OpenMP Target Offload for Heterogeneous Architectures

Aiichiro Nakano

Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Quantitative & Computational Biology
University of Southern California

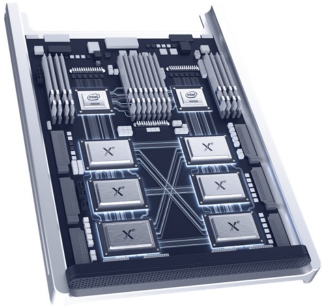
Email: anakano@usc.edu

Goal: Unified *open high-level* programming of both CPU & GPU

Exaflop/s Supercomputing

- **Diverse exaflop/s supercomputing platforms**

Summit (0.2 Exaflop/s, current)
IBM CPU/NVIDIA GPU



GPU Architecture

X^e arch-based "Ponte Vecchio" GPUTile-based, chiplets, HBM stack, Foveros 3D integration, 7nm

On-Node Interconnect

CPU-GPU: PCIe
GPU-GPU: X^e Link

Aurora (2 Exaflop/s)
Intel CPU/Intel GPU

1 HPC and AI Optimized AMD EPYC CPU
4 Purpose Built AMD Radeon Instinct GPU

FRONTIER

Frontier (1.5 Exaflop/s)
AMD CPU/AMD GPU

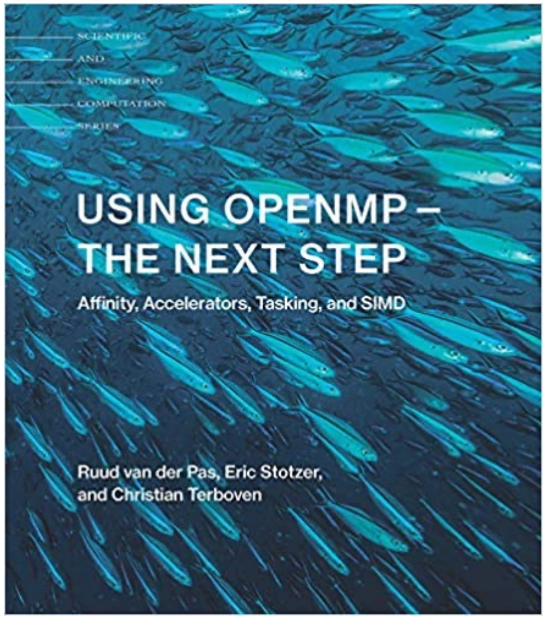
- **Need an *open* programming model for *heterogeneous* (*e.g*. GPU-accelerated) clusters (note CUDA is a proprietary language by NVIDIA)**

See http://press3.mcs.anl.gov/atpesc/files/2021/08/ATPESC-2021-Track-1-Talk-8-Muralidharan-Aurora.pdf

Open Programming Models

- **OpenACC (Open Computing Language)**
  Open standard for directive-based programming of heterogeneous devices

  https://www.openacc.org/


SCIENTIFIC AND ENGINEERING COMPUTATION SERIES
USING OPENMP – THE NEXT STEP
Affinity, Accelerators, Tasking, and SIMD
Ruud van der Pas, Eric Stotzer, and Christian Terboven

- **OpenMP 4.5/5**
  Starting specification version 4.5, OpenMP allows offloading the execution of code & data to heterogeneous devices

  `https://www.openmp.org/specifications/`

OpenMP Offload

- **Latest version of OpenMP allows one to maintain one version of a code, which can run on either a general-purpose central processing unit (CPU) or an accelerator (*e.g*. graphics processing unit, GPU; tensor processing unit, TPU; digital signal processor, DSP; field-programmable gate array, FPGA)**

- **Objective is to execute parts of the program on a heterogeneous *accelerator device* (or *target deice*), *i.e*., dedicated computer hardware outside CPU (which will be called *host device*) to execute certain functions faster than CPU**

- **In OpenMP, program execution begins on the host, which offloads the execution of parts of the code & data to accelerator**

See "OpenMP 4.5 Target Offload" (NASA Ames)
https://aiichironakano.github.io/cs653/OpenMP4.5_3-20-19.pdf

OpenMP Target Construct

- **Simple example**

```
main() {
  float a[1000],b[1000],c,d;
  ...
  #pragma omp target map(a,b,c,d)
  {
    int i;
    #pragma omp parallel for
    for (i=0; i<N; i++)
      a[i] = b[i]*c+d;
  }
  ...
}
```

- **When a host thread encounter the `#pragma omp target` directive, the target region specified by it will be executed by a new thread running on an accelerator, *cf.* CUDA GPU kernel**
- **Before the new thread starts executing the target region, the variable in the `map()` clause are mapped onto accelerator memory, which often is disjunct from the host memory, *cf.* `cudaMemcpy()`**
- **The offloaded code is usually a data-parallel structured block, which can be handled by multiple threads on accelerator using standard OpenMP constructs like `#pragma omp parallel for`**

Computing the Value of π on GPU

omp_target_pi.c

```
#include <omp.h>
#include <stdio.h>
#define NBIN 1000000
#define NTRD 96

int main() {
  float step,sum=0.0,pi;
  step = 1.0/(float)NBIN;

  #pragma omp target map(step,sum)
  {
    # pragma omp parallel for reduction(+:sum) num_threads(NTRD)
    for (long long i=0; i<NBIN; i++) {
      float x = (i+0.5)*step;
      sum += 4.0/(1.0+x*x);
    }
  }

  pi = sum*step;
  printf("PI = %f\n",pi);
  return 0;
}
```

The only addition for GPU offload

Thread reduction of sum

Specify # of GPU threads

This line is identical for CPU & GPU

1. Black: original serial code
2. Green: one-line multithreading
3. Red: another line for GPU offload

GPU: Easy & Hard Ways

Serial: pi.c

```c
#include <stdio.h>
#define NBIN 1000000000
int main() {
   double step, x, sum=0.0, pi;
   step = 1.0/NBIN;
   for (long long i=0; i<NBIN; i++) {
      x = (i+0.5)*step;
      sum += 4.0/(1.0+x*x);
   }
   pi = sum*step;
   printf("PI = %f\n",pi);
   return 0;
}
```

OpenMP: omp_target_pi.c

```c
#include <omp.h>
#include <stdio.h>
#define NBIN 1000000
int main() {
   float step,sum=0.0,pi;
   step = 1.0/(float)NBIN;
   #pragma omp target map(step,sum)
   {
      # pragma omp parallel for reduction(+:sum)
      for (long long i=0; i<NBIN; i++) {
         float x = (i+0.5)*step;
         sum += 4.0/(1.0+x*x);
      }
   }
   pi = sum*step;
   printf("PI = %f\n",pi);
   return 0;
}
```

DPC++: pi.cpp

```cpp
#include <CL/sycl.hpp>
#include <iostream>
#include <array>

using namespace cl::sycl;

#define NBIN 1000000  // # of bins for quadrature
#define NTRD 512      // # of threads

int main()
{
  float step = 1.0f/NBIN;
  std::array<float, NTRD> sum;
  for (int i=0; i<NTRD; ++i) sum[i] = 0.0f;

  queue q(gpu_selector{});

  std::cout << "Running on: " <<
  q.get_device().get_info<info::device::name>() << std::endl;

  range<1> sizeBuf{NTRD};

  {
    buffer<float, 1> sumBuf(sum.data(), sizeBuf);
    q.submit([&](handler &h){
      auto sumAccessor =
      sumBuf.get_access<access::mode::read_write>(h);
      h.parallel_for(sizeBuf, [=](id<1> tid) {
        for (int i=tid; i<NBIN; i+=NTRD) {
          float x = (i+0.5f)*step;
          sumAccessor[tid] += 4.0f/(1.0f+x*x);
        }
      });  // End parallel for
    });  // End queue submit
  }

  float pi=0.0f;
  for (int i=0; i<NTRD; i++)  // Inter-thread reduction
    pi += sum[i];
  pi *= step;  // Multiply bin width to complete integration

  std::cout << "Pi = " << pi << std::endl;

  return 0;
}
```

CUDA: pi.cu

```c
// Using CUDA device to calculate pi
#include <stdio.h>
#include <cuda.h>

#define NBIN  10000000  // Number of bins
#define NUM_BLOCK   13  // Number of thread blocks
#define NUM_THREAD 192  // Number of threads per block
int tid;
float pi = 0;

// Kernel that executes on the CUDA device
__global__ void cal_pi(float *sum, int nbin, float step, int nthreads, int nblocks) {
   int i;
   float x;
   int idx = blockIdx.x*blockDim.x+threadIdx.x;  // Sequential thread index across the blocks
   for (i=idx; i< nbin; i+=nthreads*nblocks) {
      x = (i+0.5)*step;
      sum[idx] += 4.0/(1.0+x*x);
   }
}

// Main routine that executes on the host
int main(void) {
   dim3 dimGrid(NUM_BLOCK,1,1);  // Grid dimensions
   dim3 dimBlock(NUM_THREAD,1,1);  // Block dimensions
   float *sumHost, *sumDev;  // Pointer to host & device arrays

   float step = 1.0/NBIN;  // Step size
   size_t size = NUM_BLOCK*NUM_THREAD*sizeof(float);  //Array memory size
   sumHost = (float *)malloc(size);  //  Allocate array on host
   cudaMalloc((void **) &sumDev, size);  // Allocate array on device
   // Initialize array in device to 0
   cudaMemset(sumDev, 0, size);
   // Do calculation on device
   cal_pi <<<dimGrid, dimBlock>>> (sumDev, NBIN, step, NUM_THREAD, NUM_BLOCK); // call CUDA kernel
   // Retrieve result from device and store it in host array
   cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);
   for(tid=0; tid<NUM_THREAD*NUM_BLOCK; tid++)
      pi += sumHost[tid];
   pi *= step;

   // Print results
   printf("PI = %f\n",pi);

   // Cleanup
   free(sumHost);
   cudaFree(sumDev);

   return 0;
}
```

Hierarchical Parallelization

- `#pragma omp teams`

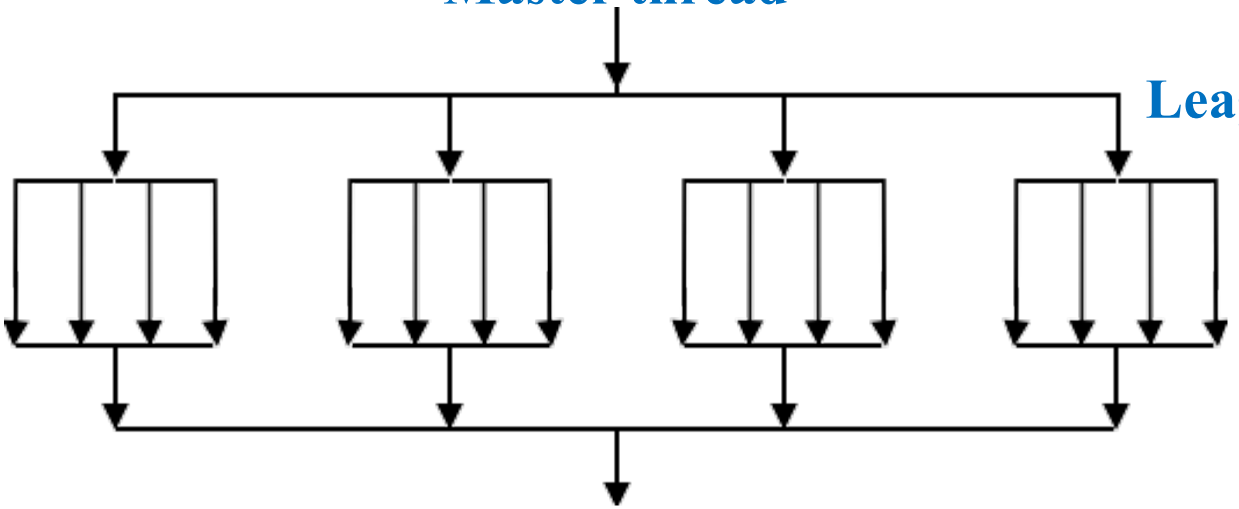
  Starts a league of multiple thread teams; `teams` construct must be nested immediately inside a `target` construct, so they are commonly used as `#pragma omp target teams`; `num_teams` clause can be used to specify the number of teams

  (Example) `#pragma omp target teams num_teams(13)`

- `#pragma omp distribute`

  Distribute the work across the teams

Master thread



League of teams

Team of threads

***cf*. CUDA grid of blocks & block of threads**

Remember NVIDIA SM/SP & Intel slice/subslice/EU?

Teams for Computing π

- **Spatial decomposition *via* offset among teams & data privatization**

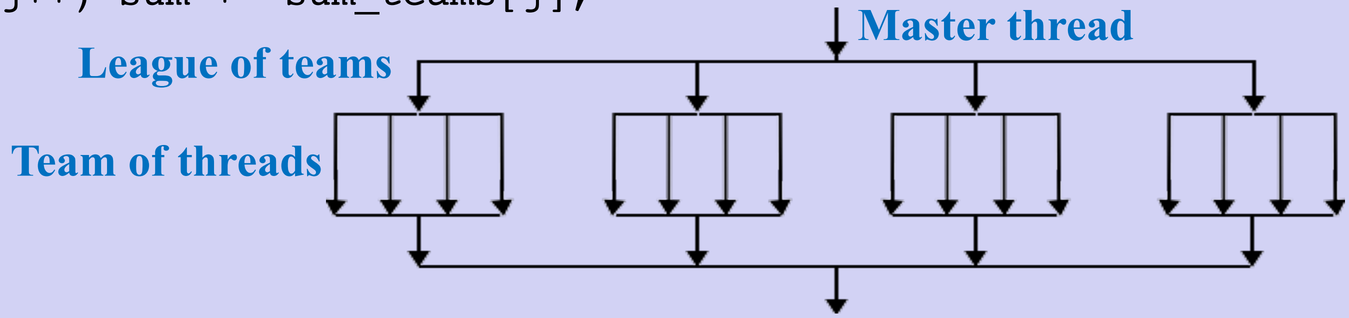
```
#define NTMS 12
float sum_teams[NTMS];
for (int j=0; j<NTMS; j++) sum_teams[j] = 0.0;
...
#pragma omp target teams map(step,sum_teams) num_teams(NTMS)
{
  #pragma omp distribute
  for (int j=0; j<NTMS; j++) {
    long long ibgn = NBIN/NTMS*j;
    long long iend = NBIN/NTMS*(j+1);
    if (j == NTMS-1) iend = NBIN;
    # pragma omp parallel for reduction(+:sum_teams[j]) num_threads(NTRD)
    for (long long i=ibgn; i<iend; i++) {
      float x = (i+0.5)*step;
      sum_teams[j] += 4.0/(1.0+x*x);
    }
  }
}

for (int j=0; j<NTMS; j++) sum += sum_teams[j];
```

Data privatization among teams

New: teams & distribute constructs

$\lfloor$NBIN/NTMS$\rfloor$*NTMS could be less than NBIN

Modified: offset & private accumulator



Using OpenMP Target on Discovery

- **Necessary module**

```
module purge
module load llvm    // Use LLVM compiler
```

- **Compilation**

Type all in one line

```
clang -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -Xopenmp-target
-march=sm_70 --gcc-toolchain=/spack/apps/gcc/8.3.0 omp_target_pi.c
-o omp_target_pi
```

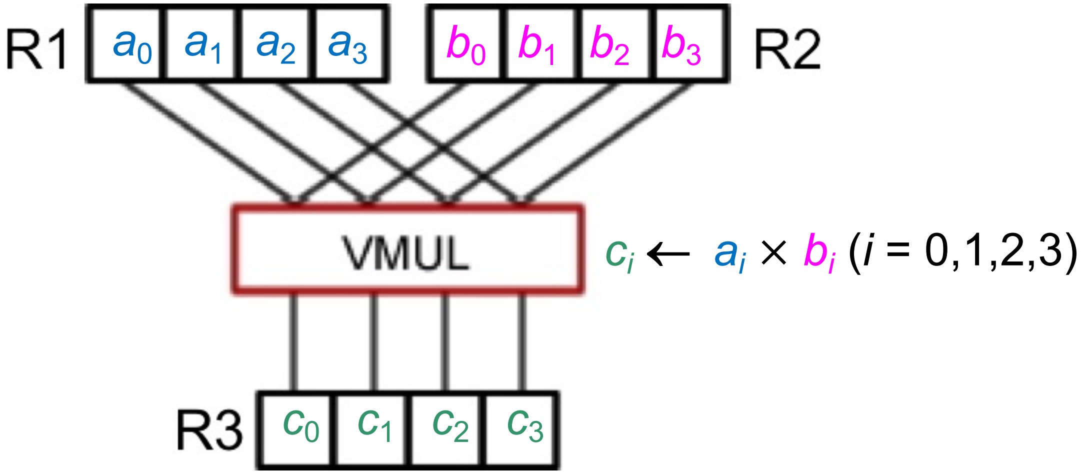
- **Execute on a GPU-accelerated node**

```
[anakano@discovery]$ salloc --partition=gpu --gres=gpu:v100:1 --time=00:30:00
salloc: Nodes d11-02 are ready for job
[anakano@d11-02]$ ./omp_target_pi
PI = 3.141593
```

Single Instruction Multiple Data

- OpenMP 4.5 & later supports several other new features
- Single-instruction multiple-data (SIMD) parallelism: An arithmetic operation is operated on multiple operand-pairs stored in vector registers (each of which can hold multiple operands) using vector instructions
- OpenMP `simd` construct instructs the compiler to vectorize the loop

```
#pragma omp for simd
for (int i=0; i<n; i++)
  a[i] = b[i]+c[i];
```



Asynchronous Offload

```
main() {
  float a[1000],b[1000],c,d;
  ...
  #pragma omp target nowait map(a,b,c,d)
  {
    int i;
    #pragma omp parallel for
    for (i=0; i<N; i++)
      a[i] = b[i]*c+d;
  }
  func(b);  // perform computation independent of device output
  #pragma omp taskwait
  func(a);  // perform computation dependent on device output
}
```

cf. MPI_Irecv() & MPI_Wait()

- **By default, the thread that encounters a device construct waits for the construct to complete before executing the next line**
- **When a `nowait` clause is added to the device construct, the encountering thread does not wait but instead continues executing the code passed the construct**
- **The `taskwait` constructs lets the original thread wait for the completion of the target task generated by it before continuing to the next line**

Overlap CPU & GPU computations for high performance

Persistent GPU Memory Allocation

- **Expensive CPU-GPU data transfer associated with the `map` clause can be minimized by making GPU memory allocation persistent**

cf. 3-phase (host-to-device copy→kernel execution→device-to-host copy) performance bottleneck

```
// Pre-allocate GPU array
#pragma omp target enter data map(alloc:psi[0:Nmax])
```

Stand-alone directive to map variables to device memory

```
...

// Keep operating on device array only from device
#pragma omp target parallel for \
map(tofrom:psi[0:Nmax]) map(to:u[0:Nmax])
for (int i=0; i< Nmax; i++) psi[i] *= u[i];
```

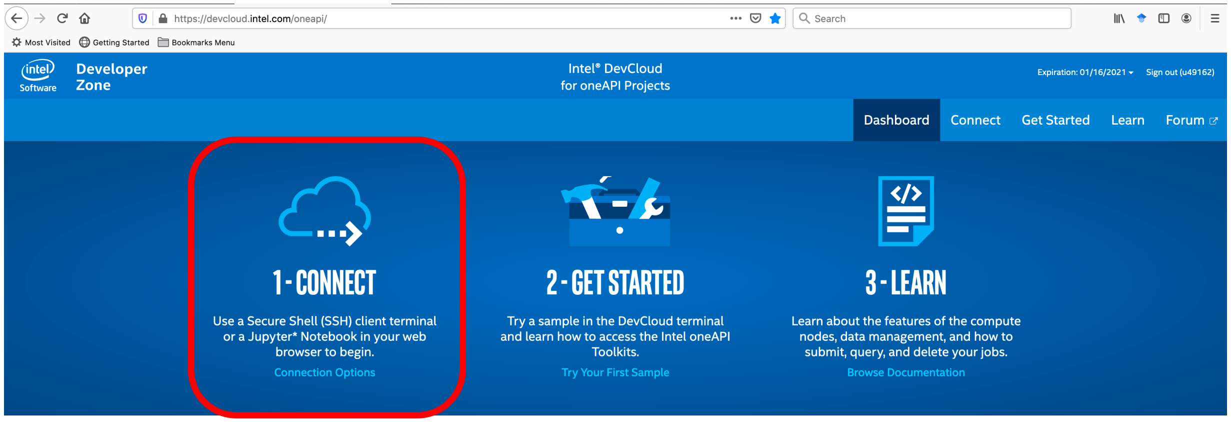
Runtime system keeps track of CPU & GPU memory access & avoids unnecessary CPU-GPU data transfer

```
...

// De-allocate GPU array
#pragma omp target exit data map(delete:psi[0:Nmax])
```

Stand-alone directive to unmap variables from device memory

(Optional) Using Intel DevCloud (1)

- Sign up at the Intel developer's cloud (DevCloud) site: https://devcloud.intel.com/oneapi



1 - CONNECT

Use a Secure Shell (SSH) client terminal or a Jupyter* Notebook in your web browser to begin.

Connection Options

2 - GET STARTED

Try a sample in the DevCloud terminal and learn how to access the Intel oneAPI Toolkits.

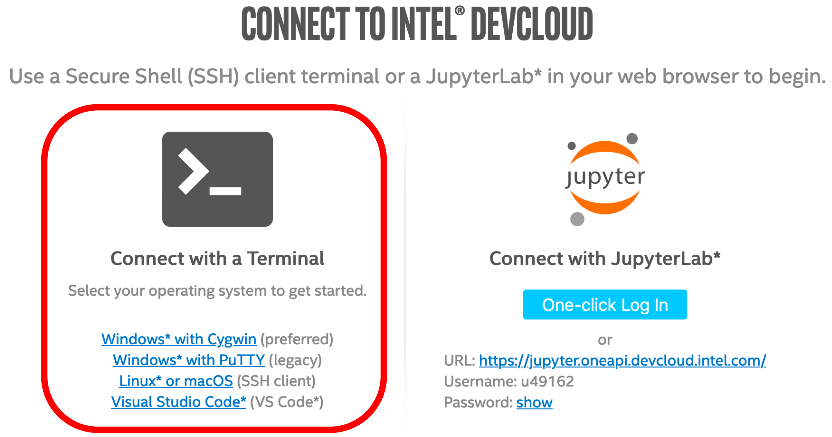
Try Your First Sample

3 - LEARN

Learn about the features of the compute nodes, data management, and how to submit, query, and delete your jobs.

Browse Documentation

- Setup terminal connection to devcloud server



CONNECT TO INTEL® DEVCLOUD

Use a Secure Shell (SSH) client terminal or a JupyterLab* in your web browser to begin.

Connect with a Terminal

Select your operating system to get started.

Windows* with Cygwin (preferred)
Windows* with PuTTY (legacy)
Linux* or macOS (SSH client)
Visual Studio Code* (VS Code*)

Connect with JupyterLab*

or

URL: https://jupyter.oneapi.devcloud.intel.com/
Username: u49162
Password: show

- Log in to devcloud server

```
macbook-pro:~ aiichironakano$ ssh devcloud
u49162@login-2:~$
```

Using Intel DevCloud (2)

- Transfer `omp_target_pi.c` to devcloud server, *e.g.*, using sftp
- Compile

```
u49162@login-2:~$ cc -o omp_target_pi omp_target_pi.c -fopenmp
```

- Interactive job on a GPU-accelerated computing node using PBS (portable batch system) job scheduler

```
u49162@login-2:~$ qsub -I -l nodes=1:gpu:ppn=2
qsub: waiting for job 694715.v-qsvr-1.aidevcloud to start
qsub: job 694715.v-qsvr-1.aidevcloud ready
u49162@s001-n177:~$
```

- Run

```
u49162@s001-n177:~$ ./omp_target_pi
PI = 3.141593
```

- Start developing your own OpenMP target offload codes for GPU acceleration!

Where to Go from Here

- **Start developing your own OpenMP target offload codes for GPU acceleration**
- **Plenty of room for performance optimization**

  ***e.g.*, target region executed by different threads happens concurrently, used for , *e.g.*, overlapping computation & data transfer**

```
#pragma omp parallel  // Spawn multiple CPU threads
{
  #pragma omp target
  {// Different GPU threads perform computation or data transfer }
}
```

See “OpenMP Offload Optimization” (Ye Luo, Argonne National Lab.)

https://aiichironakano.github.io/cs653/Luo-OpenMPoffload-SC20.pdf