LLNL-CONF-474253

# OpenMP for Accelerators

J. C. Beyer, E. J. Stotzer, A. Hart, B. R. de Supinski

March 17, 2011

International Workshop on OpenMP (IWOMP)
Chicago, IL, United States
June 13, 2011 through June 15, 2011

# OpenMP for Accelerators

James C. Beyer[1], Eric J. Stotzer[2], Alistair Hart[3], and Bronis R. de Supinski[4]

[1] Cray Inc., 380 Jackson Street, Suite 210 St. Paul, MN
[2] Texas Instruments Inc., 12500 TI Boulevard, Stafford, TX
[3] Cray European Exascale Research Initiative, c/o EPCC, University of Edinburgh
[4] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
beyerj@cray.com, estotzer@ti.com, ahart@cray.com, bronis@llnl.gov

**Abstract.** OpenMP [13] is the dominant programming model for shared-memory parallelism in C, C++ and Fortran due to its easy-to-use directive-based style, portability and broad support by compiler vendors. Similar characteristics are needed for a programming model for devices such as GPUs and DSPs that are gaining popularity to accelerate compute-intensive application regions. This paper presents extensions to OpenMP that provide that programming model. Our results demonstrate that a high-level programming model can provide accelerated performance comparable to hand-coded implementations in CUDA.

**Keywords:** OpenMP, Accelerator, GPU, DSP, CUDA

## 1 Introduction

The rapid growth in the application of GPUs as accelerators has increased interest in programming models that comprehend heterogeneous systems composed of a host and an attached accelerator. Embedded systems designers have long relied on accelerators to improve system performance for specific application areas. In general, programming models have been difficult to implement on the irregular hardware features of the accelerators in embedded systems.

Although hardware accelerator performance continues to outpace general processor performance, accelerators remain difficult to program. For example, Compute Unified Device Architecture (CUDA) [11] and Open Computing Language (OpenCL [8]), the two dominate programming models for GPUs, support high-performance accelerator algorithms but require the programmer to rewrite their code specifically for the target architecture (with CUDA, specifically NVIDIA GPUs).

We propose OpenMP extensions that support accelerators (cache coherent or not) without requiring the programmer to rewrite the code. Our extensions add the concept of execution engines (i.e., accelerators) that the runtime manages. The programmer identifies accelerator regions through directives or accelerator-specific functions. We also extend the OpenMP memory model to ensure data integrity across these regions.

Our example changes for matrix multiply and MG from the NPB suite [3] show that these OpenMP extensions provide a simple mechanism to target C, C++ and Fortran code to accelerators. We show that our prototype compiler implementation provides comparable performance to hand-coded CUDA implementations.

## 2 Motivation

The motivation for developing hardware accelerators is efficiency improvement with respect to power, performance, and silicon area. High performance applications are characterized by dynamic compute-intensive regions. Optimizing the compute-intensive regions of the application is highly desirable. Often these regions of the application contain loops with high degrees of parallelism. One way to dramatically improve the overall performance of the application is to execute the compute-intensive regions on a hardware accelerator. Because the accelerator is specialized for a specific function, it eliminates the non-essential circuitry that must be present on a general-purpose processor. This enables the accelerator to use less power to execute the same function. With respect to area, specialized accelerator hardware tends to be much smaller than the number of general-purpose processors that would be required to execute the accelerated code block in the same amount of time.

Several vendors have defined their own languages or have extended the C language to address parallelism on custom devices. For example, Clearspeed defined Cn [5], an extension of C to support their data-parallel architecture by providing the definitions of mono (scalar) and poly (parallel,or replicated) data types. The Multicore Association has proposed a message-passing model, referred to as the Communications API (CAPI) [10]. Intel proposed an extension to OpenMP in its EXOCHI [16] programming environment.

The following works were evaluated and influence, sometimes heavily, the extensions to the OpenMP specification that are presented in the paper.

StarSs extensions to OpenMP [2] This proposal, gives the programmer the ability to tell the OpenMP compiler that a particular snippet of code is to be compiled for a particular accelerator or that it will be compiled for several different accelerators and the runtime will have to figure out which accelerator is the best to use when the task is run. One of the strongest features of this proposal is that it allows the programmer to annotate the header file and to never expose the actual code to the compiler. This allows libraries to be augmented for accelerators and then called directly from user code with the correct calling sequence for the given accelerator.

hiCUDA [6] is a high-level directive-based language targeting CUDA. A prototype source-to-source compiler was implemented which showed no loss of performance between writing the code in hiCUDA and CUDA.

The hybrid multicore parallel programming (HMPP) language, from CAPS [4] Entreprises, targets GPUs via directive-based programming and introduced codelets, asynchronous tasks executed by an accelerator. The system supports C, Fortran and C++ with the goal of doing form GPUs what OpenMP has done for multi-thread programming. This system supports a diverse set of accelerators, Nvidia, ATI/AMD, SSE, CELL and OpenCL engines.

The PGI accelerator directives is a collection of compiler directives used to specify regions of code in Fortran and C programs that can be offloaded from a host CPU to an attached accelerator. The method outlined provides a model for accelerator programming that is portable across operating systems and various types of host CPUs and accelerators. [14]

OpenMPC [9] is a programming interface which builds on OpenMP to provide an abstraction of the CUDA programming model. The system addes CUDA specific directives to existing OpenMP parallel region directives. The directives are added automatically by the system to parallel regions, with the user overriding decisions when needed. The performance numbers, 88% of hand written CUDA, are impressive, however, the approach ignores the OpenMP execution model.

Accelerators are common in embedded computing, where they tend to be programmed using low-level vendor specific APIs. Currently, the most well known accelerators are GPUs, which have a separate memory space that is connected to the host (and network) via a relatively slow (high latency, low bandwidth) PCIe bus. The dominant programming models for GPUs are currently CUDA and OpenCL. Both give the programmer the power to extract performance from the accelerator but also require the programmer to rewrite their code for the target architecture at a low level. While future architectures will integrate these accelerators and full-featured cores on the same die with direct network access, we expect the accelerators will remain difficult to program [1, 7].

The low level code required by CUDA and OpenCL is often repetitive and error-prone. It often focuses on moving data between the host CPU and the accelerator. Compilers could implement significant portions of this repetitive code. Thus, we propose extensions to OpenMP that allow the programmer to accelerate key kernels or entire applications by adding directives to the original source code (Fortran, C or C++). These directives do not alter the existing code that runs well on the host CPU. Overall, our extensions support rapid, maintainable accelerator code development while leaving performance optimizations to the compiler and runtime environment. We also provide optional directive clauses that can guide these optimizations.

## 3 Changes to OpenMP Models

Our extensions require fundamental changes to the OpenMP execution and memory models. We add the concept of an *accelerator region* to the execution model. The runtime generates an explicit *accelerator task* when a thread encounters an accelerator region. If the system has multiple accelerators that can execute the region, then the runtime determines which one to use. If the system has multiple types of accelerators, the runtime ensures that an appropriate version is available for the one that it selects. Execution of that accelerator task is then tied to that accelerator type. The accelerator task must complete before the next OpenMP barrier completes. Alternatively, we provide an *accelerator task synchronization* construct that enforces completion of the accelerator task. All accelerator tasks must complete before the program exits.

Accelerators can have both private and shared accelerator-resident objects. They can access host memory via direct access or via data copying. Data motion directives provide hints to the compiler on where to place data that accelerator regions access. Accelerators do not have memory equivalent to *threadprivate* memory; all *threadprivate* data is treated as *firstprivate*. Since we target non-cache coherent accelerators as well as integrated ones that support cache coherence, host threads must use explicit synchronization to modify or to read a memory location that is duplicated on the accelerator; otherwise the result is unspecified. Both the host and the accelerator must execute flushes to ensure that global memory modifications made by one are visible by the other. However, accelerator-resident shared objects can be updated or copied to the host whenever the accelerator is not modifying them.

## 4 Directives

This section describes the syntax and behavior of our new directives and clauses.

### 4.1 Accelerator Region Construct

**Syntax:**
***C/C++***
#pragma omp acc_region *[clause [[,] clause]...]* newline
    Structured-block
***Fortran***
!$omp acc_region [clause[[,] clause]...]
    Structured-block
!$omp end acc_region
***Clauses***

**device***( integer-expression [,integer-expression])*
**if***(scalar-expression)* or **if***(scalar-logical-expression)*
**num_pes***(depth:number [, depth:number] )*
**acc_shared***(list)*
**acc_copy***(list)*
**acc_copyin***(list)*
**acc_copyout***(list)*
**host_shared***(list)*
**firstprivate***(list)*
**private***(list)*
**present***(list| *)*
**default*(acc_shared|acc_copy|firstprivate|private|none|ignore)***

This fundamental construct starts an accelerator region. Its primary purpose is to delineate the code that is to be run on an accelerator. When a thread encounters an **ACC_REGION** construct, it creates an accelerator task that can be assigned to an accelerator or to the threads in a parallel team (a new parallel team will be constructed if the construct is not inside a parallel region). If the **ACC_REGION** is inside a data environment it shall be assigned to the same device to which the **ACC_DATA_REGION** (Section 4.2) is bound.

The *device* clause causes the construct to be tied to the accelerator determined by the constant positive integer expression. The $if$ clause determines whether the region is accelerated: the region is accelerated if the expression evaluates to true; otherwise, the encountering thread runs the region. The $num_pes$ clause controls the number of processing elements that are applied to a given level. The depth refers to the level of parallelism that is to be exploited. A depth greater than the current depth causes pre-allocation of resources for when that depth is reached. A depth:number pair is ignored if the depth is less than the current depth. The compiler may generate improved code if the depth and number positive integer expressions are constants. We discuss the data clauses in Section 4.7.

## 4.2 Accelerator Data Region Construct

**Syntax:**
***C/C++***
#pragma omp acc_data *[clause [[,] clause]...]* newline
    Structured-block
***Fortran***
!$omp acc_data [clause[[,] clause]...]
    Structured-block
!$omp end acc_data
***Clauses***

**device**( integer|expression [, integer|expression])
**acc_copy***(list)*
**acc_copyin***(list)*
**acc_copyout***(list)*
**host_shared***(list)*
**acc_shared***(list)*
**present***(list| *)*
**default*(acc_shared|host_shared|acc_copy|none|ignore)***

This fundamental construct starts an accelerator data region. Its primary purpose is to define a data scope that applies to multiple accelerated constructs. When a thread encounters an **ACC_DATA_REGION** construct, it creates an accelerator task data environment. This task data environment is assigned to the default accelerator or to the accelerator specified by the device clause, which causes the construct to be tied to the accelerator determined by the constant integer expression. As stated previously, we discuss the data clauses in Section 4.7.

## 4.3 Accelerator Loop Construct

**Syntax:**
***C/C++***
#pragma omp acc_loop *[clause[, clause]...]* new-line
  For-loop-nest
***Fortran***
!$omp acc_loop *[clause[, clause]...]*
Do-construct
!$omp end acc_loop
***Clauses***

**host***( expr )*
**level***(dimension )*
**max_par_level***(expr)*
**num_pes***(depth:number [, depth:number] )*
**reduction***(operator:list)*

The accelerator loop construct specifies that the iterations of one or more associated loops will accelerated. The **ACC_LOOP** construct is associated with a loop nest consisting of one or more loops that follow the **ACC_LOOP** directive.

The host clause causes the associated loop nest to execute on the host if the expression evaluates to true. If the expression evaluates to false, or neither the host nor the hetero clause appears, all iterations execute on the accelerator. The level clause causes the associated loop to be spread across the accelerator at the dimension, or level of parallelism on the accelerator, specified. If a dimension that the constant positive integer expression specifies is not supported then the compiler will schedule the loop to be run sequentially. The *max_par_level* clause indicates that the level of parallelism in the loop will not exceed expr, which is a constant positive integer expression. The compiler uses this information to determine how to utilize resources on the accelerator. The *num_pes* clause controls the number of processing elements that are applied to a given level. As with the **ACC_REGION** construct, a depth greater than the current depth causes pre-allocation of resources while the depth:number pair is ignored if it is less than that depth. For each list item of a reduction clause, each "thread" creates a private copy that is initialized appropriately for the operator. After the end of the region, the original list item is updated to the result of combining the private copies using the specified operator.

## 4.4 Accelerator Region Loop Construct

**Syntax:**
***C/C++***

#pragma omp acc_region_loop *[clause[[,] clause]...]* new-line
    For-loop-nest
***Fortran***
!$omp acc_region_loop *[clause[[,] clause]]*
    Do-loop-nest
!$omp end acc_region_loop
***Clauses***

See component constructs.

The combined **ACC_REGION_LOOP** construct creates an accelerator region with an accelerator loop. This construct must contain a single loop nest.

## 4.5 Accelerator Call Construct

**Syntax:**
***C/C++***
#pragma omp acc_call *[clause[[,]clause]...]* newline
    Function call expression
***Fortran***
!$omp acc_call *[clause[[,] clause]...]*
    Call statement
!$omp end acc_call
***Clauses***

**device***(integer-expression)*
**if***(scalar-expression)*
**implements***(device:name [, device:name])*
**num_pes***(depth:num [, depth:num] )*
**acc_copy***(list)*
**acc_copyin***(list)*
**acc_copyout***(list)*
**present***(list| *)*

The accelerator call construct specifies that a copy of the associated call has an accelerated version in a user provided location. The **ACC_CALL** construct causes the encountering thread to request that the runtime determine the best way to accelerate the associated call. Functions that return anything will write to an accelerator memory location and this location will be copied back to the host, unless the system supports some other method of returning values. The size of the resulting data must be determinable before the call is launched on the accelerator.

If the device clause is present then only the associated type of device as determined by constant positive integer expression can be used to accelerate the call. As with the **ACC_REGION** construct, the if clause determines whether the call is accelerated. The implements clause provides a mechanism by which the programmer can tell the compiler that a prototype is implemented for "device" with name "name". This mechanism allows the programmer to provide their own accelerator version without having to determine the proper name mangling for routine. As with the **ACC_REGION** construct, the **num_pes** clause controls the number of processing elements that are applied to a given level.

### 4.6 Accelerator Update Directive

**Syntax:**
***C/C++***
#pragma omp acc_update clause*[, clause]...* new-line
***Fortran***
!$omp acc_update clause*[, clause]...*
***Clauses***

**host***(obj1[:obj2] [,obj1[:obj2]])*
**acc***(obj1[:obj2] [,obj1[:obj2]])*

The update directive is used within an explicit or implicit data region to update all or part of a host memory array with values from the corresponding array in accelerator memory, or to update all or part of an accelerator memory object with values from the corresponding object in host memory. The *host* clause causes the obj1 objects to be copied from the accelerator memory to the host memory, if the host and accelerator memory are distinct. The *acc* clause causes the obj1 objects to be copied from the host memory to the accelerator memory, if the memories are distinct. The optional *obj*2 object allows the programmer to move data between two objects that have different identifiers; when it is provided, obj1 is the source object and obj2 is the destination object. This directive allows both the host and the accelerator to work independently and then to update each other before continuing.

### 4.7 Data Environment

The following section defines the data clauses used in the various constructs.

**General Clauses:**

*Default clause:* The default clause specifies the default behavior for all objects that are used but not explicitly placed in a given memory type. A $default(none)$ clause requires all objects used inside the construct to be present on a data-sharing attribute clause list. The default is copy if no default is provided. A $default(ignore)$ clause requires any objects that are not explicitly scoped to be in a *present* clause.

*Cache clause:* The *cache* clause causes the compiler to attempt to place the object at the memory depth requested. If the depth is greater than the supported depth of the accelerator, it will be placed on the depth closest to the requested depth.

**Data-sharing Attribute clauses:**

*Host_shared clause:* The *host_shared* clause causes the objects in the list to be shared with the host. The objects are left in host memory or copied and updated automatically by the compiler if the accelerator has direct access to the hosts memory. The compiler will attempt to move data between the host and accelerator so as to ensure correct memory semantics for the accelerator region if the accelerator does not have direct access to the host memory. The compiler may demote the accelerator region to a parallel region if it cannot determine how to move the data.

*Acc_shared clause:* The *acc_shared* clause causes the objects in the list to be shared by all tasks executing on the associated accelerator.

*Private clause:* The private clause causes a unique copy of the objects in the list to be provided to each task on the accelerator.

**Data copying clauses:**

*Acc_copy clause:* The *acc_copy* clause causes the accelerator shared objects to be initialized to the hosts memory state when the region starts and then causes the hosts memory state to be updated with the accelerator memory state when the region ends. This clause combines the behavior of the *acc_copyin* and *acc_copyout* clauses. An object that appears in an *acc_copy* clause cannot appear in other data sharing clauses.

*Acc_copyin clause:* The *acc_copyin* clause causes the accelerator shared objects to be initialized to the hosts memory state when the region starts. Objects in this clause may also appear in the *acc_copyout* clause.

*Acc_Copyout clause:* The *acc_copyout* clause causes the hosts memory to be updated with the state of the accelerator shared objects when the accelerator region ends. Objects in this clause may also appear in the *acc_copyin* clause.

*Present clause:* The runtime system checks for copies of list items in a present clause already on the accelerator. If the list item also appears in an *acc_shared*, *acc_copy*, *acc_copyin* or *acc_copyout* clause and the object is not found on the accelerator then the copy clause takes effect. The behavior is unspecified if the object is not already on the accelerator and is not in another data placement clause. The special "*" list is the same as listing all objects in the lexical region in the list.

*Firstprivate clause:* The *firstprivate* clause causes all private versions of the list objects to be initialized to the state of the associated shared object.

## 4.8 Array section specifications

Array shaping syntax must be used when arrays and pointers are used inside any of the clauses and the extents are to be limited or are unknown.

**Fortran:** Fortran array syntax can be used to define the array section. The placement of Explicit, Assumed and Deferred shape array types may be modified with the array section construct. CRI pointers inherit the shape of the pointee.

**C/C++:** We provide an extended array shaping syntax for C and C++. Shaping operator ::= [¡lower bound¿ : ¡length¿ : ¡stride¿] where ¡lower bound¿, ¡length¿, and ¡stride¿ are integer expressions that represent the integer values: ¡lower bound¿, ..., ¡lower bound¿ + (¡length¿ - 1) * ¡stride¿

Successive section operators designate a sub-array of a multidimensional array object. When absent, the ¡stride¿ defaults to 1. If the ¡length¿ is less than 1, the array section is undefined. We provide [:] as shorthand for a whole array dimension if the size of the dimension is known from the array declaration or a cast. The placement of Arrays, single and multi dimensional, Pointers, single level and multi-level and C++ vectors may be modified with the array sections construct.

## 5 Examples

### 5.1 Matrix Multiply

Our simple matrix multiply example shows how to accelerate a code region with minimal effort. This example takes over two pages in CUDA [12] We only need to add two lines of code to run on an accelerator. These additions move data between the host and the accelerator and generate the code for the accelerator. The following shows the complete implementation of our OpenMP-based routine for the accelerator:

```
!$omp acc_region_loop
    do j = 1,L
      do i = 1,N
        do k = 1,M
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
         enddo
      enddo
    enddo
!$omp end acc_region_loop
```

The main line:

!$omp acc_region_loop
conveys that this is an accelerator region loop, which is analogous to a parallel do or parallel for construct. The region construct instructs the compiler to place the code on the accelerator and the loop construct then instructs the compiler to workshare the next loop on the accelerator. The compiler determines how workshare the loop, for instance by stripmining the j-loop, although clauses on the directive can guide the compiler choices. We do not need to use data placement clauses: the compiler determines correct data movements at the start and end of the region construct. This default movement can be tuned using data placement clauses. In this case, we can use the following clauses to direct the correct behavior explicitly:

!$omp acc_region_loop acc_copyin(a,b) acc_copy(c)
The *acc_copyin* instructs the compiler to transfer the values the host has in the objects a and b to the accelerator. After the region the objects can simply be freed from the accelerator memory; they do not have to be copied back. The

*acc_copy* clause instructs the system to move the data from the host to the accelerator before the region and back to the host after the region.

A final (but important) refinement is to add the present clause:

!$omp acc_region_loop acc_copyin(a,b) acc_copy(c) present(a,b,c)

If the runtime determines that any of the objects a, b or c are already on the accelerator, then we use those copies without again copying their values from the host, which overrides the *acc_copyin* and *acc_copy* clauses. The user must ensure that the data is updated on the host or accelerator as needed (e.g., by using the *acc_data* or *acc_update* directives). Overall, the present clause serves two significant purposes: composability and data reuse. The composability feature allows us the flexibility to call a routine containing the construct from different call sites where the data may or may not be on the accelerator.

### 5.2 5.2 NAS Parallel Benchmark – MG

We modified the NPB MG [3] code to use our accelerator directives to demonstrate their utility. We only had to address one minor data sharing problem and then place approximately 25 directives in the (roughly 1500) lines of code. Since the code is too large to include entirely, we briefly describe the changes. We first added an *acc_data* region directive:

!$omp acc_data acc_shared(u,v,r) acc_copyin(a,c)

We associate this directive with the  75 lines of code that contitute the main body of the computation. This block of code has more than 20 calls, some of which contain computational loops that can be executed on the accelerator. The accelerator regions are primarily called from within this data region, which allows reuse of the data objects that are placed on the accelerator just once.

The remaining directives that we added are similar to the following line:

!$omp acc_region private(r1,r2) present(r,u,c) acc_copyin(r,c) acc_copy(u)
!$omp acc_loop

These regions create private copies of some objects and conditionally copy other objects to the accelerator, before starting one or more loop nests that are work-shared.

These 25 directives were incrementally added to the code. Many, but not all, replaced standard OpenMP parallel do constructs; a working (CPU) OpenMP version is a very good template from which to start. Thus, we ported the MG code to the accelerator with minimal modifications and modest effort. The port initially achieved a modest speed up of about 2.5 (relative to a single host CPU thread). This port demonstrates that a directive-driven compiler using the proposed constructs can efficiently place an existing code with little rewriting on an accelerator. We expect that additional tuning of the compiler technology will result in even better performance.

## 6 Performance

We evaluate the performance of accelerator directives against two metrics: execution speed; and the time that the programmer spends to port existing code to the accelerator. Section 5.1 presented a simple matrix multiplication example. We accelerated this example in a stepwise manner. We first verified that the sequential code was correct. We then parallelized that code with traditional OpenMP parallel do regions. Finally, we replaced the OpenMP constructs with our proposed accelerator directives and optimized.

The experimental setup is a Dual processor quad-core Xeon E5504 running at 2.00GHz with two attached Tesla C2050s. The performance results presented in Fig. 1 are the GigaFlops per-second achieved on the host using standard OpenMP and then the GigaFlops per-second achieved using the accelerator directives, plus the gigaflops per minute of effort to switch to accelerator directives from the best OpenMP performance. The standard OpenMP performance numbers show a reasonable increase in Gflops as the number of threads is increased to four. At four threads, the OS must start placing threads on the second processor, which leads to performance problems due to the NUMA nature of the processor. The Accelerator performance numbers present both the total performance for given experiment and the Gflops achieved per minute of work spent moving the code to the accelerator.

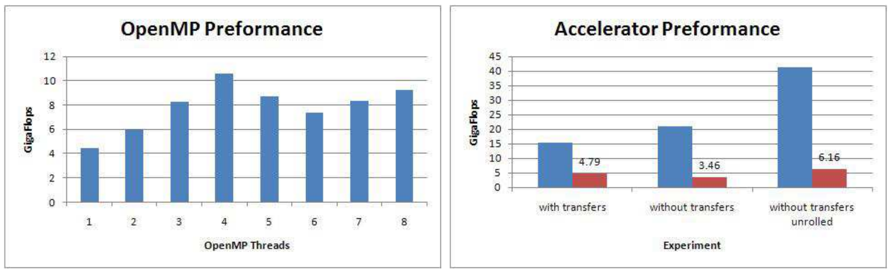

**Fig. 1.** Matrix Multiplication

The first experiment simply replaced the *parallel do* with an *acc_region_loop* construct. This took about a minute and lead to a 4.79 Gflop increase in performance over the 4 thread run. The second experiment simply moved the transfer times out of the computation by utilizing the data region construct and the present clause. This took an additional minute of work leading to a 3.46 Gflop per minute of effort improvement over the standard OpenMP version. Finally the loop body was hand unrolled to achieve a speedup that a more mature compiler could be expected to achieve without help. The unrolling almost doubled the performance on the accelerator. The performance is still significantly below peak, PGI CUDA [15] for Fortran tests have achieved 5 times the speed on this same test; however, they require the code to be rewritten in CUDA, which limits the portability of the code.

## 7 Conclusion

Accelerators have efficiency advantages that improve performance and reduce power consumption and cost. However, the challenge is to present a usable programming model. Existing models such as CUDA [11, 12] and OpenCL [8] force the programmer to rewrite their code specifically for the target architecture. We have proposed OpenMP extensions that support a wide range of accelerators without requiring the programmer to rewrite their code. With these extensions the programmer identifies regions of code and data that are offloaded to an accelerator. Using matrix multiply and the NPB MG benchmark, we have shown that these OpenMP extensions provide a simple mechanism to target C, C++ and Fortran code to accelerators

## References

1. AMD: The AMD Fusion Family of APUs (March 2011)
2. Ayguad, E., Badia, R., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., Gonzlez, M., Igual, F., Jimnez-Gonzlez, D., Labarta, J., Martinell, L., Martorell, X., Mayo, R., Prez, J., Planas, J., Quintana-Ort, E.: Extending openmp to survive the heterogeneous multi-core era. International Journal of Parallel Programming 38, 440–459 (2010), `http://dx.doi.org/10.1007/s10766-010-0135-4`, 10.1007/s10766-010-0135-4
3. Bailey, D. H. and Barszcz, E. and Barton, J. T. and Browning, D. S. and Carter, R. L. and Dagum, L. and Fatoohi, R. A. and Frederickson, P. O. and Lasinski, T. A. and Schreiber, R. S. and Simon, H. D. and Venkatakrishnan, V. and Weeratunga, S. K.: The NAS parallel benchmarks. In: International Journal of High Performance Computing Applications. vol. 5, pp. 63–73 (1991)
4. CAPS: Hmpp (November 2010), `http://www.caps-entreprise.com`
5. Clearspeed: Support (November 2010), `{http://support.clearspeed.com}`
6. Han, T.D., Abdelrahman, T.S.: ¡i¿hi¡/i¿cuda: a high-level directive-based language for gpu programming. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. pp. 52–61. GPGPU-2, ACM, New York, NY, USA (2009), `http://doi.acm.org/10.1145/1513895.1513902`
7. Intel Corp.: Intel unveils new product plans for high-performance computing (March 2011)
8. Khronos Group: The OpenCL Specification, v. 1.1 (September 2010), `http://www.khronos.org/registry/cl/`
9. Lee, Seyong and Eigenmann, Rudolf: OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. SC '10, IEEE Computer Society (2010), `{http://dx.doi.org/10.1109/SC.2010.36}`
10. MCA: The Multicore Association (2011)
11. Nvidia Corp.: NVIDIA CUDA C Programming Guide, v. 3.2 (2010)
12. Nvidia Corp.: What is CUDA (February 2011), `{http://www.nvidia.com/object/what_is_cuda_new.html}`
13. OpenMP ARB: OpenMP Application Program Interface, v. 3.0 (May 2008), `http://openmp.org/wp/openmp-specifications`

14. PGI: Accelerator (November 2011)
15. PGI: Cuda fortran (March 2011), `{http://www.pgroup.com/resources/cudafortran.htm}`
16. Wang, Perry H. and Collins, Jamison D. and Chinya, Gautham N. and Jiang, Hong and Tian, Xinmin and Girkar, Milind and Yang, Nick Y. and Lueh, Guei-Yuan and Wang, Hong: EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. pp. 156–166. ACM (2007), `{http://doi.acm.org/10.1145/1250734.1250753}`

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.