Vectorized Higher Order Finite Difference Kernels

Gerhard Zumbusch[1]

Friedrich-Schiller-Universität Jena,
Institut für Angewandte Mathematik,
07743 Jena, Germany,
gerhard.zumbusch@uni-jena.de,
WWW home page: http://cse.mathe.uni-jena.de

Abstract. Several highly optimized implementations of Finite Difference schemes are discussed. The combination of vectorization and an interleaved data layout, spatial and temporal loop tiling algorithms, loop unrolling, and parameter tuning lead to efficient computational kernels in one to three spatial dimensions, truncation errors of order two to twelve, and isotropic and compact anisotropic stencils. The kernels are implemented on and tuned for several processor architectures like recent Intel Sandy Bridge, Ivy Bridge and AMD Bulldozer CPU cores, all with AVX vector instructions as well as Nvidia Kepler and Fermi and AMD Southern and Northern Islands GPU architectures, as well as some older architectures for comparison. The kernels are either based on a cache aware spatial loop or on time-slicing to compute several time steps at once. Furthermore, vector components can either be independent, grouped in short vectors of SSE, AVX or GPU warp size or in larger virtual vectors with explicit synchronization. The optimal choice of the algorithm and its parameters depend both on the Finite Difference stencil and on the processor architecture.

1 Introduction

Finite Differences are a classical numerical scheme for the solution of differential equations. However, the stencils on structured grids are also computationally efficient on current computers, which explains their widespread use in science.

By the introduction of a new vector instruction set for x86 architecture CPUs, vector length increases from 128 bit SSE to 256 bit AVX vectors, i.e. from 4 to 8 single precision numbers (float), with a road map to even larger vectors. Other CPUs provide long vectors already (Intel Phi 512 bit, 16 floats). In GPU computing, vector lengths of 16 to 64 floats are common, which can be combined to virtual vectors of length 256 to 1024 by hardware multi-threading. Automatic vectorization of loop and array expressions in Fortran style codes has been developed successfully for classic style vector computers. However, current architecture's memory, caches or GPU local processor memories do not provide enough bandwidth anymore. Algorithmic modifications are needed to reduce memory

traffic, streamline memory access and to feed the vector units by data placed in registers and memory closer to the processor. Further issues are to provide enough instruction parallelism for long pipelines and multi-threading.

We make the following contributions: We propose interleaved data layouts 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 CPU cores with SSE and AVX vector instructions on older and most recent CPU architectures by Intel and AMD. Furthermore, OpenCL and Nvidia Cuda implementations for AMD and Nvidia GPUs are presented, again organizing Finite Difference stencil computations as vector operations and adapting the data layout accordingly. By an analysis of simple Finite Difference stencils, we obtain efficient implementation techniques and upper performance limits also applicable to more complex numerical expressions.

2 Model Problem Finite Difference Stencils

We consider Finite Difference stencil computations on structured grids. The stencils represent discretized versions of constant coefficient second order differential operators. Both application of the operator within a linear equation solver or within a time stepping scheme are included. Furthermore, an iterative solver of Jacobi type can be implemented this way. We consider arbitrary approximation orders, spatial dimensions of the structured grid, and isotropic and anisotropic self adjoint operators. Isotropic stencils of order p in one to three dimensions with constant coefficients c_l can be written as

$$u_i^{\text{new}} = c_0 u_i + \sum_{l=1}^{p/2} c_l (u_{i-l} + u_{i+l})$$

$$u_{i,j}^{\text{new}} = c_0 u_{i,j} + \sum_{l=1}^{p/2} c_l (u_{i-l,j} + u_{i+l,j} + u_{i,j-l} + u_{i,j+l})$$

$$u_{i,j,k}^{\text{new}} = c_0 u_{i,j,k} + \sum_{l=1}^{p/2} c_l (u_{i-l,j,k} + u_{i+l,j,k} + u_{i,j-l,k} + u_{i,j+l,k} + u_{i,j,k-l} + u_{i,j,k+l})$$

and may approximate the isotropic Δ Laplace operator applied to the grid function u. Anisotropic operators represent linearly distorted versions of the operator, like the two dimensional version:

$$\begin{aligned} u_{i,j}^{\text{new}} = {} & c_{0,0} u_{i,j} + \textstyle\sum_{l=1}^{p/2} \left(c_{l,0} (u_{i-l,j} + u_{i+l,j}) + c_{0,l} (u_{i,j-l} + u_{i,j+l}) \right) \\ & + \textstyle\sum_{l=1}^{p/2} c_{l,l} (u_{i-l,j-l} + u_{i+l,j+l} - u_{i+l,j-l} - u_{i-l,j+l}) \\ & + \textstyle\sum_{l}^{p/2} \sum_{m=l+1}^{p/2} c_{l,m} (u_{i-l,j-m} + u_{i-m,j-l} + u_{i+l,j+m} + u_{i+m,j+l} \\ & \qquad - u_{i+l,j-m} - u_{i+m,j-l} - u_{i-l,j+m} - u_{i-m,j+l}) \end{aligned}$$

The three dimensional anisotropic stencil of the Laplace operator approximation is a collection of two dimensional stencils along each x_i, x_j coordinate area. The

shape of the three dimensional isotropic and anisotropic stencils are depicted in Fig. 1.

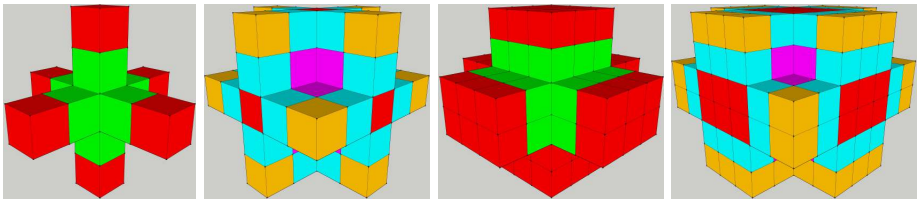

Fig. 1. Schematic 3D 4th order FD stencil, isotropic and anisotropic. Single stencils (left) and $3 \times 3 \times 2$ block (right).

The number of grid points and arithmetic operations, separated into adds and subs, multiplications (mul) and alternatively fused multiply-adds (fma) are summarized in Tab. 1. The number of fma operations equals the number of grid points, which increases with order and dimension. The number of multiplications equals the number of coefficients and is at most the number of adds.

Table 1. Finite Difference stencils of order p, number of floating point operations per grid point.

name	operator	load points	total flops	add	mul	fma
1D	D_{xx}	$1+p$	$1+\frac{3}{2}p$	p	$1+\frac{1}{2}p$	$1+p$
2D	$D_{xx}+D_{yy}$	$1+2p$	$1+\frac{5}{2}p$	$2p$	$1+\frac{1}{2}p$	$1+2p$
3D	$D_{xx}+D_{yy}+D_{zz}$	$1+3p$	$1+\frac{7}{2}p$	$3p$	$1+\frac{1}{2}p$	$1+3p$
anisotropic						
2D	$a_{11}D_{xx}+2a_{12}D_{xy}+a_{22}D_{yy}$	$(1+p)^2$	$1+\frac{13}{4}p+\frac{9}{8}p^2$	$2p+p^2$	$1+\frac{5}{4}p+\frac{1}{8}p^2$	$(1+p)^2$
3D	$a_{11}D_{xx}+2a_{12}D_{xy}+2a_{13}D_{xz}$ $+a_{22}D_{yy}+2a_{23}D_{yz}+a_{33}D_{zz}$	$1+3p+3p^2$	$1+\frac{21}{4}p+\frac{27}{8}p^2$	$3p+3p^2$	$1+\frac{9}{4}p+\frac{3}{8}p^2$	$1+3p+3p^2$

3 Target Processor Architectures

We will implement several Finite Difference algorithms for CPU and GPU (graphics processing unit) architectures. We consider single processor cores of x86 CPUs by Intel and AMD and GPUs by Nvidia and AMD, see Tab. 2 and 3. We consider the smallest independent processor unit (core), called 'streaming multiprocessor' by Nvidia, 'shader cluster' on AMD GPUs or 'module' for AMD Bulldozer. Most recent CPUs feature AVX arithmetic vector instructions, that is 256 bit vectors with 8 single precision (float) or 4-double precision values. Add and mul operations take two vectors to compute a result vector, fma takes three input vectors. Previous CPUs offered SSE vectors of half the size. The CPUs have independent floating point add and mul pipelines, with the exception of AMD Bulldozer. Hence the number of adds are an upper performance limit for the Finite Difference implementations, given that data flow between registers, caches and memory is fast enough.

Table 2. CPU and GPU processors, micro architectures and their single precision (32 bit float) performance. All numbers of a single processor core.

processor		performance				cache		
architecture	name	clock [GHz]	vector instructions	vector op/cycle	flop/s [GF]	L1 [kB]	L2 [kB]	shared LL [MB]
Intel Ivy Bridge	i5-3450	3.1/3.5	AVX	add + mul	56	32	256	6
Intel Sandy Bridge	i7-2600	3.4/3.8	AVX	add + mul	60.8	32	256	6
Intel Core	Xeon E5405	2.0	SSE4.1	add + mul	16	32	–	6
AMD Bulldozer	FX-8150	3.4/3.9/4.2	AVX, FMA4	fma	67.2	16	2048	8
AMD K10	Opteron 6168	1.9	SSE4	add + mul	15.2	64	512	6
AMD K8	Opteron 865	1.8	SSE3	1/2(add + mul)	7.2	64	–	1
Nvidia Kepler	GK110, Tesla K20c	0.705/0.758	32	6 fma	291.1	16 – 48	–	1/8
Nvidia Kepler	GK104, GTX 680	1.006/1.059	32	6 fma	406.5	16 – 48	–	1/8
Nvidia Fermi	GF108, GT 540M	1.344	32	3/2 fma	129	16 or 48	–	1/8
Nvidia Fermi	GF110, GTX 590	1.215	32	1 fma	77.8	16 or 48	–	1/8
Nvidia Fermi	GF100, Tesla C2050	1.15	32	1 fma	73.6	16 or 48	–	1/8
Nvidia GT200	GTX 260	1.296	32	1/4 fma	20.7	–	–	–
AMD South. Isl.	GCN, HD 7970	0.925	64	1 fma	118.4	16	–	1/8
AMD North. Isl.	VLIW4, HD 6990	0.83	64	1 fma	106.3	8		

GPU architectures are based on vectors of length 32 or 64 floats within a processor. Hardware multi-threading enables the program to combine the vectors transparently to larger virtual vectors of sizes 256 to 1024. The GPUs have fma floating point pipelines. Hence the number of fmas serve as an upper performance limit, like in the case of Bulldozer CPUs. We have listed the properties of a single CPU core and a single GPU processor, although usually chips and systems with different numbers of cores are available. Note that the double precision performance of the CPUs is half of the single precision in Tab. 2 and the GPU double precision performance is between 1/24 and one half of single precision.

Table 3. Double precision peak performance of a single GPU core.

processor		performance				
architecture	name	clock [GHz]	cores	vector length	vector fma/cycle	flop/s [GF]
Nvidia Kepler	GK110, Tesla K20c	0.705/0.758	13	32	2	97.0
Nvidia Kepler	GK104, GTX 680	1.006/1.059	8	32	1/4	16.9
Nvidia Fermi	GF108, GTX 540M	1.344	2	32	1/8	10.8
Nvidia Fermi	GF110, GTX 590	1.215	16	32	1/8	9.7
Nvidia Fermi	GF100, Tesla C2050	1.15	14	32	1/2	36.8
AMD South. Isl.	GCN, HD 7970	0.925	32	64	1/4	29.6
AMD North. Isl.	VLIW4, HD 6990	0.83	24	64	1/4	26.6

4 Scalar Cache Aware Algorithms and Data Layout

First of all we will discuss scalar versions of the Finite Difference algorithms before we turn them into the vectorized algorithms necessary to fully exploit the target processors. For reasons of simplicity, we ignore boundary conditions close to the border of the spatial grid and start-up procedures for algorithms with multiple time steps.

Naive implementations of a Finite Difference stencil will simply take a loop over all grid points and apply the stencil. Such an algorithm has to load the same

number of values from memory as number of fma operations are performed. This is a memory bandwidth bound algorithm in a range of about 1 GF. It is up to a good cache mechanism to make this efficient. However, even the bandwidth of the L1 cache does not match the demand of the floating point pipelines.

4.1 Sliding Window Algorithm

The memory hierarchy of main memory RAM, last level (LL) to first level (L1) caches and processor registers offers different memory capacity at vast differences of access bandwidth and latency. Only the bandwidth of the register file is able to fully match the processor floating point pipelines. An improvement of the naive Finite Difference implementation goes like this: Data re-use is explicitly organized in register space. A cache aware 1D space loop for $p+1$-point wide Finite Difference stencil (order p in Tab. 1) implementing a single time step may look like this:

```
r[0..p-1] = grid[0..p-1];                              // load memory
for (int x=0; x<stepx*(p+1); x=x+p+1) {
  for (int x0=0; x0<p+1; x0++) {                       // unroll loop
    r[(x0+p)%(p+1)] = grid[x+x0+p];                    // load memory
    grid[x+x0] = calc (r[(x0)%(p+1)..(x0+p)%(p+1)]); // store memory
  }
}
```

Values `r` and `c` are to be placed in registers, routine `calc` represents the inlined stencil and `grid` values u. For reasons of simplicity a single array u is used both input and output, over writing old values of u with new ones. The `x0` loop needs to be explicitly unrolled, at least for most of the compilers, such that index expressions for `r` can be removed. The code for example for a 3-point stencil (order $p = 2$) can be expanded to

```
r0 = grid[0]; r1 = grid[1];      // load memory
for (int x=0; x<stepx*3; x=x+3) {
  r2 = grid[x+2];                // load memory
  grid[x  ] = calc (r0, r1, r2); // store memory
  r0 = grid[x+3];                // load memory
  grid[x+1] = calc (r1, r2, r0); // store memory
  r1 = grid[x+4];                // load memory
  grid[x+2] = calc (r2, r0, r1); // store memory
}
```

by a source-to-source preprocessor. This way copying of registers can be avoided. Data re-use takes place via registers only. There is one memory load and one store per grid point, compared to e.g. $1 + p$ fma operations.

The sliding window algorithm is discussed by [1–8] (often in the GPU context) and can be generalized to higher dimensions based on the memory hierarchy: The inner most loop data re-use through the register file can be complemented by L1 and L2 caches for the surrounding loops. In this case $1 + d \cdot p$ fma operations have to be compared to one main memory load and one store per grid point and additional cache loads.

Table 4. Scalar sliding window algorithm for order p Finite Difference stencils. All memory loads expect for one can be cached.

name	register storage	load	store	fma
1D	$1+p$	1	1	$1+p$
2D isotropic	$1+2p$	$1+p$	1	$1+2p$
3D isotropic	$1+3p$	$1+2p$	1	$1+3p$
2D anisotropic	$(1+p)^2$	$1+p$	1	$(1+p)^2$
3D anisotropic	$1+\frac{21}{4}p+\frac{27}{8}p^2$	$(1+p)^2$	1	$1+\frac{21}{4}p+\frac{27}{8}p^2$

4.2 Time Slicing Algorithm

In case the ratio of arithmetic operations to memory operations is still too small, several time steps can be aggregated into the time slicing (or time skewing, temporal tiling) algorithm, see [9–12] and for more recent work [1, 5, 13–15]. Initially developed to perform most of the computations in cache rather than main memory, time slicing with wide slices does most of the computations in register with memory operations mainly to cache. A one dimensional version of time slicing with $p+1$ point stencils looks like this:

```
for (int x=(stepx-1)*s; x>=0; x=x-s) {
  r[0..s-1] = grid[x..x+s-1];                    // load memory
  for (int t=0; t<stept*p; t=t+p) {
    r[s..s+p-1] = grid[x+t+s..x+t+s+p-1];        // load cache
    r[0..s-1] = calc (r[0..s+p-1]);              // unroll
    grid[x+t+p..x+t+2*p-1] = r[0..p-1];          // store cache
  }
  grid[x+(stept+1)*p..x+stept*p+s-1] = r[p..s-1];// store memory
}
```

A spatial tile of size `s` is used to compute `stept` time steps at once. Additional input data for the inlined Finite Difference stencils `calc` is loaded from a section of `grid`, which is presumably in cache. This algorithm uses `grid` as input and output for u and as cached intermediate storage of the stencil halo zones. To make sure that `r` is mapped to registers and the number of s difference stencils are in fact unrolled, some compilers require explicit source code unrolling. Note that this implementation requires the tile size to be large enough $s \geq p$. Furthermore, an initialization of the time slices next to the border is needed, which is ignored here, see [8] for a 1D version with separate auxiliary storage.

In two dimensions the loop and storage structure is the similar. The grid pattern to compute a rectangle of $s_x \times s_y$ points is a rectangle of $(s_x+p)\times(s_y+p)$ points for the anisotropic stencils. The points to be loaded and stored in each time step are an old rectangle minus a new one, forming L-shaped domains. If we assume that data re-use of the innermost x-loop fits into L1 cache and the y-loop into L2 cache, one leg of the L-domain is mapped to L2 cache and the remaining rectangle to L1 cache. Some snapshots of a two dimensional scheme are depicted in Fig. 2.

```
for (int y=(stepy-1)*sy; y>=0; y=y-sy) {
  for (int x=(stepx-1)*sx; x>=0; x=x-sx) {
    load rectangle [0..sy-1]*[0..sx-1] at grid[y][x] from memory
    for (int t=0; t<stept*p; t=t+p) {
      load L-domain [0..sy+p-1]*[0..sx+p-1] minus [0..sy-1]*[0..sx-1]
           at grid[y+t][x+t] from caches
      calc rectangle [0..sy-1]*[0..sx-1]
      store L-domain [0..sy-1]*[0..sx-1] minus [p..sy-1]*[p..sx-1]
            at grid[y+t+p][x+t+p] to caches
```

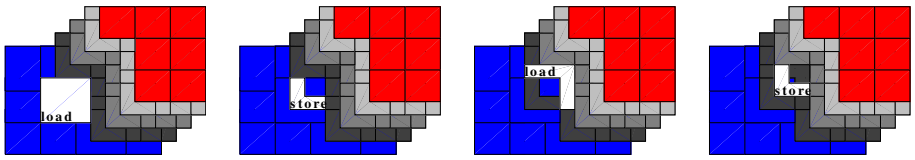

Fig. 2. Snapshots of the 2D time slicing algorithm, left to right, initial rectangle and later L-shaped pattern loads and the consecutive stores. The grid points are organized in the y-x plane. The time level is color-coded, values are shifted in x- and y-direction proportional to t. Initial values on the left (blue), results on the right (red).

```
    }
    store rectangle [0..sy-p-1]*[p..sx-p-1]
          at grid[y+stept*p][x+stept*p] to memory
  }
}
```

The storage patterns of the isotropic stencils are rectangles minus rectangles $p/2 \times p/2$ at the four corners. In three dimensions cubes and differences of cubes are mapped to three levels of cache. The memory access pattern change accordingly.

The computation to memory operation ratio is comparable to the sliding window algorithm, with the advantage to substitute main memory access by cache access, at least for large numbers of time steps.

Table 5. Scalar time slice algorithm for order p Finite Difference stencils with tile size $s \geq p$. All memory loads and stores can be cached.

name	register storage	load / store
1D	$s+p$	p / p
2D isotropic	$(s_x+p)(s_y+p)-p^2$	$(s_x+p)(s_y+p)-s_xs_y-\frac{3}{4}p^2$ $s_xs_y-(s_x-p)(s_y-p)-\frac{1}{4}p^2$
3D isotropic	$(s_x+p)(s_y+p)(s_z+p)-p^3$	$(s_x+p)(s_y+p)(s_z+p)-s_xs_ys_z-\frac{7}{8}p^3$ $s_xs_ys_z-(s_x-p)(s_y-p)(s_z-p)-\frac{1}{8}p^3$
2D anisotropic	$(s_x+p)(s_y+p)$	$(s_x+p)(s_y+p)-s_xs_y$ $s_xs_y-(s_x-p)(s_y-p)$
3D anisotropic	$\leq (s_x+p)(s_y+p)(s_z+p)$	$(s_x+p)(s_y+p)(s_z+p)-s_xs_ys_z$ $s_xs_ys_z-(s_x-p)(s_y-p)(s_z-p)$

5 Vectorization and Data Layout

All processor architectures under consideration are (parallel) vector processors. For reasons of performance, vector operations have to be used rather than scalar operations. A naive approach would be to block the innermost loop in the size of the vector length, exploit instruction level parallelism and use vector load operations not aligned to the length of a vector. However, unaligned memory access can imply more memory access if it crosses cache lines. Further, data re-use in registers is inhibited by this procedure.

An alternative approach would be to emulate unaligned memory access by loading consecutive aligned vectors and additional vector shift operations. This

is again expensive, partly because arbitrary vector shift instructions are not available. We will come back to this topic in the next subsections.

5.1 Vectors of Independent Tasks

One strategy to fully exploit the potential of vector instructions to consider them as SIMD parallel, independent tasks. Hence each vector component represents one grid, e.g. one octant of the full domain for vector length 8. Since a single step of the Finite Differences is fully parallel, a coupling just appears through the boundary conditions, which is cheap to implement.

The vector load and store operations have to be aligned for performance reasons. Hence the independent sub-grids have to be interleaved in memory accordingly. The scalar algorithm is simply vectorized by substitution of the scalar data type by the vector type and additional boundary procedures.

5.2 Shifted Vectors on CPUs

No changes in memory layout and a more efficient use of vector registers would be the introduction of vector shift instructions like

```
vec shift (vec a, vec b, int i) { // vector length n, 0<=i<=n
  return [a[i..n-1] b[0..i-1]];
}
```

A CPU of Ivy Bridge type for example is able to issue one floating point vector add, one mul and one permute operation per cycle in addition to integer and memory instructions. Unfortunately, the permute instructions have limited capabilities only:

The SSE instruction set offers the two vector argument instruction `_mm_shuffle`, which is sufficient for double precision, but in single precision can only be used for groups of two float2 values. The AVX instructions have a similar `_mm256_shuffle` instruction operating on each half-vector and `_mm256_permute2f128` to permute both half-vectors. This is again sufficient for four double values, such that one instruction can be used for shift one and the other for shift two, and the combination gives shift three. However, we still can only handle groups of float2 values. Hence, two or four interleaved parts of the domain are stored in memory in addition to some partial vector shift by one or two permute instructions.

Note that Intel Phi 512 bit vector instructions lead to flexible permute operations of groups of float4 `_mm512_mask_permute4f128`. The IBM Power AltiVec instructions set is more flexible in vector rotate, but leads to a multi instruction vector shift implementation.

5.3 Shifted Vectors on GPUs

In the GPU case vector shift can be implemented through Cuda `__shared__` memory respectively OpenCL `__local` memory. In warp synchronous parallel computing, i.e. short vectors of warp size, length 32 or 64 without explicit synchronization instructions, shared memory has to be marked as `volatile`.

Nvidia Kepler (capability $\geq$ 3.0) offers an additional vector shift for one vector without shared memory by `__shfl_up` and `__shfl_down` instructions. However, an implementation of a two argument vector shift still requires several instructions.

The hardware multi threading of GPUs allows for larger vector sizes. Vector shift can be implemented again through shared memory. Explicit synchronization is needed via Cuda `__syncthreads` or OpenCL `barrier`. Now the sliding window algorithms can be used again with shifted vectors [2].

6 Experiments

For reasons of comparison we perform experiments on a number of different generations of CPUs and GPUs. Since there are large absolute performance differences, in many cases we consider relative performance with respect to the speed of the add or the fma pipeline, i.e. peak performance of the Finite Difference stencil. The absolute numbers for single processor cores can be recovered by the number of operations per second times vector length in Tab. 2 and the number of operations per stencil in Fig. 1.

6.1 Compiler

The experiments depend on the quality of the code. We have used the compilers listed in Tab. 6. In most cases the gcc compilers gave superior x86 execution code. The larger the code and the number of unrolled instructions, the larger was gap between gcc version 4.7 and other compilers. Inspection of x86 assembly code generated by the different compilers gave no obvious explanation and we speculate that the different clock-per-instruction rates are due to the optimization level of instruction scheduling.

Table 6. List of C/C++ compilers in use. Sample compile time and performance comparison for a 3D 4th order example on an Intel Ivy Bridge CPU, compiler optimization options `-O3 -march=native`, code with AVX intrinsics.

name		version	source	compile	performance
C	C++			time [s]	[GF]
gcc	g++	4.7.0	FSF	40.907	6.09
gcc	g++	4.6.3	FSF	14.284	6.14
clang	clang++	3.2	llvm	55.899	3.59
icc	icpc	13.0.0	Intel	80.858	2.66

Source code loop unrolling and placement of vector elements in variables to be mapped to registers was done by a custom source-to-source preprocessor. The codes were written in C++ using SSE, AVX and FMA x86 vector intrinsics (if applicable), Cuda, and OpenCL respectively. Experiments were run on Linux Ubuntu 12.04 64-bit for x86 CPUs and Nvidia Cuda 5.0 for Nvidia GPUs. AMD GPUs were run with AMD APP OpenCL 2.7 on Linux Ubuntu 11.04.

6.2 Single Precision on a Single Core

The one dimensional experiments are summarized in the top rows of Figs. 3 and 5. The time slicing algorithm seems to be superior in most of the cases compared to the sliding window. The amount of required registers grows with the stencil order, such that the tile size is limited for time slicing, while the amount of data re-use grows for sliding windows. Hence there will be a turn over point at high order when sliding window is more efficient. However, this point is not reached in the 1D experiments. The 1D results show extremely high relative performance both for CPU and GPU. Some architectures have difficulties for higher order time slicing due to a shortage of registers compared to their floating point pipeline length. The detailed analysis in Fig. 4 shows that vector shift instructions can help reduce the tile sizes and the register pressure and improve higher order results.

In the 2D case, sliding window becomes superior for higher order stencils on CPUs and Nvidia GPUs, while time slicing is still better on AMD GPUs and generally for lower order stencils. Vector shift operations improve efficiency above 2nd order stencils and the optimal tile sizes are rather small, see Fig. 6. In the 3D case sliding window is superior to time slicing for all higher order stencils on CPUs and generally on GPUs. In the 3D case and in the higher order 2D case, large virtual vectors start to become superior to small warp synchronous vectors on GPUs.

Anisotropic stencils in Fig. 7 introduce substantially more operations with roughly the same memory traffic. However, the number of intermediate registers required grows, such that time slicing for higher order or 3D runs out of registers and almost constant performance sliding window starts to outperform time slicing. The problem of a small register file is much more pronounced in 3D, where performance always drops with increasing order.

6.3 Double Precision Arithmetic

A summary of the absolute performance in Tab. 7 and Tab. 8 show the performance drop with increasing spatial dimension, which is mainly due to cache access to fetch halo values in the outer loop directions. Furthermore, the time slicing seems to be superior for the 1D case, while sliding window is improving for higher dimensions.

Table 7. Absolute performance numbers of a single core CPU. Numbers of the time slicing algorithm in single and double precision arithmetic. Colored numbers indicate the shifted vector version. * marks SSE on AVX enabled CPUs.

processor		1D		2D		3D	
architecture	name	single [GF]	double [GF]	single [GF]	double [GF]	single [GF]	double [GF]
Intel Ivy Bridge	i5-3450	55.2	27.6	36.9	18.7	20.1	9.4
Intel Sandy Bridge	i7-2600	59.6	29.8	40.8	20.4	21.2	10.4
Intel Core	Xeon E5405	15.8	7.9	10.8	5.4	7.2	3.2
AMD Bulldozer	FX-8150	44.2	22.1	22.7*	11.3*	13.0	7.2*
AMD K10	Opteron 6168	15.1	7.5	9.1	4.5	5.9	2.3
AMD K8	Opteron 865	5.8	2.5	4.3	2.1	2.6	0.9

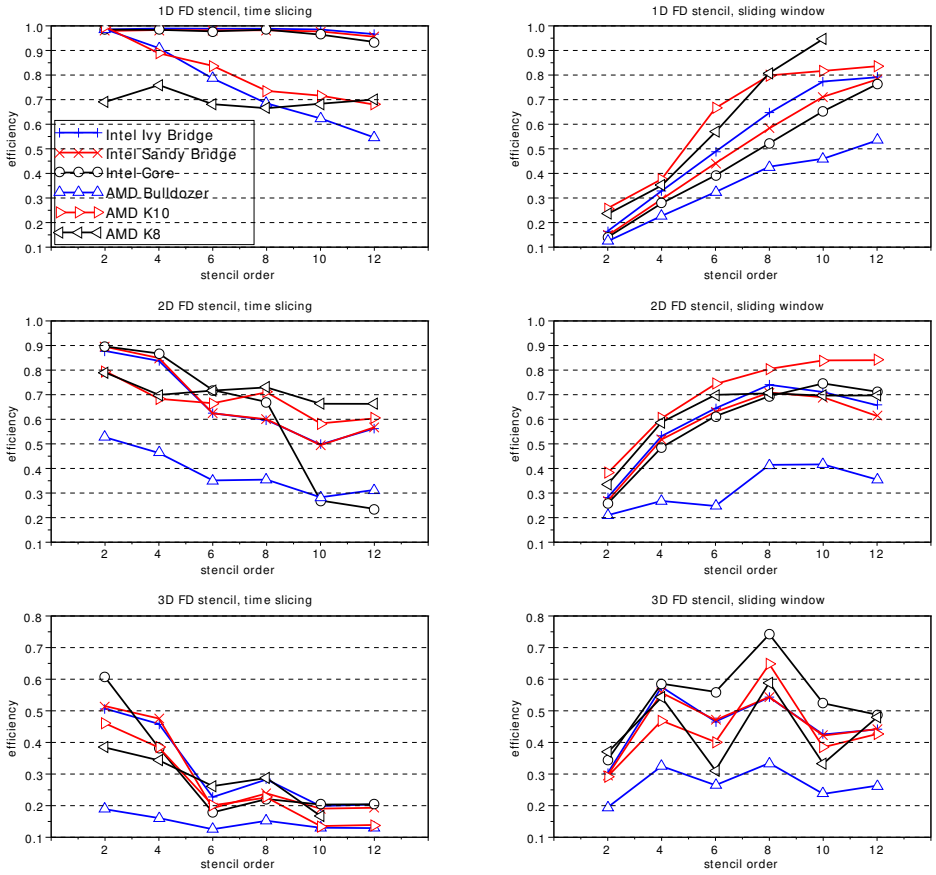

Fig. 3. Relative performance vs. stencil order of the sliding window and time slicing algorithms on CPUs.

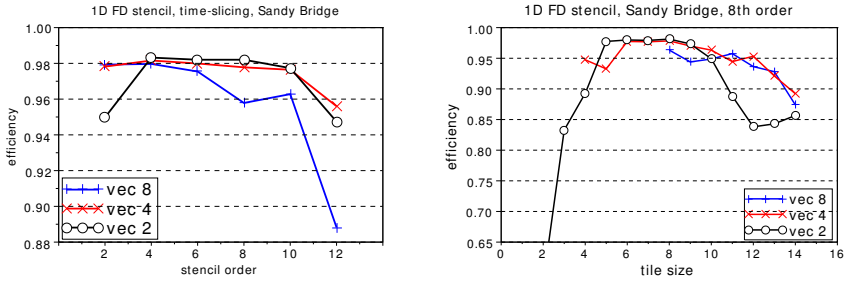

Fig. 4. 1D FD stencil. Time slicing on Sandy Bridge CPU. Independent vector components (vec 8), permute of float4 (vec 4) and permute+shuffle of float2 (vec 2). Relative performance vs. order p and tile size s.

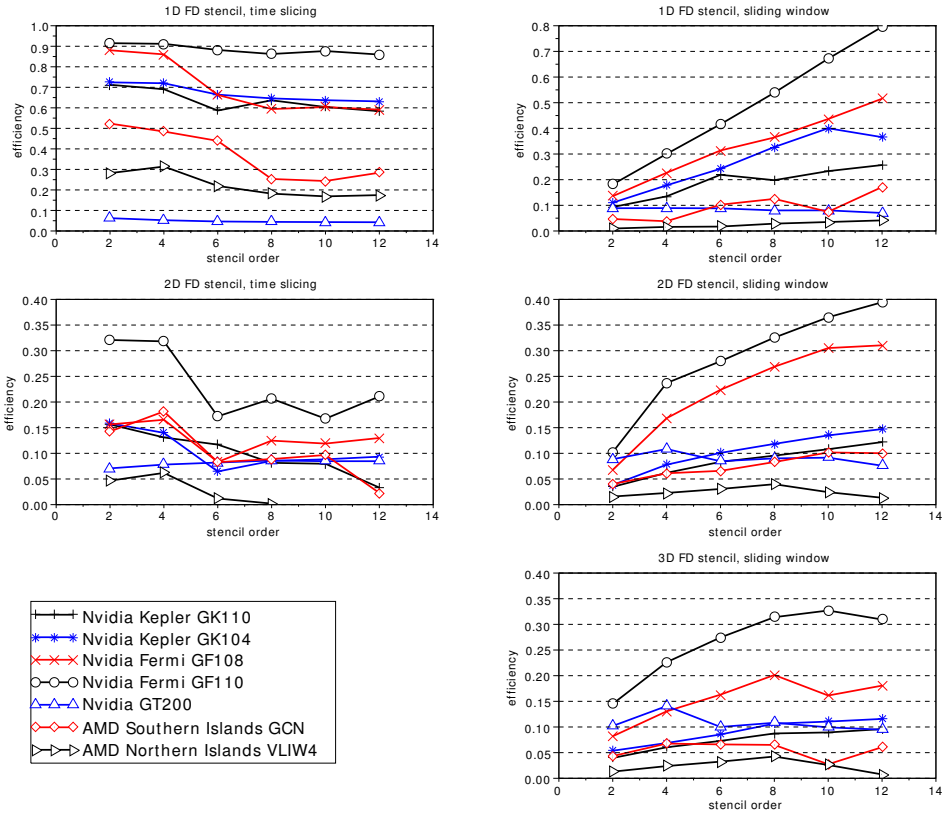

Fig. 5. Relative performance vs. stencil order of the sliding window and time slicing algorithms on GPUs.

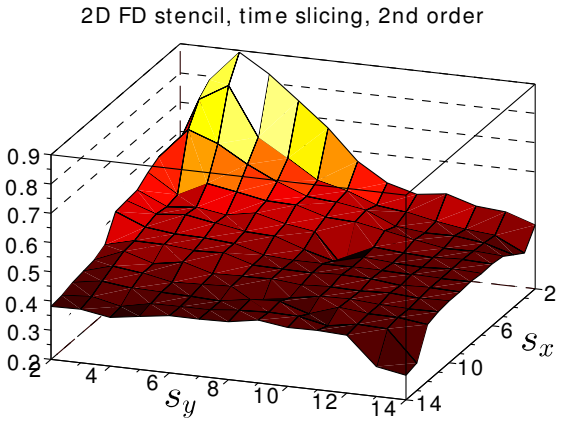

Fig. 6. 2D FD stencil. Time slicing on Sandy Bridge CPU. Relative performance vs.tile size $s_x \times s_y$.

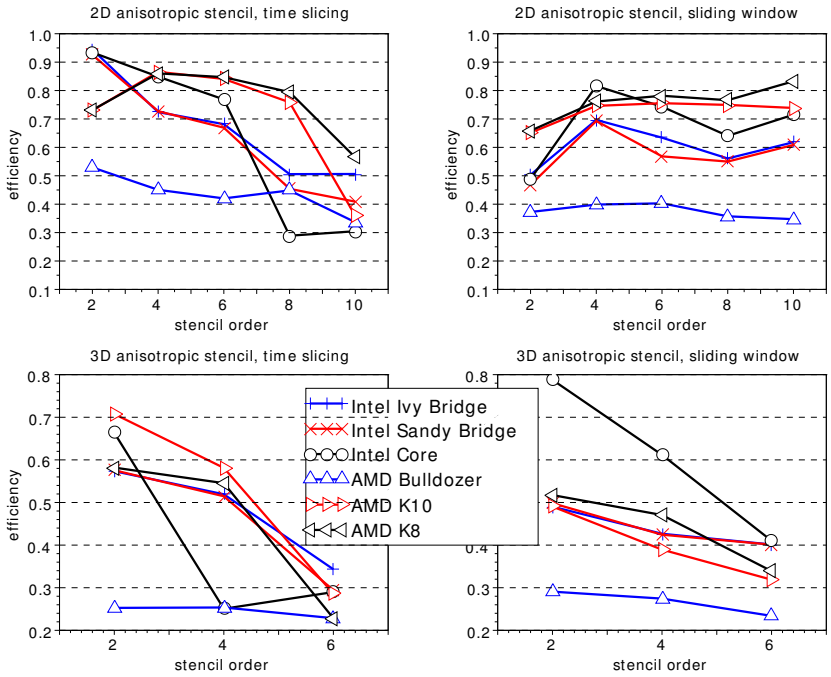

Fig. 7. 2D and 3D compact anisotropic FD stencil. Relative performance vs. stencil order of sliding window and time slicing on CPUs.

The CPU double precision numbers are consistently one half of the single precision. A 64 bit double number requires double the space in vector registers, caches and memory. The throughput and latency of the floating point pipelines remains the same in terms of vectors per time.

Double precision on GPUs is different: Cache and memory capacity and the number of available registers is halved, the throughput changes by a factor of 1/24 to 1/2, see Tab. 2. Slower floating point pipelines relative to the memory bandwidth results in an increased relative performance for memory bandwidth bound algorithms. The limited register file however results in smaller tile sizes. The Nvidia Tesla numbers demonstrate roughly half the double precision than single precision, consistent to the floating point performance. Slower double precision pipelines show higher relative performance. The ratio of single to double precision with increasing dimensions tends approach the ratio of memory throughput. Note that the AMD GPUs show even better double precision performance.

6.4 Outlook

The answer to the general question of CPU versus GPU performance mainly depends on metric of comparison, whether it be performance per core, per chip, per price or per electric power. A single CPU or GPU architecture is available as chips of different numbers of cores. These can be further aggregated into

Table 8. Absolute performance numbers of a multi-processor GPU chip. Numbers in single and double precision arithmetic of a time slicing (black) or a sliding window shifted-vector (grey) algorithm. * marks a shifted vector version of time slicing.

processor		1D		2D		3D	
architecture	name	single [GF]	double [GF]	single [GF]	double [GF]	single [GF]	double [GF]
Nvidia Kepler	GK110, Tesla K20c	1829	685	355	179	211	123
Nvidia Kepler	GK104, GTX 680	1638	98.3	309	75.0	219	60.4
Nvidia Fermi	GF100, Tesla C2050	655	302	241	116	196	98.6
Nvidia Fermi	GF110, GTX 590	794	114	304	78.5	236	69.4
AMD South. Isl.	GCN, HD 7970	1322	699	421	473	145	137
AMD North. Isl.	VLIW4, HD 6990	560	403	96.4	79.1*	62.7	52.5

shared and distributed memory systems of several chips. A comparison will have to balance the number of cores to compare, based on some criterion. So far we compared single cores, which includes instruction level parallelism and vector instructions.

Memory bandwidth bound algorithms on shared memory systems, especially on multi-core processors, will not show substantial parallel speed-ups. However, algorithms on private caches do scale. This is the case at least for one time slicing algorithm in 1D [8] and we expect it for 2D (L1 and L2 cache) and for the sliding window algorithm up to 3D. Beyond this, we expect a slow down due to shared LL cache and main memory. However, also shared LL cache and main memory usually scale for large numbers of cores. Note that the GPU experiments already take this into account, as all GPU cores execute the algorithm. Hardware multi-threading on Intel CPUs and two cores per module on AMD Bulldozer may accelerate a multi-threaded code, as long as a single thread does not fully load the floating point pipeline.

So far we neglected the treatment of the boundary nodes. However, in a parallelization based on domain decomposition, expensive inter-processor communication takes place by exchange of boundary data. Note that the algorithms differ in the communication pattern, but not in the total amount of data to transfer. The communication would again be more pronounced in higher dimensions and for higher order stencils, where the ratio of boundary nodes to inner nodes increases. We refer to [6, 14] for multi-core and to [4, 7] for distributed memory.

7 Conclusions

We were able to develop efficient vectorized sliding window and time slice implementations of Finite Differences in one, two and three dimensions, orders two to twelve and isotropic and anisotropic symmetric operators, for CPUs with x86 AVX vector instrinsics and GPUs in Cuda and OpenCL. The optimization techniques include various vectorization strategies, a change of data layout and loop unrolling. The result showed whether one of the algorithms was able to sustain high processor performance and to tolerate main memory latency: Time slicing tends to be superior for smaller Finite Difference stencils and large cache hierarchies found on current CPUs, while sliding window was better for larger stencils and larger register files like on GPUs.

Interesting generalizations of the model problem include variable coefficient difference stencils and systems of equations and hierarchies of grids with mesh refinement and multigrid algorithms [6, 11–13]. The data access patterns are more complex and the amount of data per grid point increases.

Acknowledgments. The authors would like to thank DFG for partial support under grant SFB/TR7 "gravitational wave astronomy" and the anonymous referees for their helpful comments.

References

1. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Rev. **51**(1) (2009) 129–159
2. Micikevicius, P.: 3D finite difference computation on GPUs using Cuda. In: Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, ACM (2009) 79–84
3. Weyhausen, A.: Numerical algorithms of general relativity for heterogeneous computing environments. Diplomarbeit, Universität Jena, Physics dept. (2010)
4. Maruyama, N., Nomura, T., Sato, K., Matsuoka, S.: Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In: Supercomputing, IEEE (2011)
5. Holewinski, J., Pouchet, L.N., Sadayappan, P.: High-performance code generation for stencil computations on GPU architectures. In: Proceedings of the 26th ACM international conference on Supercomputing. (2012)
6. Williams, S., Kalamkar, D., Singh, A., Deshpande, A., Straalen, B.V., Smelyanskiy, M., Almgren, A., Dubey, P., Shalf, J., Oliker, L.: Optimization of geometric multigrid for emerging multi- and manycore processors. In: Supercomputing, IEEE (2012)
7. Zhang, Y., Mueller, F.: Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In: Proc. 10th Int. Symp. Code Gen. Optim, San Jose, CA. (2012)
8. Zumbusch, G.: Tuning a finite difference computation for parallel vector processors. In: 11th Int. Symp. Parallel and Distrib. Comput. CPS, IEEE (2012) 63–70
9. Song, Y., Li, Z.: New tiling techniques to improve cache temporal locality. In: Proc. ACM SIGPLAN Conf. Prog. Lang. Design Impl., Atlanta. (1999) 215–228
10. McCalpin, J., Wonnacott, D.: Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Rutgers Univ, (1999)
11. Rivera, G., Tseng, C.: Tiling optimizations for 3D scientific computations. In: Supercomputing. (2000)
12. Weiß, C.: Data Locality Optimizations for Multigrid Methods on Structured Grids. PhD thesis, TU München (2001)
13. Stürmer, M., Treibig, J., Rüde, U.: Optimising a 3D multigrid algorithm for the IA-64 architecture. Int. J. Computational Science and Engineering **4** (2008) 29–35
14. Wellein, G., Hager, G., Zeiser, T., Wittmann, M., Fehske, H.: Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In: Int. Comput. Soft. and Applications Conf. (COMPSAC). (2009) 579–586
15. Nguyen, A., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In: Supercomputing, IEEE (2010)