

A CUDA implementation of the High Performance Conjugate Gradient benchmark

Everett Phillips and Massimiliano Fatica

NVIDIA Corporation
Santa Clara, CA 95050, USA

Abstract. The High Performance Conjugate Gradient (HPCG) benchmark has been recently proposed as a complement to the High Performance Linpack (HPL) benchmark currently used to rank supercomputers in the Top500 list. This new benchmark solves a large sparse linear system using a multigrid preconditioned conjugate gradient (PCG) algorithm. The PCG algorithm contains the computational and communication patterns prevalent in the numerical solution of partial differential equations and is designed to better represent modern application workloads which rely more heavily on memory system and network performance than HPL. GPU accelerated supercomputers have proved to be very effective, especially with regard to power efficiency, for accelerating compute intensive applications like HPL. This paper will present the details of a CUDA implementation of HPCG, and the results obtained at full scale on the largest GPU supercomputers available: the Cray XK7 at ORNL and the Cray XC30 at CSCS. The results indicate that GPU accelerated supercomputers are also very effective for this type of workload.

1 Introduction

After twenty years of the High Performance Linpack (HPL) benchmark, it is now time to complement this benchmark with a new one that can stress different components in a supercomputer. HPL solves a dense linear system using Gaussian Elimination with partial pivoting, and its performance is directly correlated with dense matrix-matrix multiplication. While there are applications with similar workload (material science codes like DCA++ or WL-LSMS, both winners of the Gordon Bell awards), the vast majority of applications cannot be recast in terms of dense linear algebra and their performance poorly correlates with the performance of HPL.

In 2013, Dongarra and Heroux [1] proposed a new benchmark designed to better represent modern application workloads that rely more heavily on memory system and network performance than HPL. The new benchmark, HPCG, solves a large sparse linear system using an iterative method. It is an evolution of one of the Mantevo Project applications from Sandia [12]. The Mantevo Project was an effort to provide open-source software packages for the analysis, prediction and improvement of high performance computing applications. This is not the

first time that a new benchmark has been proposed to replace or augment the Top 500 list. The HPCG benchmark suite [2] and the Graph 500 benchmark [4] are two well known proposals, but up to now the uptake has been limited. Graph 500 after 4 years is still listing only 160 systems.

This paper presents a CUDA implementation of HPCG and the results on large supercomputers. Although we use CUDA, the algorithms and methods are applicable in general on highly parallel processors. The paper is organized as follows: after a short introduction to CUDA, we describe the algorithmic details of HPCG. A description of the CUDA implementation and optimization is then given, followed by a section on results and comparison with available data.

2 GPU Computing and CUDA

The use of GPUs in high performance computing, sometimes referred to as *GPU computing*, is becoming very popular due to the high computational power and high memory bandwidth of these devices coupled with the availability of high level programming languages.

CUDA is an entire computing platform for C/C++/Fortran on the GPU. Using high-level languages, GPU-accelerated applications run the sequential part of their workload on the CPU - which is optimized for single-threaded performance - while accelerating parallel processing on the GPU.

CUDA follows the data-parallel model of computation. Typically each thread executes the same operation on different elements of the data in parallel. Threads are organized into a 1D, 2D or 3D grid of thread-blocks. Each block can be 1D, 2D or 3D in shape, and can consist of up to 1024 threads on current hardware. Threads within a thread block can cooperate via lightweight synchronization primitives and a high-speed on-chip shared memory cache.

Kernel invocations in CUDA are asynchronous, so it is possible to run CPU and GPU in parallel. Data movement can also be overlapped with computations and GPU can DMA directly from page-locked host memory. There are also a large number of libraries, from linear algebra to random number generation. Two libraries that are particularly relevant to this benchmark are CUBLAS [8] and CUSPARSE [9], that implement linear algebra operations on dense or sparse matrices. In the benchmark, we also used Thrust [10], a C++ template library for CUDA based on the Standard Template Library (STL), to sort and find unique values.

3 HPCG

The new HPCG benchmark is based on an additive Schwarz Preconditioned Conjugate Gradient (PCG) algorithm [3].

The benchmark has 8 distinct phases:

1. Problem and Preconditioner setups
2. Optimization phase

3. Validation testing
4. Reference sparse Matrix-vector multiply and Gauss-Seidel kernel timings
5. Reference PCG timing and residual reduction
6. Optimized PCG setup
7. Optimized PCG timing and analysis
8. Report results

During the initial setup, data structures are allocated and the sparse matrix is generated. The sparse linear system used in HPCG is based on a simple elliptic partial differential equation discretized with a 27-point stencil on a regular 3D grid. Each processor is responsible for a subset of matrix rows corresponding to a local domain of size $N_x \times N_y \times N_z$, chosen by the user in the `hpcg.dat` input file. The number of processors is automatically detected at runtime, and decomposed into $P_x \times P_y \times P_z$, where $P = P_x P_y P_z$ is the total number of processors. This creates a global domain $G_x \times G_y \times G_z$, where $G_x = P_x N_x$, $G_y = P_y N_y$, and $G_z = P_z N_z$. Although the matrix has a simple structure, it is only intended to facilitate the problem setup and validation of the solution, and may not be taken advantage of to optimize the solver.

Between the initial setup and validation, the benchmark calls a user-defined optimization routine, which allows for analysis of the matrix, reordering of the matrix rows, and transformation of data structures, in order to expose parallelism and improve performance of the SYMGS smoother. This generally requires reordering matrix rows using graph coloring for performance on highly parallel processors such as GPUs. However, this introduces a slowdown in the rate of convergence, which in turn increases the number of iterations required to reach the solution. The time for these additional iterations, as well as the time for the optimization routine, is counted against the final performance result.

Next, the benchmark calls the reference PCG solver for 50 iterations and stores the final residual. The optimized PCG is then executed for one cycle to find out how many iterations are needed to match the reference residual. Once the number of iterations is known, the code computes the number of PCG sets required to fill the entire execution time. The benchmark can complete in a matter of minutes, but official results submitted to Top500 require a minimum of one hour duration.

3.1 The PCG algorithm

The PCG algorithm solves a linear system $Ax = b$ given an initial guess x_0 with the following iterations:

Algorithm 1 Preconditioned Conjugate Gradient [13]

```
1:  $k = 0$ 
2: Compute the residual  $r_0 = b - Ax_0$ 
3: while ( $\|r_k\| < \epsilon$ ) do
4:    $z_k = M^{-1}r_k$ 
5:    $k = k + 1$ 
6:   if  $k = 1$  then
7:      $p_1 = z_0$ 
8:   else
9:      $\beta_k = r_{k-1}^T z_{k-1} / r_{k-2}^T z_{k-2}$ 
10:     $p_k = z_{k-1} + \beta_k p_{k-1}$ 
11:   end if
12:    $\alpha_k = r_{k-1}^T z_{k-1} / p_k^T A p_k$ 
13:    $x_k = x_{k-1} + \alpha_k p_k$ 
14:    $r_k = r_{k-1} - \alpha_k A p_k$ 
15: end while
16:  $x = x_k$ 
```

We can identify these basic operations:

- A. Vector inner products $\alpha := y^T z$. Each MPI process computes its local inner product and then calls a collective reduction to get the final value.
- B. Vector updates $w = \alpha y + \beta z$. These are local updates, where performance is limited by the memory system.
- C. Application of the preconditioner $w := M^{-1}y$, where M^{-1} is an approximation to A^{-1} . The preconditioner is an iterative multigrid solver using a symmetric Gauss-Seidel smoother (SYMGS). Application of SYMGS at each grid level involves neighborhood communication, followed by local computation of a forward sweep (update local elements in row order) and backward sweep (update local elements in reverse row order) of Gauss-Seidel. The ordering constraint makes the SYMGS routine difficult to parallelize, and is the main challenge of the benchmark.
- D. Matrix-vector products Ay . This operation requires neighborhood communication to collect the remote values of y owned by neighbor processors, followed by multiplication of the local matrix rows with the input vector. The pattern of data access is similar to a sweep of SYMGS, however the rows may be trivially processed in parallel since there are no data dependencies between rows (the output vector is distinct from the input vector).

All of these are BLAS1 (vector-vector) or BLAS2 (sparse matrix-vector) operations. We are not able to use BLAS3 operations, such as DGEMM, as we were able to do for HPL. An important point is that the benchmark is not about computing a highly accurate solution to this problem, but is only intended to measure performance of the algorithm.

3.2 Preconditioner

The problem is solved using a domain decomposition where each subdomain is locally preconditioned. The preconditioner in initial version (v1.x) was based on a symmetric Gauss-Seidel sweep. The latest version (v2.x) is based on a multigrid preconditioner where the pre and post smoothers are also a symmetric Gauss-Seidel sweep.

Gauss-Seidel preconditioner Since the PCG method could be used only on a symmetric positive definite matrix, the preconditioner must also be symmetric and positive definite. The matrix M is computed from lower triangular (L), diagonal (D) and upper triangular (U) parts of A :

$$M_{SGS} = (D + L)D^{-1}(D + U)$$

It is easy to verify that this matrix is symmetric and positive definite using the identity $(D + U)^T = (D + L)$. The application of the preconditioner requires the solution of upper and lower triangular systems.

Multigrid preconditioner The latest version of the benchmark is using a multigrid preconditioner instead of the simple iterative Gauss-Seidel. An iterative solver like Gauss-Seidel is very effective in damping the high frequency components of the error, but is not very effective on the low frequency ones. The idea of the multigrid is to represent the error from the initial grid on a coarser grid where the low frequency components of the original grid become high frequency components on the coarser one [14]. The multigrid V-cycle includes the following steps:

- A. Perform a number of Gauss-Seidel iterations to smooth the high frequencies and compute the residual $r^H = Ax^H - b$, where the superscript H denotes the grid spacing.
- B. Transfer the residual r^H on a coarser grid of space $2H$. This operation is often called *restriction*, and R the restriction matrix.

$$r^{2H} = Rr^H$$

- C. Perform a number of Gauss-Seidel iterations to smooth the error on the coarser grid residual equation

$$Ae^{2H} = r^{2H}$$

- D. Transfer the correction e^{2H} back on the fine grid of space H . This operation is often called *prolongation*, and P the prolongation matrix.

$$e^H = Pe^{2H}$$

The process can be extended to multiple levels. The HPCG benchmark is using a V-cycle strategy with 3 coarser levels and performs a single pre- and post- smoother Gauss-Seidel at each level.

3.3 Selecting node count

HPCG detects the number of MPI tasks at runtime and tries to build a 3D decomposition. Clearly if the number of tasks, N , is a prime, the only possible 3D decomposition is $N \times 1 \times 1$ (or a permutation). While this is a valid configuration, it is highly unlikely that a real code would run with such a configuration. We always try to select a 3D configuration that is as balanced as possible. Since the jobs on large supercomputers go through a batching system and the number of available nodes may vary due to down nodes, it is useful to know the best node count in a certain range. We have extracted the routine internally used by HPCG and made a standalone program that we use to analyze the possible decompositions. A simple criterion is to sort N_1, N_2, N_3 and compute the product of the ratios $N_{\text{max}}/N_{\text{min}}$ and $N_{\text{mid}}/N_{\text{min}}$. The closer to the unity this product is, the more balanced the decomposition is.

4 CUDA implementation

The GPU porting strategy is primarily focused on the parallelization of the Symmetric Gauss-Seidel smoother (SYMGS), which accounts for approximately two thirds of the benchmark Flops. This function is difficult to parallelize due to the data dependencies imposed by the ordering of the matrix rows. Although it is possible to use analysis of the matrix structure to build a dependency graph which exposes parallelism, we find it is more effective to reorder the rows using graph coloring.

Our implementation begins with a baseline using CUDA libraries, and progresses into our final version in the following steps:

- A. CUSPARSE (CSR)
- B. CUSPARSE + color ordering (CSR)
- C. Custom Kernels + color ordering (CSR)
- D. Custom Kernels + color ordering (ELL)

4.1 Baseline CUSPARSE

Starting with CUSPARSE has the benefit of keeping the coding effort low, and hiding the complexity of parallelizing the Symmetric-Gauss-Siedel smoother. It also allows us to easily validate the results against the reference solution, and perform experiments with matrix reordering.

With CUSPARSE, we are required to use a compatible matrix data format, which is based on compressed sparse row (CSR). The matrix elements and column index arrays must be stored in contiguous memory in row major order. An additional requirement is a `row_start` index array which gives the position of the starting element of each row. By contrast, the matrix format in HPCG uses arrays of row pointers, with a separate memory allocation for the elements and column indices for each row. There is also an array which gives the number of nonzero elements per row.

Additionally, the CUSPARSE triangular solver routine requires elements within each row to be sorted such that elements with column index smaller than the diagonal appear before the diagonal, and elements with column index larger than the diagonal appear after the diagonal. The default matrix format in HPCG violates this assumption in rows that are on the boundary of the local domain. In these rows the halo elements (those received from a neighbor processor) have column indices larger than the number of rows, but may appear before the diagonal because the order is inherited from the natural ordering of the global matrix.

Next, we describe the implementation of the SYMGS smoother, using the CUSPARSE and CUBLAS library routines. The main computational kernel, the sparse triangular solve, requires information about the structure of the matrix in order to expose parallelism. Thus, a pre-processing step is required to analyze the matrix structure using `cusparseDcsrsv_analysis` before any calls to `cusparseDcsrsv_solve` can be made. The analysis function essentially builds a task dependency graph that is later used when the solver is called. We must perform the analysis for both the upper and lower triangular portions of the matrix. This analysis phase maps nicely to the optimization phase of the benchmark, and the time spent here is recorded in the optimization timing.

The following lists the library calls that are made to perform SYMGS:

```
r <-- rhs      cublasDcopy
r <-- r - A*x  cusparseDcsrsv (SPMV)
y <-- L*y=r    cusparseDcsrsv_solve
y <-- y*D      cublasDaxpy
dx <-- U*dx=y  cusparseDcsrsv_solve
x <-- x+dx     cublasDaxpy
```

This sequence is not as efficient as the reference algorithm which combines the SPMV, vector updates, and triangular solves, reducing the number of steps and the number of times data must be accessed from memory. The WAXPBY is another example of a function which loses efficiency when implemented with library calls, in general it requires three calls: `cublasDcopy`, `cublasDscale`, and `cublasDaxpy`. Other routines are more straightforward using the libraries, Dot-Product is simply a call to `cublasDdot`, SPMV is a single call to `cusparseDcsrsv`.

The only CUDA kernels we wrote for this version, are for the routines which have irregular access patterns to gather or scatter values based on an index array. This occurs when gathering elements from the local domain that must be sent to neighbor processors, and also when performing restriction and prolongation operators (the coarse grid elements each read or write to a fine grid element given by the `f2c` index array).

4.2 Reordering with Graph Coloring

The matrix can be re-ordered based on a multi-coloring where every row is assigned a color that is not shared with any rows to which it has a connection.

Parallel algorithms have been developed to solve this problem [19,20]. The basic idea is to assign a random value to each row, and then designate a color to rows whose values are local maxima when comparing their random values with connected uncolored rows. The process is repeated, adding a new color in each step. Although this can be done completely in parallel, several iterations are required before all rows are assigned a color, and the number of colors is typically sub optimal (larger than the minimum number of colors which would be computed using a serial greedy algorithm).

We adopt several improvements proposed by Cohen et al. [21]. Namely, we replace the random number generation with an on-the-fly hash of the row index, and each row redundantly computes the hash of all neighbors. This trades off additional computation in order to avoid storing the hash values and reduces memory bandwidth requirements. We also compute two independent sets of colors in each step, one for local maxima, and another for local minima. The following code illustrates the basic coloring algorithm where `minmax_hash_step` assigns two colors in each iteration, where `A_col` is the matrix column index array, and `colors` is a vector of integers representing the color of each row:

```
while( colored < rows ){
minmax_hash_step<<<>>(A_col, colors...);
colored += thrust::count(colors, ...);
}
```

We improve the coloring quality in cases where the number of colors is too large, by performing a re-coloring. We loop over each original color, from greatest to smallest, and every row of that color attempts to reassign itself a lower color not shared with any neighbors. Since all rows of the same color are independent, we can safely update their colors in parallel. The process could be repeated to further reduce the color count, but the benefits are reduced with each pass. The following code snippet shows a single re-coloring pass:

```
if( max_color > target ){
for( color=max_color; color>0; color-- )
recolor_step<<<>>(A_col, colors...);
}
```

After the coloring is completed, we use the color information to create a permutation vector, which is used to reorder the rows in the matrix according to their colors. The permutation vector is initialized with the natural order, and then sorted by key, using colors as the key. The following code snippet shows the creation of the perm vector using the THRUST `sort_by_key` routine:

```
thrust::sort_by_key(colors, colors+rows, perm);
```

4.3 Custom kernels CSR version

Next, we replace the CUSPARSE calls with our own routines. This allows us to adopt a more flexible matrix format which simplifies the reordering of the

matrix, and removes the need for sorting of the row elements with respect to the diagonal. Using the reordered matrix, we can perform the SYMGS sweeps using the same algorithm as the reference. The following code shows the SYMGS kernel:

```

__global__ void smooth(double* A_vals, ...
{
int row_index = threadIdx.x ...
if( row_index < last_row ){
double sum = rhs[row_index];
for( i=start_index; i<end_index; i+=stride ){
if(A_col[i] != -1 )
if(A_col[i] != row_index ){
sum += -A_vals[i] * x[A_col[i]];
}else{
diag = A_vals[i];
}
}
}
x[row_index] = sum/diag;
}
}

```

The smoother is applied to one color at a time for both the forward and backward sweeps. The following is the CPU code which calls the smoother kernels:

```

for( color=0; color<num_colors; color++ )
smooth<<<>>>(A_vals, A_col, rhs, x,...);

for( color=num_colors; color>=0; color-- )
smooth<<<>>>(A_vals, A_col, rhs, x,...);

```

4.4 Optimized version

From our experience in the CUDA porting of the Himeno benchmark on clusters with GPUs [17], optimizing memory bandwidth utilization is a key design element to achieve good performance on codes with low compute intensity (the ratio between floating point operations and memory accesses). In this case most of the data access is to the matrix, so we are able to improve the performance by storing the matrix in the ELLPACK format. This allows matrix elements to be accessed in a coalesced access pattern.

In addition to the optimized matrix storage format, we also performed several other optimizations, listed here:

- A. SYMGS: removing redundant communications and work
- B. SPMV: overlapping communications with computations
- C. CG: overlapping MPI_Allreduce with vector update
- D. SYMGS + SPMV: using LDG load instructions

SYMGS: removing redundant work The SYMGS routine is called for the pre-smoother and post-smoother of the multi-grid V-cycle. The initial value of the solution at each level is set to zero, which allows us to avoid some of the communications and computations that occur during the first application of the smoother at each level. The SYMGS smoother routine begins by calling `exchange_halo`, which communicates boundary elements of the local matrix with neighbor processors. Since we know the values are all zeros, we can skip this communication step. We may also avoid processing the zero elements of the initial solution vector by restricting the forward sweep to matrix elements below the diagonal. We use a special smoother kernel for this case that checks if the column index is lower than the row index by adding `if(A_col[i] < row_index)` in the kernel code. We also note that in the CUSPARSE implementation, the zero values could allow one to skip the SPMV used to construct the residual (since the right hand side will be equal to the residual in this case), and the vector update in the last step of SYMGS where the computed delta is added to the initial solution.

SPMV: overlapping communications with computations The SPMV routine also begins with a call to `exchange_halo`, which updates the portion of the solution that is owned by other processors. However, these points, referred to as the halo points, are only required for the computation of the rows that are along the boundary of the local domain. Thus, we can safely split the computation into two phases, first computing the points which do not require the boundary, called interior, and next computing those which do require the boundary, called exterior. In this way we can overlap the computation of the interior with the halo communications.

The communications involve copying of the boundary data from GPU to CPU, MPI send/recv with neighbor processes, and copy results back to the GPU. We overlap the CPU to GPU communication by using cuda streams, with the copies placed into a different stream than the computation kernels.

While it is possible to use the same matrix structure for both interior and exterior computations, the efficiency of the exterior is greatly reduced because there is little locality in the access of the boundary matrix entries. It is more efficient to use a separate data structure, which only contains the boundary rows of the matrix, to process the boundary elements. For this purpose we also construct a boundary row index array which gives the row index of all boundary rows.

The fastest way to compute the boundary index array is to start with a copy of the already existing `elementsToSend` index array, and simply apply `thrust::sort` and `thrust::unique` functions. Then the boundary index array can be used to copy rows from the original matrix into the much smaller boundary matrix. The overhead of these operations are included in the optimization phase timing, and represent only a small fraction of the total optimization time.

CG: Overlapping MPI_Allreduce with vector update In the CG algorithm, the solution vector x is never required as an input to any of the steps. So we may delay the vector update of the solution in order to overlap the update time with the next dot product `MPI_Allreduce()` time. This scheme allows one of the three dot products in the CG solver to overlap with computations.

LDG: read-only cache load instructions The Kepler class of GPUs have a read-only data cache, which is well suited for reading data with spatial locality or with irregular access patterns. In previous GPU generations, a programmer would have to bind memory to texture objects and load data with special texture instructions to achieve this. However, on Kepler, the GPU cores can access any data with a new compute instruction called LDG. These special load instructions may be generated by the compiler provided it can detect that the data is read-only and no aliasing is occurring. This can be achieved by marking all the pointer arguments to a kernel with `__restrict` keywords. Unfortunately, this method will not always produce the best use of the memory system. For example, in the SYMGS kernels, the matrix is read-only, but the X vector is both read and written. Thus, when using `__restrict`, the compiler will use LDG for the matrix data, and regular loads for the solution vector. Ironically, the Matrix data is better suited to regular loads, since there is no data reuse and the access pattern is coalesced, while the irregular access of the solution vector is better suited to the read-only cache. By omitting the `__restrict` keywords, and using the `__ldg()` intrinsic for the load of X , we are able to increase performance by an additional 4%.

5 Results

In this section, we present results for single node and for clusters. The single node experiments allow us to have a better understanding of the relationship between HPCG performance and processor floating point and memory bandwidth capabilities.

5.1 Comparison of different versions

Before looking at the single node results on different hardware, we compare the effects of the optimizations applied in the four implementations discussed in the previous section. Figure 1 shows the timing of the four versions of the code on a K20X GPU with ECC enabled. As we can see the matrix reordering has the most relevant effect, since it exposes more parallelism in the SYMGS routine.

Figure 2 shows a detailed timing breakdown for the optimized version on a single GPU. The SYMGS kernel on all the multigrid levels takes up 55% of the time, followed by the SPMV kernel with 26%.

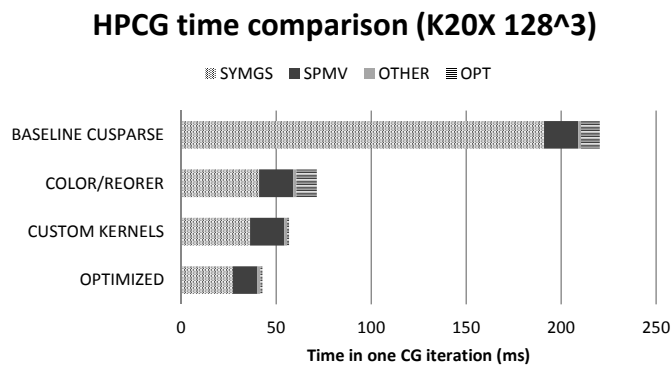


Fig. 1. Time comparison between the initial CUSPARSE implementation and the other custom versions, with 128^3 domain, on K20X with ECC enabled

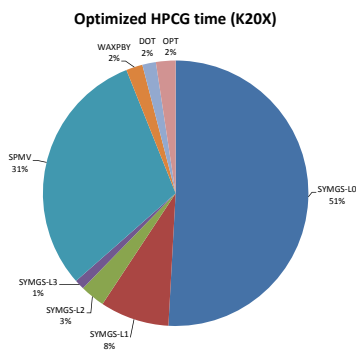


Fig. 2. Time distribution for Optimized version with 128^3 domain, on K20X with ECC enabled

5.2 Single node results

Next, we compare the performance on different classes of Kepler GPUs ranging from the smallest CUDA-capable GK20A found in the Tegra K1 mobile processor, to the highest performing Tesla K40. The Tesla K20X and K40 are both Kepler based, but they differ in the number of Symmetric Multiprocessors (SM), the amount of memory (6GB for the K20X vs 12GB for the K40) and the core/memory clocks (detailed specs are in Table 1). The K40 can also boost the core clock to 875MHz, which also results in a better memory throughput.

Table 1. Specs of the GPUs and CPU used in the benchmark, with clocks in MHz.

Processor	CC	# SM	# Cores SP/DP	Core clock	GFLOPS DP/SP	Memory clock	Memory bus width	Memory Bandwidth	DP Flops per Byte
Tegra K1	3.2	1	192/8	852	13.6/327	924	64 bit	14.7 GB/s	0.93
Tesla K10	3.0	8	1536/64	745	95/2289	2500	256 bit	160 GB/s	0.59
Tesla K20X	3.5	14	2688/896	732	1312/3935	2600	384 bit	250 GB/s	5.28
Tesla K40	3.5	15	2880/960	745	1430/4291	3000	384 bit	288 GB/s	4.96
Xeon E5-2697	N/A	N/A	12	2700	259/518	1866	256 bit	60 GB/s	4.32

The Compute intensity, or flops/bytes ratio, is a useful metric for determining whether an application will be bandwidth or floating point limited. In this case, the workload is dominated by Matrix-Vector operations, where the compute intensity may be estimated as $2 * \text{nonzerosperrowFlops} / (16 + 12 * \text{nonzerosperrow} * \text{Bytes}) = 54/340 = 0.158$. This is much lower than the *flop/byte* ratios for the hardware given in table 1. Therefore, we can expect performance to be limited much more by memory bandwidth than floating point throughput capabilities.

Figure 3 shows the scaling of HPCG performance across the GPUs used in our study. Figure 4 demonstrates the efficiency of our implementation by comparing the performance of the SYMGS and SPMV routines with the STREAM benchmark [16]. We also include the same metrics for an optimized CPU implementation developed by Park and Smelyanskiy [18]. As we can see in figure 5, there is an excellent correlation between the HPCG score and the STREAM benchmark result.

5.3 Multi node results

The cluster runs were performed on the Titan system at the Oak Ridge National Laboratory (ORNL) and on the Piz Daint system at the Swiss National Supercomputing Centre (CSCS). They are both Cray systems, but while Titan is a Cray XK7 based on AMD Opteron and a Gemini network, Piz Daint is a new Cray XC30 with Intel Xeon and the new Aries network. Titan has 18,688 nodes,

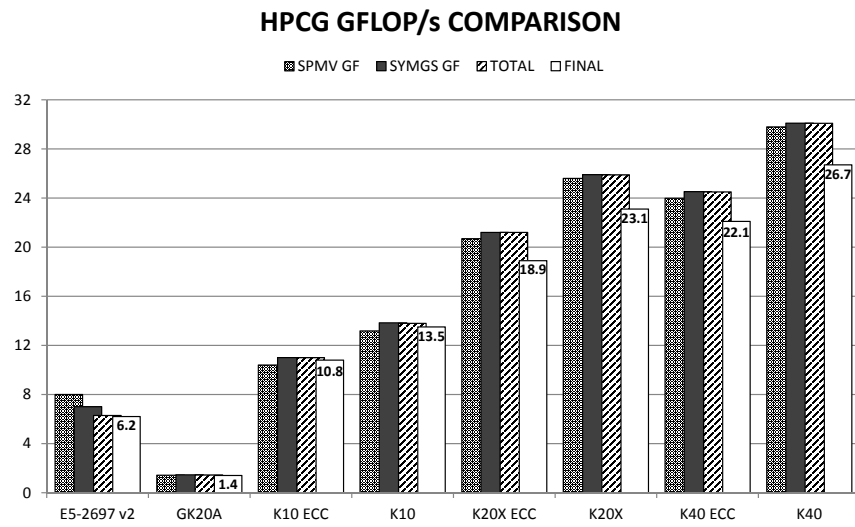


Fig. 3. Comparison of HPCG Flop Rate on single GPUs and Xeon E5-2697-v2 12-core CPU

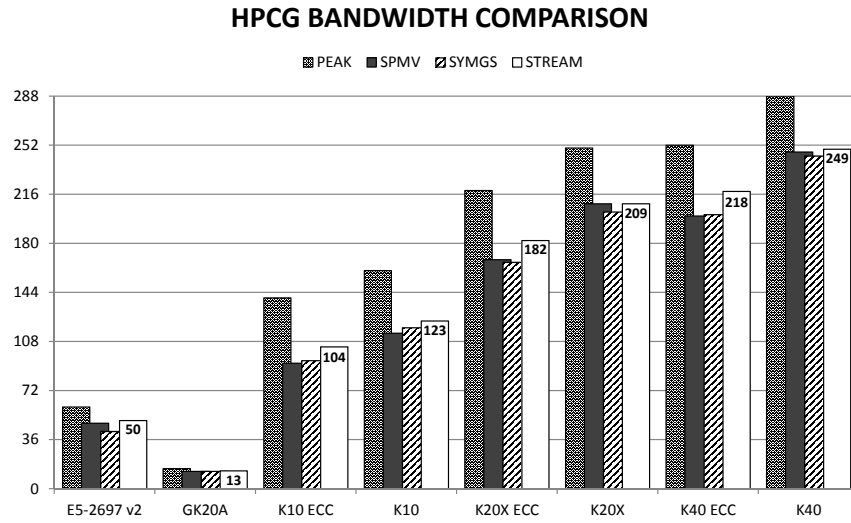


Fig. 4. Comparison of HPCG Flop Rate and Bandwidth on single GPUs and Xeon E5-2697-v2 12-core CPU

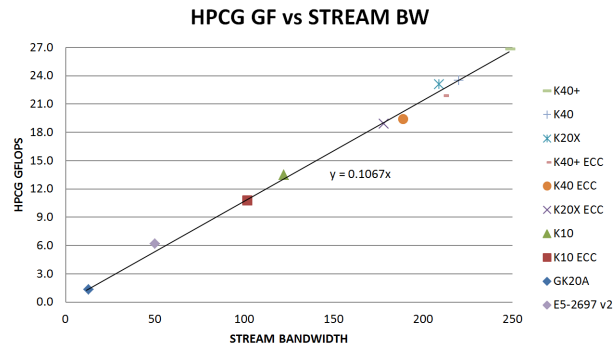


Fig. 5. Correlation between STREAM and HPCG benchmark results on single GPUs and E5-2697-v2 12-core CPU

each with a 16-core AMD Optreron processor, 32 GB of system memory and a 6GB NVIDIA K20X GPU. The network uses the Gemini routing and communications ASICs and a 3D torus network topology. Piz Daint has 5,272 nodes, each with an Intel Xeon E5 processor, 32 GB of system memory and a 6GB NVIDIA K20X GPU. The network uses the new Aries routing and communications ASICs and a dragonfly network topology.

Table 2 shows the performance of the optimized version on a wide range of nodes, up to the full size machine on Titan and Piz-Daint. The raw number is the total performance number, before the reduction due to the increased iteration count caused by the multi-coloring.

Table 2. HPCG Supercomputer Results in GFlops: local grid size $256 \times 256 \times 128$

Nodes	Titan Raw	Titan Final	Titan Eff.	Piz-Daint Raw	Piz-Daint Final	Piz-Daint Eff.
1	21.23	20.77	100.0	21.25	20.79	100.0
8	168.3	161.4	99.1	168.8	161.9	99.3
64	1321	1221	97.2	1341	1239	98.6
512	10414	9448	95.8	10719	9904	98.5
2048				42777	38806	98.3
3200	62239	56473	91.6			
5265				109089	98972	97.5
8192	158779	144071	91.3			
18648	355189	322299	89.7			

At full scale, Piz-Daint is reaching 0.098 PF, compared to the 6.2 PF during HPL. Since we are running very close to peak bandwidth and the code has no problem scaling up to the full machine, there is not much space left for large improvements. Even with no coloring overhead, the full machine will deliver only 0.1PF. Same conclusion holds for Titan, the achieved HPCG performance of 0.322PF is far away from the sustained 17.59PF during HPL.

In Figure 6, we analyze the communication time on the Titan runs. The dot products require `all_reduce` communications, that scale as the logarithm of the node count. The other communications are instead with neighbors and remain constant with the number of nodes. The ones in the SPMV phase are completely overlapped with computations, in the current version the ones in the multigrid phase are not but the overlapping will be implemented in an upcoming version.

5.4 Comparisons

The first official HPCG ranking was published at the International Supercomputing Conference in June 2014 and included 15 supercomputers. All the GPU supercomputers on the list ran the optimized version described in this paper. Table 3 summarizes the results of several of the top systems: Thiane-2 is based

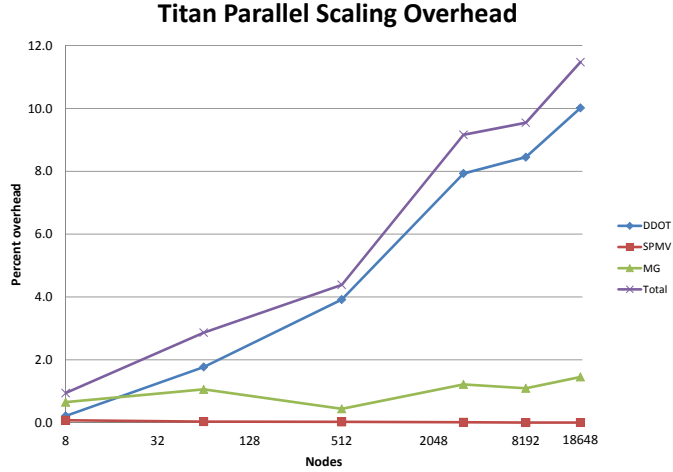


Fig. 6. Scaling overhead on Titan.

on Xeon Phi processors (currently number one in the Top500 list), K is a CPU-only system based on Sparc64 Processors. Instead of looking at the peak flops of these machines, we evaluate the efficiency based on the ratio of the HPCG result to the memory bandwidth of the processors.

Table 3. HPCG Supercomputer Results Comparison

HPCG Rank	System	HPCG GFLOPS	Iterations	#Procs	Processor Type	HPCG Per Proc	Bandwidth Per Proc	Efficiency FLOP/BYTE
1	Tianhe-2	580,109	57	46,080	Xeon-Phi-31S1P	12.59 GF	320 GB/s	0.039
2	K	426,972	51	82,944	Sparc64-viiiifx	5.15 GF	64 GB/s	0.080
3	Titan	322,321	55	18,648	Tesla-K20X+ECC	17.28 GF	250 GB/s	0.069
5	Piz-Daint	98,979	55	5,208	Tesla-K20X+ECC	19.01 GF	250 GB/s	0.076
8	HPC2	49,145	54	2,610	Tesla-K20X+ECC	18.83 GF	250 GB/s	0.075
	HPC2	60,642	54	2,600	Tesla-K20X	23.32 GF	250 GB/s	0.093

The efficiency of the GPU implementation is comparable to the one of K and the performance per processor is noticeably higher.

6 Conclusion and future plans

The results in the paper show that GPU accelerated clusters perform very well in the new HPCG benchmark. Our results are the fastest per processor ever re-

ported. GPUs, with their excellent floating point performance and high memory bandwidth, are very well-suited to tackle workloads dominated by floating point, like HPL, as well as those dominated by memory bandwidth, like HPCG.

The current implementation is all on the GPUs, but since the CPUs could give a significant contribution, we are investigating a hybrid scheme where both CPU and GPU are used together.

7 Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. We wish to thank Buddy Bland, Jack Wells and Don Maxwell of Oak Ridge National Laboratory for their support. This work was also supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID g33. We also want to acknowledge the support from Gilles Fourestey and Thomas Schulthess at CSCS. We wish to thank Lung Scheng Chien and Jonathan Cohen at NVIDIA for relevant discussions.

References

1. Jack Dongarra and Michael A. Heroux, "Toward a New Metric for Ranking High Performance Computing Systems", Sandia Report SAND2013-4744 (2013).
2. Jack Dongarra, Piotr Luszczek, "Introduction to the HPC Challenge Benchmark Suite", ICL Technical Report, ICL-UT-05-01, (Also appears as CS Dept. Tech Report UT-CS-05-544), (2005).
3. Michael A. Heroux, Jack Dongarra and Piotr Luszczek, "HPCG Technical Specification", Sandia Report SAND2013-8752 (2013).
4. Graph 500, <http://www.graph500.org>
5. Green 500, <http://www.green500.org>
6. CUDA Toolkit, <http://developer.nvidia.com/cuda-toolkit>
7. CUDA Fortran, <http://www.pgroup.com/resources/cudafortran.htm>
8. CUBLAS Library, <http://docs.nvidia.com/cuda/cublas>
9. CUSPARSE Library, <http://docs.nvidia.com/cuda/cuspars>
10. THRUST Library, <http://docs.nvidia.com/cuda/thrust>
11. <http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/>
12. Richard F. Barrett, Michael A. Heroux, Paul T. Lin, Courtenay T. Vaughan, and Alan B. Williams. "Poster: mini-applications: vehicles for co-design.", in Proceedings of the 2011 Companion on High Performance Computing Networking, Storage and Analysis Companion (SC '11 Companion). ACM, New York, NY, USA, 1-2 (2011)
13. Gene H. Golub, Charles F. Van Loan, "Matrix Computations", John Hopkins University Press, Third Edition, (1996)
14. William L. Briggs, Van Emden Henson, Steve F. McCormick, "A Multigrid Tutorial", SIAM, (2000)

15. Green 500: Energy Efficient HPC System Workloads Power Measurement Methodology,(2013)
16. John D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.
17. Everett H. Phillips, Massimiliano Fatica, "Implementing the Himeno benchmark with CUDA on GPU clusters," IPDPS, pp.1-10, 2010 IEEE International Symposium on Parallel & Distributed Processing, (2010)
18. Jongsoo Park and Mikhail Smelyanskiy, "Optimizing Gauss-Seidel Smoother in HPCG", ASCR HPCG Workshop, Bethesda MD, March 25 2014
19. Michael Luby, "A simple parallel algorithm for the maximal independent set problem," SIAM Journal on Computing, (1986)
20. Mark T. Jones and Paul E. Plassmann, "A parallel Graph Coloring Heuristic," SIAM J. SCI. COMPUT., Vol 14, pp.654-669, (1992)
21. Jonathan Cohen and Patrice Castonguay, "Efficient Graph Matching and Coloring on the GPU," GPU Technology Conference, San Jose CA, May 14-17 2012, <http://on-demand.gputechconf.com/gtc/2012/presentations/S0332-Efficient-Graph-Matching-and-Coloring-on-GPUs.pdf>