Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 1275 – 1281

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Performance of Sediment Transport Simulations on NVIDIA's Kepler Architecture

Huayou Su[a,b,c], Nan Wu[a,b], Mei Wen[a], Chunyuan Zhang[a], Xing Cai[b,c,*]

[a]*School of Computer Science, National University of Defense Technology, Changsha, Hunan 410073, China*
[b]*Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway*
[c]*Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway*

Abstract

Aiming to understand how high-performance CUDA programming can be done for NVIDIA's new Kepler architecture, we have investigated a specific case of simulating sediment transport. The arisen stencil computations have distinct features connected to the two nonlinear partial differential equations that constitute the mathematical model. Consequently, the required CUDA programming effort differs for the two corresponding CUDA kernel functions. While Kepler's new read-only data cache brings enough benefits for one kernel function, performance of the other kernel function is further enhanceable through using the shared memory and so-called halo threads. The highest achieved performance of the stencil computation amounts to 190.45 GFLOPs on a Tesla K20 GPU.

Keywords: sedimentary basin; stencil computation; CUDA programming; NVIDIA Kepler architecture

1. Introduction

While providing increased computational capabilities, new advances in computer hardware may require cumbersome software adjustments. GPU computing is certainly no exception in this respect, see e.g. [1]. As NVIDIA recently rolled out its Kepler architecture, the hardware differences between Kepler and its predecessor Fermi have to be considered by the programmer, see e.g. [2], if high computational efficiency remains the objective. According to [3], the L1 cache on Kepler is reserved for local memory accesses such as register spills and stack data, i.e., global loads are cached in L2 only. It means that Kepler's shared memory has to be given more attention again, like on the old Tesla architecture. This is somewhat contrary to the Fermi architecture where L1 caches the global loads, thus making the shared memory less important in this respect. Moreover, the newly introduced read-only data cache on Kepler can take much of the role of Fermi's L1 cache, providing an additional on-chip resource. The key to performance on Kepler is, as on Fermi, good utilization of the on-chip resources together with minimizing data traffic from/to the device memory. However, concrete strategies of CUDA performance programming can differ on the two architectures. Two examples of the existing results that address the performance of applications and data structures on both Kepler and Fermi are [4, 5].

*Corresponding author. Tel.: +47-48294368; fax: +47-67828201.
Email addresses: shyou@nudt.edu.cn (Huayou Su), nanwu@nudt.edu.cn (Nan Wu), meiwen@nudt.edu.cn (Mei Wen), cyzhang@nudt.edu.cn (Chunyuan Zhang), xingca@simula.no (Xing Cai)

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.294

As an attempt to shed more light on the specific impact of the new Kepler architecture, we investigate in this paper some performance-enhancing strategies for CUDA programming. This is done in the context of simulating sediment transport, which involves numerically solving two coupled nonlinear partial differential equations by the finite difference method. We believe that our findings from this real-world application are relevant for similar stencil-based computations.

2. Mathematical model and numerical method

2.1. A coupled system of two equations

We adopt the mathematical model of Rivenæs [6, 7] to study the transport of sediments in a basin, by considering diffusion as the driving force. The model is capable of tracking two sediment types, such as sand and mud. In the language of mathematics, the model is made up of the following coupled system of two nonlinear partial differential equations:

$$\frac{\partial h}{\partial t} = \frac{1}{C_s}\nabla\cdot(\alpha s\nabla h) + \frac{1}{C_m}\nabla\cdot(\beta(1-s)\nabla h), \tag{1}$$

$$A\frac{\partial s}{\partial t} + s\frac{\partial h}{\partial t} = \frac{1}{C_s}\nabla\cdot(\alpha s\nabla h). \tag{2}$$

The primary unknowns are $h(x, y, t)$ and $s(x, y, t)$, where h denotes the basin height as a function of space and time, while s and $1 - s$ denote the volume fraction of the two sediments. The diffusion coefficients associated with the two sediment types are known as $\alpha(x, y)$ and $\beta(x, y)$. The constants C_s and C_m are given as the compaction ratio of the two sediments, while the constant A in (2) is a prescribed value representing the thickness of a sediment-transport top layer of the basin, see [6]. As initial conditions, $h(x, y, t)$ and $s(x, y, t)$ are given at $t = 0$. As boundary conditions, the flux rate of the two sediments are prescribed over the entire boundary.

2.2. An explicit numerical method based on finite differences

Many numerical methods can be used to solve the above mathematical model (1)-(2). Five different schemes are described in [8], and the CPU computing speed of all these methods is dictated by data movement within CPU's memory hierarchy. This means that the performance of a CUDA translation is likely also bound by data traffic, not the amount of floating-point operations.

We have chosen for this paper a fully-explicit numerical method, see e.g. [9], both because earlier numerical experiments in [8] showed that this explicit method has the fastest overall computing speed on CPUs, and also because it has the best ratio of floating-point operations vs. data traffic, thus the best candidate for GPU computing.

The temporal domain $0 \le t \le T$ is divided into a series of equal-distanced time levels: $t_0, t_1, t_2, \ldots, T$, where $t_\ell = \ell\Delta t$. During each time step, the fully-explicit scheme first solves (1), and then (2). The forward Euler scheme is used in the temporal discretization, while central differences and upwind differences are used in the spatial discretization. Suppose we use $h^\ell_{i,j}$ and $s^\ell_{i,j}$ to denote the numerical solutions at $t = t_\ell$ and $x = x_i = (i-1)\Delta x$, $y = y_j = (j-1)\Delta y$. The complete numerical discretization of (1) is as follows:

$$\begin{aligned}\frac{h^{\ell+1}_{i,j} - h^\ell_{i,j}}{\Delta t} = &\frac{\left(\frac{\alpha_{i+\frac12,j}\,s^\ell_{i+\frac12,j}}{C_s} + \frac{\beta_{i+\frac12,j}\left(1-s^\ell_{i+\frac12,j}\right)}{C_m}\right)\left(h^\ell_{i+1,j} - h^\ell_{i,j}\right) - \left(\frac{\alpha_{i-\frac12,j}\,s^\ell_{i-\frac12,j}}{C_s} + \frac{\beta_{i-\frac12,j}\left(1-s^\ell_{i-\frac12,j}\right)}{C_m}\right)\left(h^\ell_{i,j} - h^\ell_{i-1,j}\right)}{\Delta x^2}\\ &+\frac{\left(\frac{\alpha_{i,j+\frac12}\,s^\ell_{i,j+\frac12}}{C_s} + \frac{\beta_{i,j+\frac12}\left(1-s^\ell_{i,j+\frac12}\right)}{C_m}\right)\left(h^\ell_{i,j+1} - h^\ell_{i,j}\right) - \left(\frac{\alpha_{i,j-\frac12}\,s^\ell_{i,j-\frac12}}{C_s} + \frac{\beta_{i,j-\frac12}\left(1-s^\ell_{i,j-\frac12}\right)}{C_m}\right)\left(h^\ell_{i,j} - h^\ell_{i,j-1}\right)}{\Delta y^2},\end{aligned} \tag{3}$$

where the half-index subscripts in the above formula mean some form of averaging for a quantity in the middle of two spatial mesh points. For example, we typically use an arithmetic mean as follows:

$$\alpha_{i+\frac12,j}s^\ell_{i+\frac12,j} = \frac{\alpha_{i,j}s^\ell_{i,j} + \alpha_{i+1,j}s^\ell_{i+1,j}}{2}.$$

For (2) the complete numerical discretization reads:

$$A\frac{s_{i,j}^{\ell+1}-s_{i,j}^{\ell}}{\Delta t}+s_{i,j}^{\ell+1}\frac{h_{i,j}^{\ell+1}-h_{i,j}^{\ell}}{\Delta t}=\frac{1}{2C_s\Delta x^2}\begin{cases}\left(\alpha_{i,j}s_{i,j}^{\ell}-\alpha_{i-1,j}s_{i-1,j}^{\ell}\right)\left(h_{i+1,j}^{\ell+1}-h_{i-1,j}^{\ell+1}\right) & \text{if } h_{i-1,j}^{\ell+1}>h_{i+1,j}^{\ell+1}\\ \left(\alpha_{i+1,j}s_{i+1,j}^{\ell}-\alpha_{i,j}s_{i,j}^{\ell}\right)\left(h_{i+1,j}^{\ell+1}-h_{i-1,j}^{\ell+1}\right) & \text{else}\end{cases}$$
$$+\frac{1}{2C_s\Delta y^2}\begin{cases}\left(\alpha_{i,j}s_{i,j}^{\ell}-\alpha_{i,j-1}s_{i,j-1}^{\ell}\right)\left(h_{i,j+1}^{\ell+1}-h_{i,j-1}^{\ell+1}\right) & \text{if } h_{i,j-1}^{\ell+1}>h_{i,j+1}^{\ell+1}\\ \left(\alpha_{i,j+1}s_{i,j+1}^{\ell}-\alpha_{i,j}s_{i,j}^{\ell}\right)\left(h_{i,j+1}^{\ell+1}-h_{i,j-1}^{\ell+1}\right) & \text{else}\end{cases} \tag{4}$$

It is remarked that (2) has been treated as a convection-diffusion equation with respect to s, where $-\nabla h$ gives the convection velocity, thus requiring the upwind differencing that is used in (4). The reason for labeling the whole numerical scheme (3)-(4) as fully-explicit is because $h^{\ell+1}$ can be calculated straightforwardly by (3) when h^{ℓ} and s^{ℓ} are known, and similarly, calculating $s^{\ell+1}$ is straightforward by (4) when h^{ℓ}, $h^{\ell+1}$ and s^{ℓ} are ready.

3. Performance programming strategies

3.1. Data structure

Suppose the number of mesh points in the x-direction is N_x, whereas N_y is for the y-direction. In order to treat the flux-type boundary conditions by central differencing, it is customary to add one layer of ghost points around the entire computational mesh. Therefore, two arrays of length $(N_x+2)(N_y+2)$ are needed in any code implementation to store, alternately, the computed values of $h_{i,j}^{\ell}$ and $h_{i,j}^{\ell+1}$. Similarly, two arrays of the same dimension store the $s_{i,j}^{\ell}$ and $s_{i,j}^{\ell+1}$ values. In addition, one array is needed for storing the given values of $\alpha_{i,j}$, another array for the $\beta_{i,j}$ values.

It can be observed from the numerical scheme (3)-(4) that a five-point computational stencil is extensively used. To calculate each $h_{i,j}^{\ell+1}$ value using (3), five values of the h^{ℓ} array (i.e., $h_{i,j}^{\ell}$ and $h_{i\pm1,j\pm1}^{\ell}$) are needed together with five values each of the s^{ℓ}, α and β arrays. To calculate each $s_{i,j}^{\ell+1}$ value using (4), five values of the $h^{\ell+1}$ array are needed, in addition to relying on the $h_{i,j}^{\ell}$ value and three values each of the s^{ℓ} and α arrays (because of upwind differencing).

3.2. Baseline CUDA implementation

A typical CUDA implementation of the fully-explicit numerical scheme (3)-(4) consists of two kernel functions, one for calculating the $h_{i,j}^{\ell+1}$ values, the other for calculating the $s_{i,j}^{\ell+1}$ values. A straightforward implementation is to let each CUDA thread calculate a single $h_{i,j}^{\ell+1}$ or $s_{i,j}^{\ell+1}$ value, while the use of on-chip shared memory is avoided for the ease of programming. Such a baseline CUDA implementation tends to have quite reasonable performance on the Fermi architecture, because the L1 cache on Fermi offers a comparable benefit as that provided by explicit usage of the shared memory on the Tesla architecture. However, as mentioned earlier, Kepler's L1 does not cache loads from the global memory, such a baseline implementation will thus incur unnecessary global memory loads that are duplicated between neighboring threads in the same thread block.

3.3. Enhancement by using Kepler's read-only data cache

Read-only data cache is a new hardware feature on the Kepler architecture, where each streaming multiprocessor (SM) has 48 KB. Programming-wise this on-chip storage is very easy to use, only requiring two special key words, `const` and `__restrict__`, to label the name of an array pointer together with using the compile option `-arch=sm_35`. The detailed implementation can be seen in lines 7–10 of the code segment in the appendix of this article. For the kernel function that calculates $h_{i,j}^{\ell+1}$, we can use the read-only data cache to store the needed segment of the α, β, h^{ℓ}, and s^{ℓ} arrays. For the kernel function that calculates $s_{i,j}^{\ell+1}$, the α, $h^{\ell+1}$ and s^{ℓ} arrays can make use of the read-only data cache. The purpose is to avoid duplicated data loads that happen in the baseline implementation, and the performance enhancement can be substantial.

3.4. Enhancement by using shared memory

Suppose the read-only data cache is large enough to store the needed global memory loads of arrays that are accessed via a five-point stencil. The on-chip shared memory can, in particular, bring performance benefits to the calculations related to (3). More specifically, each thread now only needs to compute for itself

$$\alpha_{i,j} s^{\ell}_{i,j} \quad \text{and} \quad \beta_{i,j}\left(1 - s^{\ell}_{i,j}\right)$$

and store these two intermediate results in the shared memory, such as lines 20 and 21 in the code segment shown in the appendix. Therefore, the needed values of

$$\alpha_{i-1,j} s^{\ell}_{i-1,j} \quad \text{and} \quad \beta_{i-1,j}\left(1 - s^{\ell}_{i-1,j}\right)$$

are computed by the left-neighbor thread, lying in the shared memory ready to be used. Similarly, the right-neighbor, lower-neighbor and upper-neighbor threads all contribute to the shared memory. For most threads in a thread block, twelve floating-point operations can thus be saved. For threads lying on the boundary of a thread block, more computations associated with αs and $\beta(1 - s)$ are needed, but nine or six floating-point operations can still be saved.

One comment is in order here: This performance enhancement strategy will probably not benefit the kernel function calculating $s^{\ell+1}_{i,j}$, because due to upwind differencing only two multiply operations (associated with αs) can be saved. Another comment is that this way of using the on-chip shared memory is not meant for reducing the data traffic from the device memory, which is supposed to be already taken care by the read-only data cache.

3.5. Introducing halo threads

There are more on-chip resources per SM on Kepler than on Fermi, meaning that the thread blocks can potentially be larger. Turning this statement around, it means that we may afford to not fully use all the threads of a block. Under-utilization of the threads can be beneficial in connection with the above strategy of shared memory usage, which stores intermediate calculation results and thereby reduces duplicated floating-point operations between neighboring threads.

More specifically, we can enlarge each thread block with a layer of halo threads. Shown as line 4 in the code segment in the appendix, the thread block size is configured as $(BLOCK\_X + HALO) \times (BLOCK\_Y + HALO)$, where the value of $HALO$ is 2, meaning one halo element in each direction. The introduction of halo threads can ensure that all threads calculate precisely one set of intermediate αs and $\beta(1 - s)$ values, such as shown by code lines 16–21. (Otherwise, an `if`-test is needed per side of the thread block to ask the boundary threads to do additional memory loads and calculations.) We thus avoid the scenario that the boundary threads need more time, thus delaying the interior threads at the explicit thread synchronization command (code line 22). One important remark is that, when using the halo threads, the mapping between a thread block and its designated segment of the solution domain (of size $BLOCK\_X \times BLOCK\_Y$) still only considers the number of non-halo threads. This is exemplified by code lines 15–19. After the shared memory is filled with intermediate calculation results, the remainder of the h-kernel function will only make use of $BLOCK\_X \times BLOCK\_Y$ threads to continue with the calculation of $h^{\ell}_{i,j}$, as shown after code line 23. That is, some threads (equaling the number of the halo threads) will simply skip over, thus resulting in a certain level of thread under-utilization. Nevertheless, this strategy of using halo threads may still be overall beneficial.

4. Numerical experiments

We used two NVIDIA GPUs, Kepler K20 and GTX 590, for our numerical experiments. The detailed architecture information can be found in Table 1. All the measurements were associated with a global mesh of dimension $N_x = N_y = 4096$, and all the computations used double precision. In Tables 2 and 3, we have calculated the achieved GFLOPs rates associated with the two kernel functions. (The numbers of floating-point operations needed are 55 and 34, respectively, for computing each $h^{\ell+1}_{i,j}$ and $s^{\ell+1}_{i,j}$ value.) Moreover, we have listed the on-chip resources used per thread or per thread block plus the device occupancy rate, reported by the CUDA profiler. The naming scheme of the different performance-enhancing strategies is as follows. For K20, OPT-1 refers to making

Table 1. The architectural feature of the experimental platforms.

	GTX 590	Kepler K20
#SM(X)	16	13
#CUDA cores	512	2496
Architecture	Fermi GF110	GK110
Register per SM	32768	65536
Shared Memory per SM (KB)	48	48
L1 Cache per SM (KB)	16	16
Read Only Data Cache per SM (KB)	NA	48
L2 Cache (KB)	768	1280
Peak Double Precision (GFlop/s)	161.3	1170
Theoretical Bandwidth (GB/s)	165.9	208
Tested Bandwidth (GB/s)	142.94	160.88
Compiler	CUDA 5.0	CUDA 5.0

use of the read-only data cache, as described in Section 3.3. For GTX 590, OPT-1 means configuring the L1 cache to its maximum size using the `cudaDeviceSetCacheConfig(cudaFuncCachePreferL1)` function. For both GPUs, OPT-2 refers to enhancing OPT-1 with usage of the shared memory, as described in Section 3.4, whereas OPT-3 refers to enhancing OPT-2 with the use of halo threads, as described in Section 3.5. It should be noticed that the L1 cache used the default configuration (16KB per SM), when using OPT-2 and OPT-3 on both Kepler K20 and GTX 590.

Table 2. Effect of the performance-enhancing strategies on K20.

Measurements for the *h*-kernel on K20					
Code version	Thread block	GFLOPs	Registers/thread	Shared memory/block	Occupancy
Baseline	32×4	88.43	55	0	0.562
OPT-1	32×4	178.69	52	0	0.562
OPT-2	32×4	182.36	33	3264 bytes	0.750
OPT-3	34×6	190.45	34	3264 bytes	0.656
Measurements for the *s*-kernel on K20					
Code version	Thread block	GFLOPs	Registers/thread	Shared memory/block	Occupancy
Baseline	32×4	67.99	39	0	0.750
OPT-1	32×4	122.78	40	0	0.750
OPT-2	32×4	112.84	38	1632 bytes	0.750
OPT-3	34×6	110.55	33	1632 bytes	0.656

We can observe in Table 2 that using the read-only data cache on Kepler results in a substantial performance increase for both kernel functions. For the *h*-kernel function, OPT-2 and OPT-3 deliver additional performance improvements. For the *s*-kernel function, however, OPT-2 and OPT-3 are counter productive, as feared earlier. In Table 3, we can see that the performance of the baseline implementation on GTX 590 is very close to that of K20, with the baseline version of the *s*-kernel being even faster. Although the memory bandwidth and the double-precision capability of GTX 590 are lower than those of K20, the L1 cache on the Fermi architecture seems capable of exploiting the spatial locality and effectively utilizing the memory bandwidth. Another interesting phenomenon on GTX 590 is that configuring the L1 cache size as 48KB (PreferL1) didn't bring too much performance improvement, compared with the default L1 cache size (16KB). This is probably because the advantage of an increased L1 cache size is counteracted by a reduced possibility of data reuse between the thread blocks. Similar to K20, the best performance of the *h*-kernel also comes from OPT-3 on GTX 590, while the *s*-kernel works best with OPT-1. On GTX 590, the number of registers per SM is only half of that on K20. Therefore, to relieve the pressure on the registers, we used the shared memory to store one additional array on GTX 590.

Using the best combination, i.e., OPT-3 for the *h*-kernel and OPT-1 for the *s*-kernel, the achieved overall

Table 3. Effect of the performance-enhancing strategies on GTX 590.

Measurements for the h-kernel on GTX 590					
Code version	Thread block	GFLOPs	Registers/thread	Shared memory/block	Occupancy
Baseline	32×4	76.46	55	0	0.333
OPT-1(PreferL1)	32×4	81.36	55	0	0.333
OPT-2	32×4	76.98	29	4896 bytes	0.667
OPT-3	34×6	87.97	22	4896 bytes	0.875
Measurements for the s-kernel on GTX 590					
Code version	Thread block	GFLOPs	Registers/thread	Shared memory/block	Occupancy
Baseline	32×4	72.23	39	0	0.500
OPT-1(PreferL1)	32×4	75.11	39	0	0.500
OPT-2	32×4	56.49	29	3264 bytes	0.667
OPT-3	34×6	60.39	26	3264 bytes	0.730

GFLOPs rate were 165 on K20 and 82 on GTX 590, respectively. In our earlier work [10], a CUDA implementation that made use of the shared memory achieved slightly more than 50 GFLOPs on an M2050 Fermi GPU. Interestingly, the new K20 GPU together with enhanced programming thus produces a three-fold performance increase for the same sediment transport computations.

5. Concluding remarks

It should be readily admitted that the numerical experiments that we have used are far from exhaustive. A fine performance tuning with respect to, e.g., adjusting the thread block dimensions and strategically dividing data arrays between the on-chip shared memory and read-only data cache, will result in higher GFLOPs rates. Chunking, i.e., to let each CUDA thread compute more than one $h_{i,j}^{\ell+1}$ or $s_{i,j}^{\ell+1}$ value, can also be tested. Nevertheless, we think that our preliminary investigation shows that the resurgent benefit of the shared memory and the easily obtainable benefit of the new read-only data cache call for programmers' care when using NVIDIA's Kepler architecture.

Acknowledgments

The authors gratefully acknowledge the support from the National Natural Science Foundation of China under NSFC No. 61033008, SRFDP No. 20104307110002, Innovation in Graduate School of NUDT Nos. B100603, B120605 and CJ11-06-01, the FriNatek program of the Research Council of Norway No.214113. We would also like to thank the support from China Scholarship Council and the Simula School of Research and Innovation.

References

[1] J. Nickolls, W. J. Dally, The GPU computing era, IEEE micro 30 (2) (2011) 56 – 69.
[2] Inside Kepler, http://developer.download.nvidia.com/GTC/inside-tesla-kepler-k20-family.pdf.
[3] Tuning CUDA applications for Kepler, http://docs.nvidia.com/cuda/pdf/Kepler_Tuning_Guide.pdf.
[4] T. Aila, S. Laine, T. Karras, Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum, NVIDIA Technical Report NVR-2012-02 (2012).
[5] D. Cederman, B. Chatterjee, P. Tsigas, Understanding the performance of concurrent data structures on graphics processors, in: Euro-Par 2012 Parallel Processing, Vol. 7484 of Lecture Notes in Computer Science, Springer-Verlag, 2012, pp. 883–894.
[6] J. C. Rivenæs, A computer simulation model for siliclastic basin stratigraphy, Ph.D. thesis, University of Trondheim (1993).
[7] J. C. Rivenæs, Application of a dual-lithology, depth-dependent diffusion equation in stratigraphic simulation, Basin Research 4 (2) (2007) 133 – 146.
[8] W. Wei, S. R. Clark, H. Su, M. Wen, X. Cai, Balancing efficiency and accuracy for sediment transport simulations, http://heim.ifi.uio.no/xingca/Wei-etal-2012-CG.pdf (2012).
[9] S. R. Clark, W. Wei, X. Cai, Numerical analysis of a dual-sediment transport model applied to Lake Okeechobee, Florida, in: Proceedings of the 9th International Symposium on Parallel and Distributed Computing, IEEE Computer Society Press, 2010, pp. 189–194.
[10] M. Wen, H. Su, W. Wei, N. Wu, X. Cai, C. Zhang, Using 1000+ GPUs and 10000+ CPUs for sedimentary basin simulations, in: Proceedings of IEEE CLUSTER 2012, IEEE Computer Society Press, 2012, pp. 27–35.

Appendix A. Fragments of a CUDA implementation that targets Kepler

```
1   #define BLOCK_X B_X
2   #define BLOCK_Y B_Y
3   #define HALO    2
4   dim3 block(BLOCK_X+HALO,BLOCK_Y+HALO,1);
5   dim3 grid((Nx+(BLOCK_X-1))/(BLOCK_X),(Ny+(BLOCK_Y-1))/(BLOCK_Y),1);
6   update_h<<<grid,block>>>( alpha, beta, s, h, ht, Nx);

7   __global__ void update_h(const double * __restrict__ alpha,
8                            const double * __restrict__ beta,
9                            const double * __restrict__ s,
10                           const double * __restrict__ h,
11                           double *h_new,
12                           int Nx) {
        //Shared memory per thread block
13      __shared__ double gs_alpha[(BLOCK_X+HALO)*(BLOCK_Y+HALO)];
14      __shared__ double gm_beta [(BLOCK_X+HALO)*(BLOCK_Y+HALO)];
15      int tid, gid_x, gid_y, gid;
16      tid   = threadIdx.x + threadIdx.y * blockDim.x;
17      gid_x = threadIdx.x + blockIdx.x * BLOCK_X;
18      gid_y = threadIdx.y + blockIdx.y * BLOCK_Y;
19      gid   = gid_x + gid_y * (Nx+2);
        //All threads compute the intermediate s*alpha and (1-s)*beta values
20      gs_alpha[tid] = s[gid] * alpha[gid];
21      gm_beta[tid]  = (1.0-s[gid]) * beta[gid];
22      __syncthreads();
        //Only a subset of the threads do subsequent computations
23      if(tid<(BLOCK_X*BLOCK_Y))
24      {
25          double h_center, h_left, h_right, h_up, h_down;
26          gid_x = (tid%BLOCK_X) + blockIdx.x * BLOCK_X + 1;
27          gid_y = (tid/BLOCK_X) + blockIdx.y * BLOCK_Y + 1;
28          gid   = gid_x + gid_y * (Nx+2);
39          h_center = h[gid];
30          h_left   = h[gid-1];
31          h_right  = h[gid+1];
32          h_up     = h[gid + (Nx+2)];
33          h_down   = h[gid - (Nx+2)];
            ......
      }
    }
```