Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations (Part 2: Double Precision GPUs)

Dominik Göddeke and Robert Strzodka
Applied Mathematics, Dortmund University of Technology, Germany
Max Planck Center, Max Planck Institut Informatik, Saarbrücken, Germany
http://www.mathematik.tu-dortmund.de/~goeddeke, dominik.goeddeke@math.tu-dortmund.de
http://www.mpi-inf.mpg.de/~strzodka/, strzodka@mpi-inf.mpg.de

In a previous publication, we have examined the fundamental difference between computational precision and result accuracy in the context of the iterative solution of linear systems as they typically arise in the Finite Element discretization of Partial Differential Equations (PDEs) [1]. In particular, we evaluated mixed- and emulated-precision schemes on commodity graphics processors (GPUs), which at that time only supported computations in single precision. With the advent of graphics cards that natively provide double precision, this report updates our previous results.

We demonstrate that with new co-processor hardware supporting native double precision, such as NVIDIA's G200 architecture, the situation does not change qualitatively for PDEs, and the previously introduced mixed precision schemes are still preferable to double precision alone. But the schemes achieve significant quantitative performance improvements with the more powerful hardware. In particular, we demonstrate that a Multigrid scheme can accurately solve a common test problem in Finite Element settings with one million unknowns in less than 0.1 seconds, which is truely outstanding performance. We support these conclusions by exploring the algorithmic design space enlarged by the availability of double precision directly in the hardware.

1 Introduction and motivation

We solve the Poisson problem $-\Delta\mathbf{u} = \mathbf{f}$ on a unitsquare domain $\Omega = [0,1]^2$ with Dirichlet boundary conditions. The Poisson problem is a very important prototypical representative of the class of elliptic Partial Differential Equations (PDEs), and it often appears as a sub-problem in realistic simulation codes, for example as the Pressure-Poisson problem when solving the Navier-Stokes equations in fluid dynamics with an operator-splitting approach. We use a regular subdivision scheme, so that the resulting mesh consists of $N = (2^L + 1)^2$ mesh points for a *refinement level* of $L = 1, \ldots, 10$. Bilinear, conforming Finite Elements of the Q_1 space are used to discretize the PDE on the mesh. Due to the regular structure of the mesh (and a linewise numbering of the unknowns), the resulting matrix is large and sparse, with exactly nine nonzero bands. This *logical tensorproduct structure* is independent of the location of the mesh points, and is preserved even for extremely deformed meshes occurring for instance in the context of r-adaptivity. The performance of low-level linear algebra routines consequently depends only on the special matrix structure and not on the underlying mesh. Thus, we assemble and store the matrix entries explicitly, even though for the simple unit square domain, applying only a matrix stencil would suffice. This allows us to estimate the performance of more complex, realistic grids based on the performance of this rather simple test case. For more details on the practical aspects of this approach where we cover realistic domains with patches of such a logical tensorproduct structure, we refer to a previous publication [2].

To measure accuracy, we evaluate the analytic Laplacian of a polynomial test function in the grid points and use the resulting coefficient vector as the right hand side of the linear system. We thus know the exact solution of the problem to be solved, and can calculate the error in the integral l_2 norm. According to FE theory, the error reduces by a factor of four (h^2 for the mesh width h) after refining all elements into four smaller ones.

The linear systems arising from the discretization are known to be very ill-conditioned. Table 1 (taken from the original paper [1]) illustrates that attempting to solve the system in single precision fails, while double

Level	Unknowns	single precision Error	Reduction	double precision Error	Reduction
2	25	2.391E-3		2.391E-3	
3	81	5.950E-4	4.02	5.950E-4	4.02
4	289	1.493E-4	3.98	1.493E-4	3.99
5	1,089	3.750E-5	3.98	3.728E-5	4.00
6	4,225	1.021E-5	3.67	9.304E-6	4.01
7	16,641	6.691E-6	1.53	2.323E-6	4.01
8	66,049	2.012E-5	0.33	5.801E-7	4.00
9	263,169	7.904E-5	0.25	1.449E-7	4.00
10	1,050,625	3.593E-4	0.22	3.626E-8	4.00

Table 1: Influence of solver precision on solution accuracy with increasing level of refinement (L).

precision suffices to reduce the error according to FE theory; and hence, to increase the result accuracy. With reduced computational precision however, further refinement even increases the error again, even though the solver claims to have converged well, requiring the same number of iterations as the double precision variant. This behavior is particularly bad, as the poor accuracy of the computed results can easily be overlooked, and the increased computational effort (note that $N = (L^2 + 1)^2$) is wasted. Moreover, it is difficult to notice this bad behavior without ground truth, because the solver still converges and reduces the residuals as expected.

2 Microbenchmarks

Figure 1 shows the performance of four prototypical linear algebra operations used in our solvers. We use the same input data as for the solver tests below, namely the linear system stemming from refining the underlying mesh 10 times ($L = 10$), yielding a vector size of $N = 1025^2 = 1,050,625$. We execute these benchmarks on a GeForce 8800 GTX (G80 chip) and the new GeForce GTX 280 (G200 chip). All computations are performed with CUDA 2.0 beta 2 on a Linux workstation.

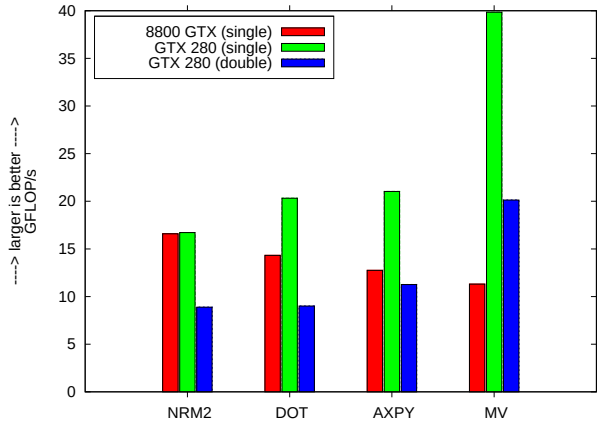

Figure 1: Performance (in GFLOP/s) of four important linear algebra kernels used in our solvers.

The operations `NRM2` and `DOT` are taken from NVIDIA's CUBLAS library. `AXPY` is our own implementation of the BLAS kernel of the same name, we do not need strided access and hence implemented our own version without the necessary conditionals, resulting in approximately 10% faster execution over `cublasS/Daxpy`. The performance of the `AXPY` kernel is completely limited by the available streaming bandwidth to off-chip memory and the ability of the hardware to hide latencies by keeping thousands of threads in-flight simultaneously. Comparing its the performance in single precision on the 8800 GTX and the GTX 280, we see the expected results: Due to the increased memory bandwidth of the newer board, single precision performance increases by roughly 2/3. The sustained bandwidth (data throughput) peaks at 76.5 GByte/s and approximately 126.2 GByte/s respectively (85% and 90% of the theoretical peak bandwidth). The

computation in single precision is faster by a factor of 1.86 than the computation in double precision, which is surprising as we expected exactly a factor of two: `AXPY` in double precision requires twice the bandwidth to perform the same number of floating point operations. The double precision version achieves 135.1 GByte/s, 96% of the theoretical peak, which continues this surprising trend. We do not know why on the newer hardware, `DOT` is faster than `NRM2`, as the dot product requires twice the bandwidth and should therefore take twice the time minus the overhead associated with the parallel reduction, which should be identical for both operations. The numbers clearly indicate some weirdness of the `NRM2` kernel, the comparison of `DOT` and `AXPY` is consistent.

As explained above, the matrix structure (nine bands at fixed offsets) is known a priori, and we can tailor the corresponding kernel specifically to this format. Listing 1 shows the corresponding CUDA code: We store the matrix bands as individual vectors of size N in memory, padded appropriately with zeros. We then compute the result vector row-wise, using one thread per row. Each thread reads nine distinct values from the matrix bands, and the reads can be perfectly coalesced into combined memory transactions by the hardware due to the alignment implied by the padding. We use the shared memory of each multiprocessor to coalesce the reads into the coefficient vector. This additionally enables (limited) data reuse of the coefficient vector in each multiprocessor, as the data needed for each row of the matrix can be directly reused by the threads that are scheduled onto the same multiprocessor.

Both effects significantly improve performance, and the performance increase on the GTX 280 is much higher than for the `AXPY` kernel that does not rely on shared memory for data reuse: In single precision, we reach almost a factor of four, and the peak double precision performance of the `MV` kernel is almost twice as high as in single precision on the G80-based board. The performance difference between single and double precision performance is almost exactly the expected factor of two.

As a side note, our alternative implementation using OpenGL and Cg is only achieving a twofold performance improvement on the new hardware.

More generally, our approach is inspired by viewing the shared memory as a software-managed cache: On modern CPUs, the hardware prefetching logic needs many cachelines to store data from the matrix bands. On NVIDIA's GPUs, it does not make sense to (manually) cache data that can be read in a fully coalesced way anyway; we ran some experiments and a modification of our `AXPY` kernel that stages reads to global memory through shared memory is consistently slower on both hardware generations. Consequently, it does not make sense for the `MV` kernel to read data through shared memory that is only used once and then discarded, so we use all available cache memory to allow data reuse in the coefficient vector.

These numbers clearly indicate that a mixed precision solution scheme is potentially very beneficial on the GTX 280 despite native double precision support.

A word of caution is required about the code shown in Listing 1. The kernel accesses the array `x` out-of-bounds, with a maximum offset of $M + 1$ ($M = \sqrt{N}$). In plain C on the CPU, this is a perfectly legitimate performance tuning technique, as we can usually rely on the fact that some garbage value read in from a memory location `x[i<0]` (or `x[i>N]`), multiplied by a zero from the padded bands gives zero. On the CUDA device however, this is not true. It took almost one week of debugging to track this error, in the process, we witnessed all kinds of weird side effects, for instance changing the linkage convention of some kernel to `extern C` caused previously working code to produce wrong results. After determining that the out-of-bound memory accesses caused the seemingly random errors, the fix was straight forward. In a first version, we zero-padded all arrays to an offset of $M + 1$, which yielded correct results but violated the coalescing rules, resulting in reduced performance. Zero-padding the arrays by rounding up the offset $M + 1$ to the nearest multiple of the block size (128 in the code example in Listing 1) thus turned out to be a good balance of trading additional memory for good performance.

```
// y=Ax, ||x||=||y||=||diag(A)||=n
__global__ void smv_row(float* y, float* x,
                        float* ll, float* ld, float* lu,
                        float* dl, float* dd, float* du,
                        float* ul, float* ud, float* uu,
                        int n, int m)  {
  extern __shared__ float  smv_cache[];
  int idx = blockDim.x*blockIdx.x+threadIdx.x;
  // runs from 0 to blockDim.x-1
  int lindex = threadIdx.x;
  float* Dcache = smv_cache;
  float* Lcache = smv_cache+blockDim.x+2;
  float* Ucache = smv_cache+2*(blockDim.x+2);

  // prefetch chunks from coefficient vector taking advantage of coalescing
  // each thread loads three elements,
  // the first and last one additionally load the border cases
  Dcache[lindex+1] = x[idx];
  Lcache[lindex+1] = x[idx-m];
  Ucache[lindex+1] = x[idx+m];
  if (lindex == 0) {
    // x_-1 in c_0
    Dcache[0] = x[blockDim.x*blockIdx.x-1];
    Lcache[0] = x[blockDim.x*blockIdx.x-m-1];
    Ucache[0] = x[blockDim.x*blockIdx.x+m-1];
   }
   if (lindex == blockDim.x-1) {
     // x_blockdim in c_blockdim+1
     Dcache[blockDim.x+1] = x[blockDim.x*(blockIdx.x+1)];
     Lcache[blockDim.x+1] = x[blockDim.x*(blockIdx.x+1)-m];
     Ucache[blockDim.x+1] = x[blockDim.x*(blockIdx.x+1)+m];
   }
   __syncthreads();
   // now, compute
   y[idx] = dd[idx]*Dcache[lindex+1] +
              (dl[idx]*Dcache[lindex] + du[idx]*Dcache[lindex+2] +
               ld[idx]*Lcache[lindex+1] + ll[idx]*Lcache[lindex] +
               lu[idx]*Lcache[lindex+2] + ud[idx]*Ucache[lindex+1] +
               ul[idx]*Ucache[lindex] + uu[idx]*Ucache[lindex+2]);
}

////////////////////////////////////////////////////////////////////

// kernel launch
block.x = 128; // or some value read in from benchmarking
int M = (int)sqrt((double)N);
grid.x = (int)ceil(N/(double)(block.x));
smv_row<<<grid, block, 3*(block.x+2)*sizeof(float)>>>
                  (y, x, LL, LD, LU, DL, DD, DU, UL, UD, UU, N, M);
```

Listing 1: Optimized row-wise matrix-vector multiplication using shared memory in CUDA. The matrix bands are stored in the arrays LL to UU.

3 Mixed precision iterative refinement

Until the introduction of the G200 architecture that provides double precision natively in hardware[1], the accurate solution of linear systems stemming from the discretization of PDE problems with Finite Elements was only possible through our proposed mixed precision iterative refinement approach or via emulated precision [1]. Native single precision yielded inaccurate results (cf. Table 1). The mixed precision solver for a linear system $A\mathbf{x} = \mathbf{b}$ basically comprises the following steps:

1. Compute $\mathbf{d} = \mathbf{b} - A\mathbf{x}$ in double precision.
2. Solve $A\mathbf{c} = \mathbf{d}$ approximately in single precision.
3. Update $\mathbf{x} = \mathbf{x} + \mathbf{c}$ in double precision.
4. Check for convergence and iterate.

With native double precision, we now have three different possibilities: We can run the entire solver in double precision (labeled `double-GPU`) on the device, we can offload only step 2 to the GPU and execute the outer loop on the CPU (labeled `mixed-CPU`, this variant has been possible before), or we can execute the mixed precision scheme completely on the GPU (labeled `mixed-GPU`).

[1] AMD's FireStream 9170 model provides native double precision, and has been available already in late 2007, but we did not have the opportunity to test our code on this hardware yet.

Independent of the choice of the inner solver, we confirm that we achieve identical error reduction rates for all three configurations, i.e. any performance gains of the mixed schemes over `double-GPU` do not compromise the final accuracy in any way. Additionally, the difference between the results obtained with a reference solver running in double precision on the CPU are in the noise, indicating that the hardware provides 'true' double precision (s53e11) instead of a composed format of two single precision values (s46e8).

4 Conjugate Gradient solver

The first solver we analyse is a standard unpreconditioned[2] Conjugate Gradient scheme, as a representative of Krylov subspace methods. We apply the solver to the linear systems arising from discretizations on refinement levels $L = 7, 8, 9, 10$, with $N = 16,641$, $66,049$, $263,169$ and $1,050,625$ unknowns respectively. The convergence criterion is set to reduce the initial defect by eight digits, and the inner solver is configured to gain two digits of accuracy. The convergence criterion is chosen such that convergence in the Finite Element sense is achieved, in other words, such that an error reduction rate of a factor of four is maintained up to $L = 10$. Table 2 lists the results[3]. The measured timings do include all transfers of right hand sides and iteration vectors, but do not include transfers of the matrices, because this is the setting we employ in our real applications [2].

	`double-GPU`				`mixed-CPU`				`mixed-GPU`			
L	#iters	time(s)	MFLOP/s	MByte/s	#iters	time(s)	MFLOP/s	MByte/s	#iters	time(s)	MFLOP/s	MByte/s
7	147	0.033	2012	13714	269	0.072	1723	5958	269	0.072	1720	5947
8	296	0.101	5233	35662	680	0.214	5720	19609	680	0.197	6205	21271
9	596	0.473	8970	61129	1597	0.842	13558	20328	1597	0.807	14138	48326
10	1202	3.041	11220	76463	3420	4.785	20328	69367	3420	4.513	21551	73541

Table 2: Number of iterations and performance results for the three variants of the Conjugate Gradient solver.

The baseline variant `double-GPU` shows the expected behavior of the Conjugate Gradient scheme, the number of iterations increases linearly in the mesh width h and doubles for each level of refinement. The measured MFLOP/s rates are in accordance to the benchmark results presented above.

As discussed in our original paper, the Conjugate Gradient algorithm reacts very delicately to the mixed precision approach. During its iterations, a search space of A-orthogonal directions is constructed, and this information is lost when the solver is restarted; resulting in a significant increase in the number of iterations required until convergence. We have designed an improved variant of the solver that maintains this search space during restarts [3], but at the time of writing, we have not ported this scheme to CUDA yet, as we want to concentrate on multigrid solvers, which are much more important for our research [2]. Another aspect is that the A-orthogonal directions are computed via dot products, and it is well-known that such operations on long vectors are very prone to accumulation of floating point errors. The mixed precision results above were obtained with our own dot product kernel that computes all intermediate sums on the GPU in single precision, but performs the final accumulation in double precision on the CPU. This is very beneficial, because a version that computes entirely in single precision needs approximately 100 additional iterations for the largest problem size.

Factoring these drawbacks out, the performance results are as expected: The mixed precision schemes are between 1.4 and 1.9 times more efficient (MFLOP/s-rates) as the native double precision variant, while requiring 2–3 times the amount of iterations. We want to gain eight digits in accuracy and the inner solver is configured to reduce the error by two digits, but the entire scheme needs five global correction iterations, which is one more than expected. We observed the identical behavior in our previous tests [1], with an

[2] Due to the regular refinement, preconditioning reduces to a uniform scaling and has no beneficial effect on convergence.

[3] We use an abstract performance and memory model, for instance, the matrix-vector multiplication $\mathbf{y} = A\mathbf{x}$, $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ is counted with $10N$ loads from and N stores to device memory: one load per matrix entry ($9N$, each value is used exactly once), one load per entry of the coefficient vector $\mathbf{x}$ (N, we abstract from data reuse through shared memory), and N stores for the result vector. The operation $\mathbf{y} = A\mathbf{x}$ is counted with $17N$ floating point operations.

equivalent implementation of `mixed-CPU` in the 'old-school' GPGPU (pre-CUDA) setting. Consequently, the `double-GPU` variant runs fastest, as it needs the fewest iterations. In the mixed precision versions, more than 97% of the arithmetic work is performed in single precision, and therefore the difference between performing the correction loop on the CPU (including all necessary data transfers of residuals and iteration vectors from device to host and vice versa) or on the GPU is rather small, the version `mixed-GPU` without data transfers executes roughly 5% faster.

5 Multigrid solver

For a Multigrid solver, the situation is fundamentally different: As we have demonstrated previously, using Multigrid as a preconditioner in a mixed precision setting has no negative effect on convergence, in fact, the convergence behavior is identical to executing the entire solver in double precision.

We continue to use the simple yet fundamental test case presented above. To reach convergence in the Finite Element sense, it suffices to configure the solver with two pre- and postsmoothing steps of a Jacobi smoother in a V-cycle. We employ the Conjugate Gradient algorithm presented above as coarse grid solver. The convergence criterion is set to reduce the initial residual by eight digits, and the mixed precision schemes gain two digits in single precision before an update step in double precision is performed. Table 3 lists the results. The measured timings do again include all transfers of right hand sides and iteration vectors, but do not include transfers of the matrices, because this is the setting we employ in our real applications [2].

	`double-GPU`				`mixed-CPU`				`mixed-GPU`			
L	#iters	time(s)	MFLOP/s	MByte/s	#iters	time(s)	MFLOP/s	MByte/s	#iters	time(s)	MFLOP/s	MByte/s
7	8	0.011	2264	14203	4*2	0.008	3024	10468	4*2	0.009	2788	9651
8	8	0.018	5302	33219	4*2	0.019	5124	17726	4*2	0.012	8086	27973
9	8	0.040	9354	58574	4*2	0.060	6605	22842	4*2	0.026	15179	52496
10	8	0.125	11960	74876	4*2	0.227	6920	23929	4*2	0.073	21406	74022

Table 3: Number of iterations and performance results for the three variants of the Multigrid solver. The notation 4*2 corresponds to four mixed precision iterations with two Multigrid preconditioning steps each.

As the Multigrid solver converges identically in all three configurations and independent of the problem size (refinement level L), the numbers indicate that even for one million unknowns, our Multigrid solver on the GTX 280 is partly bound in performance by the overhead of launching kernels and transferring data to and from the device rather than by computations or bandwidth: The number of necessary floating point operations quadruples with each refinement level, but it takes less than four times longer to solve the larger problems, so the metrics MFLOP/s and MByte/s increase rapidly (the exception is the `mixed-CPU` variant on $L = 10$). This is partly due to our implementation, we will address this in the following Section.

The mixed precision variant that uses the CPU to execute the correction iteration (`mixed-CPU`) is slower than executing the entire solver in double precision on the device (`double-GPU`). The stalls on the GPU due to data transfers to and from the CPU for the correction steps have a significant impact on performance because the solver executes so fast[4]. Equally important, double precision computations on the CPU are much more expensive than on the GPU. This is further underlined by the fact that using pageable instead of pinned memory on the host (implying lower transfer rates) does not substantially degrade performance.

Finally, the mixed precision scheme that executes entirely on the device is 1.7 times faster than the variant using double precision exclusively. This is a substantial performance improvement over the 'naive' approach of executing entirely in double precision to get accurate results. The raw performance of 21.5 GFLOP/s and more than 74 GByte/s is outstanding.

[4]In our abstract performance model, no floating point operations and no bandwidth are counted for device to host transfers and number format conversions, we are only counting 'mathematically required' operations.

6 Some remarks on performance

The underlying implementation of the Multigrid solver has not been fully tuned yet. All operations are implemented as individually tuned kernels, for instance matrix-vector multiplication, one Jacobi step, prolongation, restriction, norm computation etc. No effort has been made to fuse several operations into collective kernels, and consequently, performance is limited by the kernel launch overhead, in particular for the Conjugate Gradient scheme employed as coarse grid solver.

6.1 Performance comparison with previous generation hardware

Table 4 compares performance on the GTX 280 with the performance achieved on a GeForce 8800 GTX, which can only execute the `mixed-CPU` configuration. The code is identical, only the partitioning of the grid into thread blocks has been adapted to maximize performance.

	`mixed-GPU` GTX 280			`mixed-CPU` GTX 280			`mixed-CPU` 8800 GTX		
L	time(s)	MFLOP/s	MByte/s	time(s)	MFLOP/s	MByte/s	time(s)	MFLOP/s	MByte/s
7	0.009	2788	9651	0.008	3024	10468	0.015	1691	5853
8	0.012	8086	27973	0.019	5124	17726	0.033	2966	10262
9	0.026	15179	52496	0.060	6605	22842	0.107	3694	12776
10	0.073	21406	74022	0.227	6920	23929	0.440	3587	12404

Table 4: Performance comparison of the new hardware with the previous generation.

When executing the same configuration, `mixed-CPU`, we observe that the GTX 280 outperforms the previous hardware generation significantly for small problem sizes, while the improvement degrades to approximately 70% on the largest level of refinement. These results again confirm the reduced kernel launch overhead and the better coalescing capabilities on the new hardware, as the machine that hosts the GTX 280 does not provide a Gen2 PCIe interface, resulting in approximately the same transfer rates from host to device and vice versa as for the 8800 GTX. When taking advantage of the double precision capabilities of the G200 architecture and executing the `mixed-GPU` configuration, we observe a performance improvement of up to a factor of six, which is clearly superlinear. In particular, this means that the results from our microbenchmarks (cf. Figure 1) translate to the full application level.

6.2 Performance comparison with a CPU-based solver

Table 5 compares performance of an optimized conventional implementation executing entirely in double precision on a single core[5] of an Intel Core 2 Duo E6750 (2.66 GHz, 4 MByte L2 cache), in a machine with PC800 DDR2 memory. This is the fastest CPU and memory subsystem available to us at the moment.

	`double-CPU` Intel			`double-GPU` GTX 280				`mixed-GPU` GTX 280			
L	time(s)	MFLOP/s	MByte/s	time(s)	MFLOP/s	MByte/s	speedup	time(s)	MFLOP/s	MByte/s	speedup
7	0.021	1405	8012	0.011	2264	14203	1.9x	0.009	2788	9651	2.3x
8	0.094	1114	6352	0.018	5302	33219	5.2x	0.012	8086	27973	7.8x
9	0.453	886	5052	0.040	9354	58574	11.3x	0.026	15179	52496	17.4x
10	1.962	805	4592	0.125	11960	74876	15.7x	0.073	21406	74022	26.9x

Table 5: Performance comparison of the new hardware with a standard CPU.

While the GPU improves efficiency with increasing problem size, performance on the CPU quickly degrades as soon as the problem does not fit entirely into cache anymore. For the largest problem size, we observe an astonishing 27-fold speedup when using the faster `mixed-GPU` configuration, and a speedup of a factor of 16 for the `double-GPU` variant.

[5] We found it terribly hard to achieve strong scaling on multicore CPUs for BLAS Level 1 like operations as the ones we use for our Multigrid solver.

Anecdotally, we can state that a Multigrid solver that received the same amount of tuning (measured in the well-known metric 'working hours of a PhD student') achieves approximately 5 GFLOP/s ($L = 10$) on a single CPU of the NEC SX-8 supercomputer installation at the HLRS, Stuttgart, Germany. On the GTX 280, we achieve 21.5 GFLOP/s. This statement should of course be taken with a grain of salt.

6.3 Performance per Watt and performance per Euro

We conclude this paper with a short analyis of derived metrics. The arguments are prototypical in the sense that we do not fully use all resources, the CPU is idle while the GPU computes and vice versa. Real applications will have to implement a better heterogeneous scheduling scheme to fully exploit all resources [2]. The analysis is based on the best results achieved for the CPU (805 MFLOP/s) and the GTX 280 (21.5 GFLOP/s for the `mixed-GPU` variant). We only consider the largest problem size ($L = 10$).

The GTX 280 we use in these tests consumes 236 W under full load. We estimate the idle power consumption of our test system with 107 W, and 250 W under full load with both cores, 178.5 W under full load with one core. For the baseline workstation that uses only the CPU, we achieve approximately $805/(178.5) = 4.5$ MFLOP/s per Watt. The addition of a GPU as co-processor to the workstation[6] increases this value to approximately $21406/(178.5 + 236) = 51.6$ MFLOP/s per Watt, a factor of 11.5. This means that the observed absolute performance increase is so high that it still leads to significant improvements in the performance/Watt metric which has become very important both ecologically and economically ('green computing'). The savings are however substantially smaller (27x vs. 11.5x) than the improvements in raw performance.

A typical mid-range workstation that delivers the above performance costs approximately 800 EUR. The retail price for a GTX 280 board is 570 EUR at the time of writing. The baseline variant without the GPU co-processor board achieves approximately 1 MFLOP/s per Euro, the GPU-enhanced PC peaks at 15.6 MFLOP/s per Euro, a 16-fold improvement.

7 Conclusions and future work

We have extended our previous work on native- and mixed-precision schemes for the solution of PDE problems discretized with Finite Elements. The availablity of native double precision directly in CUDA-enabled devices allows a much broader exploration of the algorithmic design space.

A detailed performance analysis of the GTX 280 reveals that our Multigrid solver is still bound by the available memory bandwidth, but to a growing extend also by the overhead associated with performing computations on coarser grids.

Our results indicate that this is not a critical performance limitation. The achieved performance of less than 0.1 seconds to accurately solve a sparse linear system with one million degrees of freedom implies, analogously to Amdahl's Law, that local operations in our large-scale MPI-based solvers are now almost 'for free'. Much more performance can be gained by accelerating the parts of our solution scheme that must execute on the CPU as they need direct access to the interconnects such as Infiniband, or by increasing the accelerable fraction of the solver scheme with novel numerical algorithms. For details, see Figure 11 and the corresponding arguments in our application paper [2].

As raw performance does not tell the whole story, we evaluate our results in the derived metrics performance per Watt and performance per Euro, and we conclude that even though the achieved improvements are smaller, they are still substantial and economically relevant.

[6] One core is always busy, as it executes the entire control flow of the CUDA program. The second core is idle.

Acknowledgments

We would like to thank Sumit Gupta, David Luebke and Mark Harris from NVIDIA for providing both an early engineering sample to get this work started and the final board for detailed measurements; and for their support during the evaluation. Thanks to Dirk Ribbrock and Markus Geveler for helping with index battles during performance tuning. This work has been supported by the DFG in project TU102/22-1.

References

[1] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):221–256, 2007. Special Issue: Applied Parallel Computing.

[2] D. Göddeke, H. Wobker, R. Strzodka, J. Mohd-Yusof, P. McCormick, and S. Turek. Co-processor acceleration of an unmodified parallel solid mechanics code with FEASTGPU. *accepted for publication in the International Journal of Computational Science and Engineering*, 2008.

[3] R. Strzodka and D. Göddeke. Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 259–270, 2006.