Hands on with OpenMP4.5 and Unified Memory: Developing Applications for IBM's Hybrid CPU + GPU Systems (Part II)

Leopold Grinberg[1](✉), Carlo Bertolli[1], and Riyaz Haque[2]

[1] IBM Research, Yorktown Heights, USA
{leopoldgrinberg,cbertol}@us.ibm.com
[2] LLNL, Livermore, USA
haque1@llnl.gov

Abstract. Integration of multiple types of compute elements and memories in a single system requires proper support at a system-software level including operating system (OS), compilers, drivers, etc. The OS helps in scheduling work on different compute elements and manages memory operations in multiple memory pools including page migration. Compilers and programming languages provide tools for taking advantage of advanced architectural features. In this paper we encourage code developers to work with experimental versions of compilers and OpenMP standard extensions designed for hybrid OpenPOWER nodes. Specifically, we focus on nested parallelism and Unified Memory as key elements for efficient system-wide programming of CPU and GPU resources of OpenPOWER. We give implementation details using code samples and we discuss limitations of the presented approaches.

Keywords: OpenPOWER · HPC · Offloading · Directive based programming · Nested parallelism

1 Introduction

Programming applications for specific hardware components as well as taking advantage of specific system software typically have a two-fold effect: (a) achieving higher performance and productivity on a given class of systems; and (b) adversely affecting the application portability and/or performance portability to other systems. In addition to these considerations, taking advantage of hardware and system software innovations available in a subset of emerging systems sets a tone and directions for developing future systems for High Performance Computing and Analytics. It also fuels advances in language features and standard evolution.

In the first part of this two-part paper (*Hands on with OpenMP4.5 and Unified Memory: Developing applications for IBM's hybrid CPU + GPU systems (Part I)* [3] we discussed how node memory and application data can be managed

B.R. de Supinski et al. (Eds.): IWOMP 2017, LNCS 10468, pp. 17–29, 2017.
DOI: 10.1007/978-3-319-65578-9_2

using OpenMP4.5 directives. In this *Part II* we introduce methodologies taking advantage of hardware and software features which are more advanced and in part not fully supported by the OpenMP4.5 standard. Specifically, we will discuss three advanced topics: nested parallelism, use of Unified Memory and use of GPU's on-chip memory.

Our scope here is limited to programming IBM's system containing multiple POWER®CPUs and NVIDIA®GPUs with a directive based programming model. Here we employ the OpenMP4.5 standard [5] and IBM®extensions to the standard (supported in the open source CLANG®and IBM's proprietary XL®compilers) to program CPUs (*host*) and GPUs (*device*) and manage on-node memories. IBM's current hybrid CPU-GPU nodes, such as two-socket Minsky®nodes containing two ten-core POWER8®CPUs and four P-100 GPUs interconnected with NVLink 1.0 provide many opportunities for nested parallelism and concurrent execution on all compute elements. These nodes also support Unified Memory (UM) that provides a pool of memory accessible on the CPU and the GPU using a single pointer. To take advantage of UM at the present time, we rely on interoperability between OpenMP4.5 and CUDA®, and use of CUDA Managed Memory [7,8]. Use of UM substantially simplifies managing application data on heterogeneous systems. However, whereas the OpenMP4.5 standard encompasses UM support, current implementations do not support it. Pointers to buffers allocated using CUDA Managed Memory can be treated as valid device pointers inside OpenMP4.5 `target` regions, but the OpenMP compiler and runtime implementations considered in this paper do not support the concept of replacing the explicit data transfers between the *host* and *device* with features provided by the UM. Consequently, porting codes based on UM and OpenMP4.5 to systems not supporting UM may require some adaptations.

This paper makes the following contributions:

- In Sect. 2 we describe a scheme allowing nested parallelism and simultaneous execution of codes on *host* and *devices* using OpenMP4.5 directives.
- In Sect. 3 we show how architecture-specific memory support can be integrated in the codes programmed with OpenMP4.5. Specifically, we present an example making use of a section of the GPU's L1-cache which can be explicitly managed by compilers in order to host application data.
- In Sect. 4 we describe ways to develop applications using OpenMP4.5 directives and UM on systems with adequate hardware and software support. We also discuss the advantages and limitations of this approach.

To our knowledge, this is the first paper that exposes the integration of advanced system-software and hardware features in codes programmed using OpenMP4.5. Use of Unified Memory in conjunction with directive-based programming of NVIDIA GPUs is not new. For example, the PGI compiler supporting OpenACC [6] can intercept all calls to host memory allocation/de-allocation, replacing them with appropriate calls to the UM interface and rendering all data mapping operations as no-ops. UM support is an optional feature of the PGI compiler and is enabled through a compiler option. Use of UM within the

Kokkos programming framework has also been reported in [2]. Unlike these techniques, in this paper we show how programmers can make explicit use of the UM interface for memory management and still write correct code using OpenMP4.5 device constructs.

2 Concurrent Executions on CPUs and GPUs via OpenMP Nested Regions

Multiple compute resources in IBM's hybrid CPU + GPU nodes offer a range of choices for execution policies. For example a single MPI task can perform operations in parallel (using OpenMP) on the CPU cores (a model that has been widely adopted on the multicore CPUs) and it can also offload work to one or more GPUs. Each GPU can concurrently (or sequentially) support offloading work from a number of MPI tasks. In another scenario, a subset of OpenMP threads running on the *host* can offload work to one or more *devices* concurrently, while another subset of OpenMP threads can start nested parallel regions on the *host*.

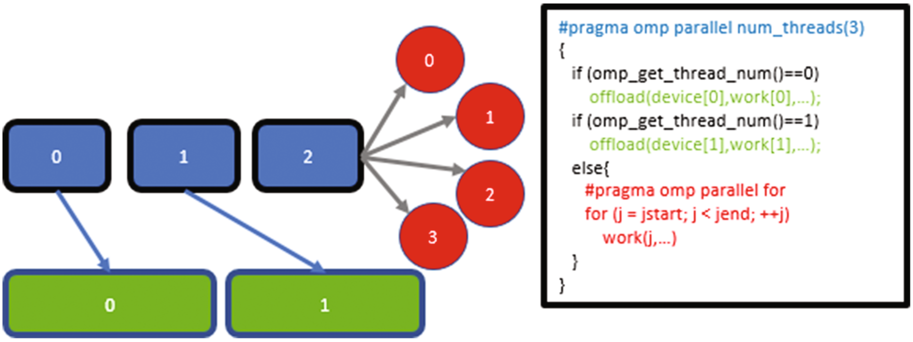
2.1 Parallel Regions on *device*: Correspondence Between CUDA and OpenMP4.5

Before diving into the topic of nested parallelism we would first like to explain the correspondance between expressing parallelism using CUDA and using OpenMP4.5 directives. The OpenMP4.5 implementation on GPU maps parallelism abstractions which are exposed to OpenMP users, to lower-level GPU programming mechanisms. `target` regions are compiled into PTX (or GPU) kernels when NVIDIA GPUs are selected as OpenMP device type. The OpenMP runtime will invoke the kernels when encountering a `target` pragma. When a `target` region contains a `teams` region, the GPU kernel is started with multiple CUDA threadblocks and threads. Each OpenMP team is mapped to a single CUDA threadblock and two teams cannot be mapped to the same threadblock. OpenMP threads within each team are mapped to CUDA threads (one OpenMP thread is one CUDA thread). When `target` does not contain a `teams` construct, only one threadblock is started. The execution of a team (or threadblock) inside a `target` and outside of `parallel` regions is sequential - a single thread (team master) within each threadblock executes the region and all other threads are waiting to be recruited for parallel work. When a `parallel` region is encountered by the team master, all necessary threads within each threadblock are activated and participate in the parallel region.

Control of CUDA grid and threadblock sizes is critical to performance tuning in GPU kernels, whenever the OpenMP runtime chosen default values can be improved. Control is exposed at the OpenMP level through clauses of the `teams` construct. `num_teams` can be used to instruct the OpenMP runtime to start a specific number of teams (threadblocks). `thread_limit` tells the OpenMP runtime not to start more than the specified number of threads. To limit the amount of threads to be recruited to execute a parallel region, users can employ the `num_threads` clause of `parallel`. Note that the OpenMP4.5 constructs `num_teams`, `thread_limit` and `num_threads` are valid on both *host* and *device*. In the following section we will make use of those constructs for execution on a *host* and on a *device*.

2.2 OpenMP4.5 and Nested Parallelism Across a Node

In this section we discuss how a certain work load can be subdivided and executed concurrently across CPU and GPU threads using all the compute resources of a node. For this purpose we will use a schematic illustration provided in Fig. 1. Here a parallel region with three OpenMP threads is created on the *host*, and threads with IDs 0 and 1 will offload work to the *devices* 0 and 1 correspondingly, while the third thread will create an inner parallel region of 4 threads on the *host*.



```
#pragma omp parallel num_threads(3)
{
  if (omp_get_thread_num()==0)
     offload(device[0],work[0],...);
  if (omp_get_thread_num()==1)
     offload(device[1],work[1],...);
  else{
    #pragma omp parallel for
    for (j = jstart; j < jend; ++j)
       work(j,...)
  }
}
```

Fig. 1. Nested parallel regions with concurrent execution on *host* and *devices*. Outer parallel region contains three CPU threads. CPU threads 0 and 1 launch kernels on devices 0 and 1 correspondingly, while CPU thread 2 creates a parallel region with four CPU threads on a *host*.

A more detailed and specific example is provided in Fig. 2. In this example we first enable nested parallelism by calling OpenMP API `omp_set_nested(1)` (line 9) and then acquire the number of visible devices (`num_devices`) by calling the OpenMP API `omp_get_num_devices()` (line 11). In the next step a parallel region with up to `num_devices`+1 threads is created on the *host*. The first `num_devices` iterations of the main for loop will offload work to the *devices* with IDs 0, ... , `num_devices`-1, and in the last iteration a parallel region will be created on the *host* and the remaining work will be executed in parallel using at most (MAX(1,`omp_get_max_threads()-num_devices`)) threads. In this example we require that 90% of the work be executed on the *devices*, while the remaining work be executed on the *host*. In general, work distribution between *host* and *device(s)* may be determined (at run time) by taking into account the *host* and *device* hardware characteristics (e.g. ratio of *device*/*host* memory bandwidth, FLOP rate, etc.), expected execution time and even the availability of *device* memory.

```
int main(){
  double *x, *y;
  int num_devices, i, chunk, j_start, N=1024*1024*10;
  double DEVICE_FRACTION = 0;
  bool USE_DEVICE;
  x = (double *) malloc(N*sizeof(double));
  y = (double *) malloc(N*sizeof(double));
  //enable nested parallel regions
  omp_set_nested(1);
  //get number of devices
  num_devices = omp_get_num_devices();
  //90% of work done on device(s)
  if (num_devices > 0)   DEVICE_FRACTION = 0.9;
  #pragma omp parallel for num_threads(num_devices+1) \
   private(chunk,j_start,USE_DEVICE)
  for (i = 0; i < (num_devices+1); ++i){
    //divide work, set default device
    if (i < num_devices){ //use device
      omp_set_default_device(i);
      chunk = DEVICE_FRACTION*N / num_devices;
      j_start = chunk*i;
      USE_DEVICE = true;
      printf("using DEVICE No %d, j_start = %d, chunk = %d\n",i,j_start,chunk);
    }
    else {    // use host
      chunk = N;  //default
      j_start = 0; //default
      USE_DEVICE = false;
      if (num_devices > 0){
        j_start = (DEVICE_FRACTION*N / num_devices) * num_devices;
            chunk = N - j_start;
      }
      printf("using HOST:  j_start = %d, chunk = %d\n",j_start,chunk);
    }
    initialize_x_and_y(x+j_start,y+j_start,chunk,j_start,USE_DEVICE);
  }
  free(x);   free(y);
  return 0;
}

void initialize_x_and_y(double *x, double *y, int N, int offset, bool USE_DEVICE)
{
  #pragma omp target  map(from:x[0:N],y[0:N]) if(USE_DEVICE)
  #pragma omp  teams distribute parallel for  if(target:USE_DEVICE)
  for (int i=0; i<N; ++i){
    x[i] = (offset+i)*0.001;
    y[i] = (offset+i)*0.003;
    if ( (!USE_DEVICE) && (i == 0))
    printf("num_threads = %d, num_teams=%d\n",omp_get_num_threads(),omp_get_num_teams());
  }
}
```

Fig. 2. Nested parallelism: concurrent execution on *host* and *devices*

```
export OMP_NUM_THREADS=20
export OMP_PLACES={0;20;8}
./a.out
using DEVICE No 0, j_start = 0, chunk = 2359296
using DEVICE No 1, j_start = 2359296, chunk = 2359296
using DEVICE No 2, j_start = 4718592, chunk = 2359296
using DEVICE No 3, j_start = 7077888, chunk = 2359296
device: CPU:  j_start = 9437184, chunk = 1048576
num_threads = 1, num_teams=16
```

Fig. 3. Concurrent execution on *host* and *devices*; nested parallelism: output of code from Fig. 2.

3 Clang's Extension for OpenMP4.5 for *device* On-chip Memory Allocation

NVIDIA GPUs allow developers to take advantage of allocating relatively small buffers in an "on-chip memory", also referred to as the *shared memory* in CUDA terminology. While there are multiple reasons for using shared memory, here we skip the discussion on use cases and refer readers to NVIDIA's programming guide [4] and NVIDIA's devblog describing using shared memory [1].

OpenMP4.5 standard does not provide developers with the means of specifically taking an advantage of the GPU's shared memory. However, IBM's extension to the OpenMP4.5 specification implemented for the Clang supports the use of shared memory. It is expected that future versions of IBM's XL compiler will also support shared memory for NVIDIA GPUs. Furthermore, OpenMP is also evolving towards incorporating special memory types as first-class citizens in the standard.

In this section we illustrate (see Fig. 4) use of shared memory in a matrix-transposition code that uses OpenMP4.5 directives. Currently, in order to allow compiler to allocate buffers in the GPU's shared memory, developers should use static memory allocation and place the corresponding code after the directive `#pragma omp target teams` but before the directive `#pragma omp distribute` (see Fig. 4, line 29). If the compiler determines that the size of the requested buffer (`VAL[BLK_SZ][BLK_SZ+1]`) is small enough to fit into the GPU's shared memory it places it there; otherwise the buffer is allocated in the global *device* memory. Note the use of the `if` clause in the code presented in Fig. 4: setting the value of the variable `USE_DEVICE` to 1 or to 0 results in code execution on the *device* or on the *host* respectively. Whether the target region is executed on a *device* or on a *host*, the buffer `VAL[BLK_SZ][BLK_SZ+1]` is designated as `team`-private, which eliminates race conditions between different teams. On the GPU *device* each `team` will be mapped to a different CUDA threadblock, and on *host* teams will be mapped to CPU threads.

At this stage of compiler development, IBM's implementation limits the size of the GPU's shared memory available to application's data to 800 bytes per team, and consequently we set `BLK_SZ=8`. In tests performed on IBM's Minsky nodes with offloading the matrix transposition to the P-100 GPU we observe effective memory BW utilization of 243 GB/s, while the achievable memory BW is in the 480–500 GB/s range. A simple (two-loop) kernel for matrix transposition not using shared memory achieves only 83 GB/s, which is expected due to non-coalesced memory access.

4 Use of Unified Memory and OpenMP4.5 Directives

The OpenMP4.5 memory model is based on the notion of heterogeneous memory address spaces (*host* and *device*) with directives for explicitly managing data movement and coherence between them. Under this model, coding is complicated by two factors. First, using OpenMP4.5 directives correctly in the presence of class member pointers is non-trivial and may involve considerable code changes to work (as illustrated in the first part of this paper). Secondly, explicitly managing coherence between two address spaces can be highly error-prone except in the simplest of cases.

Starting with the OpenMP4.5 standard, using native memory management mechanism (e.g. CUDA memory allocators) is also supported by special clauses to enable architecture-specific data allocation. For example, pointers to memory allocated using `cudaHostAlloc`, `cudaMallocHost`, `cudaMallocManaged` and `cudaMalloc` can now be used inside OpenMP4.5 `target` regions. Here we focus on the use of CUDA Managed Memory, and specifically on eliminating the need for explicit data transfers between the *host* and *devices*. Currently implicit data transfer between *host* and *devices* is not supported by the OpenMP standard, and methodology required for such a support is a considered as a research topic.

Employing CUDA Managed Memory substantially reduces the complexity of managing deep copies and also resolves the coherency issues. This is achieved by allocating data in a Unified Memory space [7,8] which is accessible on both the *host* and *device* using a single pointer.

Memory buffers associated with the Managed Memory automatically migrate between the *host* and *device* when a memory fault is encountered. The exact mechanism responsible for buffer migration is outside the scope of this paper. In this section we illustrate how to work with arrays, classes and common data structures like `std::vector` using UM and OpenMP4.5 directives. Considering that the OpenMP4.5 standard has been designed to also work with *devices* not supporting UM, we also discuss concerns with the integration of UM and OpenMP4.5 from the standpoint of code portability.

It is also important to note that for correct behavior of a code mixing OpenMP directives and CUDA API, especially on nodes with multiple visible *devices*, setting default *device* must be done twice: once using the OpenMP4.5 API `omp_set_default_device (device_ID)` and then using the CUDA API `cudaSetDevice (device_ID)`.

4.1 Eliminating Explicit Deep Copies

In Fig. 5, we consider a UM-based version of code described in the first part of this paper.

```
#define MIN(a,b) (a < b ? a : b)
#define BLK_SZ 32
int main(){

  omp_set_default_device(0);

  int Nr = 1024*8, Nc = 1024*8;
  double *U = new double[Nr*Nc];
  double *UT = new double[Nr*Nc];
  bool USE_DEVICE=1;

  //allocate U and UT in device memory
  #pragma omp target enter data map(alloc:U[0:Nr*Nc],UT[0:Nr*Nc]) if(target:USE_DEVICE)

  //initialize U
  #pragma omp target teams distribute thread_limit(512) if(target:USE_DEVICE)
  for (auto col = 0; col < Nc; ++col){
    #pragma omp parallel for if(USE_DEVICE)
    for (auto row = 0; row < Nr; ++row){
      U[row*Nc+col] = row*0.001 + col*0.0003;
    }
  }
  int nteams = (Nr*Nc + BLK_SZ*BLK_SZ - 1)/(BLK_SZ*BLK_SZ);
  int  nthreads = BLK_SZ;
  #pragma omp target teams num_teams(nteams) thread_limit(nthreads) if(target:USE_DEVICE)
  {
    // sufficiently small array VAL will be allocated in GPU's shared memory
    // otherwise in device memory
    double VAL[BLK_SZ][BLK_SZ+1];

    #pragma omp distribute collapse(2)
    for (auto rstart = 0; rstart < Nr; rstart += BLK_SZ){
      for (auto cstart = 0; cstart < Nc; cstart += BLK_SZ){

        auto rend = MIN(Nr, rstart+BLK_SZ);
        auto cend = MIN(Nc, cstart+BLK_SZ);

        //fill in temporary buffer (shared memory)
        #pragma omp parallel if(USE_DEVICE)
        {
          #pragma omp for collapse(2)
          for (auto row=rstart; row < rend; ++row){
            for (auto col=cstart; col < cend; ++col)
               VAL[row-rstart][col-cstart] = U[row*Nc + col];
          }
          //transpose and write data from shared memory to device memory
          #pragma omp for collapse(2)
          for (auto row=cstart; row < cend; ++row){
            for (auto col=rstart; col < rend; ++col)
               UT[row*Nr + col] =  VAL[col-rstart][row-cstart];
          }
        }
      }
    }
  }

  // copy data from the device to host memory and deallocate device memory
  #pragma omp target exit data map(from:U[0:Nr*Nc],UT[0:Nr*Nc]) if(USE_DEVICE)
}
```

Fig. 4. Code illustrating use of NVIDIA's GPU shared memory and OpenMP4.5 directives. Currently only IBM's extensions to OpenMP4.5 spec implemented in CLANG compiler allow use of GPU's shared memory.

```
 1 struct A {
 2   int* y;
 3   int size;
 4   A(const int* y_, const long size_) : y(y_), size(size_) {}
 5 };
 6
 7 int n = 100;
 8 int* y; cudaMallocManaged(&y, n*sizeof(int)); // Allocate in UM
 9 A* a = new A(y,n);
10
11 // Only map object a to the device using the map clause
12 #pragma omp target map(to:a[0:1])
13 {
14   // OK because a->y holds the unified address
15   a->y[3] += ...;
16 }
```

Fig. 5. Deep copy of a data structure using Managed Memory

Let us start by comparing this example to the codes presented in [3] (Figs. 7 and 9). First, the call to `malloc` on line 8 (Fig. 7 of [3]) is replaced with `cudaMallocManaged`. Second, the operation `map(to:y[0:n])` on line 12 (Fig. 7 of [3]) has been removed since UM automatically moves data between the two address spaces. Most importantly, compared to the version of this example in Fig. 7 in [3], our UM-based example in Fig. 5 works correctly. This is because being allocated in Managed Memory, the *host* address referred by `a.y` is valid on the *device* as well. This eliminates the need to update `a`'s *device* copy with the correct address. Note however, that since object `a` itself is not UM-allocated, it is still required to map it before use inside the target region (line 12, Fig. 5). In the next Sect. 4.2 we show how to allocate objects like `a` in Managed Memory.

4.2 Mapping Classes Using UM

For mapping a class using UM, we follow the approach described in [7]. We first define a class that overrides the `new` and `delete` operators as shown in Fig. 6.

Second, we further modify the code presented in Fig. 5 to make class `A` UM-allocated as shown in Fig. 7. In this example, we extend class `A` with the class `UMMapper` overriding the former's default `new` and `delete` operators with the latter's. With this change, object `a` is now allocated in UM (line 9, Fig. 7). Third, we correspondingly replace the `map` clause `map(to:a[0:1])` with the `is_device_ptr(a)` clause in Fig. 7 (line 12). Since `a` is allocated in the UM the `map` clause is not required; at the same time, however, it is necessary to inform the OpenMP4.5 runtime that `a` is a valid *device* pointer. If that is not done, the OpenMP4.5 runtime will attempt (and fail) to find the device mapping for `a`. Therefore the `is_device_ptr` clause is critical for correct execution. Note that the `is_device_ptr` clause is not required for the member pointer `a.y`; member pointers are simply moved to the *device* as part of enclosing object and no attempt is made to find their device address. If however, the pointer `y` is used directly inside a target region, that region would have to be predicated with a `is_device_ptr(y)`

```
class UMMapper {
  public:
    void* operator new(size_t len) {
      void* ptr; cudaMallocManaged(&ptr, len); return ptr;
    }
    void* operator new[](size_t len) {
      void* ptr; cudaMallocManaged(&ptr, len); return ptr;
    }
    void operator delete(void* ptr) noexcept (true) {
      cudaFree(ptr);
    }
    void operator delete[](void* ptr) noexcept (true) {
      cudaFree(ptr);
    }
};
```

Fig. 6. Overriding `new` and `delete` operators: objects derived from UMMapper will be allocated using Managed Memory

```
struct A : public UMMapper {
  int* y;
  int size;
  A(const int* y_, const long size_) : y(y_), size(size_)  {}
};

int n = 100;
int* y; cudaMallocManaged(&y, n*sizeof(int)); // Allocate y using UM
A* a = new A(y,n);

// "a" is a valid device pointer
#pragma omp target is_device_ptr(a)
{
  // OK because a->y holds the unified address
  a->y[3] += ...;
}
```

Fig. 7. Using unified memory: accessing class object and its members on *host* and *device*

clause. Note that this strategy for creating UM-based classes does not work for objects allocated outside the `new` operator, e.g. stack-allocated objects.

4.3 Working with std::vector, UM and OpenMP4.5

In this section let us consider a code section using `std::vector` (Fig. 8). Here offloading the two code loops (lines 5 and 9, Fig. 8) to the *device* would require mapping the vectors `x` and `y` to the *device* memory and deep-copying their data; something not possible using OpenMP4.5 directives alone. A way to overcome this limitation and to allow the use of `std::vector` inside target regions executed on a *device*, is to allocate the data for these vectors using UM and avoiding the deep-copy altogether. The `std::vector` can be made UM-based by specializing its memory allocator to use Managed Memory [7] as shown in Fig. 9. Accordingly, we modify the example in Fig. 8 by specializing the allocators for vectors `x` and `y` to use the `UMAllocator` as shown in Fig. 10.

The class `UMAllocator` ensures that the vector data is allocated in Managed Memory and that the data will be migrated between the *host* and *devices* upon

```
double alpha = 2.0;
int N=1024*1024*10;
vector<double> x(N);
vector<double> y(N);
for (int i = 0; i < N; ++i) {
  x[i] = i*0.01;
  y[i] = i*0.03;
}
for (int i = 0; i < N; ++i) {
  y[i] = alpha*x[i] + y[i];
}
```

Fig. 8. Using std::vector in daxpy

```
template <class T>
class UMAllocator<T> {
  public:
    typedef T value_type;
    typedef const T& const_reference;
    template <class U> UMAllocator(const UMAllocator<U>& other);
    T* allocate(std::size_t n) {
      T* ptr;
      cudaMallocManaged(&ptr, sizeof(T)*n);
      return ptr;
    }
    void deallocate(T* p, std::size_t n) {
      cudaFree(p);
    }
};
template <class T, class U>
bool operator==(const UMAllocator<T>&, const UMAllocator<U>&) {
  return true;
}
template <class T, class U>
bool operator!=(const UMAllocator<T>&, const UMAllocator<U>&) {
  return false;
}
```

Fig. 9. Specialized managed memory allocator for std::vector

```
double alpha = 2.0;
int N=1024*1024*10;
vector<double, UMAllocator<double> > x(N);
vector<double, UMAllocator<double> > y(N);
#pragma omp target teams distribute parallel for map(to:x,y
for (int i = 0; i < N; ++i) {
  x[i] = i*0.01;
  y[i] = i*0.03;
}
#pragma omp target teams distribute parallel for map(to:x,y)
for (int i = 0; i < N; ++i) {
  y[i] = alpha*x[i] + y[i];
}
```

Fig. 10. Using std::vector with specialized managed memory allocator

encountering page faults. The `map` clauses on lines 5 and 10 perform a bitwise copy of the structure of the vectors `x` and `y` to the *device* (including the data pointer to UM) allowing both loops to work correctly on *host* and *device*.

Conceivably, one might similarly want to create an "OpenMP-mapped" `std::vector` by using an allocator with additional `enter/exit data` clauses for mapping the vector's data to the *device*. This will, however, not work since mapping the vector structure (e.g. lines 5 and 10, Fig. 10) would then additionally require updating the underlying vector data pointer to the correct *device* address; something not allowed directly for the `vector` class. We further emphasize that in the code presented in the Fig. 10, although vectors `x` and `y` are used exclusively on the *device*, their initial allocation will always be on the *host*. This is because the C++ specification requires the vector data to be default constructed; there is no way to circumvent this default initialization behavior. For the same reason, any attempt at present to write a "*device*-only" allocator (e.g. one using `cudaMalloc` instead of `cudaMallocManaged`) will also fail.

4.4 Limitations of Integrating UM and OpenMP4.5

Although the techniques described above for using UM within OpenMP4.5 target regions are both convenient and elegant, it should be emphasized that mixing OpenMP4.5 and CUDA Managed Memory would require specific hardware and system-software support. For systems with NVIDIA GPUs this approach will not work with devices prior to Pascal GPUs and with versions of CUDA prior to CUDA 8.0.

5 Summary and Outlook

OpenMP is further evolving into version 5 with performance and usability critical changes. First, it will include an interface for performance profiling tools (OMPT). This defines a set of events generated by the runtime that can be intercepted by a profiling tool, and a set of hooks that can be used to inspect the internal state of the library. Second, it includes the concept of implicit declare target, which requires compilers to make function definitions available for devices even if these were not explicitly marked by the user for device compilation. This simplifies building existing host libraries for devices, including some basic STL patterns that are extensively used in technical computing applications. Lastly, the OpenMP committee is working on a set of memory-related constructs that will enable users to express different kind of storage in their program and that are currently under study as a vehicle to express non-volatile memory buffers on CPU and shared memory buffers on GPUs.

Acknowledgement. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-CONF-730616) and supported by Office of Science, Office of Advanced Scientific Computing Research.

References

1. Using shared memory in CUDA C/C++, April 2017. https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/
2. Edwards, H.C., Trott, C., Sunderland, D.: Kokkos, a manycore device performance portability library for C++ HPC applications, March 2014. http://on-demand.gputechconf.com/gtc/2014/presentations/S4213-kokkos-many core-device-perf-portability-library-hpc-apps.pdf
3. Grinberg, L., Bertolli, C., Haque, R.: Hands on with openmp4.5 and unified memory: developing applications for IBM'S hybrid CPU + GPU systems (part I). Submitted for IWOMP 2017
4. CUDA C/C++ programming guide - shared memory section, April 2017. http://docs.nvidia.com/cuda/cuda-c-programming-guide/#shared-memory
5. OpenMP Language Committee: OpenMP Application Program Interface, version 4.5 edn., July 2013. http://www.openmp.org/mp-documents/openmp-4.5.pdf
6. Sakharnykh, N.: Combine OpenACC and unified memory for productivity and performance, September 2015. https://devblogs.nvidia.com/parallelforall/combine-openacc-unified-memory-productivity-performance/
7. Unified memory in CUDA 6, April 2017. https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/
8. Beyond GPU memory limits with unified memory on Pascal, April 2017. https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/