Communication-Avoiding Optimization of Geometric Multigrid on GPUs

Amik Singh
James Demmel, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-258
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-258.html

December 14, 2012

Acknowledgement

I would like to thank Samuel Williams, Research Scientist at Lawrence Berkeley National Laboratory (LBNL) for guiding me throughout the project and giving valuable feedbacks. Also, I would like to thank my research adviser Professor James Demmel for his suggestions and feedback for my project.

Communication-Avoiding Optimization of Geometric Multigrid on GPUs

Amik Singh

EECS Department, University of California, Berkeley

Abstract – Multigrid methods are widely used to accelerate the convergence of iterative solvers for linear systems in a number of different application areas. In this report, we explore communication-avoiding implementations of Geometric Multigrid on Nvidia GPUs. We achieved an overall gain of 1.2x for the whole multigrid algorithm over baseline implementation. We also provide an insight into what future GPUs need to have in terms of on chip and shared memory for these kinds of algorithms to perform even better.

1. Introduction

Traditionally, algorithm design focuses on minimizing floating point operations as the key to performance. However, as the gap between interprocessor communication and computation continues to grow [8], focus has shifted to the problem of reducing data movement between various levels of a memory hierarchy or between different nodes on a parallel machine. Thus, the idea of a "communication-avoiding" (CA) algorithm has become a concept of critical research importance. CA algorithms may differ from traditional flop-minimizing approaches as they asymptotically minimize communication, sometimes at the cost of extra flops. Solving a problem in a CA manner may involve a significant amount of algorithmic innovation.

2. Related Work

We explore the Geometric Multigrid on GPU's. Bell et al. have explored the performance of algebraic (sparse rather than structured) multigrid on GPUs [9]. Sellapa et al. explore constant coefficient elliptical partial differential equations on structured grids [11]. Perhaps the most closely related work is that performed in Treibig's, which implements a 2D GSRB on SIMD architectures by separating and reordering the red and black elements [10], additionally a 3D multigrid on an IA-64 (Itanium) is implemented via temporal blocking. Our work expands on these efforts by providing a unique set of optimization strategies for GPUs.

3. Multigrid methods

Multigrid (MG) methods provide a powerful technique to accelerate the convergence of iterative solvers for linear systems and are therefore used extensively in a variety of numerical simulations. Conventional iterative solvers operate on data at a single resolution and often require too many iterations to be viewed as computationally efficient. Multigrid simulations create a hierarchy of grid levels and use corrections of the solution from iterations on the coarser levels to improve the convergence rate of the solution at the finest level. Ideally, multigrid is an O(N) algorithm; thus, performance optimization on our studied multigrid implementation can only yield constant speedups.

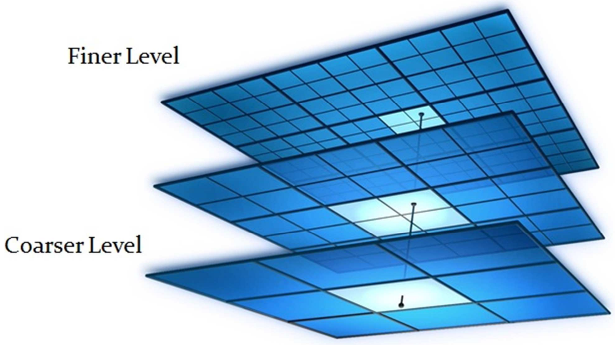


Fig 1 : Multigrid Overview

Figure 2 shows the three phases of the multigrid V-cycle for the solve of $Lu^h = f^h$. First, a series of smooths reduce the error while restrictions of the residual create progressively coarser grids.

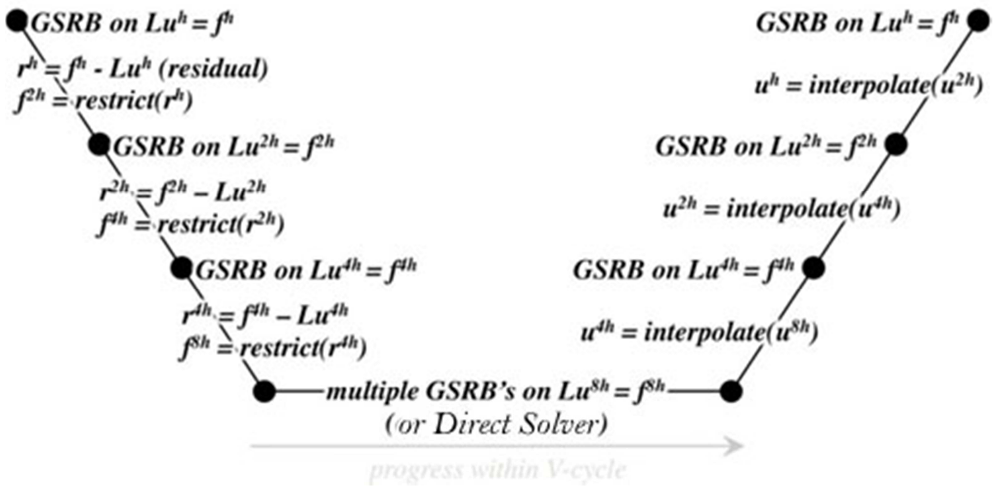

GSRB on $Lu^h = f^h$
$r^h = f^h - Lu^h$ (residual)
$f^{2h} = restrict(r^h)$
GSRB on $Lu^{2h} = f^{2h}$
$r^{2h} = f^{2h} - Lu^{2h}$
$f^{4h} = restrict(r^{2h})$
GSRB on $Lu^{4h} = f^{4h}$
$r^{4h} = f^{4h} - Lu^{4h}$
$f^{8h} = restrict(r^{4h})$
multiple GSRB's on $Lu^{8h} = f^{8h}$
(or Direct Solver)
progress within V-cycle
$u^{4h} = interpolate(u^{8h})$
GSRB on $Lu^{4h} = f^{4h}$
$u^{2h} = interpolate(u^{4h})$
GSRB on $Lu^{2h} = f^{2h}$
$u^h = interpolate(u^{2h})$
GSRB on $Lu^h = f^h$

Fig 2 : V-cycle

The smoother is a conventional relaxation such as Jacobi, successive over-relaxation (SOR), or Gauss-Seidel with Red-Black ordering (GSRB) which we used in our study as it has superior convergence properties. The restriction of the residual (f^h - Lu^h) is used to define the right-hand side at the next coarser grid. At each progressively coarser level, the correction (e.g. u^{2h}) is initialized to zero. Second, once coarsening stops (the grid size reaches one or terminated for performance when the whole grid fits into cache), the algorithm switches to a bottom solver like CG/BiCGStab. But in our case, instead of doing a bottom solver we do a lot of GSRB relaxations which is equivalent to a bottom solve. Finally, the coarsest correction is interpolated back up the V-cycle to progressively finer grids where it is smoothed.

Nominally, one expects an order of magnitude reduction in the residual per V-cycle. As each level performs O(1) operations per grid point and 1/8 (in 3D) the work of the finer grid, the overall computation is O(N) in the number of variables in u. The linear operator can be arbitrarily complex as dictated by the underlying physics, with a corresponding increase in runtime to perform the smoother computation.

4. <u>Experimental Setup</u>

We use NVIDIA's GPUs to do our experiments. NVIDIA provides CUDA as the programming paradigm to write programs on GPU. For our experiments we use Dirac GPU cluster at National Energy Research Scientific Computing Center (NERSC). Dirac is a 50 node GPU cluster with NVIDIA Fermi chips. Each GPU node also contains 2 Intel 5530 2.4 GHz, 8MB cache, 5.86GT/sec QPI Quad core Nehalem processors (8 cores per node) and 24GB DDR3-1066 Reg ECC memory. We use Tesla C2050 Fermi GPU whose specifications are given below in table 1.

Form Factor	9.75" PCIe x16 form factor
# of CUDA Core	448
Frequency of CUDA Cores	1.15 GHz
Double Precision floating point performance (peak)	515 Gflops
Single Precision floating point performance (peak)	1.03 Tflops
Total Dedicated Memory	3GB GDDR5
Memory Speed	1.5 GHz
Memory Interface	384-bit
Memory Bandwidth	144 GB/sec
Power Consumption	238W TDP

Table 1 :- Specifications of Tesla C2050 Fermi GPU

5. <u>Problem Specification</u>

The problem size is fixed to 256^3 nodes on the finest grid. We construct a compact multigrid solver benchmark on GPU that creates a global 3D domain partitioned into subdomains sized to proxy those found in real MG applications. So, the 256^3 nodes are divided into sixty four, 64^3 subdomains. All subdomains must explicitly exchange ghost zones with their neighboring subdomains, ensuring an effective communication proxy of MG codes.

We use a single-precision, finite volume discretization of the variable-coefficient operator $L = a\bar{\alpha}I + b\nabla\bar{\beta}\nabla$ with periodic boundary conditions as the linear operator within our test problem. Here a and b are scalar variable coefficients while $\bar{\alpha}$ and $\bar{\beta}$ are vectors. Variable-coefficients are an essential (yet particularly challenging) facet as most real-world applications demand it. The right-hand side (f) for our benchmarking is $\sin(\pi x)\sin(\pi y)\sin(\pi z)$ on the [0,1] cubical domain. The u, f, and α are cell-centered data, while the β's are face centered. Figure 3 shows the 3-D stencil representation of the grid.

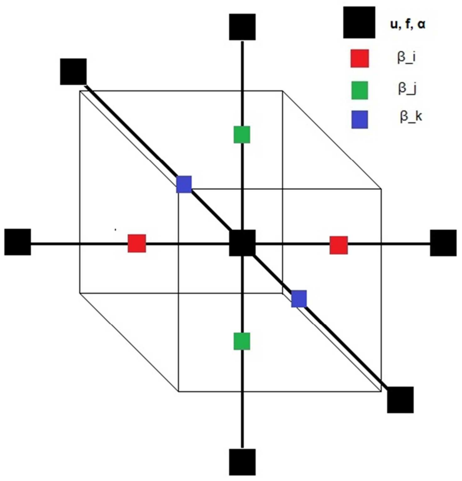
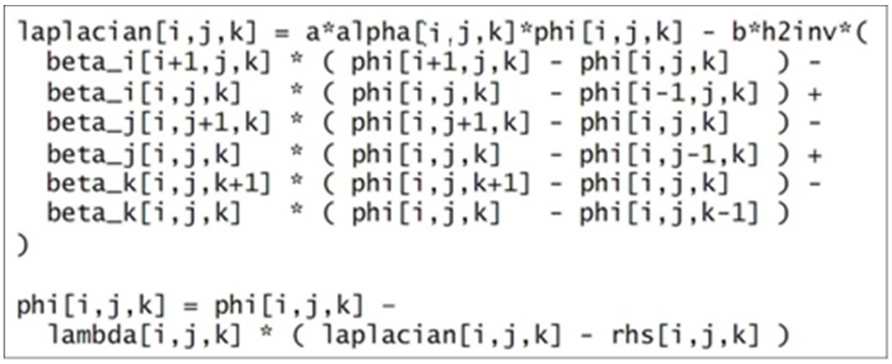


Fig 3 : 3-D stencil representation of grid

We structure a truncated V-cycle where restriction stops at the coarsest level of 4^3. We fix the number of V-cycles at 10 and perform two relaxations at each level down the V-cycle, 24 relaxations at the bottom, and two relaxations at each level up the V-cycle.

Our relaxation scheme uses Gauss-Seidel Red-Black (GSRB) which offers superior convergence compared to other methods. It consists of two grid smooths per relaxation each updating one color at a time, for a total of eight smooths per subdomain per level per V-cycle. The pseudocode for the resultant inner operation is shown in Figure 4. A similar calculation is used for calculating the residual. Nominally, these operators require a one element deep ghost zone constructed from neighboring subdomain data. However, in order to leverage communication aggregation and communication avoiding techniques, we also explore a 4-deep ghost zone that enables optimization at the expense of redundant computation.

The data structure within a level for a subdomain is a list of equally-sized grids (arrays) representing the correction, right-hand side, residual, and coefficients each stored in a separate array. Our implementations



```
laplacian[i,j,k] = a*alpha[i,j,k]*phi[i,j,k] - b*h2inv*(
  beta_i[i+1,j,k] * ( phi[i+1,j,k] - phi[i,j,k]   ) -
  beta_i[i,j,k]   * ( phi[i,j,k]   - phi[i-1,j,k] ) +
  beta_j[i,j+1,k] * ( phi[i,j+1,k] - phi[i,j,k]   ) -
  beta_j[i,j,k]   * ( phi[i,j,k]   - phi[i,j-1,k] ) +
  beta_k[i,j,k+1] * ( phi[i,j,k+1] - phi[i,j,k]   ) -
  beta_k[i,j,k]   * ( phi[i,j,k]   - phi[i,j,k-1] )
)

phi[i,j,k] = phi[i,j,k] -
  lambda[i,j,k] * ( laplacian[i,j,k] - rhs[i,j,k] )
```

Fig 4 : Smoothing Operation

ensure that the core data structures remain relatively unchanged with optimization. Although it has been shown that separation of red and black points into separate arrays can facilitate SIMDization [25], our benchmark forbids such optimizations as they lack generality and challenge other phases.

6. Single Precision Implementation

Baseline :-

The baseline implementation refers to the 1 deep ghost zone implementation. Our 256^3 grid is divided into 64 sub-grids of size 64^3 at finest level. The levels go down till the size of each subgrid is 4^3 . So with 1 deep ghost zones the grid sizes and array sizes at each level are :-

Level	Grid Size	Array Size
0	64^3	66^3
1	32^3	34^3
2	16^3	18^3
3	8^3	10^3
4	4^3	6^3

Table 2 :- Grid and array sizes at different levels for baseline

At each level the smoothing operation (figure 3) is done 4 times. After each smoothing the ghost zone cells are explicitly communicated with neighboring sub-grids. To do a smoothing operation at level 0, 66^3 subgrid is divided into smaller regions of 18x18x66 grids. Each of these 18x18x66 grids is updated by 18x18 thread block. Note that this 18x18 thread block is responsible for updating the inner 16x16x64 block. The mapping is shown schematically in figure 5. Each 18x18 thread block sweeps over 66 elements in the z direction in wavefront approach. Figure 6 shows the wavefront approach wherein only one smoothing operation is shown. We load in 3 planes into local register memory and do computation for the central plane of 18X18 points before loading in the next plane. The time taken for smoothing at the finest level was 1.03 seconds whereas total time to solve multigrid was 1.83 seconds.

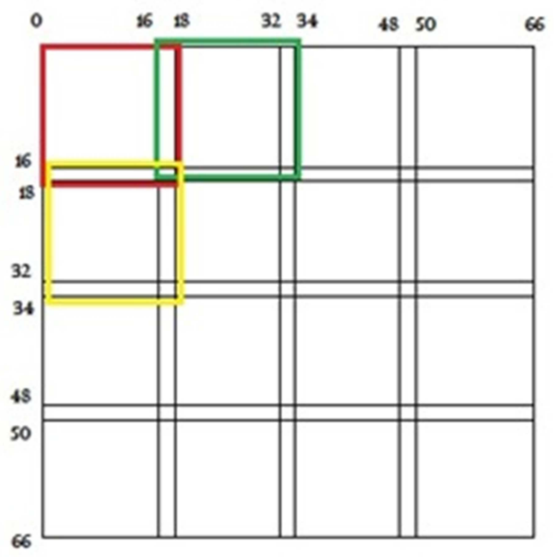


Fig 5 : Division of 66 cubed grid

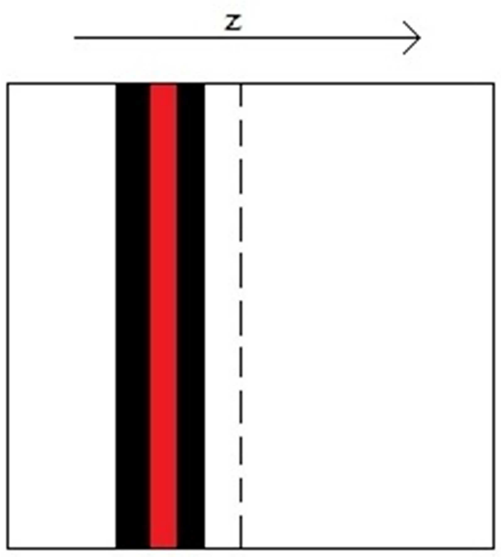


Fig 6 : Wavefront approach for baseline implementation

Communication Avoiding (CA) implementation :-

In our CA version we have 4 deep ghost zones for every sub-grid we have. So the sub-grid size at each level is:-

Level	Grid Size	Array Size
0	64^3	72^3
1	32^3	40^3
2	16^3	24^3
3	8^3	16^3
4	4^3	12^3

Table 3 :- Grid and array sizes at different levels for CA

Our smoothing kernel does 4 smooths in one pass. So we save on loading grids 4 times from global GPU memory to local memory of each thread. Ideally one should attain a decrease in memory bandwidth consumption for the smoothing operation by 0.25 and hence a speedup of 4 times for the smoothing operation only, but we have to account for extra ghost zones to be loaded. So even if we attain the maximum possible bandwidth our ideal speedup for the smoothing kernel should be (4*66*66*66)/(72*72*72) which is nearly equal to 3x. The time consumed in other operations such as restriction and interpolation remains same as we don't have to deal with ghost zones for these operations (Table 4 and 5).

The maximum thread block size on Nvidia GPUs can be 1536. So, we can't have the entire grid assigned to one thread block. We also need to have multiple thread blocks for maximum utilization of GPU cores. While dividing the smoothing operation into different thread blocks we have to load ghost zones for each sub-divided portion. Hence we also have to pay the price to load more extra memory in our thread blocks to save on inter-thread block communication for each of the CUDA thread blocks. For example, at level 0, 72^3 is divided into small nx*ny*72 thread blocks. For nx = 16 and ny = 16 the mapping is as shown below in figure 7. This 16x16x72 block is only responsible for updating inner 8x8x64 thread block. So, just to do smoothing on 8x8x72 grid we have to load in nearly 4 times the size of this block. There is no other way to save on inter thread block computation in CUDA but to do these redundant computations.

For nx = 16 and ny = 16, the amount of global memory accessed at level 0 is 16*16*72*(8*8) (words per variable). The baseline implementation accesses 4*66*66*66 in 4 smoothing operations which is fixed. So, by changing nx and ny we can change the ratio of memory accessed in CA versus memory accessed in baseline. Figure 8 shows the theoretical ratio (data to be loaded) and the measured ratio of time to solve. The data normalized series is the amount of data we have to load in our CA version compared to the baseline implementation. This data normalized curve is similar our time normalized curve which shows the measured run times. But we could not go beyond 24*24 thread block size. The Fermi GPU which we use has a maximum of 32K registers per streaming multiprocessor. To store in all the running planes (wavefront approach) we use about 56 registers per thread. Also as per CUDA programming guide each thread block is run on a single streaming multiprocessor only. So, the maximum number of threads we can have in a thread block are 32K/56 ~ 585. So we could only run a maximum of 24*24 = 576 threads in a thread block. The figure however also shows the expected data to be loaded for bigger thread block sizes and we believe the run time should also vary according to that. The points shown with red dotted lines are hypothetical as such the current GPU's cannot run such big thread blocks. Also the ideal time to run a CA version should be ¼ times that of baseline.

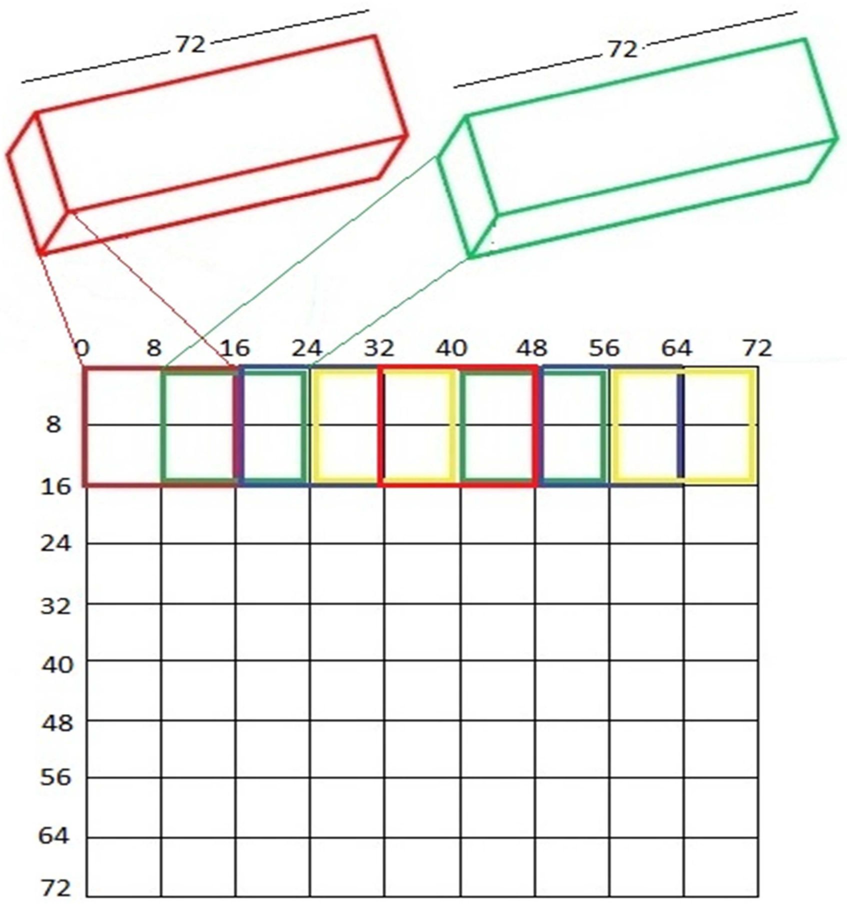


Fig 7 : Mapping of grid onto thread blocks

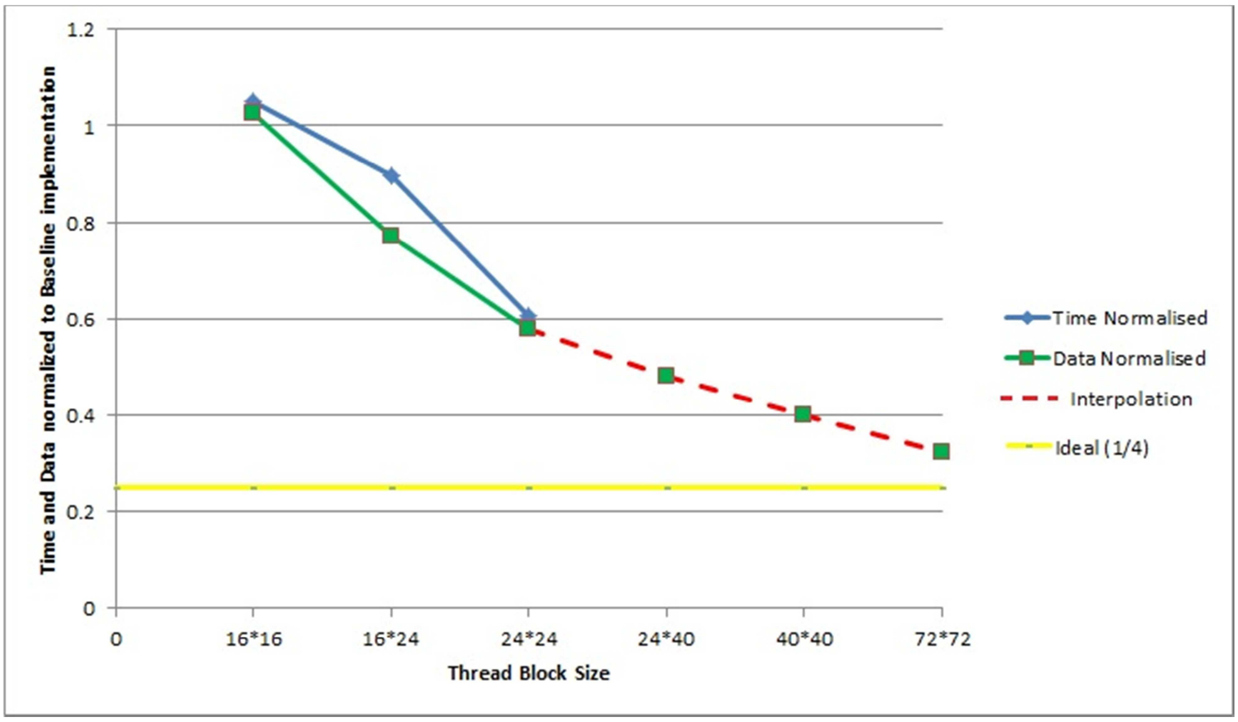


Fig 8 : Results and Prediction

Table 4 and Table 5 show our best CA implementation and baseline time analysis for different operations at different levels respectively. We were able to decrease the total smoothing time by 1.6x (1.1861/0.7425). The time for other kernels (residual, restriction and interpolation) is almost same within the errors of experiment

level	0	1	2	3	4	
size	72^3	40^3	24^3	16^3	12^3	Total
smooth (s)	0.6287	0.0774	0.0112	0.0063	0.0188	0.7425
avoiding RAW (s)	0.0549	0.0100	0.0026	0.0011	0.0000	0.0687
residual (s)	0.1158	0.0143	0.0022	0.0006	0.0000	0.1328
restriction (s)	0.0112	0.0019	0.0006	0.0004	0.0000	0.0141
interpolation (s)	0.0232	0.0036	0.0009	0.0005	0.0000	0.0283
communication (s)	0.3101	0.1162	0.0625	0.0416	0.0792	0.6095
Total (s)	1.0890	0.2134	0.0775	0.0494	0.0980	1.5272

Table 4 :- Time spent at different stages by CA version

level	0	1	2	3	4	
size	66^3	34^3	18^3	10^3	6^3	Total
smooth (s)	1.0343	0.1062	0.0191	0.0056	0.0209	1.1861
residual (s)	0.1149	0.0129	0.0018	0.0005	0.0000	0.1301
restriction (s)	0.0103	0.0017	0.0006	0.0004	0.0000	0.0129
interpolation (s)	0.0218	0.0035	0.0009	0.0005	0.0000	0.0266
communication (s)	0.2305	0.0833	0.0440	0.0327	0.0804	0.4710
Total (s)	1.4118	0.2075	0.0663	0.0397	0.1014	1.8268

Table 5 :- Time spent at different stages by baseline version

level	0	1	2	3	4
Raw CA Gflops	34.91	45.37	60.84	27.88	40.85
Useful CA Gflops	13.34	13.54	11.70	2.60	1.31
Baseline (Gflops)	8.11	9.87	6.86	2.93	1.23

Table 6 :- Gflops for the smoothing operation

level	0	1	2	3	4
Raw CA GB/s	43.21	48.78	50.55	26.63	22.59
Useful CA GB/s	24.31	33.87	50.55	26.63	22.59
Baseline GB/s	45.53	60.36	50.02	29.25	10.15

Table 7 :- Bandwidth utilization

runs as all these operations are exactly identical for both the versions. The avoiding RAW time in the CA version depicts the time we have to copy back our data to the original grid from a redundant grid used for concurrency purposes while doing 4 smooths in a GPU kernel. The communication time refers to the explicit exchange of ghost zones between different subgrids. The communication time is a bit more in the CA version because we have to communicate 4 deep ghost zones compared to 1 deep ghost zones in case of baseline version. Overall we were able to make some improvement in the total running time of multigrid solve. Table 6 shows the floating point operations done per second in the smoothing kernel of both the CA and baseline version. Raw flops include the operations done on ghost zones. The useful flops are the flops done on actual grid points divided by two because we commit only one of either red or black at each point. With other stages forming a significant time for the overall multigrid solve, even if we achieve ideal speedup for the CA version we can reduce the smoothing time to 0.395 sec (1.1861/3) and the total time to solve the multigrid to 1.18 sec (instead of 1.5272 seconds). So, the net speedup in the ideal case would be around 1.55x (1.8268/1.18) only. With limited resources on GPU, we achieve an overall speedup of 1.2x only. Note that our baseline implementation is itself optimized in the sense that it uses a wavefront approach to do the smoothing operation. Table 7 shows the bandwidth utilization of the smoothing operation at different levels. The peak bandwidth of the machine we use is 97.6 GB/s with ECC on.

7. Comments

We also explored the Double Precision implementation but with so many registers required in the single precision implementation itself, the DP version performed very poorly. Table 7 shows the run time of double precision CA version. One would expect to see a two-fold increase in runtime (due to doubling of memory to be

level	0	1	2	3	4	
size	72^3	40^3	24^3	16^3	12^3	total
smooth (s)	3.360116	0.472085	0.069302	0.009616	0.027259	3.938378
avoiding RAW (s)	0.04511	0.006516	0.001823	0.000933	0	0.054381
residual (s)	0.092837	0.013105	0.002034	0.000564	0	0.10854
restriction (s)	0.010477	0.001715	0.000547	0.000366	0	0.013105
interpolation (s)	0.025945	0.003732	0.000712	0.000461	0	0.03085
communication (s)	0.176513	0.064105	0.034783	0.023547	0.050514	0.349462
Total (s)	3.665889	0.554742	0.107379	0.034553	0.077791	4.440354

Table 7 :- Time spent at different stages by double precision CA version

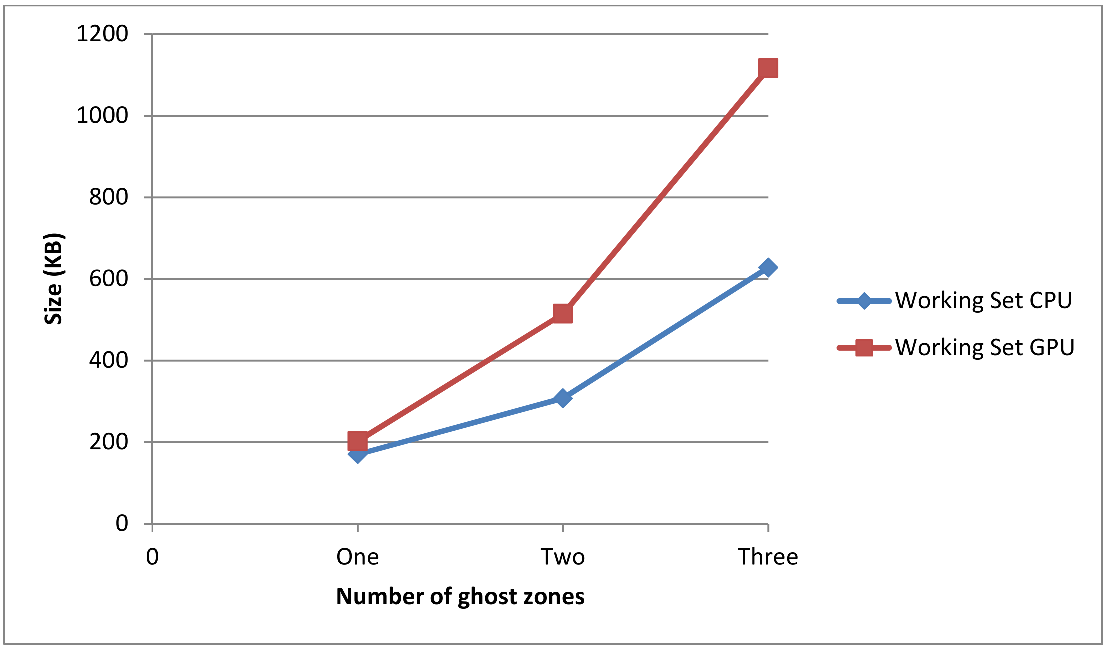


Fig 9 : Working set of memory on CPU and GPU

loaded), but the run time of this double precision version is nearly three times that of single precision implementation due to register spilling on GPU cores. We speculate that if in future GPUs more registers are available to address this issue, then our DP version would also perform well. Also for such multigrid algorithms and higher order stencil operations GPUs need to have much large memory space in terms of local registers and some mechanism to tackle inter-thread block communication to save redundant memory operations. Figure 9 shows the working set of data for CPU and GPU for different ghost zones. The working set is different because for the GPU we have to load in extra elements for each thread block to avoid inter-thread block communication. We don't have to deal with that issue on a CPU. If we try this CA approach on a Sandy Bridge CPU which has 20 MB of L3 cache, almost all data fits in L3 cache. Moreover the L2 cache (256 KB) and L1 cache (64 KB) are also very large compared to what GPU's have (48 KB of shared + L1 cache) per streaming multiprocessor. This shows GPU's are starved for on-chip memory and for writing efficient higher order stencil algorithms GPU's need to have a larger set of on-chip memory.

8. Acknowledgement

I would like to thank Samuel Williams, Research Scientist at Lawrence Berkeley National Laboratory (LBNL) for guiding me throughout the project and giving valuable feedbacks. Also, I would like to thank my research adviser Professor James Demmel for his suggestions and feedback for my project.

9. References

[1] S. Williams, D. Kalamkar, **A. Singh**, A. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, L. Oliker, "Optimization of Geometric Multigrid for Emerging Multi- and Manycore Processors", Supercomputing (SC), November 2012.

[2] Dirac Website. http://www.nersc.gov/users/computational-systems/dirac/

[3] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In Proceedings of the 2010 ACM/IEEE International Conferencefor High Performance Computing, Networking, Storage and Analysis, SC'10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[4] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. International Journal of High Performance Computing Applications, 18(1):115–133, 2004.

[5] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In Proceedings of the 3rd Conference on Computing Frontiers, New York, NY, USA, 2006.

[6] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In International Computer Software and Applications Conference, pages 579–586, 2009.

[7] N. Bell, S. Dalton, and L. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. NVIDIA Technical Report NVR-2011-002, NVIDIA Corporation, June 2011.

[8] Graham, S. L., Snir, M., and Patterson, C. A., Eds. 2004. Getting up to Speed: The Future of Supercomputing. Report of National Research Council of the National Academies Sciences. The National Academies Press, Washington, D.C. 289 pages, http://www.nap.edu.

[9] N. Bell, S. Dalton, and L. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. NVIDIA Technical Report NVR-2011-002, NVIDIA Corporation, June 2011.

[10] J. Treibig. Efficiency improvements of iterative numerical algorithms on modern architectures. PhD thesis, 2008.

[11] S. Sellappa and S. Chatterjee, "Cache-efficient multigrid algorithms," International Journal of High Performance Computing Applications, vol. 18, no. 1, pp. 115–133, 2004.