Lecture: Manycore GPU Architectures and Programming, Part 4
-- Introducing OpenMP and HOMP for Accelerators

CSCE 569 Parallel Computing

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
https://passlab.github.io/CSCE569/
Manycore GPU Architectures and Programming: Outline

• Introduction
  – GPU architectures, GPGPUs, and CUDA
• GPU Execution model
• CUDA Programming model
• Working with Memory in CUDA
  – Global memory, shared and constant memory
• Streams and concurrency
• CUDA instruction intrinsic and library
• Performance, profiling, debugging, and error handling
  ➡ Directive-based high-level programming model
  – OpenMP and OpenACC
HPC Systems with Accelerators

- Accelerator architectures become popular
  - GPUs and Xeon Phi
- Multiple accelerators are common
  - 2, 4, or 8

Programming on NVIDIA GPUs
1. CUDA and OpenCL
   - Low-level
2. Library, e.g. cublas, cufft, cuDNN
3. OpenMP, OpenACC, and others
   - Rely on compiler support
4. Application framework
   - TensorFlow, etc

https://www.anandtech.com/show/12587/nvidias-dgx2-sixteen-v100-gpus-30-tb-of-nvme-only-400k
OpenMP 4.0 for Accelerators

- **Device**: a logical execution engine
  - Host device: where OpenMP program begins, one only
  - Target devices: **1 or more** accelerators

- **Memory model**
  - Host data environment: one
  - Device data environment: one or more
  - Allow shared host and device memory

- **Execution model: Host-centric**
  - Host device: “offloads” code regions and data to accelerators/target devices
  - Target Devices: still fork-join model
  - Host waits until devices finish
  - Host executes device regions if no accelerators are available / supported
AXPY Example with OpenMP: Multicore

- \( y = \alpha \cdot x + y \)
  - \( x \) and \( y \) are vectors of size \( n \)
  - \( \alpha \) is scalar

```c
void axpy(REAL *x, REAL *y, long n, REAL a) {
    #pragma omp parallel for shared(x, y, n, a)
    for (int i = 0; i < n; ++i)
        y[i] += a * x[i];
}
```

- Data (\( x, y \) and \( a \)) are shared
  - Parallelization is relatively easy
- Other examples
  - sum: reduction
  - Stencil: halo region exchange and synchronization
AXPY Offloading To a GPU using CUDA

```c
// CUDA kernel. Each thread takes care of one element of c.
__global__ void axpy(REAL *x, REAL *y, int n, REAL a) {
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    if (id < n) y[id] += a * x[id];
}

int main( int argc, char* argv[] ) {

    // ... init host a, x and y
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_x, size);
    cudaMalloc(&d_y, size);

    // Copy host vectors to device
    cudaMemcpy( d_x, h_x, size, cudaMemcpyHostToDevice);
    cudaMemcpy( d_y, h_y, size, cudaMemcpyHostToDevice);

    int blockSize, gridSize;
    blockSize = 1024;
    gridSize = (int)ceil((float)n/blockSize);
    axpy<<<gridSize, blockSize>>>(d_x, d_y, n, a);

    // Copy array back to host
    cudaMemcpy( h_y, d_y, size, cudaMemcpyDeviceToHost );

    // Release device memory
    cudaFree(d_x);
    cudaFree(d_y);
}
```

Memory allocation on device

Memcpy from host to device

Launch parallel execution

Memcpy from device to host

Deallocation of dev memory
AXPY Example with OpenMP: single device

- $y = \alpha \cdot x + y$
  - $x$ and $y$ are vectors of size $n$
  - $\alpha$ is scalar

```c
void axpy_ompacc(REAL* x, REAL* y, int n, REAL a) {
    #pragma omp target device (0) map(tofrom: y[0:n]) \map(to: x[0:n],a,n)
    #pragma omp parallel for shared(x, y, n, a)
    for (int i = 0; i < n; ++i)
        y[i] += a * x[i];
}
```

- **target** directive: annotate an offloading code region
- **map** clause: map data between host and device $\rightarrow$ moving data
  - to|tofrom|from: mapping directions
  - Use array region
OpenMP Computation and Data Offloading

- `#pragma omp target device(id) map() if()`
  - **target**: create a data environment and offload computation on the device
  - **device (int_exp)**: specify a target device
  - **map(to|from|tofrom|alloc:var_list)**: data mapping between the current data environment and a device data environment

- `#pragma target data device (id) map() if()`
  - Create a device data environment: to be reused/inherited
target and map Examples

```c
void vec_mult(int N) {
    int i;
    float p[N], v1[N], v2[N];
    init(v1, v2, N);
    #pragma omp target map(to: v1, v2) map(from: p)
    #pragma omp parallel for
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}

void vec_mult(float *p, float *v1, float *v2, int N) {
    int i;
    init(v1, v2, N);
    #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
    #pragma omp parallel for
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}
```
Accelerator: Explicit Data Mapping

• Relatively small number of truly shared memory accelerators so far
• Require the user to explicitly map data to and from the device memory
• Use array region

```c
long a = 0x858;
long b = 0;
int anArray[100]

#pragma omp target data map(to:a) \ 
   map(tofrom:b,anArray[0:64])
{
    /* a, b and anArray are mapped
     * to the device */

    /* work on the device */
    #pragma omp target ...
    {
      ...
    }
}
/* b and anArray are mapped
 * back to the host */
```
void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    #pragma omp target data map(from: p[0:N])
    {
        #pragma omp target map(to: v1[:N], v2[:N])
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = v1[i] * v2[i];
        init_again(v1, v2, N);
        #pragma omp target map(to: v1[:N], v2[:N])
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = p[i] + (v1[i] * v2[i]);
    }
    output(p, N);
}
Accelerator: Hierarchical Parallelism

• Organize massive number of threads
  – teams of threads, e.g. map to CUDA grid/block

• Distribute loops over teams

```
#pragma omp target
#pragma omp teams num_teams(2) num_threads(8)
{
    //-- creates a “league” of teams
    //-- only local barriers permitted

#pragma omp distribute
for (int i=0; i<N; i++) {

}
```

Only `target` directive makes it as accelerator region.
teams and distribute Loop Example

```c
float dotprod_teams(float B[], float C[], int N, int num_blocks,
                    int block_threads)
{
    float sum = 0;
    int i, i0;
    #pragma omp target map(to: B[0:N], C[0:N])
    #pragma omp teams num_teams(num_blocks) thread_limit(block_threads)
        reduction(+:sum)
    #pragma omp distribute
    for (i0=0; i0<N; i0 += num_blocks)
        #pragma omp parallel for reduction(+:sum)
        for (i=i0; i< min(i0+num_blocks,N); i++)
            sum += B[i] * C[i];
    return sum;
}
```

Double-nested loops are mapped to the two levels of thread hierarchy (league and team)
#pragma omp target data device (gpu0) map(to:n, m, omega, ax, ay, b, \
  f[0:n][0:m]) map(tofrom:u[0:n][0:m]) map(alloc:uold[0:n][0:m])

while ((k<=mits)&&(error>tol))
{
  // a loop copying u[][] to uold[][] is omitted
  ...
  #pragma omp target device(gpu0) map(to:n, m, omega, ax, ay, b, f[0:n][0:m]) map(tofrom:u[0:n][0:m])
  #pragma omp parallel for private(resid,j,i)
  for (i=1;i<(n-1);i++)
    for (j=1;j<(m-1);j++)
    {
      resid = (ax*(uold[i-1][j] + uold[i+1][j]) + 
                ay*(uold[i][j-1] + uold[i][j+1]));
      u[i][j] = uold[i][j] - omega * resid;
      error = error + resid*resid;
    } // the rest code omitted ...
}
Mapping Nested Loops to GPUs

- Need to achieve coalesced memory access on GPUs

```c
#pragma acc loop gang(2) vector(2)
for ( i = x1; i < X1; i++ ) {
    #pragma acc loop gang(3) vector(4)
    for ( j = y1; j < Y1; j++ ) {...... }
}
```

Fig. 9: Double nested loop mapping.

Fig. 10: Triple nested loop mapping.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Map2_1</th>
<th>Map2_2</th>
<th>Map2_3</th>
<th>Map2_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DGEMM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaussblur</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Threads used in each loop with double loop mappings

- Shows the performance comparison in different benchmarks with different double nested loop mappings. All of Jacobi, DGEMM, and Gaussblur have double nested parallel loops but they show different performance behavior. In Jacobi, the data accessed from the inner loop are contiguous in memory while they are non-contiguous when accessed from the outer loop. In all of our four double nested loop mappings, the inner loop uses `vector` which means the threads executing the inner loop are consecutive. In both `vector` and `gang vector` cases, the threads are consecutive and the only difference is the length of concurrent threads. In Jacobi, the consecutive threads access aligned and consecutive data and therefore the memory access is coalesced. In this case, the memory access pattern and the loop mapping mechanism match perfectly. That is why the performance using all of the four loop mappings are close. Table 2 shows the number of threads used in each loop mapping. Because Map2_1 and Map2_2 have less threads than Map2_3 and Map2_4 in the inner loop, the execution time is slightly longer. Map2_1 and Map2_2 have the same performance since their threads are the same in both the outer loop and inner loop. The performance behavior of Gaussblur is similar to Jacobi because their memory access pattern and threads management are similar.

In DGEMM, the performance of Map2_2 and Map2_4 are better than the other two mappings which is because they both have enough parallelism in each block to hide memory access latency. The performance penalty in Map2_1 is due to less parallelism in each block. Map2_3 has the worst performance as it does...
We are also developing a certification suite to test emerging OpenACC implementations for completeness and semantic correctness to ensure that the implementations achieve a high degree of conformity with the standard.

The certification suite consists of applications from several well-known benchmark suites such as NAS, PARBOIL, Rodinia and others. The applications were chosen based on several domains and include a variety of computational methods.

Table 2 shows a comparison of the speedup of OpenACC for a variety of applications to sequential version, OpenMP Version 3.1 (8 cores) and CUDA (4.2 & 5.0).

<table>
<thead>
<tr>
<th>Applications</th>
<th>Domains</th>
<th>OpenACC Directive Combinations</th>
<th>Lines of Code Added vs Serial</th>
<th>Speedup Over</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>OpenMP</td>
<td>OpenACC</td>
</tr>
<tr>
<td>Needleman-Wunsch</td>
<td>Bioinformatics</td>
<td>data copy, copyin kernels present loop gang, vector, private</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Stencil</td>
<td>Cellular Automation</td>
<td>data copyin, copy, deviceptr kernels present loop collapse, independent</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Computational Fluid Dynamics (CFD)</td>
<td>Fluid Mechanics</td>
<td>data copyin, copy, deviceptr data present, deviceptr kernels deviceptr kernels loop, gang, vector, private loop gang, vector acc_malloc(), acc_free()</td>
<td>8</td>
<td>46</td>
</tr>
<tr>
<td>2D Heat (grid size 4096*4096)</td>
<td>Heat Conduction</td>
<td>data copyin, copy, deviceptr kernels present loop collapse, independent</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Clever (10Ovals)</td>
<td>Data Mining</td>
<td>data copyin kernels present, create, copyin, copy loop independent</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>FeldKemp (FDK)</td>
<td>Image Processing</td>
<td>kernels copyin, copyout loop, collapse, independent</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

We observe that the certification suite exhibits a set of behaviors. For example, OpenACC speedup ranges from 2.98 to 99.52 over the sequential version and from 1.22 to 28.63 over 8-core OpenMP version. The applications have been chosen based on their computations and communication patterns.

We also observed that OpenACC is yet to achieve good speedup compared with that of the CUDA version of the applications. It may be because the OpenACC compilers do not generate an optimized GPU code. Deeper analysis of the OpenACC code may lead to further enhancements to the code and better speedup. However, with respect to the sequential and OpenMP version of the applications, we notice improved speedup in each case. We are aware that OpenMP, a directive-based model, is good at retaining most of the code structure, and still be able to express parallelism. The LOC (Lines of Code) column shows that OpenACC also offers similar advantages. For the application CFD, we notice that LOC was significantly different to that of OpenMP, this is primarily because we have used acc_malloc(), acc_free() and runtime routines, but it is evident that OpenACC offers better speedup. Consequently, we think current OpenACC implementations allow applications to be ported to GPUs in a successful manner in terms of programmability and portability.

NAS Parallel Benchmarks for GPGPUs using a Directive-based Programming Model, Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, Yonghong Yan and Barbara Chapman 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC2014)
AXPY Example with OpenMP: Multiple device

```c
void axpy_omp_mdev(REAL* x, REAL* y, int n, REAL
int ndev = omp_get_num_devices();
#pragma omp parallel num_threads(ndev)
{
    int devid = omp_get_thread_num();
    int start, size, remnant;
    remnant = n % ndev; size = n / ndev;
    if (devid < remnant) {
        size++; start = size*devid;
    } else start = size*devid+remnant;
#pragma omp target device (devid) \ 
    map(to:from: y[start:size]) \ 
    map(to: x[start:size],a,size)
#pragma omp parallel for shared(x, y, si
for (int i = 0; i < size; ++i)
    y[i] += a * x[i];
}
```
Hybrid OpenMP (HOMP) for Multiple Accelerators

```c
/* align computation with data using ALIGN(x)*/
void axpy_homp_v1(REAL* x, REAL* y, int n, REAL a) {
    #pragma omp parallel target device (*) \ 
    map(tofrom: y[0:n] distribute(BLOCK)) \ 
    map(to: x[0:n] distribute(BLOCK),a,n)
    #pragma omp parallel for distribute(ALIGN(x))
    for (int i = 0; i < n; ++i)
        y[i] += a * x[i];
}

/* align data with computation using ALIGN*/
void axpy_homp_v2(REAL* x, REAL* y, int n, REAL a) {
    #pragma omp parallel target device (*) \ 
    map(tofrom: y[0:n] distribute(ALIGN(loop))) \ 
    map(to: x[0:n] distribute(ALIGN(loop)),a,n)
    #pragma omp parallel for distribute(AUTO)
    loop:  for (int i = 0; i < n; ++i)
        y[i] += a * x[i];
}
```

HOMP: Automated Distribution of Parallel Loops and Data in Highly Parallel Accelerator-Based Systems, Yonghong Yan, Jiawen Liu, and Kirk W. Cameron, The IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2017
Three Challenges to Implement HOMP

1. Load balance when distributing loop iterations across computational different devices (CPU, GPU, and MIC)
   – We developed 7 algorithms of loop distribution and the runtime select algorithms based on computation/data intensity

2. Only copy the associated data to the device that are needed for the loop chunks assigned to that device
   – Runtime support for ALIGN interface to move or share data between memory spaces

1. Select devices for computations for the optimal performance because more devices ≠ better performance
   – CUTOFF ratio to select device
Offloading Execution Time (ms) on 2 CPUs + 4 GPUs + 2 MICs and using CUTOFF_RATIO

| Algorithm Type                  | Execution Time (ms) | Speedup
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Profile Auto (15% CUTOFF)</td>
<td>109.93</td>
<td>2.09</td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>1482.90</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>1360.35</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>1499.78</td>
<td></td>
</tr>
<tr>
<td>Sched Profile Auto</td>
<td>5055.33</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>1422.96</td>
<td>2.68</td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>6532.80</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>10393.55</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>3664.08</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>20989.67</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>3602.80</td>
<td>1.01</td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>1723.84</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>1695.45</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>272.80</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>306.26</td>
<td>1.35</td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>514.63</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>3909.59</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>10393.55</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>3508.77</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>424.29</td>
<td></td>
</tr>
<tr>
<td>Model Profile Auto</td>
<td>1423.10</td>
<td></td>
</tr>
</tbody>
</table>

The algorithm that delivers the best performance without CUTOFF

The algorithm with 15% CUTOFF_RATIO that delivers the best performance and its speedup against the best algorithm that does not use CUTOFF
## Speedup From CUTOFF

- Apply 15% CUTOFF ratio to modeling and profiling
  - Only those devices who may compute more than 15% of total iterations will be used
- Thinking of 8 devices (1/8 = 12.5%)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Devices used</th>
<th>CUTOFF Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>axpy-10B</td>
<td>2 CPU + 4 GPUs</td>
<td>1.35</td>
</tr>
<tr>
<td>bm2d-256</td>
<td>2 CPU + 4 GPUs</td>
<td>1.01</td>
</tr>
<tr>
<td>matul-6144</td>
<td>4 GPUs</td>
<td>2.68</td>
</tr>
<tr>
<td><strong>matvec-48k</strong></td>
<td><strong>4 GPUs</strong></td>
<td><strong>0.56</strong></td>
</tr>
<tr>
<td>stencil2d-256</td>
<td>4 GPUs</td>
<td>3.43</td>
</tr>
<tr>
<td>sum-300M</td>
<td>2 CPUs + 4 GPUs</td>
<td>2.09</td>
</tr>
</tbody>
</table>