Tuning a Finite Difference Computation for Parallel Vector Processors

Gerhard Zumbusch
Institut für Angewandte Mathematik
Friedrich-Schiller-Universität Jena
Jena, Germany
Email: gerhard.zumbusch@uni-jena.de

***Abstract*—Current CPU and GPU architectures heavily use data and instruction parallelism at different levels. Floating point operations are organised in vector instructions of increasing vector length. For reasons of performance it is mandatory to use the vector instructions efficiently. Several ways of tuning a model problem finite difference stencil computation are discussed. The combination of vectorisation and an interleaved data layout, cache aware algorithms, loop unrolling, parallelisation and parameter tuning lead to optimised implementations at a level of 90% peak performance of the floating point pipelines on recent Intel Sandy Bridge and AMD Bulldozer CPU cores, both with AVX vector instructions as well as on Nvidia Fermi/ Kepler GPU architectures. Furthermore, we present numbers for parallel multi-core/ multi-processor and multi-GPU configurations. They represent regularly more than an order of speed up compared to a standard implementation. The analysis may also explain deficiencies of automatic vectorisation for linear data layout and serve as a foundation of efficient implementations of more complex expressions.**

***Keywords*-finite differences; vectorisation; GPU computing; cache aware algorithms; automatic tuning**

I. Introduction

We make the following contributions: We consider a simple Finite Difference numerical discretization scheme for the solution of differential equations. We propose an interleaved data layout of the Finite Difference grid points especially suited for vector instructions based on memory aligned vector load and store operations. We develop highly tuned implementations of Finite Difference stencil computations for single CPUs with SSE and AVX vector instructions with a single core efficiency of $91\% - 98\%$ peak floating point performance on recent CPU architectures by Intel and AMD. A parallel OpenCL implementation along the same lines achieves up to $90\% - 92\%$ peak performance on single Nvidia Fermi GPUs. All numbers are with respect to the available independent or fused-multiply-add (FMA) floating point pipelines. Furthermore, we present numbers for parallel multi-core/ multi-processor ($85\% - 94\%$ peak performance) and multi-GPU configurations ($75\% - 81\%$). These may serve as efficient node implementations for even larger distributed memory parallel systems. We discuss and compare the contributions of the different tuning techniques, some of them in contrast to 'conventional wisdom' about efficient GPU code. By an analysis of a simple Finite Difference stencil, we obtain efficient implementation techniques and upper performance limits also applicable to more complex numerical expressions.

By the introduction of a new vector instruction set (single instruction multiple data parallelism) for x86 architecture CPUs, the vector length increases from 128 bit SSE to 256 bit AVX vectors, i.e. from 4 to 8 single precision numbers (float). There is a roadmap to even larger vectors. Other CPUs like Intel Larrabee/ MIC provide long vectors already, in this case 512 bit vectors of 16 floats. In GPU computing, vector lengths of 16 or 32 floats are common. They can be combined to virtual vectors of length 256 (AMD) or 1024/2048 (Nvidia) by hardware multi-threading (simultaneous multi-threading or hyper-threading). Automatic vectorisation of loop and array expressions in Fortran and C has been developed successfully for classic style vector computers. Compiled codes were able to achieve almost peak performance for vector operands in main memory or in large vector registers. However, current architecture's memory, caches or GPU local processor memories do not provide enough bandwidth for the vector instruction pipelines any more. Algorithmic modifications are needed to reduce memory traffic and to feed the vector units by data placed in registers. Further issues are instruction parallelism for long pipelines and data parallelism for multiple execution units, cores, processors and GPUs.

II. Time stepping

A. Model Problem

We consider a one dimensional constant coefficient 3-point Finite Difference stencil computation of the type

```
for ($t$=0; $t<T$; $t$++) // iterations/ time
  for ($i$=0; $i<N$; $i$++) // space
    $u_i^{t+1} = (1-2r)\ u_i^t + r\ (u^t_{(i-1) \bmod N} + u^t_{(i+1) \bmod N})$
```

with spatial index i, left and right neighbours $i-1$ and $i+1$, and an iteration index t. The coefficients r and $(1-2r)$ remain fixed and can be treated as constant. Furthermore we apply periodic boundary conditions, that is $u_i^t = u_{i+N}^t$. The iteration loop represents an explicit time stepping scheme for a parabolic equation (heat conduction) or an iterative solver of damped Jacobi type for linear equation systems.

The stencil requires two add and two multiplication operations per grid point and per iteration, i.e. four floating point operations. Processors with independent add and multiply pipelines need at least two effective cycles per grid point. Processors with a fused-multiply-add (FMA) pipeline only need three effective cycles per grid point, namely one add, one multiply and one FMA. We will call an implementation 'efficient', if the measured number of processor instruction cycles is close to the theoretical number of $2NT$ respectively $3NT$ cycles. The floating point performance is calculated on the base of $4NT$ operations.

B. Performance of a Standard Implementation

A standard implementation will use two linear stride-1 spatial vectors u^t and u^{t+1} and swaps the role of the vectors after each time step. Vectorisation can be accomplished by compilers (Gnu gcc 4.6.2, Intel icc) for common loop expressions. Alternatively, vector intrinsics can be used in more difficult cases. With a vector length l, values of index $j = [i, \dots, i+l-1]$ are assembled in a vector register. If the addresses of u_j^t and u_j^{t+1} are multiples of l times the size of float, vector load u_j^t and store u_j^{t+1} are said to be aligned. However, vector loads u_{j-1}^t and u_{j+1}^t are now unaligned and are more expensive with respect to memory traffic.

```
loop (float *u, float *v, int n) {
  u[-1] = u[n-1];
  u[n] = u[0];
  for (int i=0; i<n; i+=1)
    *(vec*)&v[i] = stencil_vec (
       load_unaligned (u[i-1]), *(vec*)&u[i],
       load_unaligned (u[i+1]));
}
```

Unaligned instructions in registers are usually not available and can be constructed by a sequence of operations implementing a register shift/ rotate. Unaligned memory access is expensive, if available. Some PowerPC Altivec vector instruction sets require auxiliary load and register shift instructions to deal with non-aligned memory access.

Some performance numbers can be found in Figs. 1 and 2 for data placed in main memory, cache, or registers. The numbers reflect wall clock time with SSE, AVX, and FMA intrinsics in CPU code and native optimisation (except non vectorised values)), Nvidia Cuda 4.2, 64 bit Linux.

SIMD computing in registers, with sufficient loop unrolling leads to numbers close to peak performance. However, the computation is based on such small data sets that the result is usually considered useless. Caches are not fast enough to fill floating points pipelines. Now, the algorithm is memory bound. CPU main memory bandwidth throttles down computation below 6GFlop/s (GF) single and 3GF double precision, such that instructions improvements like vectorisation have a limited effect below 10% speedup for code optimised otherwise. Hence, (automatic-) vectorisation of a standard implementation is useful for data in cache only.

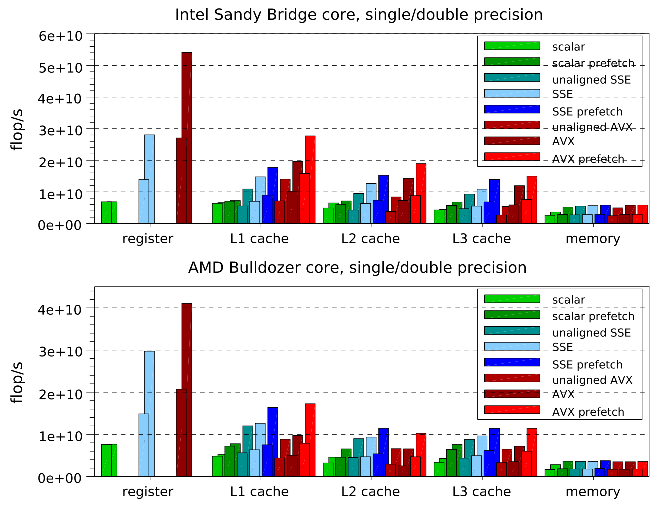

Figure 1. Performance of time-stepping implementations with respect to data placement and vector length, in single (back columns) and double (front columns) precision. Single core Intel Sandy Bridge i7-2600K and AMD Bulldozer FX-8150.

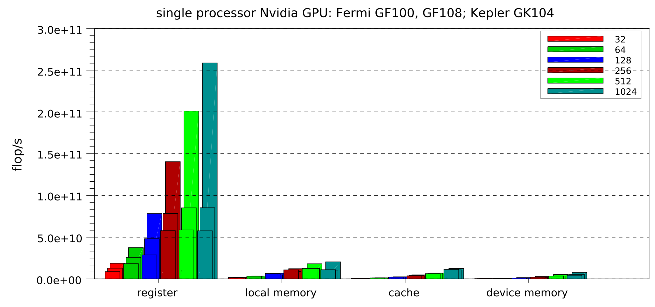

Figure 2. Performance of time-stepping implementations with respect to data placement and vector length, single multi-processor, Nvidia Fermi GTX 480 (front columns, GF100, 32 shader), GT 540M (middle columns, GF108, 48 shader), and Nvidia Kepler GTX 680 (back columns, GK104, 192 shader).

Vectorisation on GPUs is in fact essential: First, Nvidia floating point units are organised in vectors of size 32 (warp). Second, hardware multi-threading breaks down longer virtual vectors into multiple vectors of warp 32. Cached and un-chached memory vector load and store operations are blocking for hundres of cycles. Hardware muliti-threading with large thread numbers help to hide this memory latency. However, computing in memory is again bounded by $4-5.5$GF single precision per processor. Note that there are two versions of the Fermi micro architecture, Cuda compute capability 2.0 and 2.1. High performance GPUs are based on the 32 shader model, while later models of lower-end cards with lower processor numbers feature 48 shaders with similar compute in memory performance per processor, but an almost 50% improved compute in register rate, see Fig. 2. The first generation Nvidia Kepler (compute capability 3.0) raises the number of shaders to 6 groups of 32, with register layout and memory performance still comparable to Fermi

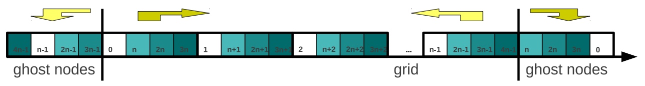

Figure 3. Vector load and store instructions can be aligned in memory, using an interleaved data layout of $4n$ grid points as vectors of length $l = 4$. The boundary ghost node vectors are rotated.

2.0. Computing in register on Kepler GPUs is accelerated by a factor of three compared to Fermi, but computing in memory is still comparable.

Note that AMD Bulldozer architecture implements AVX vectors as a combination of two SSE vectors internally, such that a single core can issue either two independent SSE instructions or one AVX instruction per cycle, see [1], [2]. This results in 33.6GF single core peak performance (30GF measured) for independent add and multiply instructions of both SSE and AVX type vectors with exclusive access to the shared floating point unit. Fused-multiply-add instructions (SSE and AVX vectors) double the performance to 67.2GF peak, of which 41GF can be achieved by the finite difference code.

C. *Interleaved Data Layout*

Intel x86 architecture allows for non-aligned vector load and store instructions, however with a performance penalty of higher memory access traffic compared to aligned vector instructions. GPU architectures can tolerate some non-aligned instructions, but honor block-wise (coalesced) memory accesses.

In order to avoid unaligned vector instructions, we change the data layout from the linear consecutive order to a permutation thereof, see Fig. 3. Grid points are stored with an increment of the vector length l. The grid is split into l pieces, which are interleaved. Using aligned vector instructions results in standard stencils, but applied to l different partitions of the grid. The performance in cache improves, see Fig. 1.

Further code optimisation prefetches the consecutive loads of u_j^t, such that one effective aligned vector load is done per grid point instead of three. Note the implementation of periodic boundary conditions.

```
loop_prefetch (vec *u, vec *v, int n) {
  vec u0 = rotate_right (u[n-1]);
  vec u1 = u[0];
  u[n] = rotate_left (u1);
  for (int i=0; i<n; i++) { // unroll space loop
    vec u2 = u[i+1];
    v[i] = stencil_vec (u0, u1, u2);
    u0 = u1; u1 = u2;
  }
}
```

Results are marked as 'prefetch' in Fig. 1. Using a 128 bit or 256 bit SIMD vector data type `vec`, vector intrinsics are used to implement `stencil_vec` $(1-2r)u_1 + r(u_0+u_2)$ (with or without FMA) and `rotate` by a single SSE shuffle or a sequence of AVX instructions.

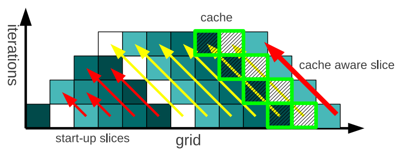

Figure 4. Time-skewing: Space-time slices in the iteration space lead to data re-use. If two old slices fit into cache, we obtain a cache aware algorithm.

The algorithm is memory bound, which can also be verified by the aid of CPU performance counters with tools like 'likwid' [3]. Note that loop unrolling and data which fit into cache are essential to the success of vectorisation.

The ratio of register performance to fastest cache/local memory performance $(2 - 2.4)$ is even worse for GPUs $(4.7 - 6.8)$, see Fig. 2. On GPUs currently there is no direct counterpart to a CPU cache to improve memory bandwidth. The GPU caches are designed as hotspot caches instead.

III. Space-Time Slicing

The algorithm so far is memory bound. The performance is critical for grids larger than cache size. We can transform this into an instruction bound algorithm by a fusion of several time steps. There are many ways in the space-time domain to order the points (i, t), such that the data dependence is granted. A systematic way uses trapezoidal shapes constructed of diagonal slices in space-time, called time-skewing [4], [5], see Fig. 4. This is effective, if at least two preceding diagonals are placed in fast (cache) memory. The amount of work of the start-up slices is wasted. Note that a straightforward implementation trades in load/ store operations in cache against memory. The improvement is limited by the ratio of bandwidth of cache to bandwidth of memory.

A. *Wide Space-Time Slices and Vectorisation*

The first improvement of time-skewing is to widen the slices to spade shaped domains in space-time, such that several grid points are computed at once, see Fig. 5. Basically two previous diagonals and two new diagonals have to fit into fast memory. We obtain an algorithm with two cache loads and two cache stores, independent of the width w of the slice and the number of floating point operations. Wide time-slices feature a ratio of memory transfers to computation of $1/w$. The number of registers is an upper limit of the slice width for an efficient computation in register.

Next, we introduce vector instructions in the spade shaped domains in space-time. Here, again unaligned vector instructions are recommended. The modified data layout of Fig. 3 transforms the grid load and store operations into aligned instructions. Applied to the time-skewing it can be re-interpreted as a decomposition into l spatial sub-domains

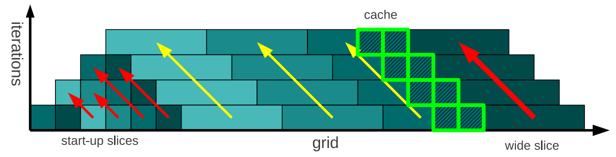

Figure 5. Time-skewing: Wider space-time slices for larger number of registers and improved computation to memory traffic ratio.

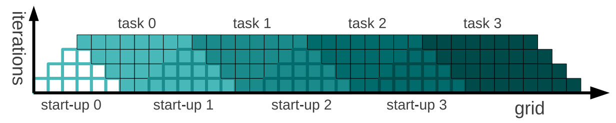

Figure 6. Decomposition of space-time slices into independent tasks. The start-up computations of each task re-do computations by a neighbour task and are parallel overhead.

with time-skewing in each sub-domain, see Fig. 6. An algorithm with e.g. a 3 grid point wide slice, aligned vector loads and stores, without outer loop and start-up looks like this:

```
slice (vec *u, vec *v, vec *a, vec *b, int m)
{
  a[0] = u[0]; a[1] = u[1];
  vec u0 = u[2], u1 = u[3], u2 = u[4];
  for (int i=0; i<m; i+=2) { // unroll
    vec a0 = a[i], a1 = a[i+1];
    vec v0 = stencil_vec (a0, a1, u0);
    vec v1 = stencil_vec (a1, u0, u1);
    vec v2 = stencil_vec (u0, u1, u2);
    b[i+2] = v1; b[i+3] = v2;
    u0 = v0; u1 = v1; u2 = v2;
  }
  v[0] = u0; v[1] = u1; v[2] = u2;
}
```

Initial and final time step grids are named u and v and temporal storage of space-time diagonals are a and b, which preferably are placed in cache or fast memory. A complete implementation adds some code for start-up code to fill a and boundary conditions. Furthermore a space loop is needed with a sliding window of u and v together with swapping the role of a and b. We end up with two temporary storage vector loads and two vector stores per slice and per time step, independent on the width of the slice.

B. Parallelisation

The next step of code optimisation is parallelisation. This is always an option, even for non-optimised single core code. Memory bounded algorithms may be harder to parallelise efficiently on shared memory architectures. Memory bandwidth does not scale. However, in our case the time slicing algorithm is instruction bound and easy to parallelise. The concept of Fig. 6 has already be used for vectorisation.

A distributed memory implementation (message passing) requires an exchange of start-up data. In Tab. II numbers of a shared memory OpenMP implementation demonstrate a perfect scaling on some server CPUs, see also Fig. 7. Small degradations of strong scaling can be observed on consumer type systems with less scalable main memory. The parallel memory system can easily sustain the low main memory demands of the time skewing algorithm.

Note that two AMD Bulldozer cores (one module) share one floating point unit, such that the parallel speedup of 8 cores i.e. 4 modules is just 4. Experimentally, 8 SSE enabled cores in 4 modules show minor advantages to 4 AVX cores, probably due to mapping threads to modules and cores.

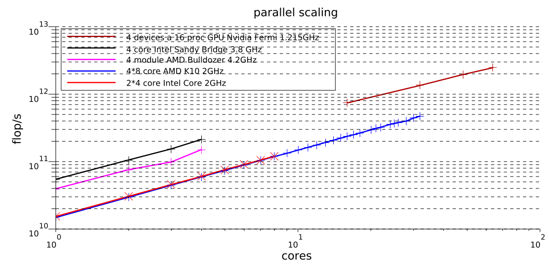

Figure 7. Performance as a function of processor numbers. A multi-GPU implementation with 1 to 4 GPUs with 16 processors each is compared to 4, 8, and 32 core CPU systems.

C. GPU implementation

The GPU implementation of the vectorised wide-time-slice algorithm in OpenCL follows the lines of the parallel CPU implementation of Sec. III-A. The OpenCL computational kernel can be translated one to one to Nvidia Cuda and performs almost identically on Nvidia GPUs. A common programming pattern in OpenCL (and Cuda) is the use of local (shared) memory to prefetch data and share data between threads together with fast synchronisation within a processor [6]. This way e.g. vector rotate can be easily implemented and we can develop an unaligned vector version of Sec. III-A. However, local memory is prohibitively slow compared to registers, see Fig. 2. Note that Cuda compute capability 3.0 of Nvidia Kepler offers faster vector rotate (warp shuffle) operations. An alternative way is to implement the aligned, CPU-like algorithm, with very long vectors. This is in fact preferable, given the low bandwidth of local memory [7].

Time-slicing was developed as a cache-aware algorithm with a fast cache. However, the fast GPU local memory and the L1 cache are even smaller than the total capacity of the registers and are not useful in our context. The large number of GPU registers allows for a large slice width w, such that computing in device memory (plus L2 cache) is still acceptable. We have to make sure to use efficient coalesced, aligned memory transfer only. The resulting algorithm does not have branch divergence, uses aligned memory access only, with one working-group per processor, no local memory and no thread synchronisation.

Table I
WORD SIZE OF DEVICE MEMORY TRANSFERS. TOTAL MEMORY BANDWIDTH IN GBYTE/S AND PERFORMANCE IN GFLOP/S.

system	peak	float	float2
1/2 GTX 590	1244GF	710.8GF	742.6GF
GF110 Fermi	163.9GB/s	88.1GB/s	95.3GB/s
GTX 680	3250GF	1035GF	1346GF
GK104 Kepler	192.3GB/s	110.3GB/s	143.4GB/s

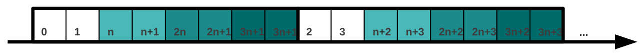

Figure 8. Interleaved data layout of $4n$ grid points for coalesced 'float2' load and store operations. Vectors length $l = 4$.

The 63 registers per thread of Nvidia Fermi and first generation Kepler GPUs allows for a large slice width. Using all of them limits the number of threads per working group to 512 on Nvidia Fermi and 1024 on Nvidia Kepler. We keep the number of grid partitions low and equal to the number of GPU processors. Hence there is only one job per processor. Furthermore, the occupancy is 50% at most, again in contrast to recommended programming patterns. The start-up overhead of time-slicing is large, due to large vector length. This can only be compensated by large problem sizes.

The implementation performs efficiently on Nvidia Fermi GPUs. However, the efficiency drops substantially on the first generation Fermi Kepler GPUs, see Tab. I. The device memory bandwidth does not scale like the floating point performance for Kepler. The code is memory bound again. In order to optimise memory bandwidth at low occupancy, the number of memory load and store operations may be reduced [8]. For example 64-bit memory accesses by loads and stores of 'float2' values instead of 32-bit 'float' values improve the effective memory bandwidth, see Tab. I. This way a 32-element coalesced vector instruction accesses 256 bytes in a single instruction. Larger memory transfers like 128-bit 'float4' introduce additional overhead and do not pay off currently.

Note that the interleaved data layout of Sec. II-C is further permuted such that two consecutive 'float' numbers are stored in one 'float2' value and the next pair of values is $l * 8$ bytes away, see of Fig. 8 .

D. Multi-GPU implementation

The multi-GPU implementation requires data transfer between the separate GPU device memories. We use a single data exchange of size $2Tl$ (vector length l) at the beginning of the time-slice algorithm in order to fill the start-up part of the grid. Data transfer goes through main memory and is initiated by CPU. Cuda and OpenCL have a different memory model. In the presence of multiple devices, there does not seem to be a good way to allocate memory on a specific OpenCL device like in Cuda. However, managing the OpenCL devices in separate contexts we loose the capability to transfer data between the devices directly. The difficult transfers introduces some parallel overhead, visible in the parallel GPU scaling. Nevertheless, we obtain roughly 2.4TFlop/s on a single PC with four GPU devices.

Similar discussion for distributed memory architectures have led to improvements of DMA data transfer ('GPUDirect') between network interface cards and GPUs. Hence it is possible to fuse MPI message passing with Cuda copy to the GPU device memory without host memory access on certain hardware.

E. Parameter Tuning

The resulting algorithm has several parameters, which are the number of time-steps T in time skewing, the slice width parameter w which indicates the size of the spade domain, the grid size, the vector length l (if configurable), and additional unrolling of the time-loop. Parameters depend on the vector instructions sets available, its number of registers and the relative size of (L1) cache. However, we employ parameter tuning to find the parameter sets.

On x86 platforms, generally a slice width w of 7 points with 28 flops, 2 loads and 2 stores and 9 registers in space is optimal due to the limited number of 16 addressable (YMM) floating point registers, see Fig. 9 (left). Note that there are larger register files of 144 (Sandy Bridge) or 160 (Bulldozer) floating point vectors [1], [2], which are used internally for register renaming due to out of order execution. A smaller slice width is less efficient due to a lack of instruction level parallelism and relatively small ratio of instructions to memory accesses. Larger slices cause register spilling, which again increases memory traffic. The optimal number of iterations in Fig. 9 highly depends on the number of processors involved and the processor's parameters. Small iteration counts lead to small gains of the cache-aware space-time slicing, large iteration counts introduce large start-up overhead. Furthermore, cache size limits the iteration count, if temporary data needs to fit into cache. Cache effects are clearly visible at 250 iterations at the limit of L1 cache size of 32 kbytes and at 2000 iterations at the L2 cache size of 256 kbytes on Intel Sandy Bridge. The effect is also visible at the L1 size of 16 kybtes of AMD Bulldozer at 125 iterations, while the L2 cache is already too large.

A similar account for Nvidia Kepler parameter tuning is in Fig. 9 (right). There are 63 registers available per thread, such that the optimal slice width is at 48. This already includes register spilling of values not needed in the innermost loop. No cache effects are visible that limit the number of iterations. Device memory is used as temporary storage. The iteration count is limited by the increasing startup-overhead of space-time slicing.

The problem size, i.e. the number of grid points does influence the efficiency of the code. The finite difference

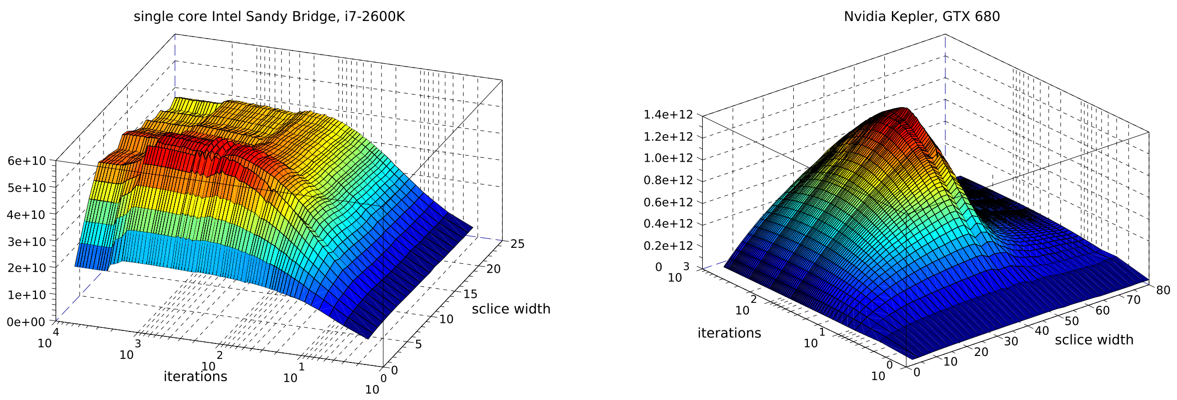

Figure 9. Performance of the space-time slicing in Flop/s. Numbers of a single core Intel Sandy Bridge i7-2600K processor and a Nvidia Kepler GTX 680 GPU. Performance is shown as a function of the number of iterations T and of the slice width.

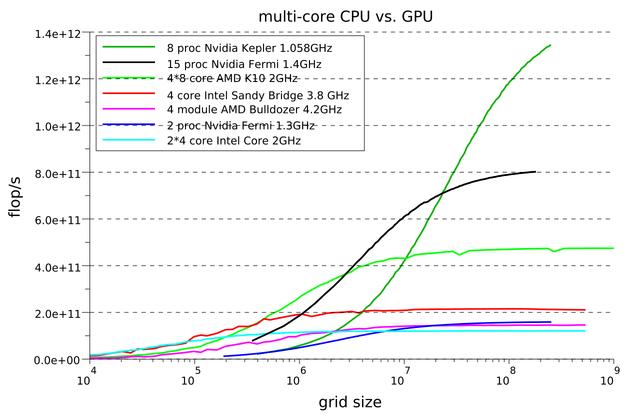

Figure 10. Performance as a function of grid size. Multi-core multi processor configurations compared to a single GPU.

algorithm is linear in the number of grid points. Larger problems require more computing time such that initialisation and start-up overhead can be compensated better, see Fig. 10. Problem size is limited by host or device memory size only. Larger numbers of processors increase the overhead, which can be compensated only at larger problem sizes.

GPUs in general involve much processor and thread parallelism, with even larger overhead. If one adds data transfer to and from a GPU, situation would be worse. However, we assume that the algorithm is just one building block of a larger application. Data is already in place and distributed in the multi-GPU case. A similar assumption is usually true in the analysis of distributed memory parallel algorithms. The accelerator programming model, which takes into account data transfer to and from the GPU does only make sense for off-loading large amounts of work.

F. Single Precision Results

The performance numbers for optimal parameters are assembled in Tab. II (left). We find single core performance well above 90% peak performance single precision, both with older generation CPUs and SSE vectors and with current CPUs and longer AVX vectors. GPU and AMD Bulldozer use FMA instructions ($3NT$ cycles), whereas other x86 CPUs have independent add and multiply pipelines ($2NT$ cycles). Data resides in main memory (CPU) or device memory (GPU). Wall clock times and theoretical cycle counts are taken.

G. Double Precision Results

Now we consider double precision (64 bit) floating point arithmetic instead of single precision. In the case of CPUs, two 'double' values fit into a 128 bit SSE vector and 4 values into a 256 bit AVX vector. Arithmetic and memory vector instructions run at the same speed as in the single precision case, but operate on half the number of values. Both theoretical and experimental numbers in Tab. II (right) are roughly halved.

The situation is completely different in the GPU case. All 'consumer' grade GPUs have much less double precision performance, if at all. For Nvidia Fermi capability 2.0 this means a reduction factor of $1/8$, Fermi 2.1 factor $1/12$, and Kepler 3.0 factor $1/24$. The memory performance is still the same, such that the experimental efficiency at this low floating point performance is almost optimal and above 97%.

The Nvidia Tesla series based on Fermi capability 2.0 offers much more double precision performance at a reduction factor of $1/2$. However, the double numbers occupy two 32 bit registers. The optimal slice width w must be reduced from 48 to 22 for 63 32-bit registers per thread. This implies a lower efficiency of 78% as in the single precision case. Note that the numbers already include the larger device memory, such that the problem size even grows compared to other Fermi numbers. Furthermore, the Tesla device offers ECC memory correction. Using this option reduces the effective memory bandwidth and the available memory size at the same time substantially, both reducing

Table II
PERFORMANCE OF VECTORISED AND PARALLELISED SPACE-TIME SLICING. NUMBERS IN GFLOP/S (GF) AND RELATIVE TO PEAK PERFORMANCE AND THEORETICAL CYCLE COUNTS.

configuration	peak	code	ops.	cycles	peak	code	ops.	cycles
	single precision				double precision			
Intel i7-2600K (Sandy Bridge)								
1 core, AVX, 3.8GHz	60.8GF	58.3GF	95.9%	95.9%	30.4GF	27.8GF	91.4%	91.4%
4 cores, AVX, 3.8GHz	243.2GF	215.1GF	88.4%	88.4%	121.6GF	107.6GF	88.5%	88.5%
AMD FX-8150 (Bulldozer)								
1 core, AVX FMA4, 4.2GHz	67.2GF	41.3GF	61.5%	92.2%	33.6GF	21.1GF	62.8%	94.2%
4 modules, SSE2 FMA4, 4.2GHz	268.8GF	151.9GF	56.5%	84.8%	134.4GF	76.0GF	56.5%	84.8%
AMD Opteron 6128 (K10)								
1 core, SSE2, 2GHz	16GF	14.8GF	92.6%	92.6%	8GF	7.2GF	89.5%	89.5%
4*8 cores, SSE2, 2GHz	512GF	473.5GF	92.5%	92.5%	256GF	230.6GF	90.1%	90.1%
Intel Xeon E5405 (Core)								
1 core, SSE2, 2GHz	16GF	15.7GF	98.1%	98.1%	8GF	7.7GF	96.3%	96.3%
2*4 cores, SSE2, 2GHz	128GF	120.5GF	94.1%	94.1%	64GF	59.4GF	92.8%	92.8%
Nvidia GTX 680 (GK104 Kepler)								
8*(6*32), 1.058GHz	3250GF	1346GF	41.4%	62.1%	135.7GF	87.9GF	64.8%	97.2%
Nvidia GT 540M (GF108 Fermi)								
2*(1.5*32), 1.344GHz	258GF	159.0GF	61.6%	92.4%	21.5GF	14.0GF	64.9%	97.4%
Nvidia GTX 590 (GF110 Fermi)								
16*32, 1.215GHz, 1/2 card	1244GF	742.6GF	59.7%	89.5%	155.5GF	103.4GF	66.5%	99.7%
2 devices, 1 card	2488GF	1348GF	54.2%	81.3%	311GF	197.6GF	63.5%	95.3%
4 devices, 2 cards	4977GF	2471GF	49.6%	74.5%	622GF	373.9GF	60.1%	90.2%
Nvidia Tesla M2070 (GF100 Fermi)								
14*32, 1.15GHz	1030GF	630.1GF	61.2%	91.7%	515GF	268.9GF	52.2%	78.3%
Nvidia GTX 480 (GF100 Fermi)								
15*32, 1.4 GHz	1345GF	804.4GF	59.9%	89.8%	168GF	111.8GF	66.5%	99.8%

the efficiency of the implementation.

IV. RELATED WORK

The problem of optimisation, tuning and vectorisation of finite difference implementations is quite old, since many of the early times numerical algorithms were based on finite differences. However, attempts to optimise for memory hierarchies, namely main memory and cache, started with a cache aware block tiling in space-time [9], the introduction of time-skewing [4], [5] and extensions to grid hierarchies [10] and a cache oblivious space-filling Z curve in space-time [11].

More recent is the overview [12], which e.g. describes a 3D time-skewing implementation at 37% peak performance on a 4.4GF peak performance, 2004 AMD Opteron processor. An automatic tuning of 3D time-stepping stencils on various double precision 10GF peak AMD and Intel cores show sequential 1.6GF and 11GF on 16 cores [13]. Another group [14] mentions 1GF on a 11.2GF peak double precision Intel 'Core' core, and 6.5GF on 8 cores for a 2D problem. A 3D higher order finite difference stencil in time-stepping is optimised with 130GF on previous generation Nvidia GPUs with 690GF peak in [15]. A coupled 3D Finite Difference Implementation on Nvidia Fermi based multi GPU cluster Tsubame 2.0 is discussed in [16].

V. CONCLUSION

We were able to develop highly efficient parallel, wide, vectorised time-slice implementations of the Finite Difference model problem for CPUs and GPUs. All optimisation techniques had to be combined. Just vectorisation or loop unrolling or time-slice did not improve the performance of a standard time-stepping scheme. The result relies on large numbers of registers to both sustain high processor performance and to hide main memory latency. Both x86 CPUs with SSE and AVX vectors and current Nvidia GPUs meet this requirement.

There are many claims comparing CPU and GPU performance, see [17]. A fair comparison may be based on a single PC or a server configuration, see table III. Other comparisons

Table III
COMPARISON CPU AND GPU. FLOP/S.

system	CPU	GPU	ratio
PC	multi-core CPU	one GPU	1:6.3
server	multi CPUs	one GPU	1:2.8
GPU server	multi CPUs	multi GPUs	1:5.2

Table IV
COMPARISON CPU AND GPU. GFLOP/S VERSUS ELECTRIC POWER OR HARDWARE PRICE OF THE DEVICE (WITHOUT HOST SYSTEM).

metric	AMD	Intel	Nvidia	Nvidia
	FX-8150	i7-2600K	GTX 590	GTX 680
	Bulldozer	Sandy Bridge	Fermi	Kepler
GF	151.9	215.1	1348	1346
GF/W	1.22	2.26	3.69	6.90
GF/$	0.62	0.68	1.93	2.69

take into account price or electric power, with or without a host system, see table IV. However, it is essential to compare implementations well tuned for each of the platforms.

Challenging generalisations of the model problem include two and three dimensional grids, longer, more complex, higher order, or variable coefficient difference stencils, systems of equations instead of a scalar value on a grid point and hierarchies of grids with mesh refinement and multigrid algorithms. The data access patterns are more complex and the computational work per grid point increases. Heavy computation on a grid point is favourable from a performance point of view. Time-slices can be generalised to 2D and 3D grid patterns. The question remains, whether the number of registers and the cache sizes are sufficient. In this sense, the performance numbers reported here are probably an upper bound for many of more difficult Finite Difference schemes. Furthermore, such an efficient single computing node implementation can be extended to even larger distributed memory parallel system code.

ACKNOWLEDGEMENT

The authors would like to thank DFG for partial support under grant SFB/TR7 “gravitational wave astronomy”, J. Treibig for a pre-release version of the tool “likwid”, J. Giesen and B. Brügmann for access to the AMD Magny-Cours and Nvidia M2070 systems respectively.

REFERENCES

[1] L. Gwennap, “Sandy Bridge spans generations,” *Microprocessor Report*, p. 8, Sep 2010, http://www.mpronline.com.

[2] D. Kanter, “Intel’s Sandy Bridge microarchitecture,” http://www.realworldtech.com/page.cfm?ArticleID=RWT091810191937, Sep 2010.

[3] J. Treibig, G. Hager, and G. Wellein, “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments,” in *Proc. PSTI2010*, San Diego CA, 2010.

[4] Y. Song and Z. Li, “New tiling techniques to improve cache temporal locality,” in *Proc. ACM SIGPLAN Conf. Programming Language Design Implementation, Atlanta*, 1999, pp. 215–228.

[5] J. McCalpin and D. Wonnacott, “Time skewing: A value-based approach to optimizing for memory locality,” Rutgers Univ., Tech. Rep. DCS-TR-379, 1999.

[6] B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL.* Morgan Kaufmann, 2012.

[7] V. Volkov and J. Demmel, “Benchmarking GPUs to tune dense linear algebra,” in *Proc. Supercomputing 2008*. IEEE, 2008.

[8] V. Volkov, “Better performance at lower occupancy,” GTC 2010 talk, 2010, http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

[9] G. Rivera and C. Tseng, “Tiling optimizations for 3D scientific computations,” in *Proc. Supercomputing 200*, 2000.

[10] C. Weiß, “Data locality optimizations for multigrid methods on structured grids,” Ph.D. dissertation, TU München, 2001.

[11] M. Frigo and V. Strumpen, “Cache oblivious stencil computations,” in *Proc. Supercomputing 2005*, 2005, pp. 361–366.

[12] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, “Optimization and performance modeling of stencil computations on modern microprocessors,” *SIAM Rev.*, vol. 51, no. 1, pp. 129–159, 2009.

[13] S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison, E. W. Bethel, and Prabhat, “A generalized framework for auto-tuning stencil computations,” in *Cray User Group Conference*, Atlanta, GA, 2009.

[14] M. Stürmer and U. Rüde, “A framework that supports in writing performance-optimized stencil-based codes,” Universität Erlangen-Nürnberg, Tech. Rep. 10–5, 2010.

[15] P. Micikevicius, “3D finite difference computation on GPUs using Cuda,” in *Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2*. ACM, 2009, pp. 79–84.

[16] T. Shimokawabe, T. Aoki, T. Takaki, A. Yamanaka, A. Nukada, T. Endo, N. Maruyama, and S. Matsuoka, “Peta-scale phase-field simulation for dendritic solidification on the Tsubame 2.0 supercomputer.” in *Proc. Supercomputing 2011*. IEEE, 2011.

[17] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, “Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU,” *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010.