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

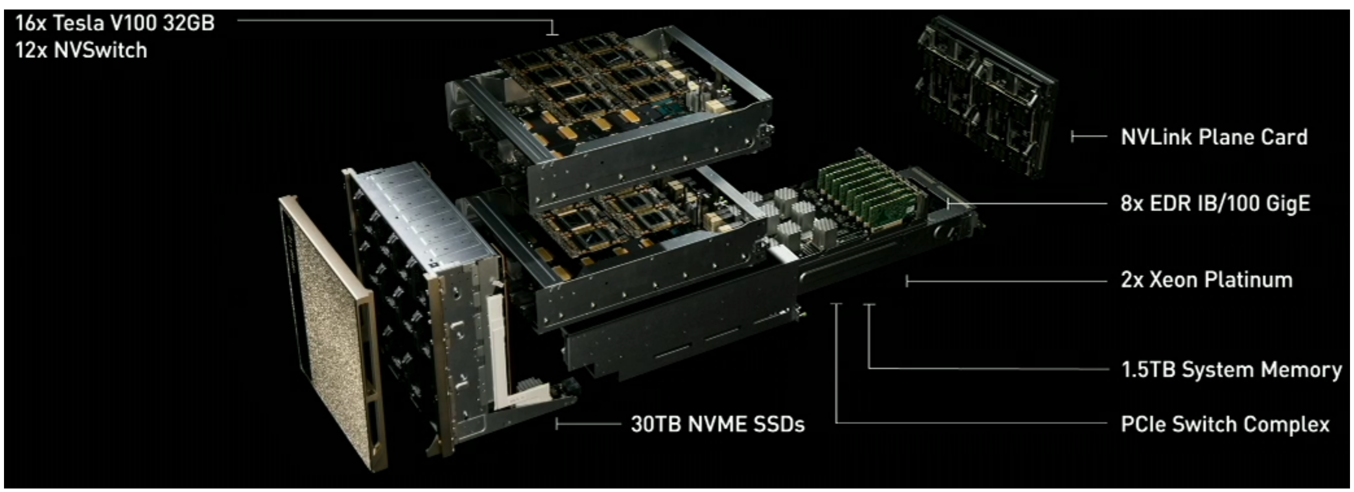
- Introduction
  - GPU architectures, GPGPUs, and CUDA
- GPU Execution model
- CUDA Programming model
- Working with Memory in CUDA
  - Global memory, shared and constant memory
- Streams and concurrency
- CUDA instruction intrinsic and library
- Performance, profiling, debugging, and error handling
- ☛ **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

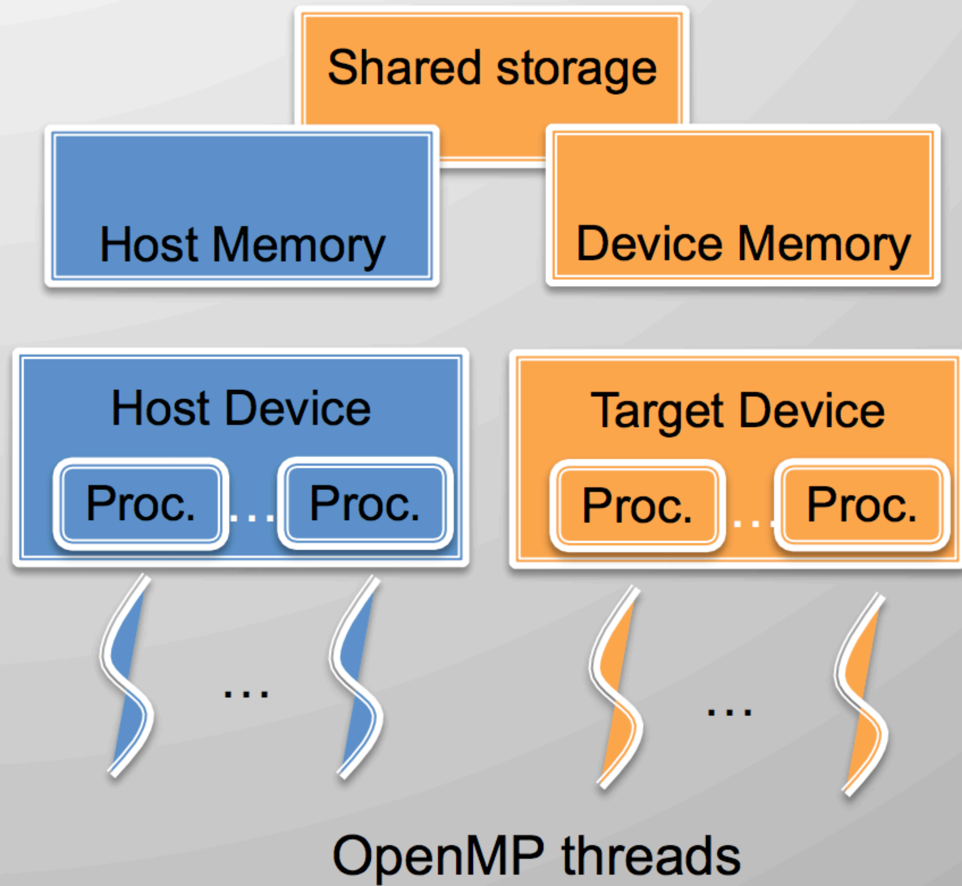
1. CUDA and OpenCL
   - **Low-level**
2. Library, e.g. cublas, cufft, cuDNN
3. **OpenMP, OpenACC, and others**
   - **Rely on compiler support**
4. Application framework
   - **TensorFlow, etc**



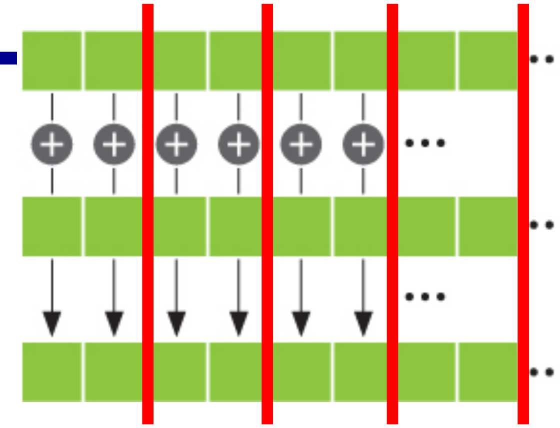
https://www.anandtech.com/show/12587/nvidias-dgx2-sixteen-v100-gpus-30-tb-of-nvme-only-400k

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



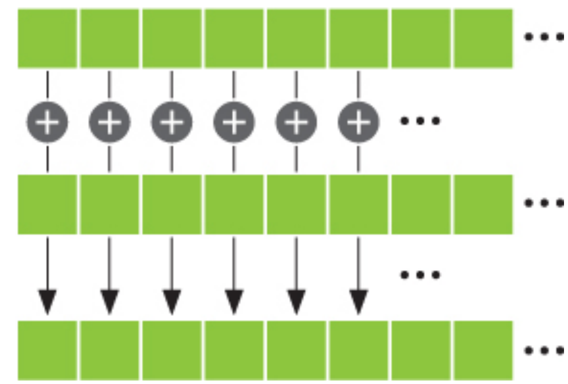
- y = α·x + y
  - x and y are vectors of size n
  - α is scalar

```
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

```
// 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

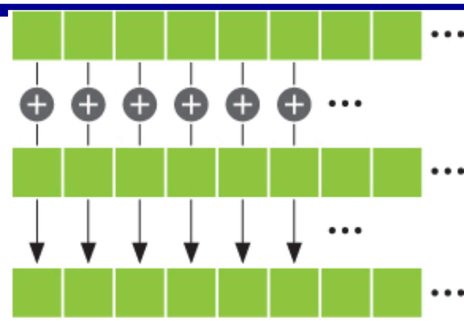
Memcpy from host to device

Launch parallel execution

Memcpy from device to host

Deallocation of dev memory

AXPY Example with OpenMP: single device



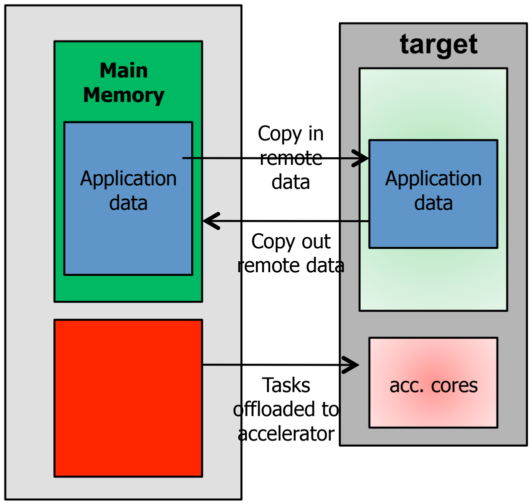
- $y = \alpha \cdot x + y$
  - x and y are vectors of size n
  - α is scalar

```
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];
}
```

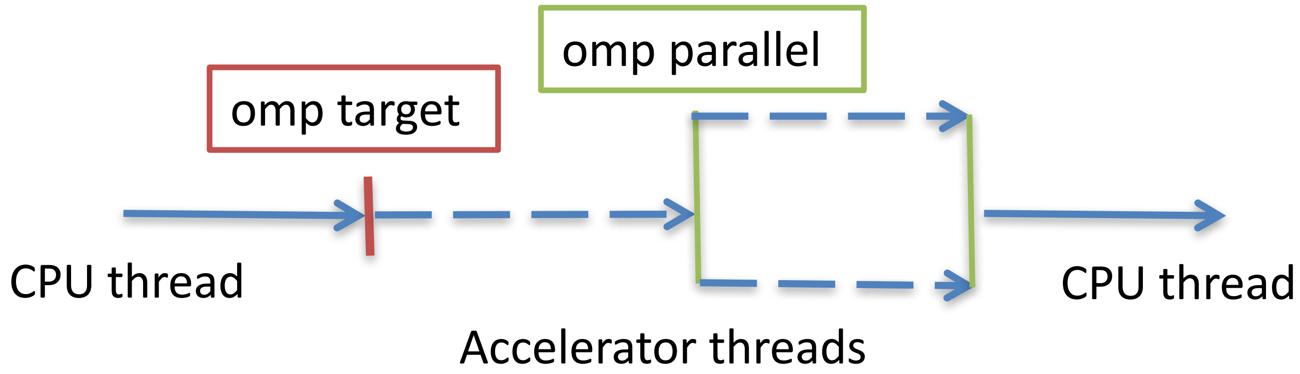
- **target** directive: annotate an offloading code region
- **map** clause: map data between host and device → moving data
  - **to|tofrom|from**: mapping directions
  - Use array region

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

```
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

```
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 */
```

target date Example

```
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);
}
```

Note mapping inheritance

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

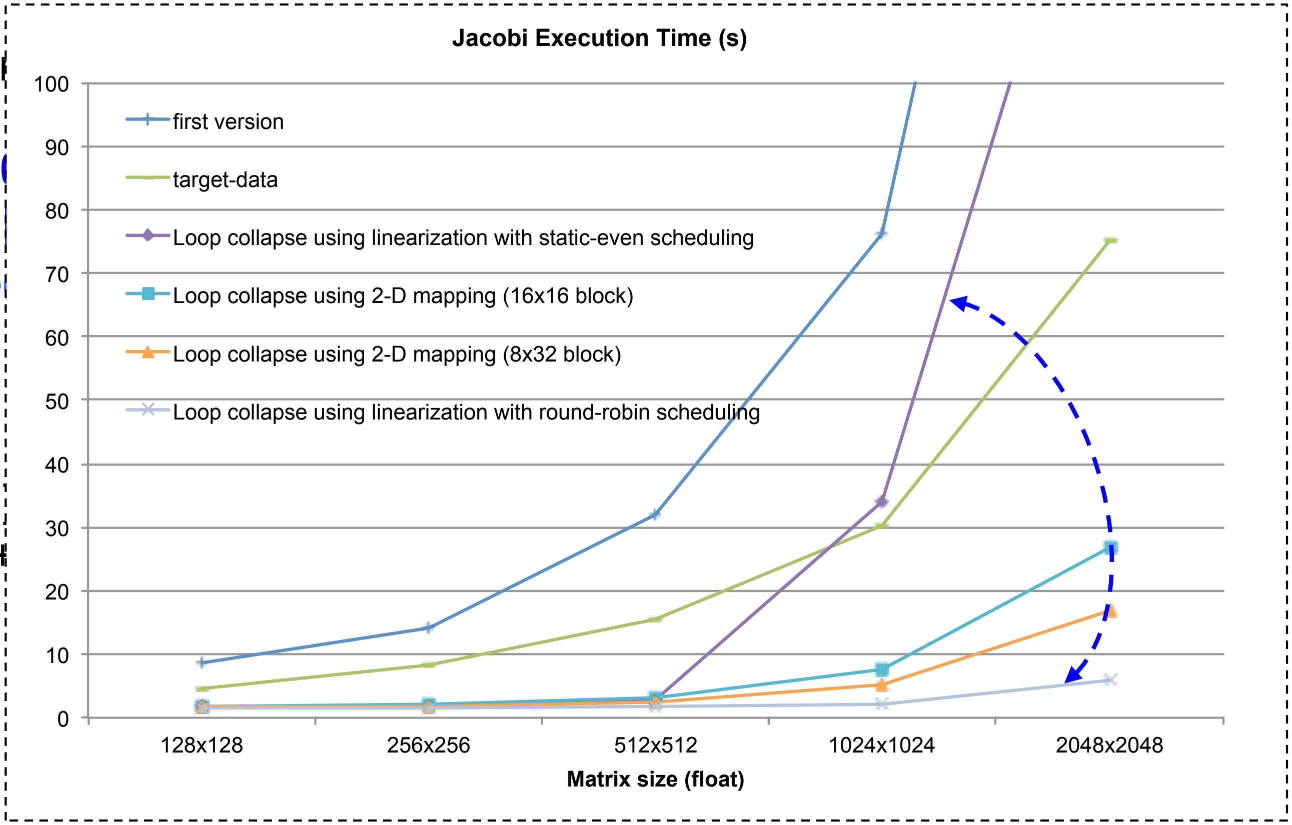
teams and distribute Loop Example

```
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)

Jacobi Example: The Impact of Compiler Transformation to Performance

```
#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 o
 …
#pragma omp target  device(gpu
    uold[0:n][0:m])  map(tofrom:u
#pragma omp parallel for private
for (i=1;i<(n-1);i++)
  for (j=1;j<(m-1);j++)
  {
    resid = (ax*(uold[i-1][j] + uold[i+1
        + 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  ...
}
```

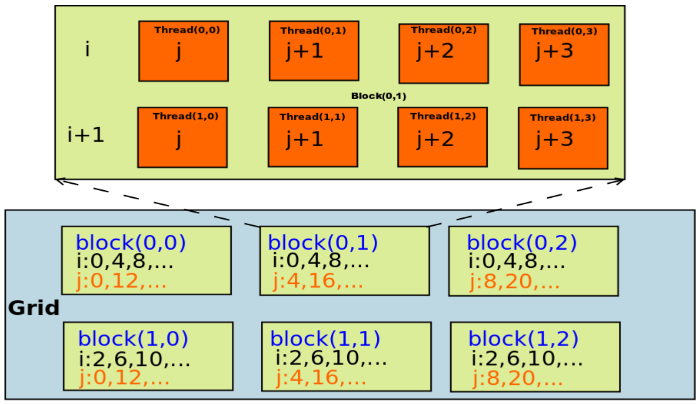


Early Experiences With The OpenMP Accelerator Model; Chunhua Liao, Yonghong Yan, Bronis R. de Supinski, Daniel J. Quinlan and Barbara Chapman; International Workshop on OpenMP (IWOMP) 2013, September 2013

Mapping Nested Loops to GPUs

- Need to achieve coalesced memory access on GPUs

```
#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++ ) {...... }
}
```



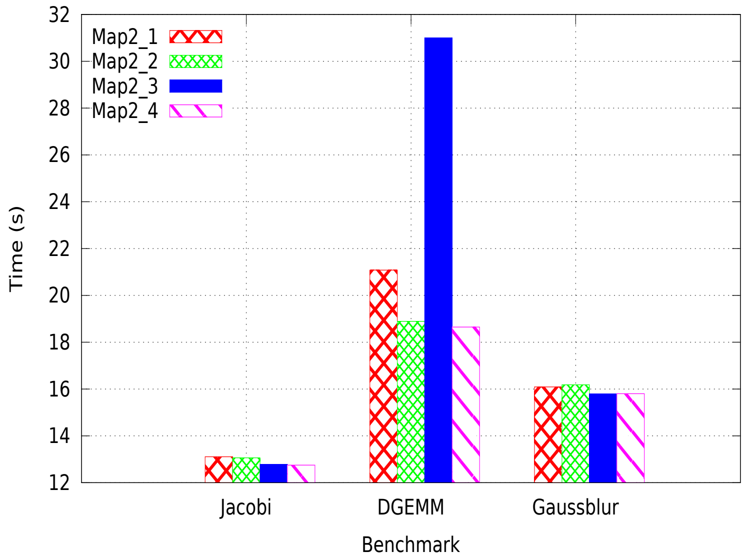


Fig. 9: Double nested loop mapping.

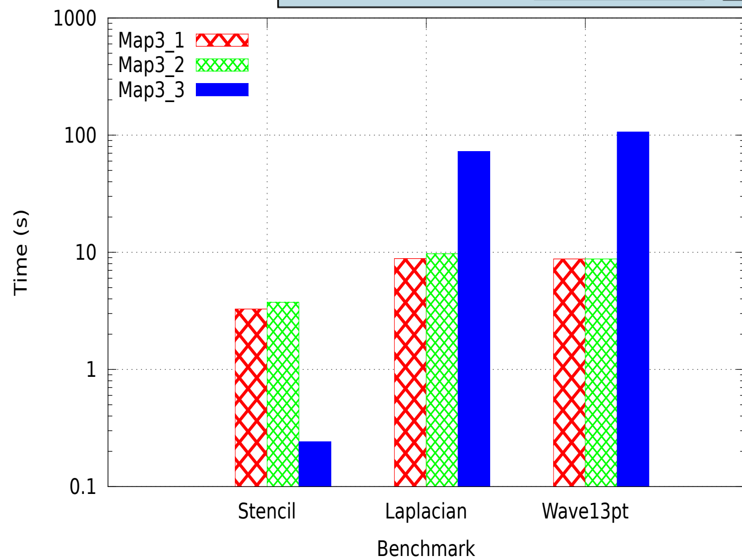


Fig. 10: Triple nested loop mapping.

Compiler Transformation of Nested Loops for GPGPUs, Xiaonan Tian, Rengan Xu, **Yonghong Yan**, Sunita Chandrasekaran, and Barbara Chapman Journal of Concurrency and Computation: Practice and Experience, August 2015

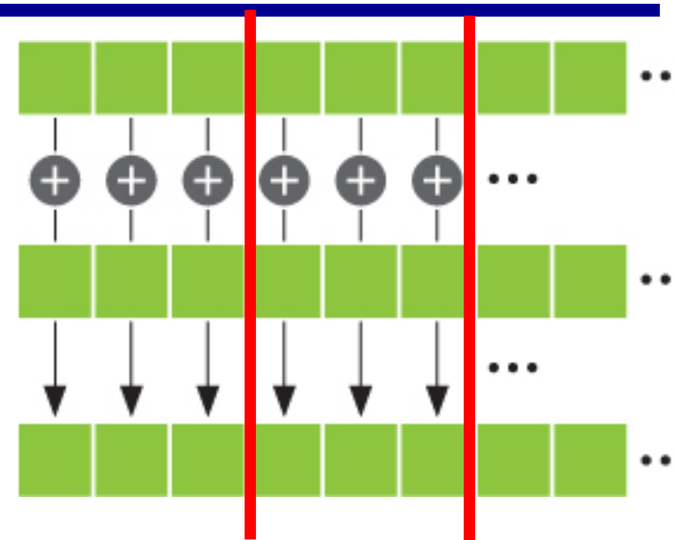
Compiler vs Hand-Written

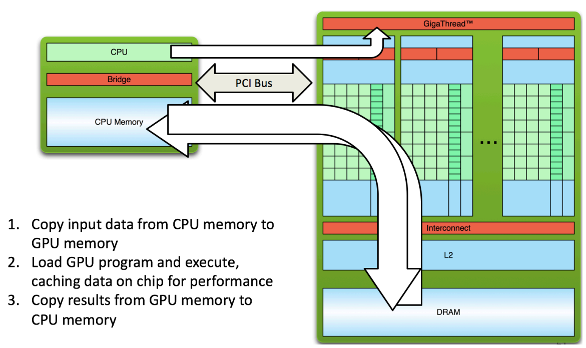
Applications	Domains	OpenACC Directive Combinations	Lines of Code Added vs Serial		Speedup Over		
			OpenMP	OpenACC	Seq	OpenMP	CUDA
Needleman-Wunsch	Bioinformatics	data copy, copyin kernels present loop gang, vector, private	6	5	2.98	1.28	0.24
Stencil	Cellular Automation	data copyin, copy, deviceptr kernels present loop collapse, independent	1	3	40.55	15.87	0.92
Computational Fluid Dyanmics (CFD)	Fluid Mechanics	data copyin, copy, deviceptr data present, deviceptr kernels deviceptr kernels loop, gang, vector, private loop gang, vector acc_malloc(), acc_free()	8	46	35.86	4.59	0.38
2D Heat (grid size 4096*4096)	Heat Conduction	data copyin, copy, deviceptr kernels present loop collapse, independent	1	3	99.52	28.63	0.90
Clever (10Ovals)	Data Mining	data copyin kernels present, create, copyin, copy loop independent	10	3	4.25	1.22	0.60
FeldKemp (FDK)	Image Processing	kernels copyin, copyout loop, collapse, independent	1	2	48.30	6.51	0.75

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(tofrom: 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];
  }
}
```





- Parallel region
  - One CPU thread per device
- Manually partition array x and y
- Each thread offload subregion of x and y
- Chunk the loop in alignment with the data partition

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

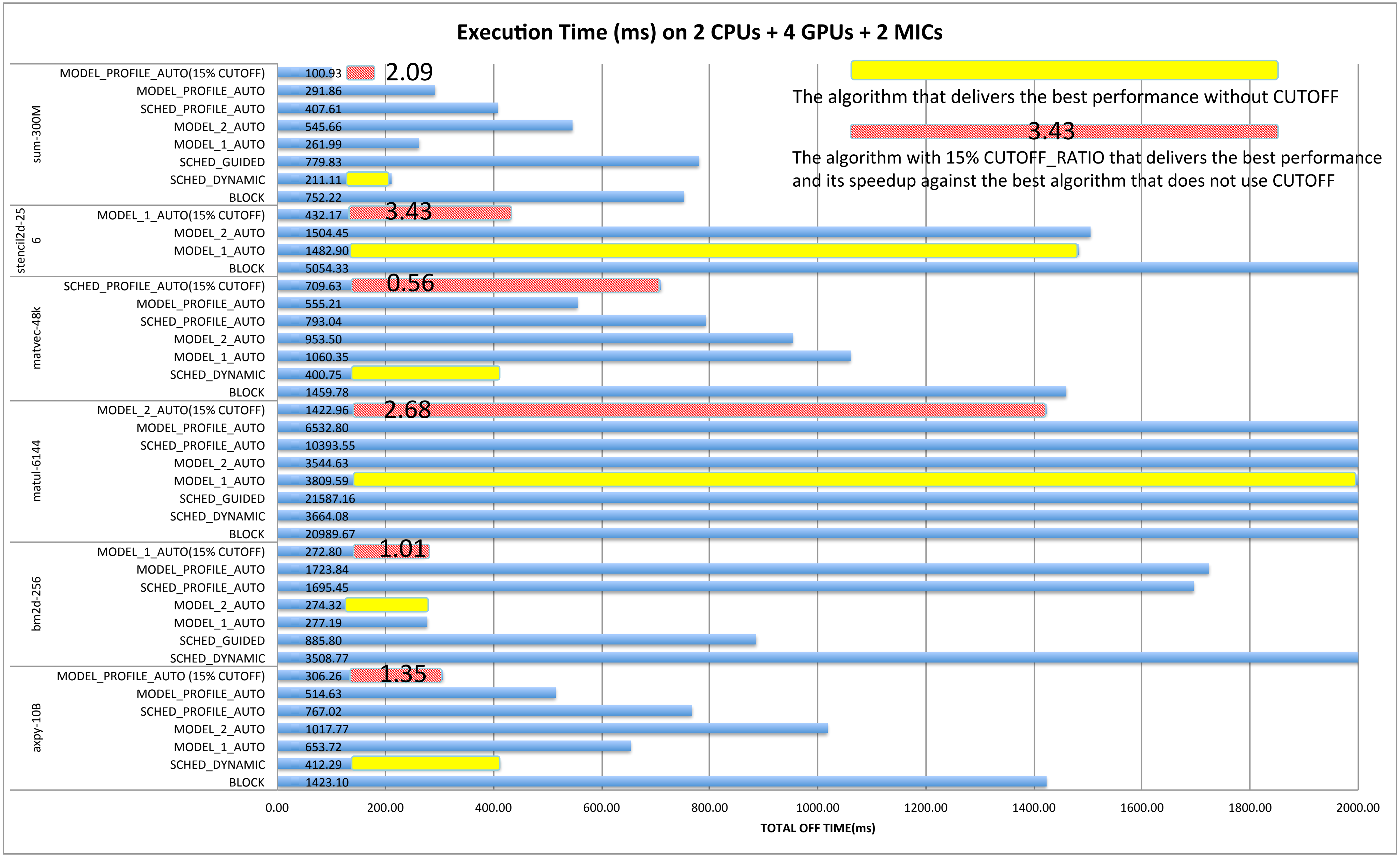
Three Challenges to Implement HOMP

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



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%)

Benchmarks	**Devices used**	**CUTOFF Speedup**
axpy-10B	2 CPU + 4 GPUs	1.35
bm2d-256	2 CPU + 4 GPUs	1.01
matul-6144	4 GPUs	2.68
matvec-48k	**4 GPUs**	0.56
stencil2d-256	4 GPUs	3.43
sum-300M	2 CPUs + 4 GPUs	2.09