Preliminary Implementation of PETSc using GPUs

Victor Minden and Barry Smith and Matthew G. Knepley

To appear in the *Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering*, Springer.

Abstract PETSc is a scalable solver library for the solution of algebraic equations arising from the discretization of partial differential equations and related problems. PETSc is organized as a class library with classes for vectors, sparse and dense matrices, Krylov methods, preconditioners, nonlinear solvers, and differential equation integrators. A new subclass of the vector class has been introduced that performs its operation on NVIDIA GPU processors. In addition, a new sparse matrix subclass that performs matrix-vector products on the GPU was introduced. The Krylov methods, nonlinear solvers, and integrators in PETSc run unchanged in parallel using these new subclasses. These can be used transparently from existing PETSc application codes in C, C++, Fortran, or Python. The implementation is done with the Thrust and Cusp C++ packages from NVIDIA.

1 Introduction

PETSc [2, 3] is a scalable solver library for the solution of algebraic equations arising from the discretization of partial differential equations and related problems. The goal of the project reported here was to allow PETSc solvers to utilize GPUs with as

Victor Minden
School of Engineering, Tufts University, Medford, MA 02155, USA, e-mail: victor.minden@tufts.edu

Barry Smith
Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4844, USA, e-mail: bsmith@mcs.anl.gov

Matthew G. Knepley
Computation Institute, University of Chicago, 5735 South Ellis Avenue, Chicago, IL 60637, USA, e-mail: knepley@ci.uchicago.edu

little change as possible to the basic design of PETSc. Specifically, a new subclass of the vector class has been introduced that performs its operation on NVIDIA GPU processors. In addition, a new sparse matrix subclass that performs matrix-vector products on the GPU was introduced. The Krylov methods, nonlinear solvers, and integrators in PETSc run unchanged in parallel using these new subclasses. These can be used transparently from existing PETSc application codes in C, C++, Fortran, or Python. The implementation uses the Thrust[1] [8] and Cusp[2] [7] C++ packages from NVIDIA.

Numerous groups have experimented with sparse matrix iterative solvers on GPUs, for example, [10, 13, 11, 12, 6, 5, 4, 9]. The Trilinos package [15, 14] already has support for NVIDIA GPUs through its Kokkos package, also using Thrust.

2 Sequential Implementation

PETSc consists of a small number of abstract classes: `Vec` and `Mat` devoted to data; and `PC`, `KSP`, `SNES` and `TS` devoted to algorithms. By abstract, we mean that each class is defined by a set of operations on the class object while any data associated with the class object is encapsulated within the object and not directly assessable outside the class. The `Vec` class is used for representing field values, discrete solutions to PDEs, right-hand sides of linear systems and so forth. PETSc provides a default implementation of the `Vec` class that stores the vector entries in a simple, one-dimensional C/Fortran array and uses BLAS 1 operations when possible for the methods, and MPI to perform reduction operations across processes needed by inner products and norms. The `Mat` and `PC` classes do not directly access the underlying array in the vector, instead they call `VecGetArray(Vec, double**)` or `VecGetArrayRead(Vec, const double**)` to access the local (on process) values of the vector. In general, the `KSP`, `SNES`, and `TS` classes never access `Vec` or `Mat` data; rather, they call methods on the `Vec` and `Mat` objects in order to perform operations on the data. The `PC` class is somewhat special in that many preconditioners are data structure specific. Thus, many `PC` implementations directly access matrix data structures, which in C++ would correspond to a *friend* class.

For this initial implementation of PETSc with GPUs, we have used the following model. PETSc runs in parallel with MPI for communication; and each PETSc process has access to a single GPU, which has its own memory, generally several gigabytes. We introduce a new `Vec` implementation, which we will call a CUDA `Vec`. Each object of this new `Vec` class must potentially manage two copies of the vector data: one in the CPU memory and one in the GPU memory. (We note that on some integrated graphics systems the GPU actually uses the usual CPU memory as its memory; we ignore this for our preliminary work). In order to manage mem-

[1] Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). Thrust provides a flexible high-level interface for GPU programming that greatly enhances developer productivity.

[2] Cusp is a library for sparse linear algebra and graph computations on CUDA that uses Thrust.

ory coherence, each CUDA `Vec` has a flag that indicates whether space in the GPU memory has been allocated and whether the memory in the GPU, in the CPU, or in both contains the most recent values. The possible flag values are given in Table 1. The flag is the only change to the base `Vec` class in PETSc. This was added to the base class rather than the derived GPU-specific `Vec` class because we want to be able to check whether the memory copy is needed, without requiring the extra clock cycles of accessing the derived class for each check.

Two utility routines are provided, `VecCUDACopyToGPU()` and `VecCUDACopyFromGPU()`, that copy vector data down to the GPU memory or up to the CPU memory based on the flag. For example, the method `VecGetArray()` for the CUDA `Vec` copies the values up from the GPU if the flag is `PETSC_CUDA_GPU` and sets the flag to `PETSC_CUDA_CPU` since the user is free to change the vector values. The `VecGetArrayRead()` would still perform the copy, but sets the flag to `PETSC_CUDA_BOTH` since the user cannot change the values in the array. For all vector operations performed on the GPU, such as `VecAXPY()`, data will be copied down from the CPU if the flag is `PETSC_CUDA_CPU`, and both allocated and copied if it is `PETSC_CUDA_UNALLOCATED`.

Table 1 Flags used to indicate the memory state of a PETSc CUDA `Vec` object.

`PETSC_CUDA_UNALLOCATED`	Memory not allocated on the GPU
`PETSC_CUDA_GPU`	Values on GPU are current (assume CPU allocated)
`PETSC_CUDA_CPU`	Values on CPU are current(assume GPU allocated)
`PETSC_CUDA_BOTH`	Values on both devices are current

Implementations of the basic vector operations is straightforward. For example, the `VecAXPY()` code is given by the following.

```
ierr = VecCUDACopyToGPU(xin);CHKERRQ(ierr);
ierr = VecCUDACopyToGPU(yin);CHKERRQ(ierr);
try {
  cusp::blas::axpy(*((Vec_CUDA*)xin->spptr)->GPUarray,
            *((Vec_CUDA*)yin->spptr)->GPUarray,alpha);
  yin->valid_GPU_array = PETSC_CUDA_GPU;
  ierr = WaitForGPU();CHKERRCUDA(ierr);
} catch(char *ex) {
  SETERRQ1(PETSC_COMM_SELF, PETSC_ERR_LIB,
                                  "CUDA error: \%s", ex);
}
```

For more sophisticated `Vec` methods, such as `VecMAXPY()`, $y = y + \sum_i \alpha_i x_i$, and `VecMDot()`, $\alpha_i = y^T x_i$ the code is more complicated. We unroll loops in order to reuse entries in the y vector. For example, we unroll the outer loop for four vectors. The multiple inner product code, written using Thrust calls, is given below.

```
for (j=j_rem; j<nv; j+=4) {
  yy0  =  yin[0]; yy1  =  yin[1];
```

```
  yy2  =  yin[2]; yy3  =  yin[3];
  ierr =  VecCUDACopyToGPU(yy0);CHKERRQ(ierr);
  ierr =  VecCUDACopyToGPU(yy1);CHKERRQ(ierr);
  ierr =  VecCUDACopyToGPU(yy2);CHKERRQ(ierr);
  ierr =  VecCUDACopyToGPU(yy3);CHKERRQ(ierr);
  try {
    result4 = thrust::transform_reduce(
                thrust::make_zip_iterator(
                  thrust::make_tuple(
          ((Vec_CUDA *)xin->spptr)->GPUarray->begin(),
          ((Vec_CUDA *)yy0->spptr)->GPUarray->begin(),
          ((Vec_CUDA *)yy1->spptr)->GPUarray->begin(),
          ((Vec_CUDA *)yy2->spptr)->GPUarray->begin(),
        ((Vec_CUDA *)yy3->spptr)->GPUarray->begin())),
                thrust::make_zip_iterator(
                  thrust::make_tuple(
            ((Vec_CUDA *)xin->spptr)->GPUarray->end(),
            ((Vec_CUDA *)yy0->spptr)->GPUarray->end(),
            ((Vec_CUDA *)yy1->spptr)->GPUarray->end(),
            ((Vec_CUDA *)yy2->spptr)->GPUarray->end(),
          ((Vec_CUDA *)yy3->spptr)->GPUarray->end())),
        cudamult4<thrust::tuple<PetscScalar,PetscScalar,
                  PetscScalar,PetscScalar,PetscScalar>,
                thrust::tuple<PetscScalar,PetscScalar,
                          PetscScalar,PetscScalar> >(),
              thrust::make_tuple(zero,zero,zero,zero),
        cudaadd4<thrust::tuple<PetscScalar,PetscScalar,
                        PetscScalar,PetscScalar> >());
    z[0] = thrust::get<0>(result4);
    z[1] = thrust::get<1>(result4);
    z[2] = thrust::get<2>(result4);
    z[3] = thrust::get<3>(result4);
  } catch(char* ex) {
    SETERRQ1(PETSC_COMM_SELF,PETSC_ERR_LIB,
                                  "CUDA error: %s", ex);
  }
  z    += 4;
  yin  += 4;
}
```

The CUDA kernel of this operation is given by the following.

```
struct VecCUDAMAXPY4 {
  template <typename Tuple>
  __host__ __device__
  void operator()(Tuple t) {
```

```
    /* y += a1*x1 +a2*x2 + 13*x3 +a4*x4 */
    thrust::get<0>(t) +=
                thrust::get<1>(t)*thrust::get<2>(t) +
                thrust::get<3>(t)*thrust::get<4>(t) +
                thrust::get<5>(t)*thrust::get<6>(t) +
                thrust::get<7>(t)*thrust::get<8>(t);
  }
};
```

Note that often the `VecCUDACopyToGPU()` calls simply verify that the vector's flag is `PETSC_CUDA_GPU` and do not need to copy the data down to the GPU. This would be the case during a Krylov solve, where only the results of norm and inner product calls are shipped back to the CPU.

The NVIDIA Cusp software provides a data structure and matrix-vector product operation for sparse matrices in Compressed Sparse Row (CSR) and several other formats. Our initial CUDA `Mat` implementation simply uses the code provided by Cusp. The matrix-vector product code in PETSc then is given by the following.

```
try {
  cusp::multiply(*cudastruct->mat,
    *((Vec_CUDA *)xx->spptr)->GPUarray,
    *((Vec_CUDA *)yy->spptr)->GPUarray);
} catch(char* ex) {
  SETERRQ1(PETSC_COMM_SELF,PETSC_ERR_LIB,
                                  "CUDA error: \%s", ex);
}
```

Our primary design goal in this initial implementation was to enable the vector and matrix data to reside on the GPU throughout an entire Krylov solve, requiring no slow copying of data between the two memories.This is now supported for all but one of the Krylov methods in PETSc, including GMRES, Bi-CG-stab, and CG, and several preconditioners including Jacobi and Cusp Smoothed-Aggregation Algebraic Multigrid. The excluded Krylov method, a varient of Bi-CG-stab that requires only one global synchronization per iteration, actually accesses the vectors directly rather than through the `Vec` class methods (since it requires many operations not supported by the class methods) and hence would need to be rewritten directly in CUDA.

3 Parallel Implementation

In the parallel case, there must be communication of vector entries between processes during the computation of the sparse matrix-vector product. In PETSc, for the built-in parallel sparse matrix formats the parallel matrix is stored in two parts: the "on-diagonal" portion of the matrix, A_d, with all the columns associated with

the rows of the vector "owned" by the given process, x_d, and the "off-diagonal" portion, A_o, associated with all the other columns (whose vector values are "owned" by other processes, x_o). The sparse matrix-vector product is computed in two steps: $y_d = A_d x_d$, then $y_d = y_d + A_o x_o$. Of course, since A_o has few columns with nonzero entries, most of x_o do not need to be communicated to the given process.

PETSc manages all communication of vector entries between processes via the `VecScatter` object. For the sparse matrix-vector product vector communication, this object is created with a list of global indices indicating from where in the source vector entries are to come from and another list of indices indicating where they are to be stored into a local work vector. The vector communication itself is done in two stages: first a `VecScatterBegin()` copies the vector entries that need to be sent into message buffers, and posts nonblocking MPI receives and sends; then `VecScatterEnd()` waits on the receives and copies the results from the message buffers into the local work vector. If we let $\hat{A}_o$ denote the nonzero columns of A_o and let $\hat{x}_o$ denote the corresponding rows of x_o, then the parallel matrix-vector code is then

```
VecScatterBegin(a->Mvctx,xd,hatxo,INSERT_VALUES,...);
MatMult(Ad,xd,yd);
VecScatterEnd(a->Mvctx,xd,hatxo,INSERT_VALUES,...);
MatMultAdd(hatAo,hatxo,yd,yd);
```

This same code can be used automatically when the A_d and $\hat{A}_o$ matrices are CUDA matrices. The difference from the standard case is that the `VecScatterBegin()` triggers a `VecCUDACopyFromGPU()` of the x_d vector (so that its entries are available in the CPU memory to be packed into the message buffers) and the `MatMultAdd()` triggers a `VecCUDACopyToGPU()` (to move the values that have arrived from other processes down to the GPU memory). Initial profiling indicated that the needed `VecCUDACopyFromGPU()` was taking substantial time. But most entries of the x_d vector are not actually needed by the vector scatter routines, only those values that are destined for other processes that will generally be only a few percent of the values. Thus we have added the routine `VecCUDACopyFromGPUSome(Vec, cusp::array1d<PetscInt, cusp::host_memory> *iCPU, cusp::array1d<PetscInt, cusp::device_memory> *iGPU)` that copies only the needed values. There are two sets of identical indices, one that resides in the CPU memory and one that lives on the GPU memory, since it would be inefficient to copy the indices between the two memories on each invocation. The constructor for the `VecScatter` determines the required indices and sets them in the two memories. With the addition of this new code the required copy time decreased significantly in the parallel matrix-vector product. This change required a small amount of additional GPU specific code in the `VecScatter` constructor and `VecScatterBegin()`.

In order to monitor the movement of data between the two memories we provided two additional `PetscEvents`, one that tracks the counts and times of copies from the GPU and one for copies to the GPU. This information can be accessed with the usual PETSc `-log_summary` option. Because CUDA calls are, by default, asynchronous, meaning the function calls in the CPU return before the GPU completes

the operation, we provide an global flag that forces a wait after each CUDA call until the operation is complete. This is neccessary whenever one wants accurate times of the individual phases of the computation. Forcing synchronization appears to cost a few percent of the runtime, in production runs this option is not needed.

4 Conclusion and Future Work

We can now run parallel linear solves (with very simple preconditioners) that utilize the GPU for all vector and matrix operations. The only vector entries that need to be passed, during the linear solve, between GPU memory and CPU memory are those destined for other processes.

This is preliminary work. Important additional work is needed in several areas.

- Performance evaluation and optimization. We have verified correctness and basic performance of the new code that utilizes the GPUs, but we have not yet done comprehensive studies.
- GPU-based preconditioners. The NVIDIA group is actively developing several of these, and they are easily added as new preconditioners in PETSc by simply deriving new `PC` subclasses that utilize the NVIDIA code.
- GPU-based nonlinear function evaluations. We have a simple, one-dimensional finite difference problem on a structured grid `src/snes/examples/tutorials/ex47cu.cu` that uses the Thrust `zip_iterator` to apply a stencil operation. As with the parallel matrix-vector product the `VecScatter` class is used to manage the communication of ghost point values between processes. More work is needed so that copies of vectors between unghosted and ghosted representations require as little as possible memory copies between GPU and CPU. We note, various groups are in the process of developing or have already developed implementations of finite element function evaluations for GPUs [21, 18, 19, 20, 17, 1, 16]. These could be used within a PETSc code.
- GPU-based Jacobian evaluations. With GPU-based Jacobian evaluations the entire nonlinear solution process (and hence also ODE integration) could be performed on GPUs without requiring any vector or matrix copies between CPU and GPU memory besides those entries required to move data between processes. This is a difficult process because the sparse matrix data structure is nontrivial and hence the efficient application of the equivalent of `MatSetValues()` on the GPU is non-trivial.

We note that because of the object-oriented design of PETSc it is possible to introduce additional vector and matrix classes, distinctly different from those discussed in this paper, that also use the NVIDIA GPUs. In fact, we hope there will be additional implementations to determine those that produce the highest performance.

When considering sparse matrix iterative solvers on GPUs, one must bear in mind that these algorithms are almost always memory-bandwidth limited. That is,

the speed of the implementation does not depend strongly on the speed or number of the processor cores but rather on the speed of the memory. Since the best GPU systems have higher memory bandwidth than do conventional processors, one expects (and actually does see) higher floating-point rates with GPU systems; but since the memory bandwidths of GPU systems are only several times faster than those of conventional processors one will see at most only several times speedup when converting a sparse matrix iterative solver from CPUs to GPUs. Speedups of 100 or more are simply not possible.

Acknowledgements We thank Nathan Bell from NVIDIA and Lisandro Dalcin for their assistance with this project. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

1. R. Abedi, B. Petracovici, and R.B. Haber. A space-time discontinuous galerkin method for linearized elastodynamics with element-wise momentum balance. *Computer Methods in Applied Mechanics and Engineering*, 195(25-28):3247–3273, 2006.
2. Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2010.
3. Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.
4. M.M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on gpus. *IBM Research Report RC24704, IBM*, 2009.
5. N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. *NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004*, 2008.
6. N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. ACM, 2009.
7. N. Bell and M. Garland. The cusp library. `http://code.google.com/p/cusp-library/`, 2010.
8. N. Bell and J. Hoberock. The thust library. `http://code.google.com/p/thrust/`, 2010.
9. J. Bolz, I. Farmer, E. Grinspun, and P. Schr
”ooder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *ACM SIGGRAPH 2003 Papers*, page 924. ACM, 2003.
10. Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 917–924, New York, NY, USA, 2003. ACM.
11. L. Buatois, G. Caumon, and B. Lévy. Concurrent number cruncher: An efficient sparse linear solver on the gpu. *High Performance Computing and Communications*, pages 358–371, 2007.
12. A. Cevahir, A. Nukada, and S. Matsuoka. Fast conjugate gradients with multiple gpus. *Computational Science–ICCS 2009*, pages 893–903, 2009.
13. Z. Feng and P. Li. Multigrid on gpu: Tackling power grid analysis on parallel simt platforms. In *IEEE/ACM International Conference on Computer-Aided Design, 2008. ICCAD 2008*, pages 647–654, 2008.

14. Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.
15. M. Heroux et al. Trilinos Web page. `http://trilinos.sandia.gov/`.
16. G.R. Joldes, A. Wittek, and K. Miller. Real-time nonlinear finite element computations on gpu-application to neurosurgical simulation. *Computer Methods in Applied Mechanics and Engineering*, 2010.
17. R. Keunings. Parallel finite element algorithms applied to computational rheology. *Computers and Chemical Engineering*, 19(6):647–670, 1995.
18. D. Komatitsch and J.P. Vilotte. The spectral element method: an efficient tool to simulate the seismic response of 2d and 3d geological structures. *Bulletin of the Seismological Society of America*, 88(2):368–392, 1998.
19. R. Liu and DY Li. A finite element model study on wear resistance of pseudoelastic tini alloy. *Materials Science and Engineering A*, 277(1-2):169–175, 2000.
20. Z.A. Taylor, M. Cheng, and S. Ourselin. Real-time nonlinear finite element analysis for surgical simulation using graphics processing units. In *Proceedings of the 10th international conference on Medical image computing and computer-assisted intervention-Volume Part I*, pages 701–708. Springer-Verlag, 2007.
21. W. Wu and P.A. Heng. A hybrid condensed finite element model with gpu acceleration for interactive 3d soft tissue cutting. *Computer Animation and Virtual Worlds*, 15(3-4):219–227, 2004.