$, with $\Omega_0 = \Omega$. Then the system matrix on level ℓ can be written as

$$\boldsymbol{A}^\ell = 2^\ell \sum_{e \subset \Omega_\ell} E_e^\ell \, \boldsymbol{T}_e^\ell \boldsymbol{A}_{\text{ref}} \boldsymbol{T}_e^{\ell T}. \tag{6}$$

Notice, that because of the incorporation of non-existing voxels the domains Ω_ℓ can grow with ℓ.

If the above coarsening procedure is applied to a homogeneous grid with the standard prolongation (trilinear interpolation) and restriction (transpose of prolongation) it corresponds to the Galerkin product [28, p.72].

We employ Chebyshev polynomial smoothers [1], a type of smoothers that scales very well. We used them successfully in ParFE in the context of a smoothed aggregation-based algebraic multigrid preconditioner [3, 4].

3. Implementation of the solver

Our iterative solver employs the conjugate gradient algorithm with a geometric multigrid preconditioner, see Algorithm 1. The pre- and postsmoothers are Chebyshev polynomial smoothers

Algorithm 1 Multigrid preconditioner

function MultiGridSolve(Matrix $\boldsymbol{A}_\ell$, Vector $\boldsymbol{x}_\ell$, Vector $\boldsymbol{b}_\ell$, int ℓ, int γ)

```
if l == max_level then
  Solve = CoarseSolve;
else
  Solve = MultiGridSolve;
end if

Presmooth(A_ℓ, x_ℓ, b_ℓ);
for number of cycles γ do
  r = Restrict(b_ℓ − A_ℓ x_ℓ);
  c = 0;
  Solve(A_{ℓ+1}, c, r, ℓ + 1);
  x_ℓ = x_ℓ + Prolongate(c);
  Postsmooth(A_ℓ, x_ℓ, b_ℓ);
end for
return
```

as suggested by Adams et al. [1] and implemented as in the multilevel preconditioner package ML of Trilinos [13, 14]. The polynomials are chosen to be small on the 'upper part' of the respective spectrum of $\boldsymbol{A}_\ell$ [1]. The latter is determined in the setup phase by a few steps of the Lanczos algorithm. The system on the coarsest grid is solved approximately by the conjugate gradient algorithm with Jacobi (diagonal) preconditioning.

The crucial operations in our PCG solver are thus the matrix-vector products and the inter-grid transfers (restriction and prolongation). Matrix-vector products have to be executed on all levels, for the computation of residuals and in the smoothing steps.

In this section we review how data is arranged in ParOSol [10, 12] and how we implemented the two crucial operations based on this storage scheme. We discuss issues that prevent it from being ported directly to a SIMD machine such as a GPGPU. Then we introduce two approaches to remedy the shortcomings [15].

3.1. ParOSol

In ParOSol, we exploit the embedding of the computational domain Ω in a regular grid with a power of 2 nodes in each direction. The nodes of the regular grid are numbered following the Morton order or z-order [5, 31]. The mapping of the node numbers in the lexicographic order and in the Morton order is accomplished by a rearrangement of bits [5, §7.2]. Since only a small fraction of nodes of the regular grid are also contained in the computational domain Ω we only store the nodal data (displacements) of those nodes that are contained in Ω in Morton order in contiguous memory locations. The mentioned fraction, called *porosity*, is typically between 10% and 20%.

Given any element in Ω, we can easily determine the numbers of its 8 vertices in lexicographic order and then compute their *keys*, i.e. their numbers in the Morton order. We now look for these 8 key values in an array `Key` that is arranged in the same way as the vector `Disp` of displacements, cf. the 2D example in Fig. 1. So, given the locations of the key values in the `Key` array, we directly

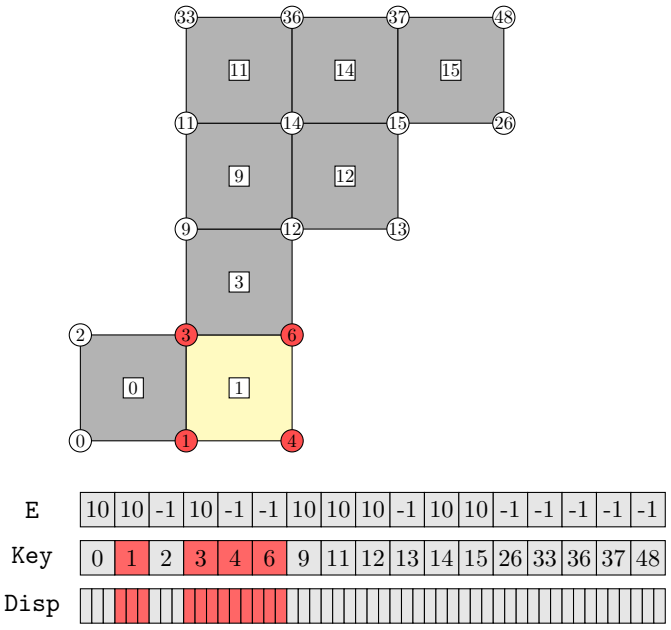

Figure 1: To store the degrees of freedom in a compact manner in an array, the index of the positions in the dense array (e.g. the displacement values) correspond to the positions of the nodes in the octree array.

get the locations of the displacement values in the `Disp` array. Note, that `Disp` serves as a template of the data layout for all node related data.

We identify elements with the smallest among the keys of its vertices. The array `E` that has a positive entry at positions that correspond to both a node and an element. At the other positions `E` has a negative value, see Fig. 1. We use `E` to store the Young's moduli of the respective elements.

Algorithm 2 Element-by-element matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{A}x$

function MatVec(Matrix $\boldsymbol{A}$, Vector $\boldsymbol{x}$, Vector $\boldsymbol{y}$)
  $\boldsymbol{y} = \boldsymbol{0}$;
  for $e \subset \Omega$ **do** {Traverse elements}
    Determine keys of vertices of element e;
    Search keys in `Key` array;
    Extract $\boldsymbol{x}_{\text{local}}$ from $\boldsymbol{x}$ array; $\{\boldsymbol{x}_{\text{local}} = \boldsymbol{T}_e^T \boldsymbol{x}\}$
    Execute local dense 24×24 MatVec; $\{\boldsymbol{y}_{\text{local}} = \boldsymbol{A}_{\text{ref}}\, \boldsymbol{x}_{\text{local}}\}$
    Apply elemental E modulus; $\{\boldsymbol{y}_{\text{local}} = E_e \boldsymbol{y}_{\text{local}}\}$
    Update components of $\boldsymbol{y}$ by $\boldsymbol{y}_{\text{local}}$; $\{\boldsymbol{y} = \boldsymbol{y} + \boldsymbol{T}_e \boldsymbol{y}_{\text{local}}\}$
  end for
return

Algorithm 2 shows how we form the sum (4). This algorithm is applied on all levels. The

procedure that is used to search the key values is given in Algorithm 3. Since in Algorithm 2 the elements are traversed in the Morton order the vertex keys are always to the right of the element key. The recursive definition of space-filling curves entails good data locality [5]. This gives reason to extend the binary search by an exponential range search to speed up the location of the indices, see Algorithm 3, see [10].

Algorithm 3 Optimized Search

```
function int SearchIndex(int start, t_octree_key key, t_tree tree)
  int range_begin = start;
  int range_end = range_begin + 1;
  while key > tree[range_end].key do
    int tmp = range_end + 1;
    range_end = (range_end − range_begin) · 8 + range_end;
    range_end = tmp;
  end while
return binarySearch(range_begin, range_end, key, tree);
```

Algorithm 2 is parallelized in a straightforward manner across elements. The loop $e \subset \Omega$ can be implemented as a loop over those elements of the array `Key` that have a positive `E` component. For the partitioning of the data we exploit once more the space-filling curve. The only restriction is that the first key of each part of `Key` must be an element key. Therefore, evenly cutting `Key` in as many pieces as there are MPI processes results in quite a well-balanced matrix-vector product. There are two statements that affect the parallel performance. **(1)** The effort of searching the key values depends on the key, causing a possible load imbalance. The average work per process is however quite evenly distributed with large problems. **(2)** There is a data dependence in the statement that updates $\boldsymbol{y}$.

3.2. Porting to the GPU: CUDA-ParOSol

A good overview on GPGPU programming can be found in [9]. The authors describe the Fermi architecture and illustrate some programming strategies. The Kepler and the Fermi architectures share most features. We will give a short overview on the Kepler architecture which we also refer to as 'GPU architecture'.

The Kepler architecture is based on streaming multiprocessors (SMX) [19]. Each SMX consists of 192 CUDA cores and 32 load/store units together with some additional double precision units. Each CUDA core has a floating-point and an integer ALU. Each SMX can concurrently execute 4 warps, a group of 32 threads. All threads share 65536 32-Bit registers and 64 kByte memory that can be used as fast shared memory or cache memory. A Tesla K20x has 14 SMX to which 6 GByte of GDDR5 memory are connected. The GPUs of the Kepler generation introduced GPUDirect™ RDMA that enables direct communication with the network card, and improved MPI communication between the GPU and network nodes.

Logically the threads are grouped into blocks. In a second step the blocks are grouped into a grid. Each block is executed on a SMX. The threads within a block synchronize and cooperate using the fast shared memory [9]. A multiprocessor splits the block into warps that have up to 32 threads. The warp is executed in the single instruction-multiple data (SIMD) mode. The difficulty is to design the program in a way that all resources are occupied most of the time. This

means that all units of the SMX have to be kept busy and enough threads should be ready to run in order to hide the memory latency. The memory bandwidth can be fully exploited only if the memory accesses are aligned [9]. This means that contiguous warps access contiguous regions of memory.

If an algorithm is ported to GPUs many details have to be considered to achieve the highest possible performance. Especially, complicated branch structures must be avoided to prevent thread divergence. Look-up tables are a way out, since memory accesses on the GPUs are very fast.

If we have a closer look at ParOSol we can easily see that the optimized search in Algorithm 3 [10, 12] does not fit on the GPU. The algorithm has many branches and the number of steps used in the search varies from thread to thread. The searches cannot be parallelized in the SIMD execution mode and get serialized.

To remedy the potential load imbalance we have implemented two different search algorithms. The first uses the perfect spacial hashing algorithm proposed by Lefebvre and Hoppe [16]. The second stores the offset of the nodes in a compressed scheme trying to benefit from the data locality that is implied by the Morton ordering.

3.2.1. Hash approach

To implement a hash algorithm in a GPU-friendly way collisions in the hash table must be avoided. This is because resolving a collision needs a branch instruction to jump to the next possible storage location and thus the number of instructions of each query varies. Our hash approach uses the perfect spatial hashing algorithm proposed by Lefebvre and Hoppe [16]. This algorithm is by definition collision-less. We adapted the three-dimensional version of the algorithm to the one-dimensional case since the Morton ordering that is used for domain decomposition can also be used to linearize the three dimensions. In one dimension evaluating the hash function (7) needs fewer instructions, because it has to be evaluated only once.

The main idea behind perfect partial hashing is to use a hashing function that consists of two simple hashing functions with a main hash table H and a much smaller offset table Φ,

$$h(k) = h_0(k) + \Phi[h_1(k)] \mod \text{size}(H), \tag{7}$$

where $h(k)$ returns the index to the hash table H. The size of H is chosen to be not much bigger than the number of elements to store. To remove collisions the small offset table is used. The hash functions $h_0(k)$ and $h_1(k)$ are defined as follow:

$$h_0(k) = k \mod \text{size}(H), \qquad h_1(k) = k \mod \text{size}(\Phi),$$

where $\text{size}(H)$ and $\text{size}(\Phi)$ are the size of the respective hash table.

The difficulty is to construct a hash table without colliding elements. This is done as proposed in [16]. First the size of the main table H is determined. Then a size of the offset table is chosen. In the third step the tables are filled. If collisions cannot be resolved with the actual offset table, a new bigger sized offset table is chosen. The filling of the hash table is implemented by the greedy algorithm proposed in [16].

This construction can be very time consuming and can hardly be parallelized. To speed up the construction, which is needed in the preprocessing phase of the simulation, the hash table on a shared memory node is split into n_t tables. For efficiency reasons, n_t is chosen as a power of two.

With the hash approach we store the elements in a separate array. Additionally we have to store the weights of the elements. The indices for the vertices are stored in the hash table. The

Sample	offset approach						hash approach		
	p_{short}	p_{medium}	p_{long}	avg. [B]	$size_{l0}$ [MB]	$size_{tot.}$ [MB]	avg. [B]	$size_{l0}$ [MB]	$size_{tot.}$ [MB]
bone1	0.658	0.304	0.038	15.2	439	540	17.9	561	650
bone16	0.655	0.303	0.042	15.3	7072	8648	22.2	11139	12568
perlin1	0.663	0.300	0.037	15.1	425	654	17.5	652	867
perlin16	0.650	0.299	0.051	15.4	6911	8532	20.2	10902	12385
sphere	0.654	0.313	0.033	15.2	6.61	8.03	17.0	7.91	9.13
cubeSolid4000	0.645	0.302	0.053	15.5	353	418	17.1	394	450

Table 1: Distribution of the different element categories and their average sizes for some problems compared to the average sizes per element of the hash approaches. The total size of the grid of the finest level and the total size of all levels is shown, too.

offset table has only two bytes per entry. In the best case (optimal hashing) we need

$$\text{key} + \text{hash table entry} + \text{weight} = 8\,\text{byte} + 4\,\text{byte} + 8\,\text{byte} = 20\,\text{byte}$$

per vertex. With 50% overhead to avoid collisions of the hash table this approach needs less than 30 bytes per vertex, or 10 per degree of freedom to store the grid information.

3.2.2. Offset approach

Instead of computing and searching the keys of the vertices in each traversal of the domain, one may directly store all indices of the vertices. Such a mapping is however memory inefficient. Here, the Morton ordering comes to our rescue. We observed that the distance between the smallest and the largest index of an element is small in most cases. Empirically we found that for at least 65% of the elements this distance is less than 256 and for at least 95% less than 65536, respectively, see Table 1. We can prove these lower bounds for the solid cube, see Appendix A. Based on this observation, we classify the elements in three groups:

1. *Short elements:* all vertices are within a distance $d < 256$. The offsets are stored in one byte.
2. *Medium elements:* all vertices are within a distance $d < 65536$. The offsets are stored in two bytes.
3. *Long elements:* there are vertices with a distance $d \geq 65536$. The offsets are stored in four bytes.

The grid is stored using the following scheme: First all *long elements* are stored. Then the *medium elements*, and finally the *short elements* are appended. Additionally, one byte per element is added that holds some information that is needed for the restriction and the prolongation.

Tests showed that the offset approach needs 13% to 25% less memory to store the grid than the hash approach. This is illustrated in Table 1.

3.2.3. Restriction and prolongation

Restriction and prolongation are implemented similarly on CPU and GPU when the hash approach is used. For the offset approach there is a difference in the way the parent-child relationships

between the two involved grid levels is found. On the CPU the finer grid is traversed and the corresponding parent is obtained by dividing the fine key by 8. On the GPU the coarser grid is traversed. Because of the Morton ordering all existing children of a node are contiguous in memory. Therefore it suffices to store the index of the first child for every node (in an additional array). An 8-bit bitmap is employed to indicate which children do exist. For the short and medium elements this bitmap is stored as the offset of the first vertex (whose offset is 0). For the long elements an extra array is allocated.

3.2.4. Implementation

For the distribution of the computational domain and the communication between GPUs the same layout and implementation as in ParOSol is used, cf. Section 3.1.

Because the GPU has a smaller memory, two vectors needed in the PCG algorithm that are not used in the application of the preconditioner are swapped out to the main memory of the CPU. This makes it possible to simulate 20% bigger bone samples.

The most important operation in ParOSol is the matrix-vector multiplication that is executed on all levels, in particular in the smoothers. The MatVec is implemented in the element-by-element fashion, as given in Algorithm 2. Each elementwise MatVec constitutes a thread. So, each thread loads the element data and the vertex data of that element. The vertex data consist of 24 double values that are stored in eight triples in contiguous memory locations. Then the 24×24 matrix-vector product is executed which amounts to $24^2 = 576$ fused multiply-adds. Finally, the result is summed into the result vector.

Since there is an abundance of elements (cf. Table 2) and the arithmetic intensity of the approach is quite high we can expect CUDA-ParOSol to perform well. Nevertheless, since potentially up to eight threads deal with the same vertex, synchronization is necessary among the threads when they update the result vector. We use atomic operations to avoid race conditions. Because of the massive number of threads this synchronization overhead can easily be hidden behind computation.

Note that computing the matrix-vector product with multiple threads is not worthwhile for several reasons. First, there are enough elements available to keep all cores of the GPU busy. Second, any splitting of the matrix-vector product would introduce new synchronization overhead. Since the local stiffness matrix is constant for all elements/threads, the matrix can be hold in the constant memory and the broadcast mechanism is used to load the data. If all threads access the same address then reading from the constant memory is as fast as reading from a register.

To improve the performance of the memory accesses we have all threads on a thread block cooperate in the reading and writing of the (collective) node data. Remember that the threads in a thread block can communicate through the shared memory. Loading from and storing to global memory are fastest if contiguous threads read contiguous memory addresses. To exploit this fact we proceed as follows. Let `tid` be the identifier of a thread in a block of `blksize` threads, $0 \leq$`tid`$<$`blksize`. Each thread determines the memory locations where the x-, y- and z-displacements of element vertex 0 are stored and writes the three (contiguous) addresses at positions `3*tid` to `3*tid+2` of an index array of length `3*blksize` in the shared memory. All threads write into different entries of the index array. Further, since all elements have a different vertex 0, there are no repeated entries in this index array. After having filled the index array the threads `tid` read those displacements from memory that are at positions `tid`, `tid+blksize` and `tid+2*blksize` in the index array and writes them at the same positions in a shared displacement array. This is precisely what we need, since contiguous threads read contiguous memory addresses. Now, a first portion of the MatVec is executed in each thread. Afterwards, or synchronously with the MatVec,

the data of vertex 1 are transfered to the shared memory by the threads in the thread block in an analogous manner as the data of vertex 0. And so on with the data of the vertices 2 to 7. At the end of the procedure each thread has the result of the MatVec in a 24-vector. All threads in the thread block cooperate in writing back the result to memory. (This is actually an addition into the global result vector.) This can be done in eight steps 'in reversed order' as the orchestrated read operations above. There is no data dependence in the writing of the various threads in the thread block. There can however be race conditions across thread blocks that enforce atomic operations.

This interleaving of accessing the memory and computing improves the speed of the elementwise matrix vector product by about 30%.

4. Experiments

We performed weak and strong scalability tests to compare ParOSol with the new CUDA-ParOSol. We ported the new offset approach used on the GPU back to work on the CPU implementation to ensure a fair comparison. Thus, we have a 4-way benchmark with the original code, the offset approach implemented on both the CPU and the GPU, and the hash approach on the GPU.

All test were executed on Tödi, a Cray XK7 supercomputer at the Swiss National Supercomputing Center (CSCS)[1]. The system has 272 nodes, each equipped with a 16-core AMD Opteron CPU, 32 GB main memory, and one NVIDA Tesla K20x GPU with 6 GB of GDDR5 memory. The Cray XK7 has a 3D-torus interconnect based on the Gemini ASIC from Cray.

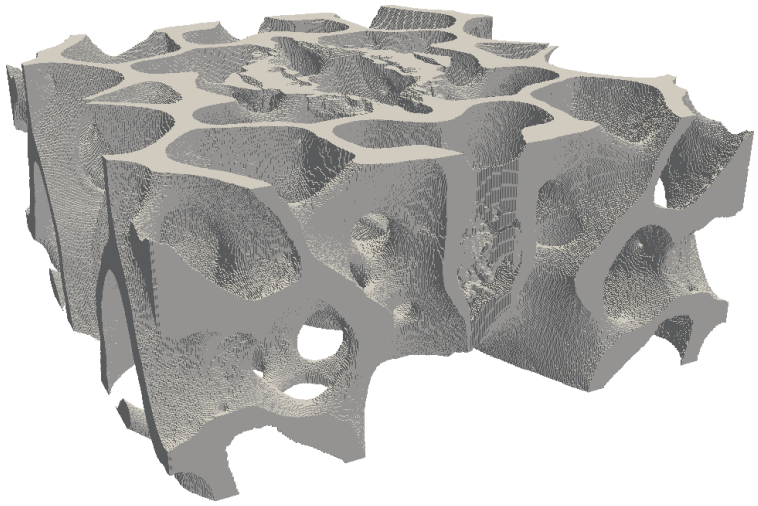

Figure 2: The `bone1` mesh is a cubical portion of a μCT scan of a real bone.

4.1. Weak scalability tests

In a weak scalability test the problem size per compute node stays fixed. If the volume of the data that is communicated is proportional to the problem size and the communication is mostly local then good weak scalability can be expected.

[1] http://www.cscs.ch/computers/toedi/

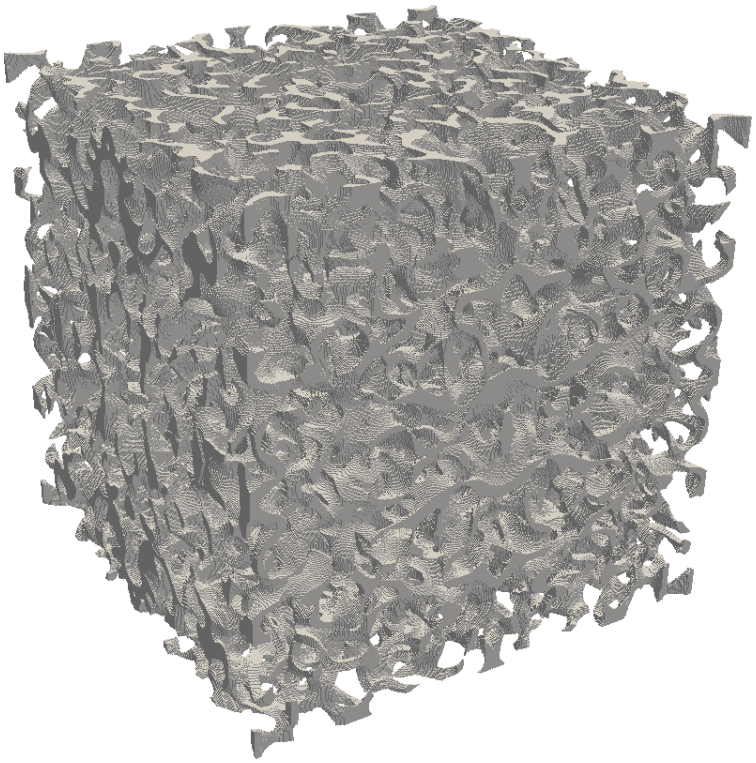

Figure 3: `perlin1` is a mesh generated artificially by Perlin noise. It consists of $30 \cdot 10^6$ cubical elements.

In this benchmark we use two base meshes. The first base mesh, `bone1` displayed in Figure 2, corresponds to a cubical portion of a real bone sample [25]. The base mesh is 3D-mirrored [3] to generate meshes of varying sizes, see Table 2.

The second mesh has been artificially generated with Perlin noise [22]. Perlin noise is a coherent noise function. Thus, in the near neighborhood of a point the noise function changes smoothly. A fast implementation of the Perlin noise can be obtained from the library libnoise [17]. To generate the artificial bone mesh we used the intersection of two thresholded Perlin noise functions. Figure 3 shows the base mesh. This artificial mesh has been 3d-mirrored in all direction. This saves a lot of computing time to generate the artificial meshes. Bigger meshes are obtained by evaluating the same Perlin noise functions on finer grids. This can be considered a discretization of the same bone but with a higher resolution.

The original solver (ParOSol) was designed to scale well on distributed memory supercomputers with MPI-based communication [10, 12]. To the best of our knowledge, ParOSol is at present the fastest and most memory efficient solver for bone structure simulations.

We now compare the GPU implementations of CUDA-ParOSol with the original ParOSol. Figures 4 and 5 show that both implementations scale perfectly on the GPUs in the weak sense on both meshes. This means that the time per iteration stays constant when the number of computing nodes and the problem size are increased proportionally. On the Tesla GPU an iteration step completes five times faster than on an Opteron CPU with 16 cores. The 5-fold speedup is remarkable and is comparable to sparse matrix-vector multiplication implemented in CUDA [7].

When looking at the execution times (Figures 6 and 7) we observe a similar behavior. The

Sample	dofs 10^6	elements 10^6	dim x	dim y	dim z	iterations
bone1	99	30	628	628	284	7
bone2	197	61	628	628	568	6
bone4	394	121	1256	628	568	7
bone8	788	243	1256	1256	568	7
bone16	1575	486	1256	1256	1136	5
bone32	3149	971	2512	1256	1136	5
bone64	6295	1942	2512	2512	1136	5
bone128	12586	3884	2512	2512	2272	5
bone256	25167	7769	5024	2512	2272	5
perlin1	112	30	550	550	550	16
perlin2	213	59	692	692	692	15
perlin4	415	120	874	874	874	12
perlin8	805	238	1100	1100	1100	10
perlin16	1573	477	1386	1386	1386	9
perlin32	3086	954	1746	1746	1746	8
perlin64	6081	1908	2000	2000	2000	7
perlin128	12016	3816	2772	2772	2772	7
perlin256	23789	7629	3492	3492	3492	6

Table 2: Some information on the solved meshes

solving time of the simulation with the real bone mesh is varying, see Figure 6. This effect arises from the varying number of iteration steps to convergence with the different meshes. With the Perlin meshes (Figure 7) the solving time decreases with the number of nodes used. This can be explained by the preconditioner that works better with finer resolutions of the same mesh.

The weak scalability benchmark also shows that our offset approach is about 30% faster than the hash approach. We have ported the offset approach back to the CPU implementation. This increased the performance wrt. the original code by 34%.

4.2. Strong scalability tests

In strong scalability tests the problem size stays constant as the processor number is increased. Good strong scalability can be expected if the overhead introduced by the parallelization, in particular communication, is little. Since the computational volume decreases as the number of computing elements increases the overhead will prevail sooner or later.

We ran the strong scaling benchmark with the `perlin1` mesh, see Table 2. This mesh has about $112 \cdot 10^6$ degrees of freedom and $30 \cdot 10^6$ bone elements. Initially one computing node is used. This means that the test was run on one GPU or on 16 CPU cores, respectively. The solution time was 296.9 seconds on the GPU and 1560 seconds on the CPU. These timings correspond to the numbers of the weak scalability benchmark displayed in Figure 7.

In Table 3 we list execution times and efficiencies for the strong scalability test. In Figure 8 we depict the graphs of the execution times. We see that the algorithm running on the CPUs scales almost perfectly. With the largest node number 128 we still obtain an efficiency of 72%. The dashed lines in Figure 8 indicate ideal speedup, i.e., times proportional to the reciprocal of

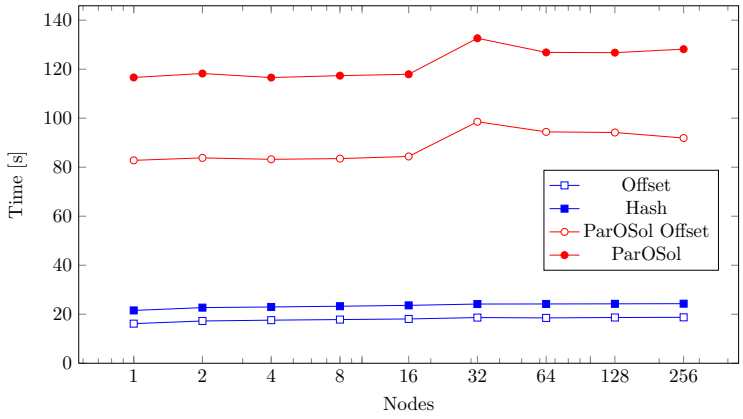

Figure 4: Time per iteration for boneX meshes.

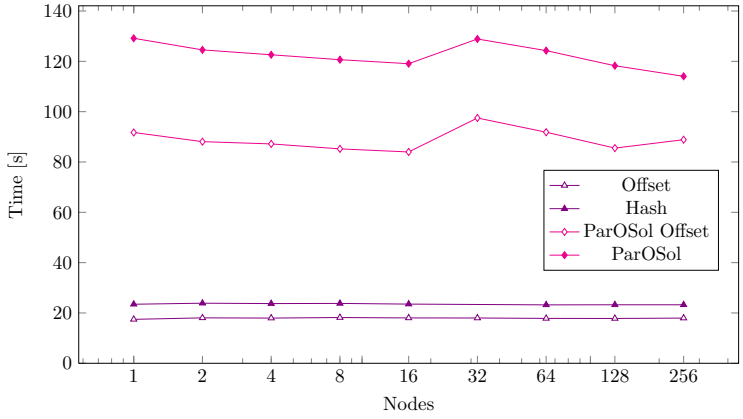

Figure 5: Time per iteration for perlinX meshes.

the node number. In contrast, the algorithm running on the GPUs does not perform as well. Here, the efficiency drops to 72% already with 4 nodes. This is due to the 5-fold faster execution of the numerical operations on the GPU which entails that the communication overhead accounts for a much bigger fraction of the overall execution time. Note that the structure of the communication is the same in both CPU and GPU implementations and that the GPU communicates via the CPU. Nevertheless, the GPU runs are faster than the CPU runs on up to 32 nodes.

In conclusion, if simulations are run in a clinical environment and fast response times are needed, then for a problem with a similar size as the `perlin1` mesh up the four GPUs can be used efficiently. The solution time is then mere 100 seconds and the parallel efficiency stays at 72%. In order to obtain a similar response time 16 CPU nodes with 16 cores are needed. For execution times of around a minute, 8 GPUs or 32 CPU nodes are required. Note that in our application most of the CPU cores are not utilized in the GPU runs.

5. Conclusions

We have presented a GPGPU implementation of the highly parallel solver ParOSol for voxel-based μFE bone analysis. Also the new solver CUDA-ParOSol is based on the conjugate gradient

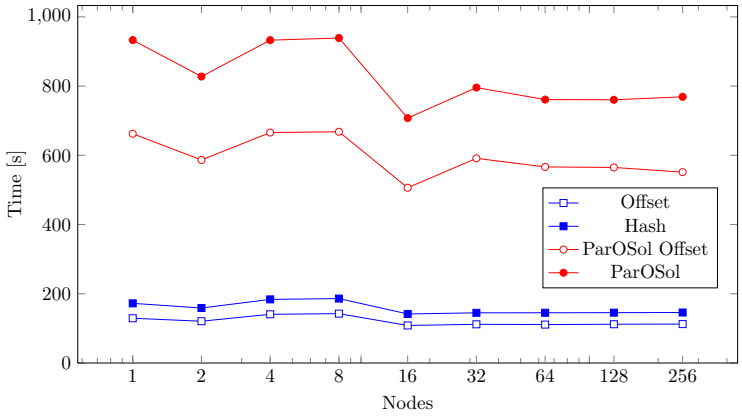

Figure 6: Time to solution for boneX meshes.

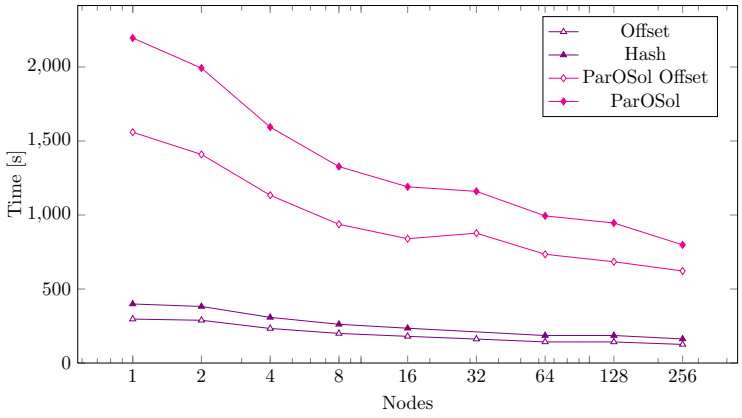

Figure 7: Time to solution for perlinX meshes.

algorithm complemented by a geometric multigrid preconditioner. The GPU implementation differs from the orginal ParOSol mainly in the way the voxel data are stored and accessed. Instead of searching for the vertex keys in a vector of Morton keys the offsets of the vertex keys are stored elementwise. The locality of the z-curve makes it possible to store most of the offsets in a single byte. Additionally, a clever strategy to read and write data of overlapping elements avoids race conditions among threads of a thread block.

We present a GPU implementation of ParOSol that exhibits a five-fold speedup over an equal number of CPU nodes with 16 cores. The GPU implementation scales ideally in the weak sense and reasonably well in the strong sense. Introducing the modifications for the GPU into the CPU code results in an improvement in run times of 35% compared with the original code. This code scales very well in both the weak and strong sense.

Appendix A. Differences of element vertex numbers for solid cubes

We want to determine the fraction of elements that have a difference smaller than 2^b among its node numbers in the Morton ordering for solid cubes. The three-dimensional Morton key μ of the node with coordinates $(x, y, z) \in \mathbb{Z}_+^3$ is obtained by interleaving the bits of the coordinates [5,

gpus	Offset	Ideal	Efficiency	ParOSol_Offset	Ideal	Efficiency
1	296.9	296.9	100%	1560.	1560.	100%
2	173.2	148.5	86%	780.8	780.1	100%
4	103.6	74.23	72%	395.2	390.1	99%
8	65.10	37.11	57%	197.8	195.0	99%
16	46.00	18.56	40%	99.59	97.5	98%
32	38.61	9.278	24%	51.29	48.76	95%
64	33.72	4.639	14%	27.85	24.38	88%
128	32.79	2.320	7%	17.04	12.19	72%

Table 3: Strong scaling test. Times to solution and efficiencies for the `perlin1` mesh.

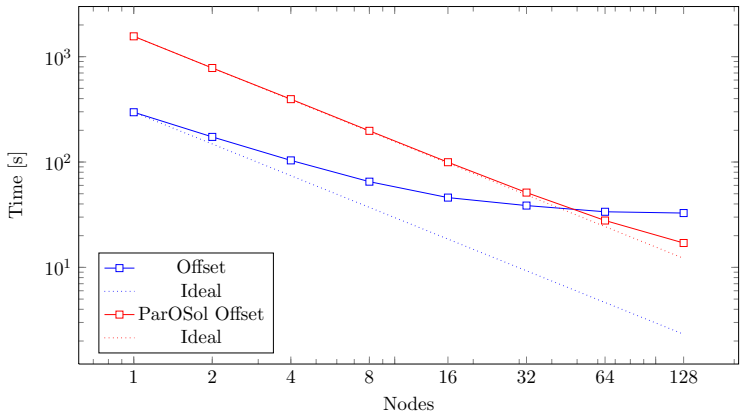

Figure 8: Strong scaling test. Times to solution for the `perlin1` mesh.

§7.2],

$$\mu = \mu(x, y, z) = \ldots z_3 y_3 x_3 z_2 y_2 x_2 z_1 y_1 x_1 z_0 y_0 x_0. \tag{A.1}$$

We are in particular interested in the cases $b = 8$ and $b = 16$, i.e., the number of bits that we use to store the offsets. We write $b = b_x + b_y + b_z$ where b_x, b_y, b_z denote the number of bits devoted to the x-, y-, and z-directions in the b lowest bits in the Morton ordering (A.1). For $b = 8$ we have $b_x = b_y = 3$, $b_z = 2$; for $b = 16$ we have $b_x = 6$, $b_y = b_z = 5$.

We can tile the solid cube by building blocks of $2^{b_x} \times 2^{b_y} \times 2^{b_z}$ elements. Due to the recursive structure of the Morton ordering the node numbers in each building block are equal, up to a block-dependent shift. Therefore, we can restrict our investigation to a basic building block.

The node (x, y, z) with local coordinates $(0, 0, 0)$ has the smallest Morton key among the nodes of an element, the node $(x+1, y+1, z+1)$ with local coordinates $(1, 1, 1)$ has the largest. Therefore, our investigate amounts to determine the number of elements of a building for which

$$\mu(x + 1, y + 1, z + 1) - \mu(x, y, z) < 2^b. \tag{A.2}$$

The desired fraction is this number divided by 2^b.

(1) If *each* of the three bit sequences $x_{b_x-1} \ldots, x_0$, $y_{b_y-1} \ldots, y_0$, and $z_{b_z-1} \ldots, z_0$ in (A.1) contain a zero bit then $\mu(x + 1, y + 1, z + 1)$ differs from $\mu(x, y, z)$ only in the b lowest bits, whence (A.2)

is satisfied. There are

$$\prod_{i\in\{x,y,z\}} (2^{b_i}-1) = (2^{b_x}-1)(2^{b_y}-1)(2^{b_z}-1)$$

such patterns in the basic building block.

(2) The bit patterns that are not covered in **(1)** have all ones in at least one of the three bit sequences. Adding one to that coordinate will entail a carry to bits with number b or higher. Let $w \in \{x, y, z\}$ be the coordinate associated with the b-th bit. (For $b = 8$ we have $w = z$ and for $b = 16$ we have $w = y$.) If one of the coordinates that are not w have all ones, then adding one to that coordinate will alter a bit with a number bigger than b such that (A.2) does not hold. If the ones are in the w coordinate then we distinguish the two cases where bit b is 0 or 1. If bit b is 1 then adding one to the w-coordinate introduces a carry at position $b + 3$ or higher. So, the only possibility for (A.2) to hold is when bit b is 0. Thus, we have the following patterns, displayed for the case $b = 8$,

$$\mu(x, y, z) = \ldots z_2|y_2x_2z_1y_1x_1z_0y_0x_0 = \ldots 0|y_2x_21y_1x_11y_0x_0,$$
$$\mu(x + 1, y + 1, z + 1) = \ldots 1|y'_2x'_20y'_1x'_10y'_0x'_0.$$

In order that $\mu(x+1, y+1, z+1)$ and $\mu(x, y, z)$ differ by less than 2^b the bits b and $b-1$ must not differ. Therefore, there must be a zero in the low order bits $0, 1, \ldots b_i - 1$ for $i \in \{x, y, z\} \setminus \{w\}$. There are

$$4 \cdot \frac{1}{2} \prod_{i\in\{x,y,z\}\setminus\{w\}} (2^{b_i-1}-1)$$

such patterns in the basic building block. The factor 4 reflects the four possible values for the bits $b-1$ and $b-2$. The factor $1/2$ stems from the requirement that bit $b+1$ needs to be 0.

Summarizing we have

Lemma. *The fraction of elements the node numbers of which differ by at most $2^b - 1$ is bounded below by*

$$l_{\text{bound}}(b) = \frac{1}{2^b}\left(\prod_{i\in\{x,y,z\}} (2^{b_i}-1) + 2 \prod_{i\in\{x,y,z\}\setminus\{w\}} (2^{b_i-1}-1)\right) \tag{A.3}$$

For the cases $b = 8$ and $b = 16$ we get the lower bounds

$$l_{\text{bound}}(8) = \frac{165}{256} \gtrapprox 0.644, \qquad l_{\text{bound}}(16) = \frac{61473}{65536} \gtrapprox 0.938.$$

It can be verified by direct computation that these two bounds are exact.

Acknowledgments

The computations on the Cray XK7 have been performed in the framework of a Development Project grant of the Swiss National Supercomputing Centre (CSCS).

References

[1] M. Adams, M. Brezina, J. Hu, and R. Tuminaro. Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *J. Comput. Phys.*, 188(2):593–610, 2003.

[2] M. F. Adams, H. H. Bayraktar, T. M. Keaveny, and P. Papadopoulos. Ultrascalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, 2004.

[3] P. Arbenz, G. H. van Lenthe, U. Mennel, R. Müller, and M. Sala. Multi-level μ-finite element analysis for human bone structures. In B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 240–250, Berlin, 2007. Springer. (Lecture Notes in Computer Science, 4699).

[4] P. Arbenz, G. H. van Lenthe, U. Mennel, R. Müller, and M. Sala. A scalable multi-level preconditioner for matrix-free μ-finite element analysis of human bone structures. *Internat. J. Numer. Methods Eng.*, 73(7):927–947, 2008.

[5] M. Bader. *Space-Filling Curves*. Springer, Berlin, Heidelberg, 2013.

[6] C. Bekas, A. Curioni, P. Arbenz, C. Flaig, G.H. van Lenthe, R. Müller, and A.J. Wirth. Extreme scalability challenges in micro-finite element simulations of human bone. *Concurrency Computat.: Pract. Exper.*, 22(16):2282–2296, 2010.

[7] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[8] D. Braess. *Finite elements: theory, fast solvers and applications in solid mechanics*. Cambridge University Press, Cambridge, 2nd edition, 2001.

[9] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *J. Parallel Distrib. Comput.*, 73(1):4–13, 2013.

[10] C. Flaig. *A highly scalable memory efficient multigrid solver for μ-finite element analyses*. PhD Thesis No. 20712, ETH Zürich, 2012.

[11] C. Flaig and P. Arbenz. A scalable memory efficient multigrid solver for micro-finite element analyses based on CT images. *Parallel Comput.*, 37(12):846–854, 2011.

[12] C. Flaig and P. Arbenz. A highly scalable matrix-free multigrid solver for μFE analysis based on a pointer-less octree. In I. Lirkov, S. Margenov, and J. Waśniewski, editors, *Large Scale Scientific Computing (LSSC 2011)*, pages 498–506. Springer, Berlin, Heidelberg, 2012. (Lecture Notes in Computer Science, 7116).

[13] M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, and M. G. Sala. ML 5.0 smoothed aggregation user's guide. Tech. Report SAND2006-2649, Sandia National Laboratories, 2006.

[14] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[15] D. Kellenberger. Bone structure analysis with GPGPUs. Master thesis, ETH Zurich, Computer Science Department, June 2013.

[16] S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM Trans. Graph.*, 25(3):579–588, 2006.

[17] Libnoise: A portable, open-source, coherent noise-generating library for C++, 2013.

[18] T.L. Mueller, A.J. Wirth, R. Müller, and G.H. van Lenthe. Computational bone mechanics to determine bone strength of the human radius. Transactions of the ORS annual meeting, vol 33. San Francisco, CA., 2008.

[19] NVIDIAs next generation CUDA compute architecture: Kepler GK110. White paper. NVIDIA Corporation, 2012. Available from http://www.nvidia.com/object/nvidia-kepler.html.

[20] The ParFE Project Home Page, 2013.

[21] The ParOSol Project Home Page, 2013.

[22] K. Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[23] W. Pistoia, B. van Rietbergen, E.-M. Lochmüller, C.A. Lill, F. Eckstein, and P. Rüegsegger. Estimation of distal radius failure load with micro-finite element analysis models based on three-dimensional peripheral quantitative computed tomography images. *Bone*, 30(6):842–848, 2002.

[24] B. van Rietbergen, H. Weinans, R. Huiskes, and B. J. W. Polman. Computational strategies for iterative solutions of large FEM applications employing voxel data. *Internat. J. Numer. Methods Eng.*, 39(16):2743–2767, 1996.

[25] D. Ruffoni, A. J. Wirth, J. A. Steiner, I. H. Parkinson, R. Müller, and G. H. van Lenthe. The different contributions of cortical and trabecular bone to implant anchorage in a human vertebra. *Bone*, 50(3):733–738, 2012.

[26] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, 2nd edition, 2003.

[27] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16:187–260, 1984.
[28] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, London, 2001.
[29] P. Varga, E. Dall'Ara, D. H. Pahr, M. Pretterklieber, and P. K. Zysset. Validation of an HR-pQCT-based homogenized finite element approach using mechanical testing of ultra-distal radius sections. *Biomech. Model. Mechanobiol.*, 10(4):431–444, 2011.
[30] A. J. Wirth, Th. L. Mueller, W. Vereecken, C. Flaig, P. Arbenz, R. Müller, and G. H. van Lenthe. Mechanical competence of bone-implant systems can accurately be determined by image-based micro-finite element analyses. *Arch. Appl. Mech.*, 80(5):513–525, 2010.
[31] G. Zumbusch. *Parallel Multilevel Methods.* Teubner, Wiesbaden, 2003.