Using C++ AMP to Accelerate HPC Applications on Multiple Platforms

M. Graham Lopez[1(✉)], Christopher Bergstrom[2], Ying Wai Li[3], Wael Elwasif[1], and Oscar Hernandez[1]

[1] Computer Science and Mathematics Division,
Oak Ridge National Laboratory, Oak Ridge, TN, USA
{lopezmg,elwasifwr,oscar}@ornl.gov
[2] Pathscale Inc., Wilmington, DE, USA
cbergstrom@pathscale.com
[3] National Center for Computational Sciences,
Oak Ridge National Laboratory, Oak Ridge, TN, USA
yingwaili@ornl.gov

Abstract. Many high-end HPC systems support accelerators in their compute nodes to target a variety of workloads including high-performance computing simulations, big data / data analytics codes and visualization. To program both the CPU cores and attached accelerators, users now have multiple programming models available such as CUDA, OpenMP 4, OpenACC, C++14, etc., but some of these models fall short in their support for C++ on accelerators because they can have difficulty supporting advanced C++ features e.g. templating, class members, loops with iterators, lambdas, deep copy, etc. Usually, they either rely on unified memory, or the programming language is not aware of accelerators (e.g. C++14). In this paper, we explore a base-language solution called C++ Accelerated Massive Parallelism (AMP), which was developed by Microsoft and implemented by the PathScale ENZO compiler to program GPUs on a variety of HPC architectures including OpenPOWER and Intel Xeon. We report some prelminary in-progress results using C++ AMP to accelerate a matrix multiplication and quantum Monte Carlo application kernel, examining its expressiveness and performance using NVIDIA GPUs and the PathScale ENZO compiler. We hope that this preliminary report will provide a data point that will inform the

M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS 9945, pp. 563–576, 2016.
DOI: 10.1007/978-3-319-46079-6_38

functionality needed for future C++ standards to support accelerators with discrete memory spaces.

Keywords: HPC · C++ for Accelerators · C++ AMP · Accelerator programming

1 Introduction and Background

With various accelerator architectures emerging in the HPC space, there have been renewed concerns about available programming models and their ability to provide performance portability for applications across platforms. To address these issues, there are a few efforts in development that, like C++ Accelerated Massive Parallelism (AMP), make heavy use of C++ language features. Kokkos [1] and RAJA [2] are template-based library solutions that attempt to hide low-level implementation details from the application developer to achieve good performance on multiple architectures. Unlike Kokkos/RAJA, AMP attempts to extend the C++ language directly to deal with accelerator programming and non-contiguous memories. There is slow progress being made in the C++ language standard as well, with a conservative proposal [3] to include preliminary support for generic parallelism in the upcoming C++17 standard. However, the present proposal lacks sufficient expressiveness to deal with the multiple memory address spaces and complex compute and memory hierarchies found in today's accelerator platforms. OpenMP [4,5] 4 and OpenACC [6] provide directive-based approaches to program C++ on accelerators but fall short on supporting many advanced C++ features (deep copy, STLs, etc.) and alternative approaches need to be explored. Perhaps the most similar programming model to AMP is NVIDIA's CUDA. Both allow the programmer to specify arbitrary compute kernels in the (slightly extended) native language, as well as directly managing the data transfer between host and device or relying on implicit transfer features of the runtime. The main differences are that CUDA is a single-vendor defined model optimized for a specific architecture, while AMP is an open standard that can be implemented by any compiler to target various accelerators. AMP also hides the low-level details a little bit more by discarding the concepts of threads, grid blocks, etc. that are usually specified in the CUDA programming model.

While C++ AMP is not widely implemented at present, it does attempt to offer a complete and open language-based solution for programming GPUs with descrete memory spaces while allowing the application to continue to use advanced features of C++. In this paper, we first give a brief overview of the main syntactic and semantic features of C++ AMP which provide the context for a preliminary evaluation of C++ AMP using both a well-understood and compute-bound kernel, matrix-matrix multiplication (GEMM), as well as an "in-the-wild" kernel from a quantum Monte Carlo application called QMCPACK. We then show the code transformations involved when using this model for each kernel and present some preliminary performance impressions using an experimental

compiler-based implementation. Finally, we discuss our experiences with AMP in the context of new and upcoming C++ language standards as they apply to accelerator programming.

1.1 C++ AMP

C++ AMP is an open specification [7] based on a namespace that provides accelerator programming extensions to the C++ programming language. It is published by Microsoft Corporation, with input from PathScale Inc, NVIDIA Corporation, and Advanced Micro Devices Inc. (AMD). It supports offload of data-parallel algorithms to discrete accelerators like GPUs. The first implementation for C++ AMP was introduced in Microsoft Visual Studio 2012 [8], and experimental support has emerged in the PathScale [9] and LLVM/Clang [10,11] compilers as well.

When using C++ AMP, the programmer describes the computation to be performed on the accelerator by specifying the iteration space and the kernel to be applied over that space. The `parallel_for_each()` routine provides the mechanism for iterating through a domain. The computational kernel to execute on the accelerator is given by a lambda with the `restrict(amp)` keyword which indicates that the kernel contains the restricted subset of the C++ language that AMP is able to accelerate. The set of threads used for parallel execution on the accelerator is specified by creating `extent` or `tiled_extent` objects. Additionally, double-precision precise math and fast math libraries are provided for use on the accelerator, as well as several common numerical libraries that have been released for the C++ AMP programming model under the open-source Apache License, including random number generation (RNG), fast Fourier transform (FFT), basic linear algebra subroutines (BLAS), and linear algebra package (LAPACK).

The primary way to transfer data to the accelerator is by using the C++ AMP `array` and/or `array_view` objects. These objects need four pieces of information to describe the data: the rank (logical shape) of the data and the datatype of the elements are passed as type parameters, while the data itself and the physical shape of the array in memory are specified using constructor parameters. The `array` class causes a deep copy of the data when the object is constructed with a pointer to the original data set. The accelerator is able to access and modify its copy of the data, and after computation, the data must be copied out of the object to the source data structure. `array_view` objects can be constructed and accessed similarly, but instead of explicit data transfer happening upon construction, data is transferred implicitly to the accelerator on-demand at kernel execution time. After kernel execution, the data can be directly accessed on the host, and synchronization can be guaranteed using a provided method. For both `array` and `array_view` objects, shapes must be rectangular (in N dimensions), and can either be specified manually for each dimension or by using the C++ AMP `extent` class.

2 Preliminary Results

We have prototyped the use of C++ AMP for both a benchmark GEMM and a QMCPACK application kernel using the Pathscale ENZO 6.0.9 compiler. This work illustrates the use of the basic C++ AMP building blocks to parallelize the execution of nested loops used in both GEMM and QMCPACK. For a preliminary evaluation, we used two HPC platforms that are of significant relevance to the INCITE [12] and CORAL [13] programs of which QMCPACK is a part: one based on a representative node of Titan [14] containing a 16-core AMD Opteron 6274 CPU attached to an NVIDIA Tesla K20X GPU via PCIe v2, and the second a Summit [15] test node containing a Power8E CPU @ 2.61GHz processor with a NVIDIA Tesla K40m connected via PCIe v3.

2.1 Benchmark Kernels

Matrix Multiplication. To evaluate C++ AMP functionality, programmability and baseline performance, we wrote a simple matrix multiplication kernel. Below is the code snippet [16] that was used:

Listing 1.1. Matrix Multiplication Kernel in C++ AMP

```
double *ha, *hb, *hc;
// allocate and initiliaze host data
void MatrixMultiply(ha, hb, hc) {

    array_view<double, 2> a(SIZE, SIZE, ha);
    array_view<double, 2> b(SIZE, SIZE, hb);
    array_view<double, 2> product(SIZE, SIZE, hc);

    parallel_for_each(product.extent,
        [=](index<2> idx) restrict(amp) {
            int row = idx[0];
            int col = idx[1];
            for (int inner = 0; inner < SIZE; inner++) {
                product[idx] += a(row, inner) * b(inner, col);
            }
    });
    product.synchronize();
}
```

First, the 1-D host-memory arrays `ha`, `hb`, and `hc` are allocated and initialized to size `SIZE*SIZE*sizeof(double)`. Then these arrays are associated with the `array_view` objects `a`, `b`, and `product`. The `array_view` can only be initialized with 1-D arrays or rectangular blocks of memory. Next, the `parallel_for_each()` construct is used to parallelize the kernel over the row and columns of the matrix multiplication. After the computation completes, the `array_view` object on the host and accelerator are synchronized to ensure data coherency. We compiled and ran the C++ AMP matrix multiplication kernel on the Titan and test Summit nodes using the Pathscale ENZO 6.0.9 compiler

that supports C++ AMP on multiple GPUs. For comparing the results in Fig. 1, we show the code listing for the tiled GEMM implementation in listing 1.2, but omit detailed discussion as this is available in other materials [16]:

Listing 1.2. Matrix Multiplication Kernel in C++ AMP with Tiling

```
double *ha, *hb, *hc;
// allocate and initiliaze host data
void MatrixMultiply(ha, hb, hc) {

    array_view<double, 2> a(n, n, A);
    array_view<double, 2> b(n, n, B);
    array_view<double, 2> product(n, n, C);

    // Call parallel_for_each by using 2x2 tiles.
    parallel_for_each(product.extent.tile< TS, TS >(),
        [=] (tiled_index< TS, TS> t_idx) restrict(amp)
        {
            int row = t_idx.local[0];
            int col = t_idx.local[1];
            int rowGlobal = t_idx.global[0];
            int colGlobal = t_idx.global[1];
            int sum = 0;

            for (int i = 0; i < n; i += TS) {
                tile_static int locA[TS][TS];
                tile_static int locB[TS][TS];
                locA[row][col] = a(rowGlobal, col + i);
                locB[row][col] = b(row + i, colGlobal);
                t_idx.barrier.wait();

                for (int k = 0; k < TS; k++) {
                    sum += locA[row][k] * locB[k][col];
                }

                t_idx.barrier.wait();
            }

            product[t_idx.global] = sum;
        });
        product.synchronize();

}
```

QMCPACK - three-body Jastrow factor QMCPACK [17,18] is an open-source software package that enables quantum Monte Carlo (QMC) simulations of realistic materials on large parallel computers. It is implemented using C++ object-oriented and generic programming design patterns, and achieves efficient parallelism through the hybrid use of MPI/OpenMP and inlined specializations to use SIMD intrinsics. Additionally, a port to CUDA for NVIDIA GPU acceleration

was done, but some of the data structures and algorithms needed refactoring for efficient execution on the accelerator. QMCPACK is one of the applications participating in the CORAL [13] application readiness program (CAAR) for the POWER-based Summit [15] system to be deployed as the next leadership-class machine at Oak Ridge National Lab (ORNL).

Quantum Monte Carlo methods are a class of stochastic-based, ab initio electronic structure calculations to solve the time-independent Schrödinger equation in quantum mechanics for the ground state energy and its corresponding physical state, or the so-called wavefunction. Regardless of the algorithm employed, the code takes a trial wavefunction as an initial input. It then employs an iterative Monte Carlo procedure to optimize the wavefunction and obtains the ground state.

One commonly used type of wavefunction is composed of a product of Slater determinants and Jastrow factors. The Slater determinants encapsulate the electrons' distribution, whereas the Jastrow factors capture the Coulombic interactions among the electrons or ions. The kernel that we are porting here to C++ AMP is a prototype of the evaluation of the three-body Jastrow factor, which accounts for the interactions among any two electrons and an ion for the entire system. Thus, there are three nested `for` loops in the kernel, two of which loop over the number of electron-ion pairs, and one which loops over the number of electron-electron pairs in the physical system. It is for this reason the calculation of the three-body Jastrow is computationally intensive, as the number of electrons in a typical calculation could be few hundred up to thousands.

Listing 1.3 shows the original version of the QMCPACK Jastrow kernel. The code uses several custom linear vector and tensor classes `TinyVector`, `Tensor`, and `MyVector`. The result of the kernel is captured in the `grad` and `hess` arguments.

Listing 1.3. Original QMCPACK Kernel

```
inline
real_type evaluate(real_type r_12, real_type r_1I,
        real_type r_2I, TinyVector<real_type,3> &grad,
        Tensor<real_type,3> &hess,
        MyVector &gamma){
  real_type val = 0.0;  grad = 0.0;  hess = 0.0;
  real_type r2l(1.0), r2l_1(0.0), r2l_2(0.0), lf(0.0);
  for (int l=0; l<=N_eI; l++) {
    real_type r2m(1.0), r2m_1(0.0), r2m_2(0.0), mf(0.0);
    for (int m=0; m<=N_eI; m++) {
      real_type r2n(1.0), r2n_1(0.0), r2n_2(0.0), nf(0.0);
      for (int n=0; n<=N_ee; n++) {
        real_type g = gamma(l,m,n);
        val += g*r2l*r2m*r2n;
        grad[0] += nf * g *r2l * r2m * r2n_1;
        // Omit code for grd[1] and grd[2]
        hess(0,0) += nf*(nf-1.0) * g * r2l   * r2m  * r2n_2;
        // Omit code for calculating other hess() entries
```

```
          r2n_2 = r2n_1; r2n_1 = r2n; r2n *= r_12; nf += 1.0;
        }
        r2m_2 = r2m_1; r2m_1 = r2m; r2m *= r_2I; mf += 1.0;
      }
      r2l_2 = r2l_1; r2l_1 = r2l; r2l *= r_1I; lf += 1.0;
    }
    for (int i=0; i<C; i++){
      hess(0,0)=(r_1I - L)*(r_2I - L)*hess(0,0);
      // Omit code for updating other hess() entries
      grad[0] = (r_1I - L)*(r_2I - L)*grad[0];
      // Omit code for updating other grad() entries
      val *= (r_1I - L)*(r_2I - L);
    }
    hess(1,0) = hess(0,1);
    hess(2,0) = hess(0,2);
    hess(2,1) = hess(1,2);
    return val;
  }
```

The motivation to explore the use of C++ AMP for this kernel came from the fact that it had not been ported to CUDA yet, and initial attempts to use directives for accelerator offload were not satisfactory, requiring reduced usage of custom C++ classes and data structures needed in the application. Furthermore, developer investment in CUDA is being reduced for this application for portability reasons. Using C++ AMP to parallelize the two loops requires capturing the main data structures into `array<>` objects for access on the accelerator inside the `parallel_for_each` looping construct. Listing 1.4 shows the C++ AMP code corresponding to the three nested `for` loops making up the first part of the QMCPACK kernel. Note that we omit some of the common code elements and present primarily the parts that illustrate the modifications needed to adapt the kernel to the C++ AMP interface.

Listing 1.4. QMCPACK Kernel Using C++ AMP

```
    real_type grd_acc[3] = {0.0, 0.0, 0.0};
    real_type hess_acc[6] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
    // local for grad
    real_type grd0, grd1, grd2;
    grd0 = grd1 = grd2 = 0.0;
   // local for gamma (need to be changed to a 1D array)
    int g_size = (N_eI+1) * (N_eI+1) * (N_ee+1);
    real_type gmm[g_size];
    for (int l=0; l<=N_eI; l++)
      for (int m=0; m<=N_eI; m++)
        for (int n=0; n<=N_ee; n++)
          gmm[l*N_eI*N_eI + m*N_eI + n] = 1.0;

    real_type _r2l[N_eI], _r2l_1[N_eI], _r2l_2[N_eI];
    _r2l[0] = 1.0; _r2l_1[0] = _r2l_2[0] = 0.0;
    for (int l=0; l<N_eI-1; l++){
      _r2l_2[l] = _r2l_1[l-1];
      _r2l_1[l] = _r2l[l-1];
      _r2l[l] = _r2l[l-1] * r_1I;
    }
```

```
  {
      extent<2> e(N_eI, N_eI);
24    array_view<real_type, 1> av_gmm(g_size, gmm);
      array_view<real_type, 1> av_r2l(N_eI, _r2l);
      array_view<real_type, 1> av_r2l_1(N_eI, _r2l_1);
      array_view<real_type, 1> av_r2l_2(N_eI, _r2l_2);
      array<real_type, 2> value(e);
29    array<real_type[3], 2> grd(e);
      array<real_type[6], 2> hss(e);
      parallel_for_each(e,
          [=, &value, &grd, &hss](index<2> idx) restrict(amp) {
        int l = idx[0]; int m = idx[1];
34      real_type r2n(1.0), r2n_1(0.0), r2n_2(0.0), nf(0.0);
        real_type mf = (real_type)m;   real_type lf = (real_type)l;
        real_type r2l = av_r2l[l];     real_type r2l_1 = av_r2l_1[l];
        real_type r2l_2 = av_r2l_2[l]; real_type r2m = av_r2l[m];
        real_type r2m_1 = av_r2l_1[m]; real_type r2m_2 = av_r2l_2[m];
39      for (int n=0; n<=N_ee; n++){
          const real_type g = av_gmm[l *N_eI * N_eI + m * N_eI + n];
          value[idx] += g*r2l*r2m*r2n;
          grd[idx][0] += nf * g *r2l   * r2m   * r2n_1;
          // Omit code for other grd[] entries
44        hss[idx][0] += nf*(nf-1.0) * g * r2l   * r2m   * r2n_2;
          // Omit code for other hss[] entries
          r2n_2 = r2n_1; r2n_1 = r2n;  r2n *= r_12;
          nf += 1.0;
        } // end for n
49    } // end parallel_for lambda function
      ); //end parallel_for
  }
```

The code illustrates the general approach for porting an existing application to the C++ AMP programming model. Since the C++ AMP data model is implemented primarily using the `array<>` and `array_view<>` classes, existing data generally needs to go through a copy-in/copy-out process to the corresponding C++ AMP data structure. The overhead for creating and accessing data through the C++ AMP data structures will depend on how compatible the underlying memory layout is with the layout supported by C++ AMP (`array_views`s can be created using raw pointers as shown in Listing 1.4).

The listing also shows an example of creating and using *accelerator-only* data structures to control data movement into and out of an accelerator with disjoint memory. The variables `value, grd` and `hss` are used in the listed part of the kernel. Their lifetime extends to the rest of the kernel (not shown above) where the reduction operation is performed. They are then explicitly copied over to their host counterparts at the end of the kernel function execution.

2.2 Preliminary Performance Evaluation

The first panel of Fig. 1 shows the performance achieved using different matrix sizes on the test Summit node, and second panel of Fig. 1 shows the performance achieved using different matrix sizes on the Titan node. The execution times shown include the data transfer time between host and device. Each GEMM experiment uses double-precision data and compares the C++ AMP code represented by listing 1.1 to a more optimized C++ AMP implementation using a tiled algorithm as well as the highly-tuned NVIDIA CUBLAS DGEMM routine. While this kernel realizes the expected performance improvement when moving

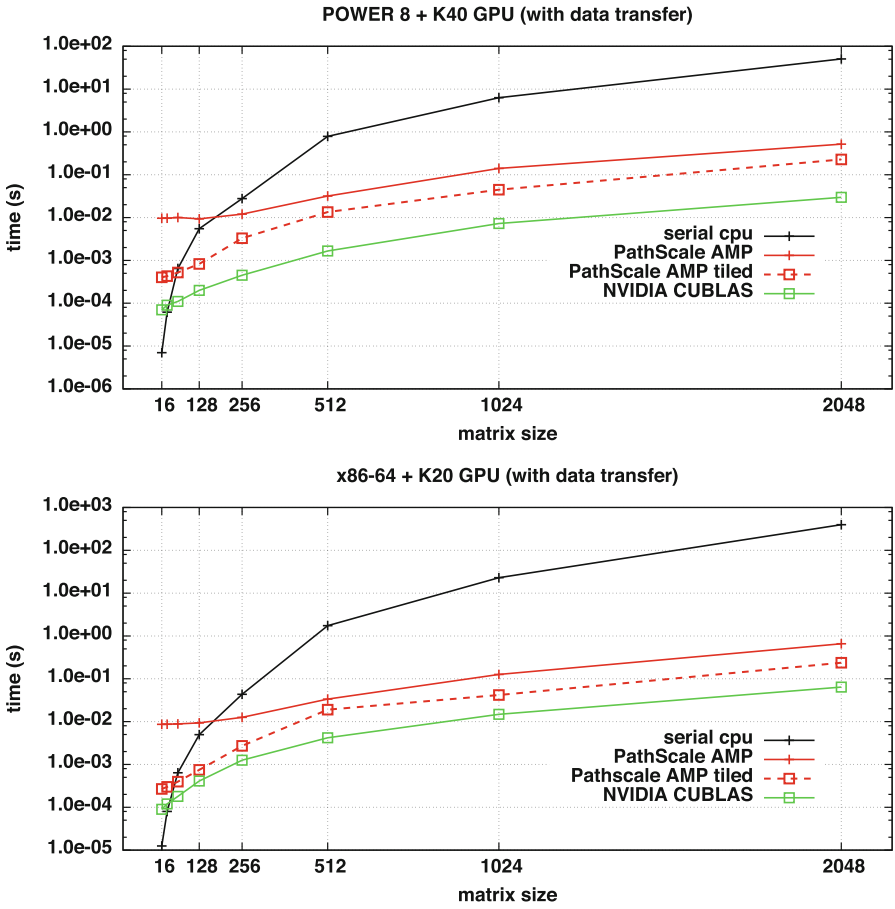

Fig. 1. Matrix multiplication Kernel in C++ AMP on a preliminary test Summit node (Power8E CPU @ 2.61GHz processor with a NVIDIA Tesla K40 m connected via PCIe v3) and a Titan node (16-core AMD Opteron 6274 CPU attached to an NVIDIA Tesla K20X GPU via PCIe v2).

from the K20 to the K40, the relatively basic AMP implementations do not see quite the amount of improvement of the hyper-tuned CUBLAS implementation. Also, as this is a compute-bound kernel, we do not expect the improved PCIe bandwidth of the POWER8 node to play a significant role in this case.

Figure 2 shows the performance speedup of the Jastrow QMC application kernel on our two HPC node types as described in Sect. 2. These timings include the time required for data transfer between host and device, as well as the manually-implemented reduction operation as explained below. The performance gain for large particle numbers is about an order of magnitude, and while the kernel involves a triply-nested loop, the computation to memory bandwidth density isn't quite as high as the GEMM algorithm. While we were able to run the kernel and accelerate it on the GPUs, C++ AMP currently lacks a reduction

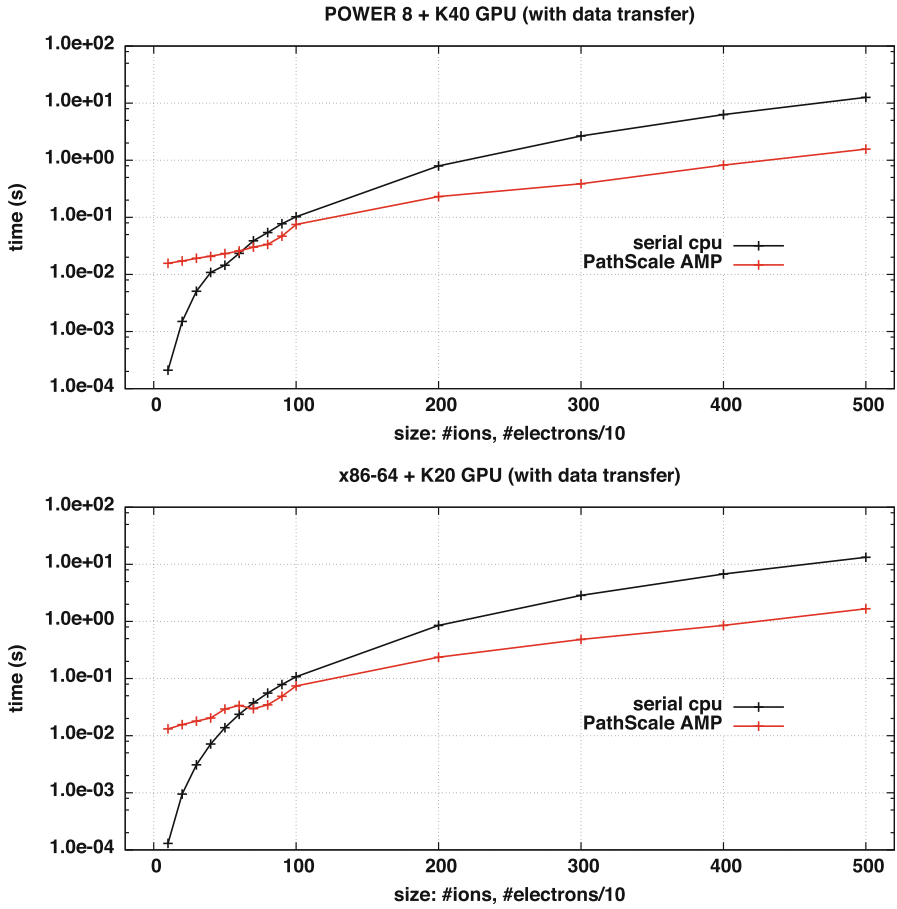

Fig. 2. QMC 3-body Jastrow kernel implemented in C++ AMP on a preliminary test Summit node (Power8E CPU @ 2.61 GHz processor with a NVIDIA Tesla K40m connected via PCIe v3) and a Titan node (16-core AMD Opteron 6274 CPU attached to an NVIDIA Tesla K20X GPU via PCIe v2).

construct. This led us to implement the reduction manually in the application. This is an area where C++ AMP needs improvement. The reduction was implemented using local arrays of type `tile_static` in order to share work among compute elements during the reduction operation. Listing 1.5 shows the implementation for the reduction for the calculated value, gradient, and Hessian of the Jastrow terms. We believe that the performance gain by moving from the CPU implementation to the accelerated AMP implementation could be further improved by having natively supported and well-optimized constructs for reduction operations. This would also increase the programmer productivity and code brevity regarding this kernel.

Listing 1.5. QMCPACK Kernel Using C++ AMP

```
extent<1> e2((N_eI+1)*(N_eI+1));
parallel_for_each(e2.tile<N_eI+1>(),
    [&](tiled_index<1> idx) restrict(amp) {
        int l = idx.tile[0];
        int m = idx.local[0];
        tile_static real_type v[N_eI+1];

        v[m] = value[l][m];
        idx.barrier.wait();
        if (m == 0) {
            for (int i=1; i<=N_eI; ++i) {
              v[0] += v[i];
            }
            value[l][0] = v[0];
        }
        for (int i=0; i<3; ++i) {
            v[m] = grd[l][m][i];
            idx.barrier.wait();
            if(m == 0) {
                for (int j=1; j<=N_eI; ++j) {
                    v[0] += v[j];
                }
                grd[l][0][i] = v[0];
            }
        }
        for (int i=0; i<6; ++i) {
            v[m] = hss[l][m][i];
            idx.barrier.wait();
            if(m == 0) {
                for (int j=1; j<=N_eI; ++j) {
                    v[0] += v[j];
                }
                hss[l][0][i] = v[0];
            }
        }
    });

    for (int i=0; i<=N_eI; ++i) {
      val += value[i][0];
      for (int j=0; j<3; ++j) grd_acc[j] += grd[i][0][j];
      for (int j=0; j<6; ++j) hess_acc[j] += hss[i][0][j];
    }
```

3 Discussion

The C++ AMP programming model could be attractive for some C++ application developers because it offers a language-based solution for discrete accelerator

offload, yet works well with native language features. Ideally, HPC applications would be well-supported by features in the C++ language standard itself, and indeed progress is being made in this direction. NVIDIA, Microsoft, and Intel independently proposed library approaches for standardized C++ parallelism, and these authors were eventually asked to submit a joint proposal to the committee, which was then refined over two years and informed along the way by experimental implementations. The result of this effort can be found in the parallelism technical specification (TS) N4507 which was subsequently included into the C++17 standard.

The parallelism features that have been included in C++17 show some similarities to the AMP model, defining execution policies and methods to specify computational kernels. It even includes exception handling, which is not covered by the AMP specification. However, the main feature set missing from C++17 that may prevent its wide adoption among HPC applications is the lack of data handling facilities. For heterogeneous systems with accelerators that have discrete memory address spaces, there is currently no way to specify which data should be moved between memory spaces and when the movement should take place. However, the concurrency and parallelism subgroup of the C++ language committee is working on followups to both technical specifications that will further augment the features that are included in the C++17 standard. Features are being considered [19,20] from HPX [21] and OpenCL [22] because, even though they include an HPC domain view-point, they are modeled after the existing parallel and concurrency TSs and so retain appropriateness for the consumer domain as well.

4 Early Conclusions and Future Work

In this paper we describe how C++ AMP works and can potentially be used on different platforms including x86-64 and OpenPOWER systems with NVIDIA GPUs. We describe the language constructs that C++ AMP provides to accelerate applications written in C++. We were able to use C++ AMP to accelerate a matrix multiplication kernel and important computational regions from the QMCPACK application. The success from AMP is its ability to use parallel primitives and data constructs that fit the native C++ programing model. Evaluating the C++ AMP programming model is a step toward a C++ solution to program accelerators. One of the differences with C++ AMP and the upcoming C++17 draft is that C++ AMP is aware of the different memory spaces between the accelerator and host; the language provides namespaces and objects to manage and synchronize shared data objects between the host and the accelerator. Upcoming explorations will include immediate concerns such as a more generalized yet performant way to handle data reductions within AMP parallel regions and exploring more target accelerator and multicore architectures. Longer-term studies in which we are interested include more detailed comparisons with the newly released C++17 concurrency and parallelism features which are only recently emerging in compiler implementations.

Acknowledgements. This material is based upon work supported by the U.S. Department of Energy, Office of science, and 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.

References

1. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). Domain-Specific Languages and High-Level Frameworks High-Performance Computing. http://www.sciencedirect.com/science/article/pii/S0743731514001257
2. Hornung, R.D., Keasler, J.A.: The RAJA portability layer: Overview and status (2014). https://e-reports-ext.llnl.gov/pdf/782261.pdf
3. Hoberock, J.: Working draft, technical specification for C++ extensions for parallelism (2014). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4071.htm
4. Beyer, J.C., Stotzer, E.J., Hart, A., de Supinski, B.R.: OpenMP for accelerators. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 108–121. Springer, Heidelberg (2011)
5. Liao, C., Yan, Y., de Supinski, B.R., Quinlan, D.J., Chapman, B.: Early experiences with the OpenMP accelerator model. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 84–98. Springer, Heidelberg (2013)
6. CAPS, CRAY and NVIDIA, PGI: The OpenACC application programming interface (2013). http://openacc.org
7. Microsoft Corporation: C++ AMP: Language and programming model (2013). http://download.microsoft.com/download/2/2/9/22972859-15C2-4D96-97AE-93344241D56C/CppAMPOpenSpecificationV12.pdf
8. Microsoft Corporation "Reference (C++ AMP)" (2012). http://msdn.microsoft.com/en-us/library/hh28939028v=vs.11029.aspx
9. PathSCale Inc.: PathScale EKOPath Compiler & ENZO GPGPU Solutions (2016). http://www.pathscale.com
10. Sharlet, D., Kunze, A., Junkins, S., Joshi, D.: Shevlin Park: ImplementingC++ AMP with Clang/LLVM and OpenCL 2012 LLVM Developers' Meeting (2012). http://llvm.org/devmtg/201211#talk10
11. HSA Foundation: Bringing C++ AMP Beyond Windows via CLANG and LLVM (2013). http://www.hsafoundation.com/bringing-camp-beyond-windows-via-clang-llvm/
12. INCITE program. http://www.doeleadershipcomputing.org/incite-program/
13. CORAL fact sheet. http://www.anl.gov/sites/anl.gov/files/CORAL%20Fact%20Sheet.pdf
14. Bland, A.S., Wells, J.C., Messer, O.E., Hernandez, O.R., Rogers, J.H.: Titan: early experience with the cray XK6 at Oak Ridge National Laboratory. In: Proceedings of Cray User Group Conference (CUG) (2012)
15. SUMMIT: Scale new heights. Discover new solutions. https://www.olcf.ornl.gov/summit/
16. Walkthrough: Matrix multiplication. https://msdn.microsoft.com/en-us/library/hh873134.aspx

17. Kim, J., Esler, K.P., McMinis, J., Morales, M.A., Clark, B.K., Shulenburger, L., Ceperley, D.M.: Hybrid algorithms in quantum Monte Carlo. J. Phys.: Conf. Ser. **402**(1), 012008 (2012). http://stacks.iop.org/1742-6596/402/i=1/a=012008
18. Esler, K.P., Kim, J., Schulenburger, L., Ceperley, D.: Fully accelerating quantum monte carlo simulations of real materials on GPU clusters. Comput. Sci. Eng. **13**(5), 1–9 (2011)
19. Wong, M., Kaiser, H., Heller, T.: Towards Massive Parallelism (aka Heterogeneous Devices/Accelerator/GPGPU) support in C++ with HPX (2015). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0234r0.pdf
20. Wong, M., Richards, A., Rovatsou, M., Reyes, R.: Kronos's OpenCL SYCL to support Heterogeneous Devices for C++ (2016). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0236r0.pdf
21. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: HPX: a task based programming model in a global address space. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, ser PGAS 2014, pp. 6:1–6:11. ACM, New York (2014). http://doi.acm.org/10.1145/2676870.2676883
22. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. IEEE Des. Test **12**(3), 66–73 (2010). http://dx.doi.org/10.1109/MCSE.2010.69