3D Finite Difference Computation on GPUs using CUDA

Paulius Micikevicius
NVIDIA
2701 San Tomas Expressway
Santa Clara, CA 95050

pauliusm@nvidia.com

ABSTRACT

In this paper we describe a GPU parallelization of the 3D finite difference computation using CUDA. Data access redundancy is used as the metric to determine the optimal implementation for both the stencil-only computation, as well as the discretization of the wave equation, which is currently of great interest in seismic computing. For the larger stencils, the described approach achieves the throughput of between 2,400 to over 3,000 million of output points per second on a single Tesla 10-series GPU. This is roughly an order of magnitude higher than a 4-core Harpertown CPU running a similar code from seismic industry. Multi-GPU parallelization is also described, achieving linear scaling with GPUs by overlapping inter-GPU communication with computation.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming.

General Terms

Algorithms, Performance, Measurement.

Keywords

Finite Difference, GPU, CUDA, Parallel Algorithms.

1. INTRODUCTION

In this paper we describe a parallelization of the 3D finite difference computation, intended for GPUs and implemented using NVIDIA's CUDA framework. The approach utilizes thousands of threads, traversing the volume slice-by-slice as a 2D "front" of threads in order to maximize data reuse from shared memory. GPU performance is measured for the stencil-only computation (Equation 1), as well as for the finite difference discretization of the wave equation. The latter is the basis for the reverse time migration algorithm (RTM) [6] in seismic computing.

An order-*k* in space stencil refers to a stencil that requires *k* input elements in each dimension, not counting the element at the intersection. Alternatively, one could refer to the 3D order-*k* stencil as a $(3k + 1)$-point stencil. Equation below defines the stencil computation for a three-dimensional, isotropic case. $D^t_{x,y,z}$ refers to data point at position (x,y,z), computed at time-step t, c_j is a multiplicative coefficient applied to elements at distance j from the output point.

$$D^{t+1}_{x,y,z} = c_0 D^t_{x,y,z} + \sum_{i=1}^{k/2} c_i \left(D^t_{x-i,y,z} + D^t_{x+i,y,z} + D^t_{x,y-i,z} + D^t_{x,y+i,z} + D^t_{x,y,z-i} + D^t_{x,y,z+i} \right)$$

The paper is organized as follows. Section 2 reviews the CUDA programming model and GPU architecture. CUDA implementation of the 3D stencil computation is described in Section 3. Performance results are presented in Section 4. Section 5 includes the conclusions and some directions for future work.

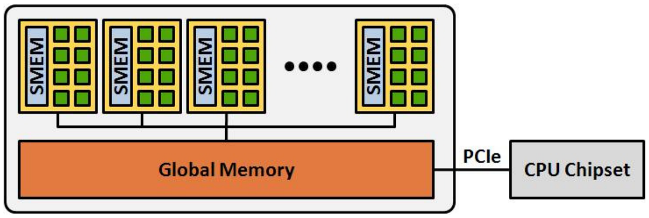

Figure 1. A high-level view of GPU architecture.

2. CUDA AND GPU ARCHITECTURE

CUDA allows the programming of GPUs for parallel computation without any graphics knowledge [5][7]. As shown in Figure 1, a GPU is presented as a set of multiprocessors, each with its own stream processors and shared memory (user-managed cache). The stream processors are fully capable of executing integer and single precision floating point arithmetic, with additional cores used for double-precision. All multiprocessors have access to global device memory, which is not cached by the hardware. Memory latency is hidden by executing thousands of threads concurrently. Register and shared memory resources are partitioned among the currently executing threads. There are two major differences between CPU and GPU threads. First, context switching between threads is essentially free – state does not have to be stored/restored because GPU resources are partitioned. Second, while CPUs execute efficiently when the number of threads per core is small (often one or two), GPUs achieve high performance when thousands of threads execute concurrently.

CUDA arranges threads into threadblocks. All threads in a threadblock can read and write any shared memory location assigned to that threadblock. Consequently, threads within a threadblock can communicate via shared memory, or use shared memory as a user-managed cache since shared memory latency is two orders of magnitude lower than that of global memory. A

GPGPU2, March 8, 2009, Washington D.C., US.
Copyright 2009 ACM 978-1-60558-517-8/09/03…$5.00.

barrier primitive is provided so that all threads in a threadblock can synchronize their execution. More detailed descriptions of the architecture and programming model can be found in [5] [7][2].

The Tesla 10-series GPUs (Tesla S1060/S1070) contain 30 multiprocessors, each multiprocessor contains 8 streaming processors (for a total of 240), 16K 32-bit registers, and 16 KB of shared memory. Theoretical global-memory bandwidth is 102 GB/s, available global memory is 4GB.

3. EFFICIENT 3D STENCIL COMPUTATION ON TESLA GPUs

Utilizing memory bandwidth efficiently is key to algorithm implementation on GPUs, since global memory accesses are not implicitly cached by the hardware. We use *memory access redundancy* as a metric to assess implementation efficiency. Specifically, redundancy is the ratio between the number of elements accessed and the number of elements processed. Where applicable, we refer to *read redundancy*, which does not include writes in the access count. Write redundancy is defined similarly. Ideally, read and write redundancies would be 1 for the stencil-only computation, overall redundancy would be 2.

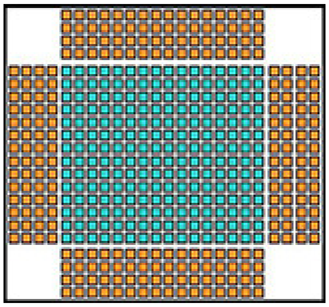

Figure 2. 16x16 data tile and halos for order-8 stencil in shared memory

The naïve approach to compute an order-k stencil refetches all $(3k + 1)$ input elements for every output value, leading to $(3k + 1)$ read redundancy. Our implementation reduces redundancy by performing calculations from shared memory. Since 16KB of shared memory available per multiprocessor is not sufficient to store a significantly large 3D subdomain of a problem, a 2D tile is stored instead (Figure 2). Extension of the computation to the 3rd dimension is discussed in the next section. Threads are grouped into 2D threadblocks to match data tiling, assigning one thread per output element. Given an order-k stencil and $n \times m$ threadblocks and output tiles, an $(n + k)\times(m + k)$ shared memory array is needed to accommodate the data as well the four halo regions. Even though space for k^2 elements could be saved by storing the halos separately (the four $(k/2)\times(k/2)$ corners of the array are not used), savings are not significant enough to justify increased code complexity. Since halo elements are read by at least two threadblocks, the read redundancy of loading the data into shared memory arrays is $(n\cdot m + k\cdot n + k\cdot m)/(n\cdot m)$. For example, read redundancy is 2 for an order-8 stencil when using threadblocks configured as 16x16 threads (24×24 shared memory array). Increasing threadblock and tile dimensions to 32×32 reduces redundancy to 1.5, in this case threadblocks contain 512 threads (arranged as 32x16), each thread computing two output values.

3.1 Extending the Computation to 3D

Once a 2D tile and halos are loaded into shared memory, each threadblock straightforwardly computes the 2D stencil for its output tile. All the data is fetched from shared memory, the latency of which is two orders of magnitude lower than that of global memory. There are two approaches to extend the computation to three dimensions: two-pass and single-pass. These are similar to stencil approaches described for Cell processors in [1][4].

As the name suggests, the two-pass approach traverses the input volume twice. During the first pass only the 2D stencil values are computed. The 1D stencil (in the remaining dimension) is computed during the second pass, combining it with the partial results from the first pass. Consequently, read redundancy is 3 plus redundancy due to halos – the first pass reads tiles and halos into shared memory, the second pass reads the original input and partial results. Returning to the example of an order-8 stencil and 16×16 tiles, read redundancy is 4. Since each pass writes output, write redundancy is 2. Overall redundancy is 6, which is an improvement over the naïve approach, which has redundancy of 26 (counting reads and writes). Efficiency can be further increased with the single-pass approach.

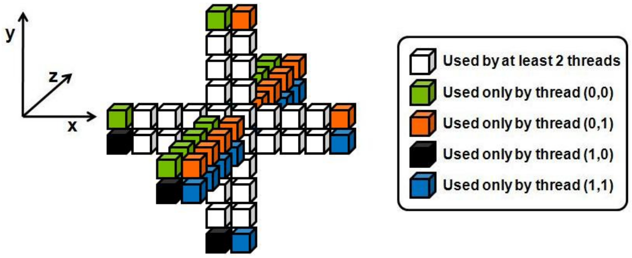
y
z
x
Used by at least 2 threads
Used only by thread (0,0)
Used only by thread (0,1)
Used only by thread (1,0)
Used only by thread (1,1)

Figure 3. Element re-use by a group of threads

The two passes described above can be merged into a single one. Let z be the slowest varying dimension. If we assign a thread to compute output values for a given column along z, no additional shared memory storage is needed. Threads of a given threadblock coherently traverse the volume along z, computing output for each slice. While the elements in the current slice are needed for computation by multiple threads, elements in the slices preceding and succeeding the current z-position are used only by the threads corresponding to the elements' (x, y) position (Figure 3). Thus, input elements in the current slice are stored in shared memory, while each thread stores the input from the preceding/succeeding $k/2$ slices it needs in local variables (which in CUDA are usually placed in registers). Using the case depicted in Figure 3, the four threads would access the 32 elements in the xy-plane from shared memory, while the elements along the z axis would be stored in corresponding thread's registers. Once all the threads in a threadblock write the results for the current slice, values in the local variables are "shifted," reading in a new element at distance $(k/2 + 1)$: the k local variables and shared memory are used as a queue. Output is written exactly once, input is read with $(n\cdot m + k\cdot n + k\cdot m)/(n\cdot m)$ redundancy due to halos. For example, redundancy for an order-8 stencil with 16x16 tiles is 3, compared to 6 of the two-pass approach.

$$D^{t+1}_{x,y,z} = 2D^{t}_{x,y,z} - D^{t-1}_{x,y,z} + v_{x,y,z}\left(c_0 D^{t}_{x,y,zk} + \sum_{i=1}^{k/2} c_i \left(D^{t}_{x-i,y,z} + D^{t}_{x+i,y,z} + D^{t}_{x,y-i,z} + D^{t}_{x,y+i,z} + D^{t}_{x,y,z-i} + D^{t}_{x,y,z+i} \right)\right) \qquad \text{(Equation 2)}$$

4. EXPERIMENTAL RESULTS

This section describes experiments with two types of kernels – stencil-only and finite difference of the wave equation. Performance was measured on Tesla S1070 servers, containing four GPUs each. S1070 servers were connected to cluster CPU nodes running CUDA 2.1 toolkit and driver (64-bit RHEL). Throughput in millions of output points per second (Mpoints/s) was used as the metric. Multi-GPU experiments are also included in this section, since practical working sets for the finite-difference in time domain of the wave equation exceed the 4GB memory capacity of currently available Tesla GPUs.

The prototype kernels do not account for boundary conditions. In order to avoid out-of-bounds accesses, the order-*k* kernel does not compute output for the first and last *k*/2 slices in the slowest dimension, while the *k*/2 boundary slices in each of the remaining 4 directions are computed with data fetched at usual offsets. Consequently, the 4 boundaries are incorrect as at least one "radius" of the stencil extends into inappropriate data. Since the intent of this study is to determine peak performance, we chose to ignore boundary conditions. Furthermore, boundary processing varies based on application as well as implementation, making experiments with "general" code not feasible. While we expect the performance to decrease once boundary handling is integrated, we believe performance reported below to be a reliable indication of current Tesla GPU capabilities.

4.1 Stencil-only Computation

Table 1 summarizes performance of the stencil-only kernel for varying orders in space as well as varying input volume sizes. All configurations were processed by 16×16 threadblocks, operating on 16×16 output tiles. Since tile size was fixed, the halos for the higher orders consumed a larger percentage of read bandwidth. For example, for the 8th order in space, halos (four 4×16 regions) consume as much bandwidth as the tile itself, leading to 2x read redundancy. For the 12th order in space, read redundancy is 2.5x. Using 32×32 tiles would reduce redundancy to 1.5x and 1.75x, respectively.

Table 1. 3D stencil-only throughput in Mpoints/s for various orders in space

Dimensions	Order in space			
	6	8	10	12
480x480x400	4,455	4,269	3,435	2,885
544x544x400	4,389	4,214	3,347	2,816
640x640x400	4,168	3,932	3,331	2,802
800x800x400	3,807	3,717	3,236	2,752
800x800x800	3,780	3,656	3,247	2,302

For fixed volume dimensions, throughput decrease with increased orders is largely due to higher read redundancy, additional arithmetic being another contributing factor. For a fixed order in space, increased memory footprint of larger volumes affects the TLB performance (480×480×400 working set is 703MB, while 800×800×800 requires 3.8GB). While throughput in Mpoints/s varied significantly across the configurations, memory throughput in GB/s (counting both reads and writes) was much more consistent, varying between 45 and 55 GB/s.

4.2 3D Finite Difference of the Wave-Equation

Finite difference discretization of the wave equation is a major building block for the Reverse Time Migration (RTM) [6][1], a technique of great interest in seismic imaging. While RTM has been known since 1980s, until very recently its computational cost has been too high for practical purposes. Due to advances in computer architecture and the potential for higher quality results, adoption of the RTM for production seismic computing has started in the last couple of years. The stencil-only computation is easily extended to the time-domain finite difference of the wave equation, second order in time (Equation 2 above).

In addition to reading data from the past two time steps, array *v* (inverse of velocity squared, in practice) is added to the input. Computing an output element requires (7*k*/2) + 4 floating point operations and 4 memory accesses, not accounting for redundancy due to halos. Therefore, ideal redundancy would be 4. CUDA source code for the 4th order in space wave equation kernel (using 16x16 tiles and threadblocks) is listed in Appendix A.

Table 2. 3D FDTD (8th order in space, 2nd order in time) throughput in Mpoints/s

data dimensions			tile dimensions	
dimx	dimy	dimz	16x16	32x32
320	320	400	2,870.7	2,783.5
480	480	480	2,965.5	3,050.5
544	544	544	2,786.5	3,121.6
640	640	400	2,686.9	3,046.8
800	800	200	2,518.3	3,196.9

Performance measurements for the 8th order in space are summarized in Table 2. Two kernel versions were implemented. The first one utilizes 16×16 threadblocks and output tiles (redundancy is 5). The second implementation used 32×16 threadblocks to compute 32×32 output tiles (redundancy is 4.5). GPU performance is roughly an order of magnitude higher than a single 4-core Harpertown Xeon, running an optimized implementation of the same computation.

4.3 Multi-GPU Implementation of the Wave-Equation Finite Difference

The working set for practical finite difference computations sometimes exceeds the 4 GB of memory available on a single GPU. For example, processing a single shot (a unit of computation for RTM) can easily require more than 10 GB of storage. It is therefore necessary to partition the data among several GPUs, at the same time scaling the performance. For simplicity we will examine a 2-GPU case, which is straightforwardly extended to more GPUs.

Table 3. Multi-GPU 3D FDTD (8th order in space, 2nd order in time) performance and scaling

Data dimensions			1 GPU		2 GPUs		4 GPUs	
dimx	dimy	dimz	Mpnts/s	scaling	Mpnts/s	scaling	Mpnts/s	scaling
480	480	800	2,986.85	1.00	5,944.98	1.99	11,845.90	3.97
544	544	400	2,826.35	1.00	5,545.63	1.96	6,453.15	2.28
544	544	800	2,736.89	1.00	5,459.69	1.99	11,047.20	4.04
640	640	640	2,487.17	1.00	5,380.89	2.16	10,298.97	4.14
640	640	800	2,433.94	1.00	5,269.04	2.16	10,845.55	4.46

Given two GPUs and a computation of order *k* in space, data is partitioned by assigning each GPU half the data set plus (*k*/2) slices of ghost nodes (Figure 4). Each GPU updates its half of the output, receiving the updated ghost nodes from the neighbor. Data is divided along the slowest varying dimension so that contiguous memory regions are copied during ghost node exchanges. In order to maximize scaling, we overlap the exchange of ghost nodes with kernel execution. Each time step is executed in two phases, as shown in Figure 5. In the first phase, a GPU computes the region corresponding to the ghost nodes in the neighboring GPU. In the second phase, a GPU executes the compute kernel on its remaining data, at the same time exchanging the ghost nodes with its neighbor. For each CPU process controlling a GPU, the exchange involves three memory copies: GPU->CPU, CPU->CPU, and CPU->GPU. CUDA provides asynchronous kernel execution and memory copy calls, which allow both the GPU and the CPU to continue processing during ghost node exchange.

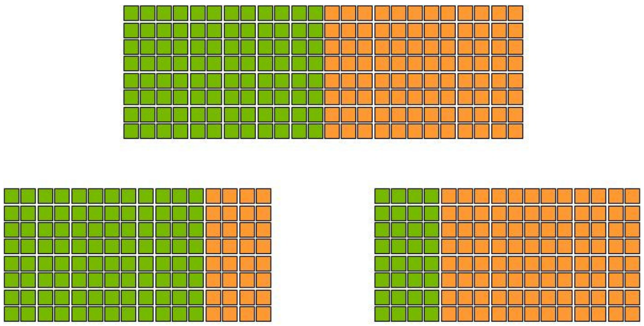

Figure 4. Data distribution between two GPUs

Extending the 2-GPU approach to more GPUs doubles the cost of ghost node exchange (each GPU has to communicate with two neighbors). The increased communication cost is still effectively hidden for data sets large enough to warrant data partitioning among GPUs. Performance results (using 16x16 tiles and threadblocks arranged as 16x16 threads) for up to 4 GPUs are summarized in Table 3. As in the 2-GPU case, memory copies were optimized by partitioning data only along the slowest-varying dimension. Measurements were collected on an Infiniband-connected cluster, where each CPU node was connected to two Tesla 10-series GPUs (one half of a Tesla S1070 1-U server, containing 4 GPUs).

CPU-GPU communication was carried out via cudaMemcpyAsync calls, using page-locked memory on the CPU side. MPI was used to spawn one CPU process per GPU. CPU processes exchanged ghost nodes with MPI_Sendrcv calls.

Table 3 indicates that communication and computation in Phase 2 are effectively overlapped when using either 2 or 4 GPUs. Scaling is the speedup over a single GPU, achieved by the corresponding GPU number. Note that only the smallest case (544×544×400 does not scale linearly with 4 GPUs. This is due to the fact that each GPU computes only 100 slices in Phase 2, which takes significantly less time than corresponding communication. Our experiments show that communication overhead is hidden as long as the number of slices per GPU is 200 or greater. Furthermore, we found that 2/3 of the communication time is spend in MPI_Sendrcv, the time for which should be further reduced by using the non-buffered version. The superlinear speedup for the larger data sets is due to the decreased pressure on TLB when a data set is partitioned among several GPUs – each GPU traverses a fraction of the address space that a single GPU has to access.

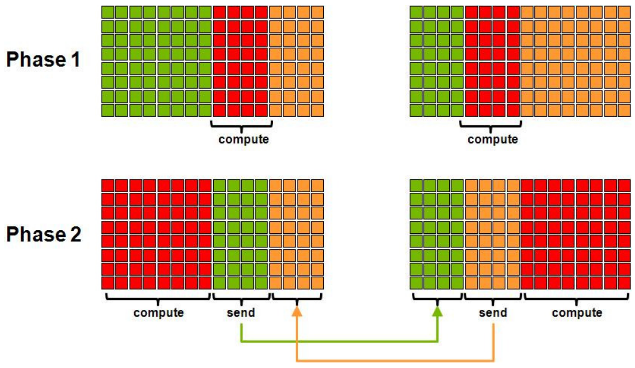

Figure 5. Two phases of a time step for a 2-GPU implementation of FD

5. CONCLUSTIONS AND FUTURE WORK

We have described a GPU parallelization approach for 3D finite difference stencil computation that achieves approximately an order of magnitude speedup over similar seismic industry standard codes. We also described the approach for utilizing multiple GPUs to solve the problem, achieving linear scaling with GPUs by using asynchronous communication and computation. This allows for GPU processing of large data sets common in practice, often exceeding 10 GB in size. There are at least two directions for further work. One, it would be interesting to modify the parallelization to carry out several time-steps in shared memory for the smaller stencils. Two, multi-GPU parallelization could also benefit from storing larger ghost regions and computing more than one time-step before communicating. This would be particularly interesting for the smaller data sets, where communication overhead is close to, or even greater, than the computation time.

6. ACKNOWLEDGMENTS

The author would like to thank Scott Morton of Hess Corporation for extensive assistance with the finite difference discretization of the wave equation.

7. REFERENCES

[1] Baysal, E., Kosloff, D. D., and Sherwood, J. W. C. 1983. Reverse-time migration. Geophysics, 48, 1514-1524.

[2] CUDA Programming Guide, 2.1, NVIDIA. http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf

[3] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Austin, Texas, November 15 - 21, 2008). Conference on High Performance Networking and Computing. IEEE Press, Piscataway, NJ, 1-12.

[4] Kamil, S., Datta, K., Williams, S., Oliker, L., Shalf, J., and Yelick, K. 2006. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness* (San Jose, California, October 22 - 22, 2006). MSPC '06. ACM, New York, NY, 51-60.

[5] Lindholm, E., Nickolls, J., Oberman, S., Montrym, J. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro 28, 2 (Mar. 2008), 39-55.

[6] McMechan, G. A. 1983. Migration by extrapolation of time-dependent boundary values. Geophys. Prosp., 31, 413-420.

[7] Nickolls, J., Buck, I., Garland, M., and Skadron, K. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (Mar. 2008), 40-53.

Appendix A: CUDA Source Code for a 25-Point Stencil

```
__global__ void fwd_3D_16x16_order8(TYPE *g_output, TYPE *g_input, TYPE *g_vsq, // output initially contains (t-2) step
                                    const int dimx, const int dimy, const int dimz)
{
#define BDIMX    16 // tile (and threadblock) size in x
#define BDIMY    16 // tile (and threadblock) size in y
#define radius   4  // half of the order in space (k/2)

    __shared__ float s_data[BDIMY+2*radius][BDIMX+2*radius];

    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int in_idx  = iy*dimx + ix;   // index for reading input
    int out_idx = 0;              // index for writing output
    int stride  = dimx*dimy;      // distance between 2D slices (in elements)

    float infront1, infront2, infront3, infront4;  // variables for input “in front of” the current slice
    float behind1, behind2, behind3, behind4;      // variables for input “behind” the current slice
    floatcurrent;                                  // input value in the current slice

    int tx = threadIdx.x + radius;  // thread’s x-index into corresponding shared memory tile (adjusted for halos)
    int ty = threadIdx.y + radius;  // thread’s y-index into corresponding shared memory tile (adjusted for halos)

    // fill the "in-front" and "behind" data
    behind3  = g_input[in_idx];   in_idx += stride;
    behind2  = g_input[in_idx];   in_idx += stride;
    behind1  = g_input[in_idx];   in_idx += stride;
    current  = g_input[in_idx];   out_idx = in_idx;in_idx += stride;
    infront1 = g_input[in_idx];   in_idx += stride;
    infront2 = g_input[in_idx];   in_idx += stride;
    infront3 = g_input[in_idx];   in_idx += stride;
    infront4 = g_input[in_idx];   in_idx += stride;

    for(int i=radius; i<dimz-radius; i++)
    {
        //////////////////////////////////////////
        // advance the slice (move the thread–front)
        behind4  = behind3;
        behind3  = behind2;
        behind2  = behind1;
        behind1  = current;
        current  = infront1;
        infront1 = infront2;
        infront2 = infront3;
        infront3 = infront4;
        infront4 = g_input[in_idx];

        in_idx  += stride;
        out_idx += stride;
        __syncthreads();

        /////////////////////////////////////////
        // update the data slice in smem
        if(threadIdx.y<radius)    // halo above/below
        {
            s_data[threadIdx.y][tx]              = g_input[out_idx-radius*dimx];
            s_data[threadIdx.y+BDIMY+radius][tx] = g_input[out_idx+BDIMY*dimx];
        }
        if(threadIdx.x<radius)    // halo left/right
        {
            s_data[ty][threadIdx.x]              = g_input[out_idx-radius];
            s_data[ty][threadIdx.x+BDIMX+radius] = g_input[out_idx+BDIMX];
        }

        // update the slice in smem
        s_data[ty][tx] = current;
        __syncthreads();

        /////////////////////////////////////////
        // compute the output value
        float temp = 2.f*current – g_output[out_idx];
        float div  = c_coeff[0] * current;
        div += c_coeff[1]*( infront1 + behind1
                    + s_data[ty–1][tx] + s_data[ty+1][tx] + s_data[ty][tx–1] + s_data[ty][tx+1] );
        div += c_coeff[2]*( infront2 + behind2
                    + s_data[ty–2][tx] + s_data[ty+2][tx] + s_data[ty][tx–2] + s_data[ty][tx+2] );
        div += c_coeff[3]*( infront3 + behind3
                    + s_data[ty–3][tx] + s_data[ty+3][tx] + s_data[ty][tx–3] + s_data[ty][tx+3] );
        div += c_coeff[4]*( infront4 + behind4
                    + s_data[ty–4][tx] + s_data[ty+4][tx] + s_data[ty][tx–4] + s_data[ty][tx+4] );
        g_output[out_idx] = temp + div*g_vsq[out_idx];
    }
}
```