Parallel Computing 39 (2013) 533–548

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Efficient 3D stencil computations using CUDA

Marcin Krotkiewski *, Marcin Dabrowski

Physics of Geological Processes, University of Oslo, Norway

ARTICLE INFO

Article history:
Received 4 October 2011
Received in revised form 13 December 2012
Accepted 10 August 2013
Available online 23 August 2013

Keywords:
Stencil computation
3D convolution
Memory bandwidth optimization
Multigrid
CUDA

ABSTRACT

We present an efficient implementation of 7-point and 27-point stencils on high-end Nvidia GPUs. A new method of reading the data from the global memory to the shared memory of thread blocks is developed. The method avoids conditional statements and requires only two coalesced instructions to load the tile data with the halo (*ghost zone*). Additional optimizations include storing only one XY tile of data at a time in the shared memory to lower shared memory requirements, *common subexpression elimination* to reduce the number of instructions, and software prefetching to overlap arithmetic and memory instructions, and enhance latency hiding. The efficiency of our implementation is analyzed using a simple stencil memory footprint model that takes into account the actual halo overhead due to the minimum memory transaction size on the GPUs. Through experiments we demonstrate that in our implementation the memory overhead due to the halos is largely eliminated by good reuse of the halo data in the memory caches, and that our method of reading the data is close to optimal in terms of memory bandwidth usage. Detailed performance analysis for single precision stencil computations, and performance results for single and double precision arithmetic on two Tesla cards are presented. Our stencil implementations are more efficient than any other implementation described in the literature to date. On Tesla C2050 with single and double precision arithmetic our 7-point stencil achieves an average throughput of 12.3 and 6.5 Gpts/s, respectively (98 GFLOP/s and 52 GFLOP/s, respectively). The symmetric 27-point stencil sustains a throughput of 10.9 and 5.8 Gpts/s, respectively.

1. Introduction

The computational power of processors has been growing significantly faster than the speed of the global memory. The hardware *flop to byte ratio* (F2B) is defined as the peak hardware performance in terms of *Floating Point Operations per Second* (FLOP/s) divided by the available memory bandwidth (bytes/s). A growing hardware F2B recently became one of the recognized challenges in high performance computing [1]. The performance of many codes is nowadays limited by the memory bandwidth and not by the peak hardware FLOP/s capabilities. This is partly due to a low algorithmic *flop to byte ratio*, i.e., the number of arithmetic instructions required by many algorithms is low compared to the amount of data processed. Often enough, implementations have a higher bandwidth demand than is strictly required by the algorithms, possibly due to the characteristics of the computer architecture or a suboptimal implementation. The challenge is to design and implement algorithms that utilize the memory bandwidth and/or the arithmetic units in an optimal way (i.e., 100% peak memory bandwidth or 100% peak FLOP/s), and to avoid inefficiencies due to excessive memory traffic and unnecessary computations.

The *Graphics Processing Units* (GPUs) are an architecture that recently gained a lot of interest from the *High Performance Computing* (HPC) community. GPUs offer an impressive computational performance exceeding a Tera-FLOP/s, but most

* Corresponding author. Tel.: +47 228 560 51.
E-mail address: marcink@fys.uio.no (M. Krotkiewski).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.08.002

importantly – a memory bandwidth that is often larger than 150 GB/s. Although at first glance these numbers seem impressive, modern CPU-based architectures can deliver similar performance as the GPUs. The often reported 100-fold speedups with GPUs are mostly a consequence of using a single CPU core and/or inefficient CPU implementations in the comparison. For many practical problems the potential speedup of the GPUs with respect to CPU-based architectures is limited [2]. Nonetheless, the GPUs are considered an attractive and promising alternative because of their high peak performance relative to the cost and power usage. The impact made by the graphics cards on the HPC hardware can clearly be seen on the Top 500 list of supercomputers, where a number of the fastest installations in the world have GPUs as co-processors. Hence, developing new programming techniques and optimizing computational codes for this architecture benefits the HPC community.

GPUs just as the CPUs suffer from the same problem of a limited memory bandwidth. Consider the Nvidia Tesla C2050, which has a peak double precision performance of 515 GFLOP/s and a peak memory bandwidth of 144 GB/s ($F2B \approx 3.6$). Similar performance can nowadays easily be reached by CPU-based shared memory architectures, e.g., a 4-way system with a total of 48 2.2 GHz Opteron cores delivering 422 GFLOP/s peak performance and supported with around 170 GB/s of memory bandwidth ($F2B \approx 2.6$). Notice that in this configuration the CPU delivers more bandwidth relative to floating point capabilities than the GPU. What is also important is the fact that *F2B* is relatively high on both architectures, which means that even the computationally intensive codes must carefully optimize the memory access and limit the redundant memory operations to be truly compute bounded.

GPUs are similar to vector computers – warps of threads on the GPUs can be perceived as vector units. To hide the latency of instructions and memory access, and operate at peak instruction and memory throughput, GPUs rely on concurrent execution of thousands of threads. Hence, inherently serial problems and programs with a complicated work-flow logic are likely better suited for CPU-based computers. In addition, there are strict rules on how the threads should access the global memory to efficiently use the available memory bandwidth. Finally, the GPU architecture is throughput oriented, i.e., it is efficient for problems that involve processing of data that is structured in the global memory, preferably performing the same set of operations for all data points.

Due to the unique characteristics of the GPU hardware, most algorithms that have been designed for CPUs need to be adjusted or completely reworked to run efficiently on the GPUs. As an example, significant effort has been put into developing general purpose high-performance GPU libraries, e.g., Fast Fourier Transform (e.g., [3,4]), Monte-Carlo methods and random number generation (e.g., [5]), dense linear algebra (e.g., [6,7]) and sparse linear algebra (e.g., [8]). Obtaining significant speedups over CPUs can be challenging. Bell and Garland [9] presented an efficient implementation of the memory bandwidth bounded *Sparse Matrix–Vector* product (SpMV) for matrices resulting from unstructured computational meshes. The performance reached 10 GFLOP/s on a GTX280 GPU with the peak memory bandwidth of 142 GB/s. For symmetric matrices, Krotkiewski and Dabrowski [10] obtained a performance of up to 4.2 GFLOP/s on one node of a Cray XT4 with a 4-core Opteron and 12.8 GB/s memory bandwidth – only 2.4 times slower on a system with 11 times lower memory bandwidth.

In this paper we optimize stencil computations on modern Nvidia GPUs – an important kernel in numerical solvers of *Partial Differential Equations* (PDEs). A novel three-dimensional CUDA implementation of the single and double precision symmetric 7- and 27-point stencils, and the general 27-point stencil ($3 \times 3 \times 3$ convolution filter) is presented. These algorithms are well suited for the GPUs since the computations can be performed in parallel for all grid points and the data is structured in the memory. We employ the following optimizations that specifically address the outlined challenges of programing for modern GPUs:

- A new method of reading the required data from the global memory to the shared memory of the thread blocks is developed. The method avoids conditional statements and requires only two instructions to load the block data and the halo regions (*ghost zones*). Both read instructions are coalesced, which enhances the effective memory bandwidth and decreases the total memory latency.
- The data is loaded into the shared memory one XY tile at a time. All stencil computations that require this plane are performed and the partial stencil results are kept in registers. This saves the shared memory and enhances the performance by performing most of the computations in registers. Moreover, in this approach *common subexpression elimination* can be used to reduce the number of *floating point operations* (FLOPs) performed by the symmetric 27-point stencil from 30 to 18 per grid point.
- Software prefetching is used to overlap arithmetic and memory instructions, and thus enhance hiding of the memory latency.

The data in the halos comprises a memory overhead, since it is loaded multiple times for the neighboring thread blocks. We analyze the efficiency of our implementation using a simple stencil memory footprint model that takes into account the actual halo overhead due to the minimum memory transaction size on the GPUs. We use the model to demonstrate that the overhead due to the halos is largely eliminated by good reuse of the halo data in the global memory and texture caches. Our implementation of the 7-point stencil is shown to be memory bandwidth bounded for both single and double precision on the two tested Tesla GPUs, and close to optimal on Tesla C2050, i.e., it runs only 8% slower than a memory copy routine with no halo overhead. The performance of the single precision symmetric 27-point stencil is similar to and 10% worse than that of the 7-point stencil on Tesla C1060 and Tesla C2050 GPUs, respectively.

The paper is organized as follows. Section 2 explains the algorithm behind stencil computations and outlines an example application to the solution of the Poisson equation. In Section 4 we develop a new memory footprint model that takes into

account the actual halo overhead due to the minimum memory transaction size on the GPUs. Section 5 describes the implementation in detail. Section 6 demonstrates the performance and a detailed efficiency analysis of our stencil implementations. Section 7 presents our algorithms and performance results in the context of previous work on optimizing stencil computations for the GPUs. Finally, Section 8 briefly discusses application of our stencil code in a Multigrid solver.

2. 3D stencil computations

Consider the Poisson's boundary value problem

$$\begin{aligned} -\nabla^2\phi &= \psi \quad \text{in } \Omega \\ \frac{\partial\phi}{\partial\mathbf{n}} &= 0 \quad \text{on } \Gamma \\ \phi &= \phi_D \quad \text{on } \partial\Omega \setminus \Gamma \end{aligned} \tag{1}$$

where Ω is the spatial domain, zero-flux through the boundary is prescribed on Γ (Neuman boundary conditions), values of $\phi = \phi_D$ are prescribed on $\partial\Omega \setminus \Gamma$ (Dirichlet boundary conditions), ψ is the source term.

The above boundary value problem can be discretized using e.g., Finite Difference Method (FDM), or Finite Element Method (FEM). Finding a solution to the discretized problem involves solving a system of linear equations

$$Au = f \tag{2}$$

where A is the discretized $-\nabla^2$ operator, u and f are discrete counterparts of the sought field ϕ and the source term ψ, respectively. On a uniform Cartesian grid a general $3 \times 3 \times 3$ discretization of the isotropic and homogeneous Poisson's operator (1) in points far from the grid boundaries yields

$$\begin{aligned} & C_0 \cdot u_{i,j,k} + C_1 \cdot (u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1}) \\ & + C_2 \cdot (u_{i-1,j-1,k} + u_{i+1,j-1,k} + u_{i-1,j+1,k} + u_{i+1,j+1,k} + u_{i-1,j,k-1} + u_{i+1,j,k-1} + u_{i,j-1,k-1} \\ & \quad + u_{i,j+1,k-1} + u_{i-1,j,k+1} + u_{i+1,j,k+1} + u_{i,j-1,k+1} + u_{i,j+1,k+1}) \\ & + C_3 \cdot (u_{i-1,j-1,k-1} + u_{i+1,j-1,k-1} + u_{i-1,j+1,k-1} + u_{i+1,j+1,k-1} + u_{i-1,j-1,k+1} + u_{i+1,j-1,k+1} \\ & \quad + u_{i-1,j+1,k+1} + u_{i+1,j+1,k+1}) = f_{i,j,k} \cdot h^2 \end{aligned} \tag{3}$$

where h is the grid spacing and i, j, k are indices of the grid points in X, Y, Z spatial dimensions, respectively.

The left-hand side of (3) is referred to as a stencil. The result of computation of a $3 \times 3 \times 3$ stencil in a grid point (i, j, k) is given by a weighted sum of u in point (i, j, k) and its 26 neighbors using stencil coefficients $C_{0\ldots3}$ that depend on the discretization method. For example, setting $C_0 = 1, C_1 = -1/6, C_2 = C_3 = 0$ yields the 7-point Finite Difference stencil, choosing $C_0 = 8/3, C_1 = 0, C_2 = -1/6, C_3 = -1/12$ yields the Finite Elements stencil for an 8-node brick (trilinear) element. Stencil computation in every grid point is the major part of the work performed by various numerical methods for the solution of (1) such as relaxation methods (e.g., Jacobi, Gauss–Seidel) often employed in conjunction with multigrid techniques, and Krylov space iterative methods (e.g., Conjugate Gradients).

A general (i.e., not necessarily related to the Poisson's problem) $3 \times 3 \times 3$ stencil computation in point (i, j, k) far from the grid boundaries can be performed as

$$\hat{u}^K_{i,j,k} = \sum_{\hat{i}=-1}^{1} \sum_{\hat{j}=-1}^{1} \sum_{\hat{k}=-1}^{1} K_{\hat{i},\hat{j},\hat{k}} \cdot u_{i+\hat{i},j+\hat{j},k+\hat{k}} \tag{4}$$

where K is the stencil kernel – a three-dimensional array of size $3 \times 3 \times 3$ – that holds the stencil coefficients. In the field of computer graphics, (4) is known as the general (non-separable) 3D *convolution filter*.

3. Stencil computations on GPUs

In our analysis we assume some prior knowledge about GPU programming, terminology and code optimization. For an informative description of the crucial aspects of GPU programming the reader is referred to the "CUDA Overview" chapter in the paper by Cohen and Molemake [11] and to the Nvidia CUDA C Programming Guide ([12]). Volkov and Demmel [6] and Cui et al. [13] presented detailed guidelines regarding the optimization of programs dedicated for the GPUs.

Parallel computations on GPUs involve execution of thousands of concurrent threads. Multi-threading is used to parallelize the computations, but also as a means to hide memory and instruction latency: when active threads stall, schedulers choose threads ready to be executed from the queue. On the finest level the threads are arranged into essentially synchronous groups of 32 called *warps*, which are scheduled for execution independently of each other by the *Streaming Multiprocessors (SM)*. Individual threads within the same warp are allowed to take a different execution path due to e.g., conditional statements. However, since a warp executes one common instruction at a time, branching results in some threads of that warp being disabled until different execution paths merge [12]. Consequently, branching effectively increases the number of instructions, which are executed by a warp, to the sum of the instruction counts of all execution paths taken. This behavior favors branch-free implementations and must be taken into account, especially when optimizing compute bounded codes.

Warps are further grouped into 1D, 2D or 3D regular *thread blocks*. Warps belonging to the same thread block are executed on the same SM. The order in which different thread blocks are executed is undefined. Threads within a thread block can quickly synchronize and exchange data through a common *shared memory*. The position of a thread inside a block is described by a set of local `threadIdx.x`, `threadIdx.y`, `threadIdx.z` indices. Each thread block has a similar set of indices (`blockIdx.x`, `blockIdx.y`, `blockIdx.z`) describing the position of the block in the global data grid. In one approach to 3D stencil computations the grid is processed using 3D thread blocks (e.g., [14]). Based on `blockIdx` and `threadIdx` coordinates the threads compute the global indices *(i,j,k)* of the processed data points.

We employ a more efficient and widely used method of *2.5D blocking* [15–17]. Two-dimensional thread blocks are used to tile the grid in the XY plane. This provides the threads with the (i,j) indices of the grid points. A loop is then used to traverse the grid in the Z-dimension, providing the final k index. This method has several advantages to using 3D thread blocks. The occupancy (and hence the performance) is much higher due to significantly lower shared memory requirements: both methods require the data and the halos to be stored in the shared memory caches, but the 3D thread blocks require additional space for the Z-dimension halos. There are no Z-dimension halos in the 2.5D method since the data is streamed by the thread blocks plane by plane. This ensures data reuse in the Z-dimension and large memory bandwidth savings. Finally, in the 2.5D the initialization costs of the thread blocks (computing thread and grid indices, setting boundary conditions, etc.) is amortized over a larger number of grid points processed by every thread.

4. Memory footprint

The purpose of the proposed memory footprint model is to estimate the efficiency of our stencil implementations in terms of memory bandwidth usage and memory overhead due to redundant data access. We calculate the theoretical minimum and the hardware/algorithm constrained minimum amount of data read/written in a single stencil computation. We assume there is no data reuse between subsequent stencil computations, which is the case in applications where stencils computations are interleaved by other operations on the results of the stencil, e.g. the dot product in iterative methods such as Conjugate Gradients.

In a memory-optimal implementation of the 27-point stencil (4), i.e., one which performs only the strictly required global memory operations, the value of u in every point is read from the global GPU memory once, the stencil is computed, and the result is written back to the global memory once. Using *single precision floating point* numbers (SPFP) with 4 bytes per value, the optimal memory footprint – number of bytes of memory traffic per grid point (BpP) – of a stencil computation is therefore

$$BpP^{optimal} = 4(read) + 4(write) = 8[byte/point] \tag{5}$$

On GPUs, thread blocks executed by the SMs run independently of each other and can not explicitly exchange data. Hence, every thread block has to read from the global memory the interior data of the block and a layer of neighboring values for every XY tile – recall from (4) that neighbors of all points are required. In total, `(blockDim.x + 2)*(blockDim.y + 2)` values of u are required for every XY tile. The values outside of the block boundaries are often referred to as the *halo* (or *ghost zone*). Note that the values in the halo are interior values of neighboring thread blocks. Consequently, the memory requirements are increased, since some of the data is read from the global memory more than once: first as a part of the block interior, and again – as a part of the halo required by neighboring thread blocks. This memory overhead is decreased by enlarging the thread block dimensions, which yields a smaller number of halo data values relatively to the number of values in the data block. However, the overhead can not be nullified due to the limitations on the admissible size of the thread blocks. Since the thread blocks can not explicitly exchange data, the overhead introduced by reading of the halo can not be avoided and should be included in the calculations of the memory footprint.

Memory is accessed on the GPUs via 32, 64, or 128 byte transactions on memory segments that must be aligned correspondingly. In order to achieve high fractions of memory bandwidth, reading and writing of the data on GPUs should be performed in a coalesced manner. The exact rules differ for different GPUs. In short, threads in a warp should access consecutive addresses within the aligned data segments. Full memory coalescing is obtained when the width of the thread block and the most frequently changing dimension of the data grid are a multiple of the warp size (or half the warp size for 1.$\times$ devices) [12]. Reading and writing of the block interior data can be fully coalesced if the data grid is correctly aligned, and the thread blocks have correct dimensions. Since the data values in the Y-dimension halos are also correctly aligned in the memory, they are also read in a coalesced way. On the other hand, reading of the X-dimension halo and the corner points is more challenging from the point of view of performance. The minimum memory transaction size on the GPUs is 32 bytes. Even though only 1 data point of halo is required on each side of the data block, in the best case scenario a memory read transaction of 32 bytes will be issued for each of them. Consequently, for every thread block of dimensions $BX \times BY$ the minimum number of data values requested when reading a block of data with a halo of width 1 is

$$BX \cdot BY + 2 \cdot BX + 2 \cdot (BY + 2) \cdot 32/size(FP) \tag{6}$$

where $size(FP)$ denotes size of the data value in bytes – 4 and 8 for single and double precision arithmetic, respectively. $BX \cdot BY$ is the number of values in the data block interior. The remaining terms describe the halo overhead: $2 \cdot BX$ is the number of values read for the Y-dimension halo (to the top and bottom of the data block), and $2 \cdot (BY + 2) \cdot 32/size(FP)$ is the

minimum number of data values effectively read for the X-dimension halo – to the left and right of the data block, including the four corner points. For simplicity, in the following detailed analysis we deal with single precision floating point numbers. In this case formula (6) reads:

$$BX \cdot BY + 2 \cdot BX + 2 \cdot (BY + 2) \cdot 8 \tag{7}$$

Formula (7) describes the minimum amount of SPFP values including the halo overhead fetched by a thread block for every XY tile. One approach to reduce the memory overhead is to find the correct (large) thread block size. Instead, we choose to rely on the global memory caches, which can ensure that a lower number of values is transferred from the global memory than shown by (7). In our tests we use the older Tesla C1060, which has a texture cache, and the new Fermi-based Tesla C2050 additionally equipped with L1 and L2 global memory caches. On these architectures the global memory overhead introduced by the halos can be partly or entirely eliminated when reuse of data is observed between different thread blocks. Maximizing the cache reuse requires control on the order in which the scheduler executes the warps, which can not be done on present generation GPUs. Hence, the cache reuse can only be measured, but not really guaranteed in the implementation.

In the case of a perfect cache reuse the halo overhead is nullified, and the number of bytes effectively transferred from/to the global memory per grid point is equal to the optimal number (5): $BpP^{eff} = BpP^{optimal} = 8$. This is our definition of a *memory optimal stencil implementation*, i.e., one that is memory bandwidth bounded and does not incur any memory overhead. On the other hand, with no cache reuse BpP^{eff} is higher due to the halo overhead shown in (7), and the exact value depends on the thread block size.

We note that a memory optimal implementation can not be further improved otherwise than by decreasing the memory footprint. Some applications that use stencils do not process the results between subsequent stencil computations (e.g.,

```
1 __global__ void stencil_3d(out, in, nx, ny, nz)
2 {
3   extern __shared__ float shm[];        // shared memory holds the data block and the halo
4
5   const uint tx = threadIdx.x;
6   const uint ty = threadIdx.y;
7
8   float r1, r2, r3;                     // values in neighboring grid points are explicitly read
9   float r4, r5, r6;                     // from shm to the registers before computations
10  float r7, r8, r9;
11
12  float t1, t2;                         // intermediate stencil results
13
14  readBlockAndHalo(in, 0, tx, ty, shm); // read block of data from 0th XY-plane
15
16  shm2regs(shm, tx, ty);                // copy required thread data from shm to registers
17                                        // software prefetching:
18  readBlockAndHalo(in, 1, tx, ty, shm); // 1. initiate reading of the next block of data to shm
19  t1 = computeStencil_plane0();         // 2. compute stencil using data available in registers
20
21  shm2regs(shm, tx, ty);
22  readBlockAndHalo(in, 2, tx, ty, shm);
23
24  t2 = computeStencil_plane0();
25  t1+= computeStencil_plane1();
26
27  out += ix + iy*nx;                    // adjust per-thread output pointer
28  for(uint k=3; k<nz-1; k++){           // Z-dimension loop
29
30    shm2regs(shm, tx, ty);
31    readBlockAndHalo(in, k, tx, ty, shm);
32
33    out    = out + nx*ny;
34    out[0] = t1  + computeStencil_plane2();
35    t1     = t2  + computeStencil_plane1();
36    t2     = computeStencil_plane0();
37  }
38  shm2regs(shm, tx, ty);
39  out[0] = t1 + computeStencil_plane2();
40}
```

Fig. 1. Pseudo-code of the general 27-point stencil routine.

explicit wave propagation solver, Jacobi iterative solver). In these cases BpP^{eff} can be lowered by exploiting the time-dimension data reuse using the temporal blocking optimization (e.g., [17,18,14]).

5. Implementation

A CUDA pseudo-code of our implementation of the general 27-point stencil (4) is presented in Fig. 1. For clarity, the pseudo-code does not handle the boundary conditions in X and Y dimensions and the stencil is not computed for the first and last XY planes. The algorithm is explained in detail in the following sections. Briefly, `readBlockAndHalo` reads one XY tile with halos from the global memory to the shared memory `shm` of the thread blocks; in `shm2regs` every thread copies 9 values around the processed point *(i,j,k)* from the XY tile stored in the shared memory to the registers; `computeStencil_plane0..2` perform stencil calculations on the values stored in registers, i.e., all coefficients of the stencil kernel are applied to the data in one *k*-loop iteration.

The usual algorithm for stencil computations is formulated from the point of view of the output plane: for every output point the required data is read from the global memory, the stencil is computed, and the result is stored. On GPUs, computing a stencil for the *k*th output tile is implemented by reading tiles $k-1, k$, and $k+1$ into the shared memory of the thread block. The stencil can then be computed directly as shown in (4). Reusing the k and $k+1$ input planes in the calculations of the next $k+1$ output plane is done using e.g., the circular queue [19].

In contrast, our algorithm is formulated from the point of view of the input plane. The thread block reads from the global memory to the shared memory only one XY tile of data, including the halos (Fig. 1, line 31). This tile from the *k*th input plane is used as part of the stencil computations of three output planes: $k-1, k$, and $k+1$. The calculations are performed in the `computeStencil_plane0..2` routines, which apply all XY planes of the kernel K to the input tile (lines 34–36). The obtained partial stencil results are accumulated in registers $t1$ and $t2$. Once partial results from the three neighboring planes have been accumulated, the result is stored in the output array (line 34).

Our method has the advantage that the data values can be loaded from the shared memory into registers using only 9 load instructions per grid point. In the usual approach of storing three input planes in the shared memory the values are either loaded from the shared memory into registers directly before computations (27 load instructions per grid point), or the arithmetic operations have to use operands in the shared memory. This is slower than our approach since arithmetic operations using shared memory operands are slower than using registers [6]. Moreover, in our implementation we can use *common subexpression elimination* to take advantage of the symmetry of the stencil, which reduces the number of arithmetic operations. For a symmetric stencil $K_{i,j,0} = K_{i,j,2}$ and hence `computeStencil_plane0` and `computeStencil_plane2` are equivalent. Consequently, line 36 can be removed.

In the following we discuss in detail the memory reading method (`readBlockAndHalo`) and the implementation of software prefetching.

5.1. Global memory reading

Fig. 2 shows a straightforward implementation of `readBlockAndHalo`, in which a thread block reads one XY tile of data into the shared memory (`shm`). Similar implementation is found e.g., in the Nvidia FD3D finite difference code from the SDK examples [15]. For simplicity, the boundaries of the domain are not considered. Note that the neighbor indices are in fact

```
1    readBlockAndHalo(in, k, tx, ty, shm)
2    {
3      uint bx = (blockDim.x+2);
4      uint ix = blockIdx.x*blockDim.x + threadIdx.x;
5      uint iy = blockIdx.y*blockDim.y + threadIdx.y;
6
7      shm[tx+0+(ty+0)*bx] = in[ix + iy*nx + k*nx*ny];
8      if(tx==1) {
9                            shm[tx-1+(ty+0)*bx] = in[ix-1 + (iy+0)*nx + k*nx*ny];
10       if(ty==1)           shm[tx-1+(ty-1)*bx] = in[ix-1 + (iy-1)*nx + k*nx*ny];
11       if(ty==blockDim.y) shm[tx-1+(ty+1)*bx] = in[ix-1 + (iy+1)*nx + k*nx*ny];
12     }
13     if(tx==blockDim.x) {
14                           shm[tx+1+(ty+0)*bx] = in[ix+1 + (iy+0)*nx + k*nx*ny];
15       if(ty==1)           shm[tx+1+(ty-1)*bx] = in[ix+1 + (iy-1)*nx + k*nx*ny];
16       if(ty==blockDim.y) shm[tx+1+(ty+1)*bx] = in[ix+1 + (iy+1)*nx + k*nx*ny];
17     }
18     if(ty==1)             shm[tx+0+(ty-1)*bx] = in[ix+0 + (iy-1)*nx + k*nx*ny];
19     if(ty==blockDim.y)    shm[tx+0+(ty+1)*bx] = in[ix+0 + (iy+1)*nx + k*nx*ny];
20   }
```

Fig. 2. Pseudo-code of a straightforward `readBlockAndHalo` implementation. In line 7 every thread reads "its own" data value. The halo is read by the boundary threads. For simplicity, boundaries of the domain are not considered.

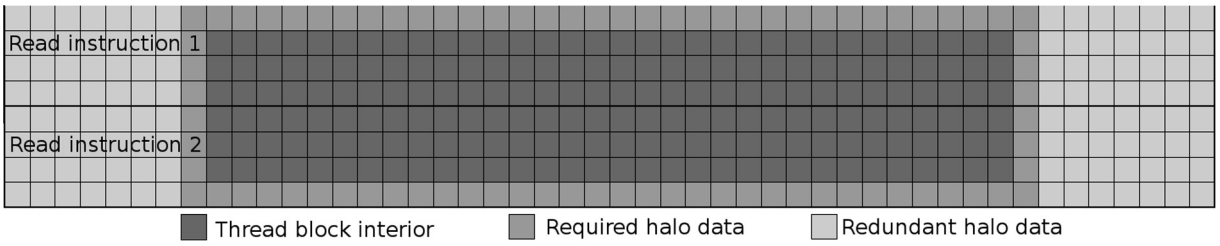

Fig. 3. The proposed method of reading the data and the halo for thread block size of 32×6. The threads are remapped onto two thread blocks of size 48×4. The data block including the halos is then read using only two coalesced instructions.

precomputed before the *k*-loop, at the beginning of the `stencil_3d` routine, and are only shown here for clarity. Since every thread processes the entire *Z*-column, only the *k*-component of the indices changes and needs to be updated inside the *k*-loop.

In line 7 every thread reads the data point that corresponds to the thread's id. The halo is read only by the boundary threads, which requires the use of conditional statements and effectively increases the total number of instructions executed by every warp. To avoid the conditional statements Zhang and Mueller [16] statically assigned the halo nodes to individual threads at the beginning of the stencil routine. The threads first read the interior of the domain into the shared memory (same as Fig. 2, line 7). The following instructions read the assigned halo nodes. Depending on the thread block size, one or more instructions are required to read the halo nodes, some threads may be idle during that time, and some threads may need to read multiple nodes. Also, the X-dimension halo nodes are read in a non-coalesced way. Conditional statements have also been removed in the method developed by Phillips and Fatica [20] and Krotkiewski [21]. This method is quite efficient, but it always requires 4 texture fetches.

Our novel method of reading the data and the halo from the global memory into the shared memory draws on the observations regarding the total memory overhead introduced by reading of the halos given by (7). We find data blocks of size $BX \cdot BY$ such that the number of values in the block interior is a multiple of the number of values in the halo (including the redundant values read due to the 32 byte memory operations). For single precision arithmetic, this requirement is obtained from (7) as

$$M \cdot (2 \cdot BX + 2 \cdot (BY + 2) \cdot 8) = BX \cdot BY \tag{8}$$

where M is an integer number and the number of threads in a thread block is $BX \cdot BY$ divided by M. With such arrangement reading of the data block and the halos can be done using $M + 1$ coalesced memory read instructions: M for the block interior and 1 for the halos. Suitable thread block sizes must meet several criteria:

- The number of threads in a thread block is limited by the hardware constraints – 512 threads on $1.\times$ architectures and 1024 threads on newer architectures.
- For the memory operations to be coalesced into few memory transactions BX should be a multiple of the warp size, or half warp size on the $1.\times$ architectures.

For $M = 1$ there are as many values in the halos as there are in the interior block. For $M = 2$ the interior block holds twice as many values as the halos, hence the overhead is relatively lower and a better performance could be expected. Since in this case the thread blocks are twice smaller than the data block interior, every thread performs stencil calculations for two grid points on a given XY plane. This results in high per-thread register requirements and limits the occupancy.

As an example, in our tests we use $M = 1$ and a thread block size of 32×6. It meets the above criteria, works on both tested architectures and exhibits high occupancies. As shown in Fig. 3, reading of the data with the halos is performed using two coalesced memory read instructions. The pseudo-code in Fig. 4 shows the implementation of our algorithm: threads are first remapped onto two thread blocks of size 48×4. The number 48 comes from the fact that the interior data block has 32 values in the X dimension, and 16 values are read as the X dimension halos (a 32 byte transaction, or 8 single precision values on each side).

The memory footprint that takes into account the halo overhead (7) is

$$BpP = 8(read) + 4(write) = 12 \tag{9}$$

since the number of data values in the halos is equal to the number of values in the block interior. This is a 50% overhead compared to the optimal scenario (5).

Full coalescing during writing is obtained naturally for the chosen block size. Assuming the 3D data grid is appropriately aligned in the memory, for every thread block the data values belonging to the block interior start at a 128-byte aligned memory address. Hence, writes performed by every warp can be coalesced into a single 128-byte memory transaction on the 2.x architecture, and into two 64-byte transactions on the 1.3 architecture. On the other hand, data reads require more than one memory transaction for at least some warps. The mapping to a thread block size of 48×4 introduces misalignment,

```
1    readBlockAndHalo_32x6(texData1D, k, shm)
2    {
3       // input thread block size is 32x6
4
5       // convert thread x/y ids to linear index 0..191
6       uint ti  = blockIdx.y*blockDim.x + threadIdx.x;
7
8       // remap the linear index onto a thread block of size 48x4
9       uint tx  = ti%48;
10      uint ty1 = ti/48;
11      uint ty2 = ty1+4;
12
13      // calculate input array indices
14      int  ix  = blockIdx.x*blockDim.x + tx  - 8;  // 8 words offset due to X-dimension halo overhead
15      int  iy1 = blockIdx.y*blockDim.y + ty1 - 1;  // Y-dimension halos
16      int  iy2 = blockIdx.y*blockDim.y + ty2 - 1;
17
18      // data and halos is read as two read instructions
19      // of thread blocks of size 48x4
20      shm[tx + ty1*48] = in[ix + iy1*nx + k*nx*ny];
21      shm[tx + ty2*48] = in[ix + iy2*nx + k*nx*ny];
22   }
```

Fig. 4. Pseudo-code of the developed method of reading the data and the halo for thread block size of 32×6. The threads are remapped onto two thread blocks of size 48×4. The data block including the halos is then read using only two coalesced instructions.

since 48 is not divisible by 32 – the warp size. Hence, some warps access non-contiguous memory areas. Also, data fetched as the left side halo starts with a 32 byte offset from the 128-byte aligned interior data. Consequently, one read instruction may be performed by the hardware as a combination of a separate 32 byte transaction and at least one other transaction. We demonstrate through performance tests that this misalignment does not incur a significant penalty if the data is accessed through textures.

5.2. Software prefetching

We implement software prefetching as shown in Fig. 1 using the following sequence.

1. `readBlockAndHalo` – initialize the algorithm by scheduling of the reading of the first XY tile from global memory to shared memory (Fig. 1, lines before the *k*-loop),
2. `shm2regs` – synchronize to finish the scheduled reads and copy required data from shared memory to registers (line 30),
3. `readBlockAndHalo` – schedule the reading of the next block of data from global memory to shared memory (line 31),
4. `computeStencil` – perform stencil computations on values present in registers (lines 34–36),
5. Go to step 2 (iterate the *k*-loop).

The data read from the global memory in step 3 is not immediately used in the stencil computations. Instead, computations are done on the data scheduled for reading in the previous *k*-loop iteration and copied to the registers in step 2. Instructions in steps 3 and 4 can therefore be overlapped by the hardware as they have no common operands. This approach enhances hiding of the global memory latency and ensures an overlap of the time required for the arithmetic instructions and the time needed to read the data from the device memory. While the hardware itself can overlap arithmetic and memory instructions – during the time one warp waits for the data the arithmetic instructions of another warp can be performed – our implementation leaves more opportunity for that at the level of a single warp. Hence, overlapping of arithmetic and memory instructions is effective for lower occupancies.

We explicitly copy the data from shared memory to registers in our implementation (`shm2regs`). This requires additional instructions and allocation of 9 registers per thread, and may seem expensive. However, inspection of the assembly generated for the non-prefetched code compiled for the Tesla C2050 reveals that the compiler anyway generates explicit loads of the operands from the shared memory into registers. In the code generated for the older Tesla C1060 the operands of arithmetic instructions reside in the shared memory and the prefetched implementation indeed executes more instructions. However, as noted by Volkov and Demmel [6], when one of the operands of an arithmetic instruction resides in the shared memory the achievable performance is around 66% of the peak FLOP/s performance. Thus, our approach is expected not only to enhance overlapping of memory and arithmetic operations, but also to improve the throughput of the arithmetic instructions.

6. Performance tests

In this section we analyze the efficiency of our stencil implementations by comparing the measured performance to the memory and compute bounds of the tested hardware. Performance bounds due to the available memory bandwidth are esti-

Table 1
Hardware specifications of the tested GPUs. Peak FLOPs performance for single and double precision *Multiply–Add* instructions.

	Clock Cycle GHz	Peak mem. bandwidth GB/s	SP performance (MAD) GFLOP/s	DP performance (MAD) GFLOP/s
Tesla C1060	1.300	102	622	78
Tesla C2050	1.115	144	1030	515

mated using memory bandwidth benchmarks and the presented memory footprint models. The compute bounds are estimated by counting the number of instructions in the assembly code generated by the compiler.

First, we test a zero-overhead memory copy routine for 3D grids of single precision data with a memory footprint $BpP = 8$ (4 bytes read and 4 bytes written per grid point). The results of this benchmark are an estimate of the peak achievable memory bandwidth and an absolute bound on stencil performance. Next, we test a routine that implements all the data transfers required in our stencil implementations: the thread blocks of size 32×6 use the proposed method to read the data blocks with the halos and write the interior data block ($BpP = 12$). Comparing the results of these two benchmarks allows us to estimate the efficiency and memory overhead of our new method of reading the data. Finally, comparing the performance results of the stencil implementations to the memory bandwidth benchmarks we estimate how close they are to optimal in the sense of the optimal memory footprint (5).

We test the performance on the Tesla C1060 and Tesla C2050 with error correction turned off. The relevant hardware specifications are shown in Table 1. During the tests the grid size was adjusted so that both `nx` and `ny` are multiples of the thread block size. All the codes were compiled using Nvidia CUDA 4.0 for the 1.3 GPU architecture (`-arch = compute_13 -code = sm_13,sm_20`). Compiling for `compute_20` architecture on Tesla C2050 in most cases resulted in a code that performed significantly worse due to a much higher register usage, register spills and a significantly lower occupancy. We consider this to be a compiler problem: it seems that *nvcc* applies less strict register saving rules on the 2.0 architecture. This might be a good strategy in some cases, but it is not for our stencil code.

6.1. Benchmarking the memory bandwidth

The memory bandwidth is benchmarked using the `copycube` kernel presented in Fig. 5. We measure the time required to copy a 3D data cube of SPFP values inside the device memory. The *k*-loop structure of `copycube` reflects our 2.5D-blocking implementation. The effective memory bandwidth is computed as

$$B^{eff} = \frac{nx \cdot ny \cdot nz \cdot BpP}{t \cdot 10^9} \left[\frac{GB}{s}\right] \tag{10}$$

where t is the time in seconds of one `copycube` call and the memory footprint is $BpP = 8$. Iteration time t is obtained by calling `copycube` n times and calculating $t = t_n/n$, where t_n is the elapsed time of n executions. Table 2 shows B^{eff}, iteration

```
__global__ void copycube(float *out, const float *in, uint nx, uint ny, uint nz)
{
  const  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  const  int iy = blockIdx.y*blockDim.y + threadIdx.y;

  for(int k=0; k<nz; k++)
    out[ix + iy*nx + k*nx*ny] = in[ix + iy*nx + k*nx*ny];
}
```

Fig. 5. `copycube` kernel used for benchmarking of the achievable streaming memory bandwidth.

Table 2
Benchmarked device memory bandwidth (B^{eff}), the execution time of one `copycube` kernel using SPFP and the throughput in billions of points per second for a grid size of $256 \times 252 \times 256$ are shown. Second last column shows the throughput averaged over a range of grid sizes. For comparison, value returned by the *bandwidthTest* from CUDA SDK are given.

	`copycube`				*bandwidthTest*[1]
	$256 \times 252 \times 256$			avg. throughput	
	B^{eff} [GB/s]	[Gpts/s]	t [s]	[Gpts/s]	[GB/s]
Tesla C1060	74.1	9.3	$1.78 \cdot 10^{-3}$	9.2	77.1
Tesla C2050	109.8	13.7	$1.20 \cdot 10^{-3}$	13.3	95.8

```
__global__ void copycube_halo(float *out, const float *in, uint nx, uint ny, uint nz)
{
  const  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  const  int iy = blockIdx.y*blockDim.y + threadIdx.y;

  // pre-compute the global and shared memory indices t1, t2, i1, i2, tx, ty

  out += ix + iy*nx;
  for(int k=0; k<nz; k++){
    shm[t1] = in[i1];
    shm[t2] = in[i2];
    i1 += nx*ny;
    i2 += nx*ny;

    __syncthreads();
    out[0] = shm[tx+8 + (ty+1)*48];
    out   += nx*ny;
    __syncthreads();
  }
}
```

Fig. 6. `copycube_halo` kernel used to analyze the efficiency of the proposed method to read the data and the halo.

time t and throughput in grid points per second obtained for a grid size of $256 \times 252 \times 256$ on both GPUs. For comparison, the results of the *bandwidthTest* distributed with the NVIDIA SDK are included.[1] Average throughput of the `copycube` routine for a range of grid sizes from $192 \times 192 \times 192$ to $512 \times 510 \times 512$ is also presented. The average effective memory bandwidth of `copycube` is roughly 75% of the theoretical memory bandwidth on both GPUs (compare Table 1). The achieved fraction of the peak memory bandwidth is lower than the reported 89% achieved by the streaming benchmark used by Volkov and Demmel [6] when copying arrays of 64-bit words (two floats or one double per array entry), but the results are similar when 32 bit words are used. In practice the `copycube` routine achieves an average throughput of 13.3 billion of points copied per second on Tesla C2050 and 9.2 billion of points per second on Tesla C1060. Unless the achieved fraction of the peak hardware memory bandwidth can be increased, these throughput numbers are the absolute upper bound on SPFP stencil implementation.

6.2. Benchmarking reading of the halo

Fig. 6 presents the `copycube_halo` routine used to analyze the efficiency of our new method of reading the data from the memory. Table 3 shows the results obtained for the grid size of $256 \times 252 \times 256$ and average results for a range of grid sizes. The effective memory bandwidth B^{eff} is computed as in (10) using memory footprint (9): $BpP = 12$. We tested our method in two configurations: by binding the data cube to a 1D texture[2] and reading the values using `texlDfetch`, and by directly accessing the values through a global memory pointer.

Table 3
Measured effective memory bandwidth (B^{eff}), the execution time of one `copycube_halo` call using SPFP, and the throughput in billions of points per second for a grid size of $256 \times 252 \times 256$ are shown. Last column shows the throughput averaged over a range of grid sizes, percent of the average `copycube` throughput given in parentheses. Results using global memory pointer and texture fetches.

copycube_halo with global memory pointer				
	$256 \times 252 \times 256$			avg. throughput
	B^{eff} [GB/s]	[Gpts/s]	t [s]	[Gpts/s]
Tesla C1060	58.6	4.9	$3.38 \cdot 10^{-3}$	4.8 (52%)
Tesla C2050	125.5	10.5	$1.58 \cdot 10^{-3}$	8.5 (64%)
copycube_halo with texture fetches				
Tesla C1060	88.9	7.4	$2.22 \cdot 10^{-3}$	7.4 (81%)
Tesla C2050	160.0	13.3	$1.23 \cdot 10^{-3}$	12.4 (93%)

[1] The results obtained from *bandwidthTest* were multiplied by 1.049 to account for a different definition of Giga: $2^{20} \cdot 1000$ vs. 10^9 in our tests.
[2] Since the size of the 1D texture is limited on the GPUs, large grids need to be processed in chunks. We used chunks of maximum 2^{22} words on Tesla C1060, and 2^{24} words on Tesla C2050.

On Tesla C2050 the implementation that uses textures shows an effective memory bandwidth higher than the peak hardware bandwidth. In fact, for the grid size of $256 \times 252 \times 256$ the execution time of `copycube_halo` is only ~3% worse than in the case of the `copycube` routine. This suggests almost perfect halo data reuse in global memory caches. We confirmed this by comparing the `dram_reads` and `dram_writes` performance counters in the Visual Profiler. On the other hand, accessing the memory through a global memory pointer instead of the textures is significantly slower.[3] These results show that the texture cache is effective in eliminating the performance hit of the misalignment introduced by the halos.

On the older Tesla C1060 the effective memory bandwidth of `copycube_halo` that uses textures is around 86% of the peak hardware memory bandwidth and is higher than that of the `copycube` routine. Since the global memory has no L1/L2 caches on this GPU, these results could suggest either a more efficient memory bandwidth utilization, or – to some degree – texture cache reuse between different thread blocks. As in the case of Tesla C2050, the results obtained using pointers are significantly worse than when using textures.

We conclude that on both GPU architectures the proposed method works best when the global data is read using texture references. The last column in Table 3 presents the throughput of the `copycube_halo` routine averaged over a range of grid sizes. Comparing these figures to the average throughput of the `copycube` routine (Table 2) we can calculate the effective memory footprint of the developed method. On Tesla C2050, $BpP^{eff} = 13.3/12.4 \cdot 8 \approx 8.6$ bytes per point, which is roughly 8% more than in a theoretically optimal implementation (5). Note that the average performance is lower than that obtained for $256 \times 252 \times 256$ grid points, which is due to a lower cache reuse – as reported by the Visual Profiler. The average effective memory footprint on Tesla C1060 is $BpP^{eff} \approx 9.9$, around 24% more than optimal. This is a surprisingly good result considering that the global memory is not cached. This could indicate a good deal of data reuse in the texture caches among different thread blocks.

To summarize, potential performance improvements to our method of reading the data and the halo to achieve an optimal, zero-overhead method with $BpP^{optimal} = 8$ are roughly 8% on Tesla C2050 and 24% on Tesla C1060. The presented throughputs are a practical upper bound on memory bandwidth bounded stencil implementations.

6.3. Single precision stencil performance

Computation of the general 27-point stencil requires 27 multiply and 26 add operations – 53 *floating point operations* (FLOPs) in total. Symmetric 27-point stencil as shown in (3) requires 30 FLOPs. In our implementation *common subexpression elimination* is used to reduce the number of FLOPs to 18 (12 adds and 6 multiplies). Both implementations read and write the same amount of data.

On the Nvidia GPUs subsequent multiply and add operations are merged into a single hardware instruction: *Multiply–Add* (MAD) on Tesla C1060 and *Fused Multiply–Add* (FMA) on Tesla C2050. For these architectures Table 6 gives the theoretical peak FLOP/s performance for the MAD/FMA, which perform 2 floating point operations in a single instruction. For FADD/FMUL instructions the peak FLOP/s performance is half of that – only 1 FLOP per instruction is performed. We investigated the assembly of the stencil routines compiled for the Tesla C2050 GPU using `cuobjdump`. 27 floating point instructions are executed for the general stencil (26 FMA + 1 FMUL) and 13 for the symmetric stencil (7 FADD, 5 FMA, 1 FMUL) in every iteration of the k-loop. For the general stencil the k-loop contains in total 53 instructions: 27 are floating point instructions, the remaining auxiliary instructions are copying of the data from the shared memory to the registers, thread synchronization, global memory access, integer indices updates and loop branch. Assuming all instructions are executed with the peak floating point instruction throughput of 515 billion instructions per second (one instruction per clock cycle per thread), a compute bound general 27-point stencil can perform at maximum $515/53 \approx 9.7$ Gpts/s. Performance in GFLOP/s is computed by multiplying the throughput by the number of FLOPs performed per grid point. Peak floating point performance of the general 27-point stencil (53 FLOPs per grid point) is thus $9.7 \cdot 53 \approx 515$ GFLOP/s. In the case of the symmetric stencil there are in total 40 instructions in the k-loop, out of which 13 have floating point operands. Hence, a compute bounded implementation can have a maximum throughput of $515/40 \approx 12.9$ Gpts/s, with a peak floating point performance of $12.9 \cdot 18 \approx 232$ GFLOP/s.

Fig. 7 presents the throughput of our symmetric and general 27-point stencil implementations on Tesla C2050, together with memory bandwidth and compute bounds. The memory bandwidth bound on the performance is the average throughput of the `copycube_halo` routine (solid line). The peak instruction throughput discussed above is used as the computation speed bound (dashed line). For the general stencil the bound on instruction throughput is lower than the bound on the memory bandwidth, hence the implementation is compute bounded. In the case of the symmetric stencil the memory bandwidth bound is slightly lower than the compute bound. Note however that the compute and memory bounds are close to each other, which suggests that the code is in fact well balanced in terms of instruction and memory requirements.

For both the symmetric and the general 27-point stencil the observed performance is close to the upper bound. This indicates that the time needed to access the memory is largely overlapped by the hardware with the time needed to execute the instructions. It is due to two factors. The code has a high occupancy (75% on the Tesla C2050), i.e., there is a large number of resident warps per SM. As a result, during the time a memory transfer is performed there is a high likelihood that the SMs

[3] We tested the performance of the global memory references on Tesla C2050 with L1 memory cache turned on and off (compiler options `-Xptax -cg`). The performance in both cases was not significantly different.

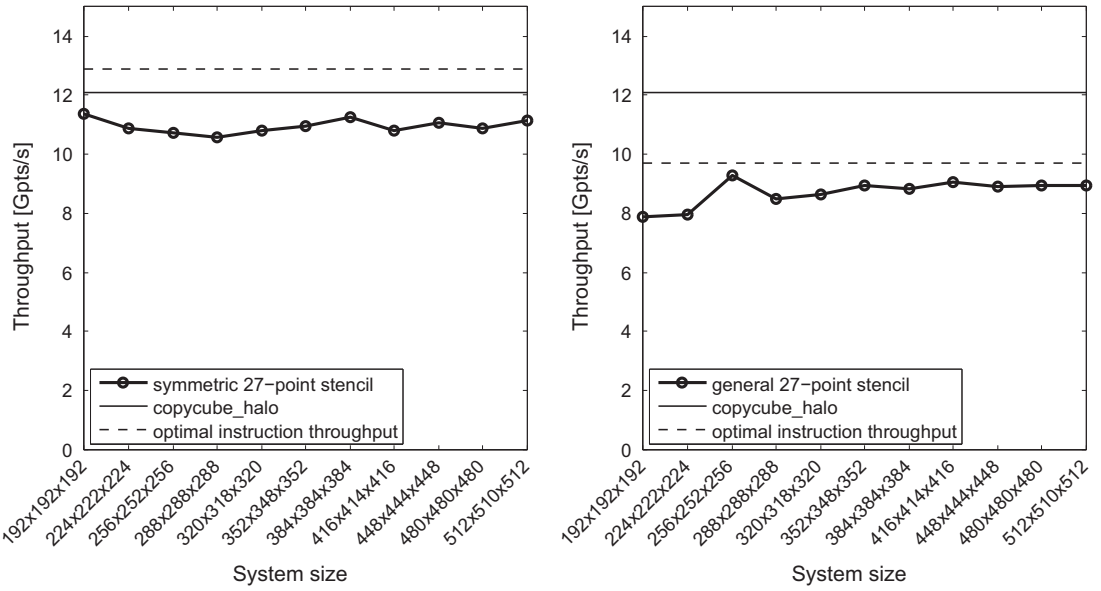

Fig. 7. Throughput of symmetric (left) and general (right) 27-point stencil for single precision numbers, Tesla C2050. Performance bounds based on memory bandwidth (throughput of `copycube_halo`) and peak hardware SPFP instruction throughput.

Table 4
Average performance of the single precision symmetric 27-point stencil and the general 27-point stencil. Fraction of the peak *Instructions Per Cycle* throughput (IPC) assuming SPFP instructions reported by the Visual Profiler.

symmetric 27-point stencil				
	Gpts/s	IPC [%]	GFLOP/s	`copycube_halo` [%]
Tesla C1060	7.2	–	130	97
Tesla C2050	10.9	86	196	88
general 27-point stencil				
Tesla C1060	5.9	–	313	79
Tesla C2050	8.6	91	456	71

can schedule arithmetic instructions for some warp. This is further enhanced by the fact that software prefetching introduces an explicit overlap between the memory transfers and computations performed by a single warp.

Table 4 shows the performance of the developed stencil implementations on two Nvidia GPUs averaged over a range of grid sizes. On both tested platforms the general 27-point stencil achieves roughly 50% of the peak FLOP/s performance. For the Tesla C2050 the percent fraction of the peak *Instructions Per Cycle* throughput (IPC) obtained from the Visual Profiler is given. On average, the code operates at 86% and 91% of peak instruction throughput for the symmetric and the general 27-point stencil, respectively.

Table 4 also shows the percent fraction of the `copycube_halo` throughput. For a memory bandwidth bounded implementation this value should be close to 100%. That is almost the case for the symmetric 27-point stencil on Tesla C1060. On Tesla C2050 the symmetric stencil performs at 88% of the throughput of `copycube_halo` routine and at 86% of the peak instruction throughput, which shows that the code is well balanced in terms of memory and arithmetic requirements. For the general stencil the achieved fraction of the streaming memory bandwidth is lower since the code is compute bounded.

6.4. Double precision stencil performance

Table 5 presents a performance summary of our stencil implementations including double precision results and the 7-point stencil. Since the amount of data in the double precision case is twice larger than in the single precision case, a memory bandwidth bounded implementation should perform at 50% of the single precision throughput. This is roughly the case for the `copycube_halo` routine, although we observe a slightly better memory bandwidth utilization with double precision values. The results also show that our 7-point stencil implementation is essentially memory bandwidth bounded for both single and double precision arithmetic and on both tested architectures – its throughput is close to the throughput of the `copycube_halo` routine.

Table 5
Average performance of our 27-point and 7-point stencil implementations compared to numbers reported by other authors.

Tesla C1060					
	Stencil type	Single precision		Double precision	
		Gpts/s	GFLOP/s	Gpts/s	GFLOP/s
This work	`copycube_halo`	7.4	–	3.8	–
	7pt	7.2	58	3.4	27
	Symmetric 27pt	7.2	130	2.1	38
	General 27pt	5.9	313	1.0	56
FDTD3d	7pt	4.3	34	–	–
Zhang and Mueller [16]	7pt	7.2	57	3.6	29
	Symmetric 27pt	3.2	95	1	29
Tesla C2050					
	Stencil type	Single precision		Double precision	
		Gpts/s	GFLOP/s	Gpts/s	GFLOP/s
This work	`copycube_halo`	12.4	–	6.5	–
	7pt	12.3	98	6.5	52
	Symmetric 27pt	10.9	196	5.8	104
	General 27pt	8.6	456	2.9	164
FDTD3d	7pt	6.7	54	–	–
Holewinski et al. [18]	7pt	5.9	48	3.2	26
Datta et al. [19] (GTX280)	7pt	–	–	4.5	36
Kamil et al. [22] (GTX280)	7pt	–	–	1.6	13
Nguyen et al. [17] (GTX285)	7pt	9.2	74	4.6	37
	7pt, temporal blk.	17	136	–	–
Christen et al. [23]	7pt	3	24		
Zhang and Mueller [16]	7pt	10.9	87	5.7	46
	Symmetric 27pt	5.3	158	3.3	98

CPU implementations			
	Stencil type	Double precision	
		Gpts/s	GFLOP/s
Kamil et al. [22] (Xeon X5550)	7pt	1.4	11
Tang et al. [24] (Xeon X5650)	7pt	2.5	20
	Symmetric 27pt	0.9	27
Datta et al. [25] (Xeon X5550)	Symmetric 27pt	1.2	22

The peak performance on Tesla 1060 is 78 GFLOP/s for MAD instructions and half of that for FADD/FMUL instructions, which is 8 times lower than for the single precision arithmetic. In our general 27-point stencil implementation 26 out of 27 floating point instructions performed per grid point are MAD instructions. Hence, the peak FLOPs performance in this case is $26/27 \cdot 78 + 1/27 \cdot 39 \approx 76$ GFLOP/s. Our code delivers 56 GFLOP/s – around 70% of the hardware FLOP/s capabilities. This is a good result considering that there are other, auxiliary instructions, such as integer index calculations and loop management. For the symmetric 27-point stencil 5 out of 13 instructions are MADs and the peak hardware FLOP/s performance in this case is roughly 54 GFLOP/s. Our code delivers 38 GFLOP/s, roughly 70% of the hardware peak.

On Tesla C2050 the peak double precision FLOP/s performance is half the single precision performance. Notice that the auxiliary instructions take the same amount of time in both cases. Hence, a double precision code can potentially perform at more than 50% of the single precision performance – the auxiliary instructions take less time relative to the floating point instructions. The results show that our double precision general 27-point stencil implementation operates at 34% of the single precision performance. Nvidia's `cuobjdump` revealed that for the general 27-point stencil the total number of instructions is around 90% higher than in the single precision case. This is caused by the fact that for the general 27-point stencil the coefficients of the stencil kernel K are stored in the constant memory on the device. While in the single precision case the arithmetic instructions use stencil coefficients directly from the constant memory, in the double precision case the compiler explicitly generates additional instructions that load the stencil coefficients from the constant memory to the registers before every arithmetic instruction. This increases the number of instructions, and decreases the occupancy due to a much larger register usage. Both factors results in a significantly worse performance. In contrast, our implementations of the 7-point stencil and the symmetric 27-point stencil store the stencil the coefficients in registers. In these cases the double precision throughput is above 50% of the single precision throughput.

6.5. Performance summary

- In the single precision case, our new method of reading the block data for 32×6 thread blocks has a memory footprint of $BpP = 12$. The overhead is due to the halo values in the X-dimension. Relative to the benchmarked memory bandwidth, the effective memory footprint of our implementation is $BpP^{eff} = 8.6$ on Tesla C2050 and $BpP^{eff} = 9.9$ on Tesla C1060. This

shows that the halo overhead is largely eliminated by the global memory and texture caches. The memory footprint of a memory optimal implementation (5) is $BpP^{optimal} = 8$, hence our new method can not be improved more than 8% on Tesla C2050 and 24% on Tesla C1060.

- Our implementation of the 7-point stencil is memory bandwidth bounded and runs with the speed of the `copy-cube_halo` routine on both architectures and for both single and double precision arithmetic.
- The performance of the single precision symmetric 27-point stencil is similar to and 10% worse than that of the `copy-cube_halo` on Tesla C1060 and Tesla C2050, respectively. As expected, on Tesla C2050 the double precision version runs at roughly half the performance of the single precision code.
- The single precision general 27-point stencil is compute bounded on both tested architectures, and 20% and 30% slower than the `copycube_halo` on Tesla C1060 and Tesla C2050, respectively. The performance in the double precision case is less than 50% of the single precision version due to hardware limitations (low DP capabilities of Tesla C1060) and a larger total number of instructions generated by the compiler.

7. Discussion and related work

Table 5 shows a comparison of the performance of our implementation to numbers reported by other authors. In one case, the FDTD3d stencil code [15] distributed with the Nvidia SDK examples, we ran the tests ourselves. In this case we report performance obtained for the best thread block size and compiler options we could find.

A number of stencil optimization techniques for GPUs has been previously developed. Micikevicius [15] showed an implementation of a high order 3D stencil (FDTD3d). Major optimizations included using the shared memory to load XY tiles of data with the halos, and using registers to save shared memory space. FDTD3d is widely available through the CUDA SDK examples and many programmers use a similar approach. Our implementation of the 7-point stencil is almost twice faster that FDTD3d on both tested GPUs. Moreover, FDTD3d does not implement the 27-point stencil.

Datta et al. [19] investigated the performance of a 7-point stencil implementation on an Nvidia GTX 280. The authors used thread blocking and employed the Z-dimension loop with the circular queue optimization that reduce the memory traffic. The latter requires that three data planes are stored in the shared memory. In comparison, in our approach only one plane is stored in the shared memory, which increases the GPU occupancy and results in a better performance. For the double precision 7-point stencil the authors reported 36 GFLOP/s. On Tesla C2050, a similar card in terms of the memory bandwidth, our implementation achieves 52 GFLOP/s.

Reading of the data and the halo from the global memory has been the target of optimizations in the past. Micikevicius [15] loaded the data with the halos into the shared memory of the thread blocks before computations. Using the shared memory as a cache greatly reduces the pressure on the global memory bandwidth and is now routinely used by most of the state-of-the-art implementations. However, it is of crucial importance how the data is read into the shared memory. In this respect our new method significantly differs from all other approaches found in the literature. A naive method as in Fig. 2 results in around 100 instructions in 1 iteration of the k-loop (investigated using `cuobjdump`), which makes it compute bounded. Phillips and Fatica [20] and Krotkiewski [21] both developed the same method of reading the data and the halo to the shared memory avoiding conditional statements, but using 4 texture fetches. Phillips and Fatica [20] reported 51 GFLOP/s ($\sim$1.5 Gpts/s) for the 19-point stencil applied in the Himeno benchmark on Tesla 1060. In the studied problem the amount of data accessed in the global memory was requirements were almost 6 times larger, so the results are not directly comparable. Recently, Zhang and Mueller [16] statically assigned halo data to individual threads during initialization of the thread blocks. Like ours, the method avoids conditional statements, but requires at least two instructions, and the halos in the most frequently changing dimension are read in a non-coalesced way. Comparison shows that the performance of our new 7-point stencil implementation is essentially the same on the older Tesla 1060, but is significantly better on the newer Tesla C2050. Moreover, our implementation of the symmetric 27-point stencil is roughly two times faster on both architectures. This is a combined effect of our new, coalesced memory reading method and the fact that in our implementation we have reduced the number of floating point operations per grid point from 30 to 18, making the symmetric 27-point essentially memory bandwidth bounded.

In our memory footprint analysis we assumed that the data needs to be both read and written from/to the global memory once per a stencil computation. This is the case e.g., in the Conjugate Gradients method and in the Multigrid method (computation of the residual, but also interpolation/restriction operators, which can be implemented as modified stencils). Some applications do not process the data between subsequent stencil computations (e.g., explicit wave propagation solver, Jacobi iterative solver). In these cases the arithmetic intensity of memory bounded stencil codes can be increased by exploiting the time-dimension data reuse with the temporal blocking optimization (e.g., [14,18,17]). The idea is to apply many stencil iterations to the data available in the shared memory of the GPU, or in the CPU caches. Note that this optimization can only work for memory bandwidth bounded codes. A common approach on the GPUs is to use larger *ghost zones* [14,18] and trade redundant computations for a decreased memory footprint. Holewinski et al. [18] claimed that due to the amount of redundant computations, temporal blocking is not very effective for 3D stencils. In contrast, Nguyen et al. [17] presented performance results of a single precision 7-point stencil with temporal blocking on a GTX 285, which is roughly 1.4 times better than ours. However, their code without temporal blocking performs 25% worse than our implementation. Nguyen et al. [17] also claimed that the temporal blocking optimization is not effective for the 27-point stencil, for which the implementations available till now were not memory bandwidth bounded. Our new method of reading the data, combined with the *common*

Table 6
Number of V-cycles per second for different grid sizes and architectures. Single precision arithmetic. Two pre-smoothers were used on the finest grid level. Otherwise, one pre- and one post-smoothing Jacobi iteration was used.

	129^3	257^3	$513 \times 513 \times 257$
Tesla C1060	150	35	10
Tesla C2050	233	53	15

subexpression elimination, resulted in a memory bandwidth bounded symmetric 27-point stencil on Tesla C1060. Hence, the temporal blocking can potentially improve its performance. We note that implementation of temporal blocking using ghost zones is straightforward with our memory reading method. The ghost cells in the X-dimension are already loaded into the shared memory as part of the X-dimension halos. Reading additional two layers of Y-dimension ghost cells can be done with only one additional coalesced memory read instruction. Similarly, our method can be extended to support stencils with radius larger than 1.

Many authors have investigated automatic stencil code generation and tuning, especially in the context of modern multi-core architectures. The auto-tuning approach promises to deliver a portable high performance across computing architectures and stencil types, at the same considerably speeding up the code development process. An auto-tuning framework for modern parallel multi-core CPU architectures was developed by Datta et al. [19] and Kamil et al. [22]. Datta et al. [25] used the framework to tune the 27-point stencil for the CPUs. Tang et al. [24] designed a parallelizing stencil compiler for modern CPU architectures that translates stencils defined in a domain-specific language to highly optimized, parallel code. Table 5 shows that these implementations have 5–10 times worse performance than our code, although the CPUs used in those studies have only around 3 times less memory bandwidth and 6 times lower peak FLOPs performance than Tesla C2050. This shows that GPUs are a well suited architecture for this type of computations. Recent work in this area targeting specifically GPUs was presented by Holewinski et al. [18], Christen et al. [23] and Zhang and Mueller [16]. The idea is to search through the space of parameters that are known to influence the performance on GPUs, such as thread block size, loop unrolling factor, shared memory vs. registers usage, and referencing memory through textures. However, due to its versatility, this approach usually leads to inferior performance compared to a problem-specific hand-tuned code. For the simplest case of the 7-point stencil Zhang and Mueller [16] presented results similar to ours, but only on Tesla C1060. In other cases our code delivers better performance (see Table 5).

8. Applications – geometric multigrid

Here we briefly summarize our implementation of the Multigrid algorithm to solve the Poisson problem, (1). We use the symmetric 27-point stencil implementation as the Jacobi smoother and computation of the residual. Restriction and interpolation of errors between grid levels are implemented as a modified stencil computation. Before executing the solver we scale the source vector to remove the mesh size dependence from the Poisson stencil. Thus, in our case the coefficients of the stencil are the same for all grid levels. On the coarsest level we solve the system using Jacobi relaxation.

Performance in V-cycles per second of our Multigrid implementation is summarized in Table 6. In these test we used two pre-smoothers on the finest level and 1 pre-smoother on all coarse grids. One post-smoother was used on all grid levels. For single precision arithmetic and using the Finite Element stencil around 6–7 V-cycles are required to converge to the result within machine precision. One V-cycle takes the time of approximately 10–12 fine-grid stencil computations.

9. Conclusions

We presented a performance study of our CUDA-based implementation of the symmetric 7-point and 27-point, and the general 27-point stencil ($3 \times 3 \times 3$ convolution) for modern GPUs.

1. A novel method of reading the data from the global memory to shared memory of thread blocks has been developed. The thread block size is chosen so that the number of values in the data block interior is a multiple of the number of values in the halo. In this study we used a thread block size of 32×6. In this case the data is read from the global memory using only two coalesced memory read instructions, with no conditional statements. On Tesla C2050 the memory overhead due to the halos is largely eliminated by the global memory and texture caches. The effective memory footprint (bytes accessed in the global memory per data point) for single precision data is $BpP^{eff} \approx 8.6$, only 8% worse than the optimal value of 8. On Tesla C1060, $BpP^{eff} \approx 9.9$ and our method can not be improved by more than 24%.
2. The proposed method works significantly better when global memory is read using texture fetches than when using global pointer reference.
3. In our implementation only one XY tile is stored in the shared memory of the thread blocks at any time. This approach limits the shared memory usage, increases occupancy, and allows for *common subexpression elimination*. The latter reduces the number of FLOPs in the 27-point stencil from 30 to 18.

4. Our implementation of the 7-point stencil is memory bandwidth bounded and runs close to the speed of the memory copy routine on both architectures and for both single and double precision arithmetic.
5. The performance of the single precision symmetric 27-point stencil is similar to and 10% worse than that of the `copycube_halo` on Tesla C1060 and Tesla C2050, respectively. As expected, on Tesla C2050 the double precision version runs at roughly half the performance of the single precision code.
6. The single precision general 27-point stencil is compute bounded on both tested architectures, and 20% and 30% slower than the `copycube_halo` on Tesla C1060 and Tesla C2050, respectively. The performance in the double precision case is less than 50% of the single precision version due to hardware limitations and a larger number of instructions generated by the compiler.
7. Single precision throughput on the Tesla C2050 is 12.3, 10.9 and 8.6 Gpts/s for the 7-point stencil, and the symmetric and general 27-point stencils, respectively. Double precision throughput is 6.5, 5.8 and 2.9 Gpts/s, respectively.
8. The codes performed better with compiler optimizations for the 1.3 architecture (`-arch = compute_13`). Compiling for the 2.0 architecture (Tesla C2050) resulted in lower throughput due to a much higher per-thread register usage and a lower occupancy.

Acknowledgments

We would like to thank the reviewers for insightful comments and useful suggestions, which helped us to improve the presentation of our work. This work was supported by a Center of Excellence grant from the Norwegian Research Council to PGP (Physics of Geological Processes) at the University of Oslo.

References

[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R.S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R.S. Williams, K. Yelick, Exascale computing study: Technology challenges in achieving exascale systems, 2008.
[2] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, A. Shringarpure, On the limits of GPU acceleration, in: HotPar'10: Proceedings of the 2nd USENIX conference on Hot topics in parallelism, USENIX Association, Berkeley, CA, USA, 2010, pp. 13.
[3] CUFFT, CUDA CUFFT Library, Technical Report, Nvidia Corporation, 2010.
[4] V. Volkov, B. Kazian, Fitting FFT onto the G80 Architecture, Technical Report CS 258 Final Project Report, University of California, Berkeley, 2008.
[5] L. Howes, D. Thomas, Efficient random number generation and application using CUDA, in: GPU Gems 3, 2010.
[6] V. Volkov, J.W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–11.
[7] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Comput. 36 (2010) 232–240.
[8] J. Krüger, R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, ACM Trans. Graphics 22 (2003) 908–916.
[9] N. Bell, M. Garland, Implementing sparse matrix–vector multiplication on throughput-oriented processors, in: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, NY, USA, 2009, pp. 1–11.
[10] M. Krotkiewski, M. Dabrowski, Parallel symmetric sparse matrix–vector product on scalar multi-core CPUs, Parallel Comput. 36 (2010) 181–198.
[11] J.M. Cohen, J. Molemake, A fast double precision CFD code using CUDA, in: 21st International Conference on Parallel Computational, Fluid Dynamics (ParCFD2009).
[12] NVIDIA Corporation, CUDA C Programming Guide programming guide, 2012. Version 5.0.
[13] X. Cui, Y. Chen, H. Mei, Improving performance of matrix multiplication and FFT on GPU, in: ICPADS'09: Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems, IEEE Computer Society, Washington, DC, USA, 2009, pp. 42–48.
[14] J. Meng, K. Skadron, A performance study for iterative stencil loops on GPUs with ghost zone optimizations, Int. J. Parallel Program. 39 (2011) 115–142.
[15] P. Micikevicius, 3D finite difference computation on gpus using CUDA, in: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, ACM, New York, NY, USA, 2009, pp. 79–84.
[16] Y. Zhang, F. Mueller, Auto-generation and auto-tuning of 3D stencil codes on GPU clusters, in: Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO'12, ACM, New York, NY, USA, 2012, pp. 155–164.
[17] A. Nguyen, N. Satish, J. Chhugani, C. Kim, P. Dubey, 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–13.
[18] J. Holewinski, L.-N. Pouchet, P. Sadayappan, High-performance code generation for stencil computations on gpu architectures, in: Proceedings of the 26th ACM International Conference on Supercomputing, ICS'12, ACM, New York, NY, USA, 2012, pp. 311–320.
[19] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC'08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 4:1–4:12.
[20] E. Phillips, M. Fatica, Implementing the Himeno benchmark with CUDA on GPU clusters, in: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–10.
[21] M. Krotkiewski, Efficient implementations of numerical models for geological applications on modern computer architectures, Ph.D. Thesis, University of Oslo, 2010.
[22] S. Kamil, C. Chan, L. Oliker, J. Shalf, S. Williams, An auto-tuning framework for parallel multicore stencil computations, in: IPDPS, pp. 1–12.
[23] M. Christen, O. Schenk, H. Burkhart, Patus: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures, in: Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE, International, pp. 676–687.
[24] Y. Tang, R.A. Chowdhury, B.C. Kuszmaul, C.-K. Luk, C.E. Leiserson, The pochoir stencil compiler, in: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA'11, ACM, New York, NY, USA, 2011, pp. 117–128.
[25] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, K. Yelick, Auto-tuning the 27-point stencil for multicore, in: Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning.