A Source-to-Source OpenACC Compiler for CUDA

Akihiro Tabuchi[1], Masahiro Nakao[2], and Mitsuhisa Sato[1]

[1] Graduate School of Systems and Information Engineering, University of Tsukuba
[2] Center for Computational Sciences, University of Tsukuba

Abstract. OpenACC is a new directive-based programming interface for accelerators such as GPGPU. OpenACC allows the programmer to express the offloading of data and computations to accelerators to simplify the porting process for legacy CPU-based applications. In this paper, we present the design and implementation of an open-source OpenACC compiler that translates C code with OpenACC directives to C code with the CUDA API, which is the most widely used GPU programming environment provided for NVIDIA GPU. We adopt a source-to-source approach using the Omni compiler infrastructure for source code analysis and translations. Our approach leaves detailed machine-specific code optimization to the mature NVIDIA CUDA compiler. An experimental evaluation of the implementation shows that some parallel benchmark codes compiled by our compiler achieve speeds up to 31 times greater than those of CPUs, and that it is competitive with commercial implementations. However, the results also indicate the optimization of OpenACC programs has several problems, such as assigning iterations to GPU threads.

Keywords: OpenACC, GPU, compiler, CUDA.

1 Introduction

An accelerator such as a GPU is a promising device for further increasing computing performance in the field of high-performance computing. Several vendors are providing programming models for accelerators. For example, NVIDIA provides CUDA, which is an extension of C and C++ and provides GPU data and thread management functions, and OpenCL provides a portable programming model for various accelerators. Although CUDA is the most mature and extended approach to GPU programming, achieving a good performance rate usually requires a noticeable coding and optimization effort. To offload a fragment of code to an accelerator using OpenCL or CUDA, programmers need to rewrite a parallel loop to a kernel function executed on each thread of the GPU and to manage data transfers between host and device in an explicit way.

OpenACC [1] is a directive-based programming model that has been proposed as a solution to the complicated programming for offloading codes on accelerators. OpenACC provides a set of OpenMP-like loop directives for the

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 178–187, 2014.

programming of accelerators in an implicit and portable way. The OpenACC compiler generates codes that transfer data between host and device, and GPU code executed on the device.

In this paper, we present the design and implementation of an open-source OpenACC compiler that translates C code with OpenACC directives to that with CUDA API. This source-to-source approach means detailed machine-specific code optimization is left to the mature CUDA compiler by NVIDIA.

To implement our OpenACC compiler, we use the Omni compiler infrastructure [2] of C and Fortran95 for source code analysis and translations. The public availability of Omni means the research community is able to conduct experiments such as the study of new extensions of OpenACC and program analyses for optimizations. Although CUDA supports only NVIDIA devices, it would be easy to generate codes using the proposed OpenCL as a common programming interface for the heterogeneous devices supported by many manufacturers.

The contributions of this paper are summarized as follows:

- We describe a compiler to translate OpenACC programs into CUDA. The generated CUDA code is compiled by the NVIDIA CUDA compiler.
- Our approach enables the mature CUDA compiler to optimize the GPU code, sometimes better than a commercial compiler that generates the parallel thread execution (PTX) code.
- We present performance measurements to validate our approach by using some parallel benchmark codes.

As related work, efforts have been made to generate CUDA from OpenMP and the directive extension for GPUs. OMPCUDA [5] is the OpenMP compiler that translates to CUDA by using the Omni compiler. OpenMPC [6] also translates OpenMP to CUDA, but it has original directives that can tune performance. Additionally, the block size and other parameters can be tuned by a parameter search. hiCUDA [7] is a directive-based programming model for CUDA. In hiCUDA, the user has to specify the number of blocks and threads explicitly. It is more like CUDA, so it may obtain performance equal to that written by CUDA. OpenHMPP [8] can specify the data transfer between host and GPU and launch GPU kernels by original directives. OpenHMPP can use not only NVIDIA GPU but also AMD GPU by using CUDA and OpenCL as the backend. In addition, accULL [9] is an OpenACC compiler that uses CUDA and OpenCL as the backend. By using OpenCL, accULL can use accelerators supporting OpenCL. This work is similar to ours.

The rest of this paper is organized as follows. In Section 2, we provide an overview of the translation of OpenACC to CUDA with sample code. Section 3 describes the implementation of our OpenACC compiler and the details of code translation. We report our performance evaluation using some benchmark programs in Section 4 and conclude our work in Section 5.

```
1  #define N 1024
2  int main(){
3    int i, a[N], b[N];
4  #pragma acc data copyin(a) copyout(b)
5    {
6  #pragma acc parallel loop
7      for(i = 0; i < N; i++){
8        b[i] = a[i] + 1;
9      }
10   }
11 }
```

Fig. 1. Sample code (sample.c)

2 Translation of OpenACC into CUDA

2.1 OpenACC

OpenACC defines constructs by directives to specify the offloaded region of codes and data in standard C and Fortran programs. The offloaded regions are defined with either parallel or kernel directives, and are executed on an accelerator. Programs with OpenACC directives can be compiled into a hybrid code by OpenACC-compliant compilers.

In the offloaded region, a parallel loop is specified by the loop directive. OpenACC supports three-level parallelism: gang, worker, and vector. The parallel loop is executed in parallel by multiple workers, each of which can also have single instruction, multiple data (SIMD) operations. Similar to the thread blocks in CUDA, a group of workers is managed as a gang in OpenACC.

In the OpenACC model, the host and device memory spaces are independent. Therefore, data referenced in computations in the accelerator need to be transferred between host and device memory. The data transfers are implicitly executed by the compiler, but the programmer can specify them by data directives and reduce unnecessary transfers. The data directive specifies the data environment so that the coherency of the specified data is taken at the region boundary. In addition, the data directive defines data allocated on device memory. The data is copied from host to device at the beginning of the structured block or copied from device to host at the end of the structured block.

Fig. 1 shows the OpenACC sample code. First, line 4 allocates device memory by using the data construct. The values of array *a* are referenced but not updated in the device, and the values are specified as a *copyin* to transfer them from host memory to device memory at the beginning of the data region. The values of array *c* are not referenced but updated in the device, so it is specified as a *copyout* to transfer it from device memory to host memory at the end of the data region. Then, line 6 specifies the loop on *i* as a parallel region with parallelization of the loop by the *parallel loop* directive.

2.2 CUDA

CUDA is the programming environment for NVIDIA GPU, which is the most widely used for GPGPU. In CUDA, a kernel function is written in C and C++.

```
1  int main(void)
2  {
3    int i; int a[1024]; int b[1024];
4    {
5      void *_ACC_DEVICE_ADDR_a,*_ACC_HOST_DESC_a,*_ACC_DEVICE_ADDR_b,*_ACC_HOST_DESC_b;
6      _ACC_gpu_init_data(&(_ACC_HOST_DESC_a), &(_ACC_DEVICE_ADDR_a), a, ...);
7      _ACC_gpu_init_data(&(_ACC_HOST_DESC_b), &(_ACC_DEVICE_ADDR_b), b, ...);
8      _ACC_gpu_copy_data(_ACC_HOST_DESC_a, 400);
9      {
10     _ACC_GPU_FUNC_0(_ACC_DEVICE_ADDR_b, _ACC_DEVICE_ADDR_a);
11     }
12     _ACC_gpu_copy_data(_ACC_HOST_DESC_b, 401);
13     _ACC_gpu_finalize_data(_ACC_HOST_DESC_a);
14     _ACC_gpu_finalize_data(_ACC_HOST_DESC_b);
15   }
16 }
```

Fig. 2. Sample code (sample_tmp.c)

A kernel is a function callable from the host and executed on the CUDA device simultaneously by many threads in parallel. A thread is a minimum unit of execution. The threads are organized as a two-level hierarchy: block and grid. A block is a group of threads. A grid is a group of blocks that executes a kernel function. Each thread can be identified by the index of the block to which it belongs and the index of the thread in the block. By using these identifiers, a parallel loop is executed as a kernel function that computes each data.

Device memory is independent of host memory. Device kernels essentially can access only device memory, so device memory allocations and releases and data transfers between host and device are needed for executing kernels. CUDA provides a runtime library for managing device memory, and host programs call them as needed.

2.3 Translation to CUDA

The translation of sample code shown in the previous section is shown in Figs. 2 and 3. The OpenACC code is compiled into two parts. In sample_tmp.c, the data directive is converted to the block to set up the data transfer. The runtime call _ACC_gpu_init_data allocates device memory for arrays a and b, then the runtime call _ACC_gpu_copy_data transfers a from host to device memory at the beginning of the data region. The parallel region is compiled into the separate function _ACC_GPU_FUNC_0 generated by the compiler. _ACC_gpu_copy_data transfers b from device to host memory at the end of the data region. Finally, _ACC_gpu_finalize_data frees device memory for a and b.

The code sample.cu is a CUDA source code that includes a GPU kernel function and a function that launches it. Function _ACC_GPU_FUNC_0 calculates the number of thread blocks and threads from the number of loop iterations and launches the GPU kernel. After that, _ACC_GPU_M_BARRIER_KERNEL waits until the end of GPU kernel execution. Function _ACC_GPU_FUNC_0_DEVICE is a GPU kernel, from which each thread gets an induction variable value i from _ACC_gpu_calc_idx and calculates b[i]=a[i]+1.

```
__global__ static
void _ACC_GPU_FUNC_0_DEVICE(int b[1024], int a[1024])
{
  int i, idx, init, cond, step;
  _ACC_gpu_init_block_thread_x_iter(&init,&cond,&step,0,1024,1);

  for(idx=init;idx<cond;idx+=step){
    _ACC_gpu_calc_idx(idx,&i,0,1024,1);
    (b[i])=((a[i])+(1));
  }
}
extern "C"
void _ACC_GPU_FUNC_0(int b[1024],int a[1024])
{
  int _ACC_block_x=(((1023)/(256))+(1)), _ACC_block_y=(1), _ACC_block_z=(1);
  int _ACC_thread_x=(256), _ACC_thread_y=(1), _ACC_thread_z=(1);
  dim3 _ACC_DIM3_block(_ACC_block_x,_ACC_block_y,_ACC_block_z);
  dim3 _ACC_DIM3_thread(_ACC_thread_x,_ACC_thread_y,_ACC_thread_z);

  _ACC_GPU_FUNC_0_DEVICE<<<_ACC_DIM3_block,_ACC_DIM3_thread>>>(b,a);
  _ACC_GPU_M_BARRIER_KERNEL();
}
```

Fig. 3. Sample code (sample.cu)

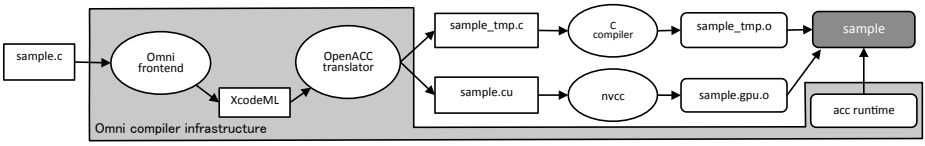


Fig. 4. Flow of compilation

3 Implementation of OpenACC Compiler

3.1 Omni Compiler Infrastructure

Our OpenACC compiler uses the Omni compiler infrastructure, which is a set of programs for a source-to-source compiler with code analysis and transformation. This compiler is also used to implement XcalableMP-dev [4], a GPU extension of XcalableMP [3] that shares a part of code with our OpenACC compiler.

Fig. 4 shows the flow of the compilation. When source code sample.c written in C and OpenACC is compiled, the Omni frontend translates it to an XcodeML file, which is intermediate code written in XML, and the OpenACC translator reads the XcodeML file and generates middle code sample_tmp.c and sample.cu.

Here, sample_tmp.c is host code compiled by the C compiler, and sample.cu is GPU code compiled by the NVIDIA CUDA compiler (NVCC). The compiled object files sample_tmp.o and sample.gpu.o are linked with the OpenACC runtime library. Then, the compiler generates an executable file.

3.2 Compilation of Parallel and Loop Constructs

For each region offloaded by parallel constructs, a function is generated to set up the dimension of thread blocks and to launch the kernel function to a GPU.

```
#pragma acc parallel num_gangs(4) vector_length(128)
{
  /* codes in parallel region */
}
```

(a) ACC parallel

```
__global__ static
void _ACC_GPU_FUNC_0_DEVICE(...)
{
  /* codes in parallel region */
}
extern "C"
void _ACC_GPU_FUNC_0(...)
{
  int _ACC_block_x=(4), _ACC_block_y=(1), _ACC_block_z=(1);
  int _ACC_thread_x=(128), _ACC_thread_y=(1), _ACC_thread_z=(1);
  dim3 _ACC_DIM3_block(_ACC_block_x,_ACC_block_y,_ACC_block_z);
  dim3 _ACC_DIM3_thread(_ACC_thread_x,_ACC_thread_y,_ACC_thread_z);

  _ACC_GPU_FUNC_0_DEVICE<<<_ACC_DIM3_block,_ACC_DIM3_thread>>>(...);
  _ACC_GPU_M_BARRIER_KERNEL();
}
```

(b) translated code

Fig. 5. Code translation of parallel construct

OpenACC supports three-level parallelism: gang, worker, and vector. However, only the block and thread models are provided in CUDA. In our compiler, gang and vector correspond to block and thread, respectively. This means that a gang has a worker, and a worker can use vector operations. The number of blocks and threads can be specified by the *num_gangs* and *vector_length* clauses. If the number of threads is not specified, the number of threads is 256, which is often used for Fermi architecture GPUs, and the number of blocks is determined by the number of loop iterations contained in the region.

When a loop is specified as parallelism by a loop construct, the kernel function executes that loop in parallel. The function executes a part of the loop by _ACC_gpu_init_block_thread_x_iter and calculates the loop index by the inline function _ACC_calc_idx. Additionally, our compiler supports two- or three-dimensional blocking for nested loops. If multiple gang or vector clauses are specified for nested loops, these loops are executed by two- or three-dimensional grids or blocks in CUDA, respectively. This execution may increase data locality in a block and the performance of loop execution may improve. At the moment, our compiler does not allow function calls inside the parallel region except for mathematical functions supported in CUDA.

Fig. 5 shows an example of the parallel construct and the translated code. This parallel construct specifies that the next structured block is executed by 4 gangs and 128 vector operations on the accelerator. In the translated code, the number of blocks and threads are calculated, and the device kernel is launched with the configuration. After that, the host waits for the finish of the device kernel.

Fig. 6 shows an example of the loop construct and the translated code. This loop construct specifies the following loop executed by vector operation. In the translated code, the inline function _ACC_gpu_init_thread_x_iter calculates the range of the loop for the thread. For our loop iteration, the inline function

```
/* inside parallel region */
#pragma acc loop vector
for(i = 0; i < N; i++){
  a[i]++;
}
```

(a) ACC loop

```
/* inside gpu kernel function */
int i, _ACC_idx, _ACC_init, _ACC_cond, _ACC_step;
_ACC_gpu_init_thread_x_iter(&_ACC_init, &_ACC_cond, &_ACC_step,0,N,1);

for(_ACC_idx=_ACC_init;_ACC_idx<_ACC_cond;_ACC_idx+=_ACC_step){
  _ACC_gpu_calc_idx(_ACC_idx,&i,0,N,1);
  a[i]++;
}
```

(b) translated code

Fig. 6. Code translation of loop construct

_ACC_gpu_calc_idx calculates the loop index and executes the increment of the array a element.

3.3 Compilation of Data Construct

To translate a region with a data construct, codes for device memory allocations and data transfers from host to device are inserted at the beginning, and codes for device memory releases and data transfers from device to host are inserted at the end. The runtime function _ACC_gpu_init_data allocates device memory for a host variable or an array. The pointer _ACC_DEVICE_ADDR_*name* is the address of device memory for the *name*, and the pointer _ACC_HOST_DESC_*name* is the address of the structure that contains the host address, device address, and size of that variable. Data is transferred between host and device memory by the runtime function _ACC_gpu_copy_data. The direction of transfer is given by the second argument. At the end of the region, device memory is freed by the runtime function _ACC_gpu_finalize_data.

Fig. 7 shows an example of the data construct and the translated code. This data construct specifies that arrays a and variable b are allocated on the device at the beginning of the region and freed at the end of the region. Array a in the copy clause is transferred at the beginning and end of the region, and variable b in the copyout clause is transferred at the end of the region.

4 Performance Evaluation

For the performance evaluation, we used the following benchmark programs: Matrix multiplication (MatMul), N-body problem, and CG benchmark in the NAS Parallel Benchmarks (NPB). The Matrix multiplication and the N-body problem are naive implementations. We measured the execution time of each program compiled by our compiler. The execution time includes the time of data movement between the host and GPU and excludes the GPU initialization

```
int a[100], b;
#pragma acc data copy(a) copyout(b)
{
  /* some codes using a and b */
}
```

(a) ACC data

```
int a[100], b;
{
  void *_ACC_DEVICE_ADDR_a,*_ACC_HOST_DESC_a,*_ACC_DEVICE_ADDR_b,*_ACC_HOST_DESC_b;
  _ACC_gpu_init_data(&(_ACC_HOST_DESC_a),&(_ACC_DEVICE_ADDR_a),a,(100)*sizeof(int));
  _ACC_gpu_init_data(&(_ACC_HOST_DESC_b),&(_ACC_DEVICE_ADDR_b),&(b),sizeof(int));
  _ACC_gpu_copy_data(_ACC_HOST_DESC_a, 400); /* 400 means host -> device */
  {
    /* some codes using a and b */
  }
  _ACC_gpu_copy_data(_ACC_HOST_DESC_a, 401); /* 401 means device -> host */
  _ACC_gpu_copy_data(_ACC_HOST_DESC_b, 401);
  _ACC_gpu_finalize_data(_ACC_HOST_DESC_a);
  _ACC_gpu_finalize_data(_ACC_HOST_DESC_b);
}
```

(b) translated code

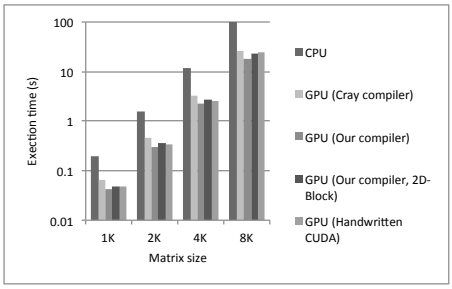
Fig. 7. Code translation of data construct

time. We ran the codes on one node of a CrayXK6m-200. The CPU of the Cray is AMD Opteron Processor 6272 and its GPU is NVIDIA X2090 (for MatMul and N-body) or NVIDIA K20 (for CG). We used the Cray compiler 8.1.0.143 (for MatMul and N-body) or 8.1.6 (for CG) with the flag *-O3*.

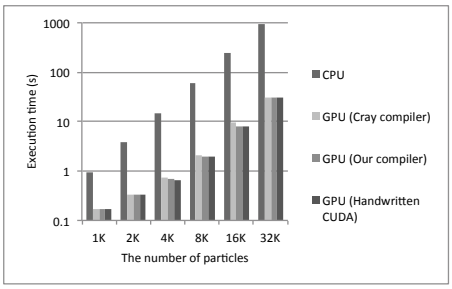
Fig. 8(a) and 8(b) shows the execution time of the Matrix multiplication and the N-body program, respectively. Fig. 8(c) shows the performance of the CG benchmark in Mops/s. In these results, "CPU" indicates the results of the CPU (single core) for code compiled by the Cray compiler, and "GPU (Cray compiler)", "GPU (Our compiler)" and "GPU (Handwritten CUDA)" indicate the results of the Cray OpenACC compiler, our compiler, and the handwritten CUDA code, respectively. Note that handwritten code does not use the shared memory of the GPU. For the Matrix multiplication, the result by 2D blocking of our compiler is also shown.

For the Matrix multiplication, the code by our compiler is 4.6–5.5 times faster than that of the CPU single core, and 1.4–1.5 times faster than that compiled by the Cray compiler. The reason for this difference is the quality of PTX code generated by the Cray compiler, since the Cray compiler generates the PTX code directly, not via CUDA. We found that the PTX code generated by the Cray compiler executes more operations in the innermost loop. Our compiler leaves PTX code generation to NVCC so that we can make use of mature NVIDIA compiler technology for their architecture.

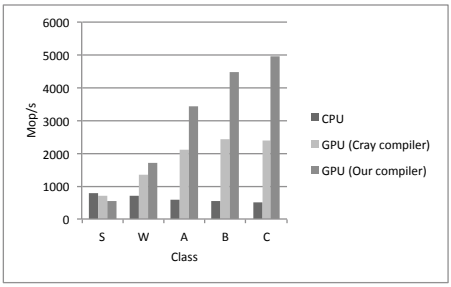
The performance of our compiler using 2D blocking is slightly lower than that of non-2D blocking code because the default 2D block size (16x16 in this case) is not adequate for this program. The handwritten CUDA code also uses a 2D block (16x16). We measured the performance in various 2D block sizes, and the best result was 512x2. In general, it is difficult to know the optimal block size without executing in various block sizes.



(a) Matrix multiplication



(b) N-body



(c) CG benchmark

Fig. 8. Performance comparison using benchmarks

For N-body, our compiler achieves a speedup of 5.4–31 times greater than the CPU single core and 0.95–1.2 times greater than the Cray compiler. The speedup became larger as the problem size increased and the intensity of calculations increased. At the small problem size, the performance of our compiler is lower than that of the Cray compiler because our compiler used fewer Streaming Multiprocessors (SMs). The GPU kernel is executed by the SMs per thread block. If the number of blocks is smaller than that of the SMs, the performance of the GPU kernel becomes low. The default block size of our compiler is 256 threads and that of the Cray compiler is 128 threads. When the problem size is small, the number of launched blocks is too small and fewer SMs are used in comparison with the Cray compiler. Our compiler needs to be improved to determine the block size to avoid choosing too small a number of blocks.

For CG, our compiler achieves a speedup of 0.66–9.7 times over the CPU single core and 0.74–2.1 times over the Cray compiler. For class S in the CG benchmark, the performance of the GPU is lower than that of the CPU because overheads, which include launching kernel functions, synchronization with the device and data transfers, are large compared with the execution time of kernel functions. Additionally, the overhead of reduction in our compiler is large because a reduction kernel uses a temporal array allocated before and freed after the kernel execution. However, the performance of our compiler is higher than that of the Cray compiler in the larger classes, W–C, so the performance of the GPU kernel is better than that of the Cray compiler.

5 Conclusion

In this paper, we presented the design and implementation of a source-to-source OpenACC compiler. The compiler translates C code with OpenACC directives to that with CUDA API. Our compiler implementation uses the Omni compiler infrastructure for analyzing and translating code.

We measured our compiler's performance for Matrix multiplication, N-body, and CG in the NAS parallel benchmarks, and compared the performance with that of a commercial compiler and handwritten CUDA. In most cases, the performance of the GPU program using our compiler is higher than that of the CPU. We observed a speedup of up to 31 times over the CPU single core for the N-body problem. Our compiler makes use of the CUDA backend successfully by the source-to-source approach, and the performance of our compiler is often better than that of the Cray compiler. On the other hand, we found that the grid size and block size are unsuitable for some programs and overheads are larger than that of the Cray compiler.

We are currently working on optimization of the tuning block size at compile time and the reduction of overheads. Our compiler will support the full set of directives for conforming to the OpenACC specification, and we will compare the performance of our compiler against that of PGI and accULL.

References

1. "OpenACC Home", `http://www.openacc-standard.org/`
2. "Omni Compiler Project", `http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/`
3. "XcalableMP", `http://www.xcalablemp.org`
4. Lee, J., Tran, M.T., Odajima, T., Boku, T., Sato, M.: An Extension of XcalableMP PGAS Lanaguage for Multi-node GPU Clusters. In: Alexander, M., D'Ambra, P., Belloum, A., Bosilca, G., Cannataro, M., Danelutto, M., Di Martino, B., Gerndt, M., Jeannot, E., Namyst, R., Roman, J., Scott, S.L., Traff, J.L., Vallée, G., Weidendorfer, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 7155, pp. 429–439. Springer, Heidelberg (2012)
5. Ohshima, S., Hirasawa, S., Honda, H.: OMPCUDA: OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 161–173. Springer, Heidelberg (2010)
6. Lee, S., Eigenmann, R.: 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, SC 2010, pp. 1–11 (2010)
7. Han, T.D., Abdelrahman, T.S.: hiCUDA: a High-level Directive-based Language for GPU Programming. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2), pp. 52–61 (2009)
8. "OpenHMPP", `http://www.caps-entreprise.com/openhmpp-directives/`
9. Reyes, R., López-Rodríguez, I., Fumero, J.J., de Sande, F.: `accULL`: An OpenACC Implementation with CUDA and OpenCL Support. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 871–882. Springer, Heidelberg (2012)