Concurrent Systems (ComS 527)

Ali Jannesari

Department of Computer Science

Iowa State University, Spring 2023

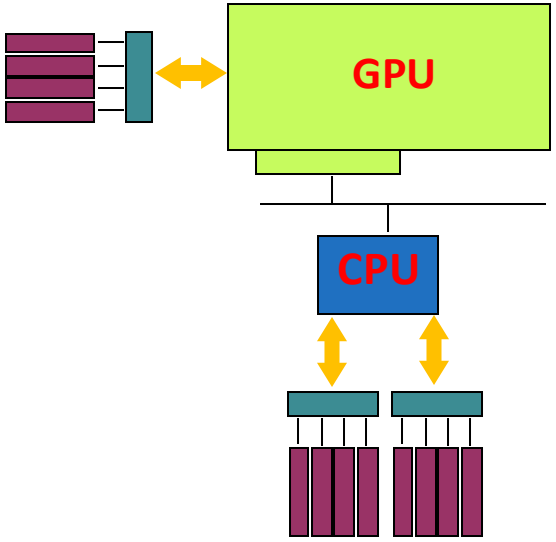
TARGETING GPUS WITH OPENMP

Outline

- Programming Model for GPGPU
- Data Movement
- Device Execution
- Asynchronous Target Execution
- Example – Jacobi Iteration
- Appendix: GCC compiler for OpenMP offloading (self-study)

Programming Heterogeneous Systems

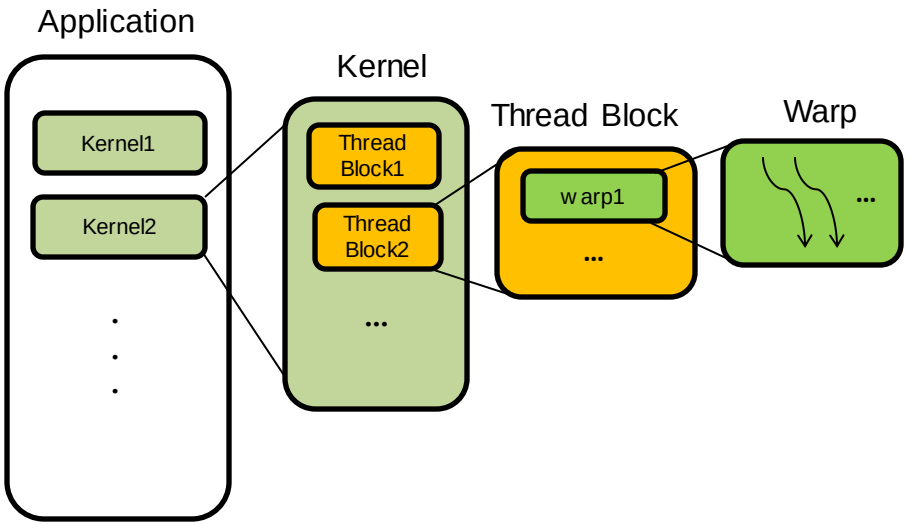
- Modern high-performance applications must exploit **heterogeneous resources** in a performance portable manner



- What Programming Models should we use?
- What's the right level of abstraction?

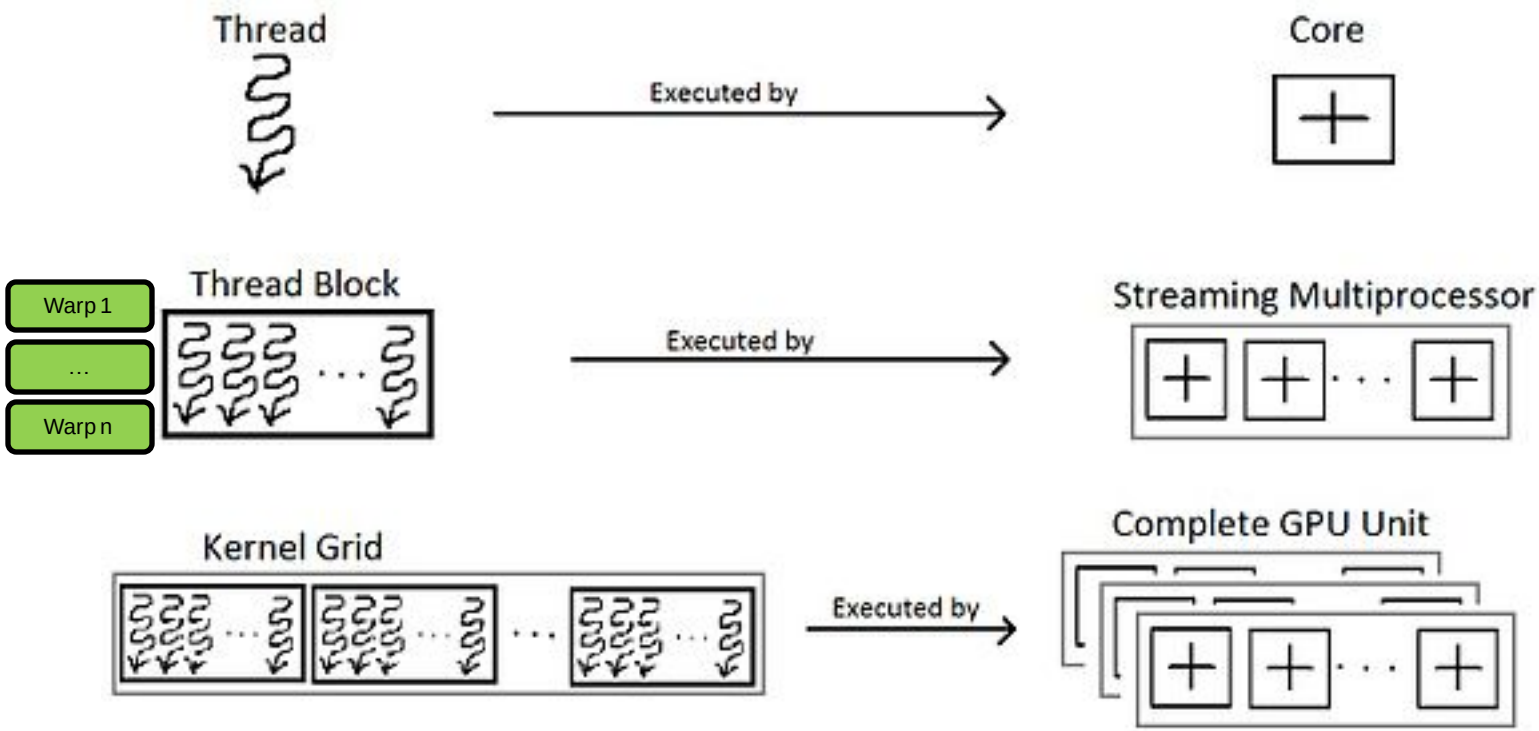
Programming Model for GPGPU

- **Warp**: a group of 32 threads
- Volta GV100, supports up to 64 warps per SM (has 80 SMs)
  - Up to 2048 threads per SM
  - Each kernel is split into groups of threads called **thread blocks** or concurrent thread arrays (CTA).



Programming Model for GPGPU

- A **Thread Block** is a basic workload unit assigned to an SM (streaming Multiprocessor) in a GPU



Programming Heterogeneous Systems

With **OpenMP 4.0/4.5/5.0/5.1/5.2**:

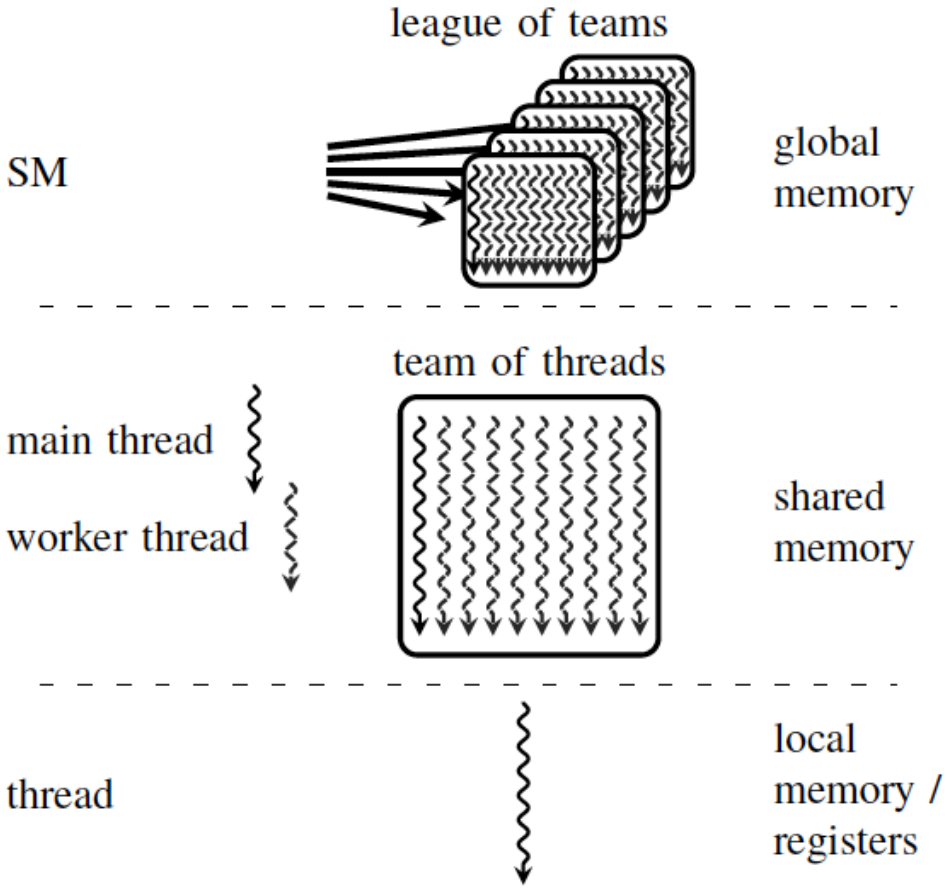
- You can use the same standard to program the GPU, the SIMD units, and the CPU threads
- Better yet: you can do it in a portable way
- New features have been added to provide support for offloading computation to accelerators
- `#pragma omp target`
  - Moves a region of code to the GPU and implicitly maps data

OpenMP GPU Execution Model

Top row: Outermost parallelism across SMs onto which OpenMP teams are mapped. All threads on this level share the global memory.

Middle row: A single SM corresponding to a team of threads in OpenMP. Both shared and global memory are accessible by these threads.

Bottom row: A single GPU/OpenMP thread which has exclusive access to local memory and registers.



SAXPY in OpenMP – CPU

CPU

```
double x[N], y[N], s[N], a; //calculate s[i]=a*x[i]+y[i]

#pragma omp parallel for
for (int i=0; i<N; i++)
       s[i] = a*x[i] + y[i];
```

SAXPY in OpenMP – GPU

GPU

```
double x[N], y[N], s[N], a; //calculate z=a*x+y

#pragma omp target
{
#pragma omp parallel for
for (int i=0; i<N; i++)
        s[i] = a*x[i] + y[i];
}
```

Compiler: Generate code for GPU

Runtime: Run code on device if possible, copy data from/to GPU

- Code is unmodified except for the pragma
- Data is **implicitly** copied
  - Moves this region of code to the GPU and implicitly maps data
- All calculation done on **device**

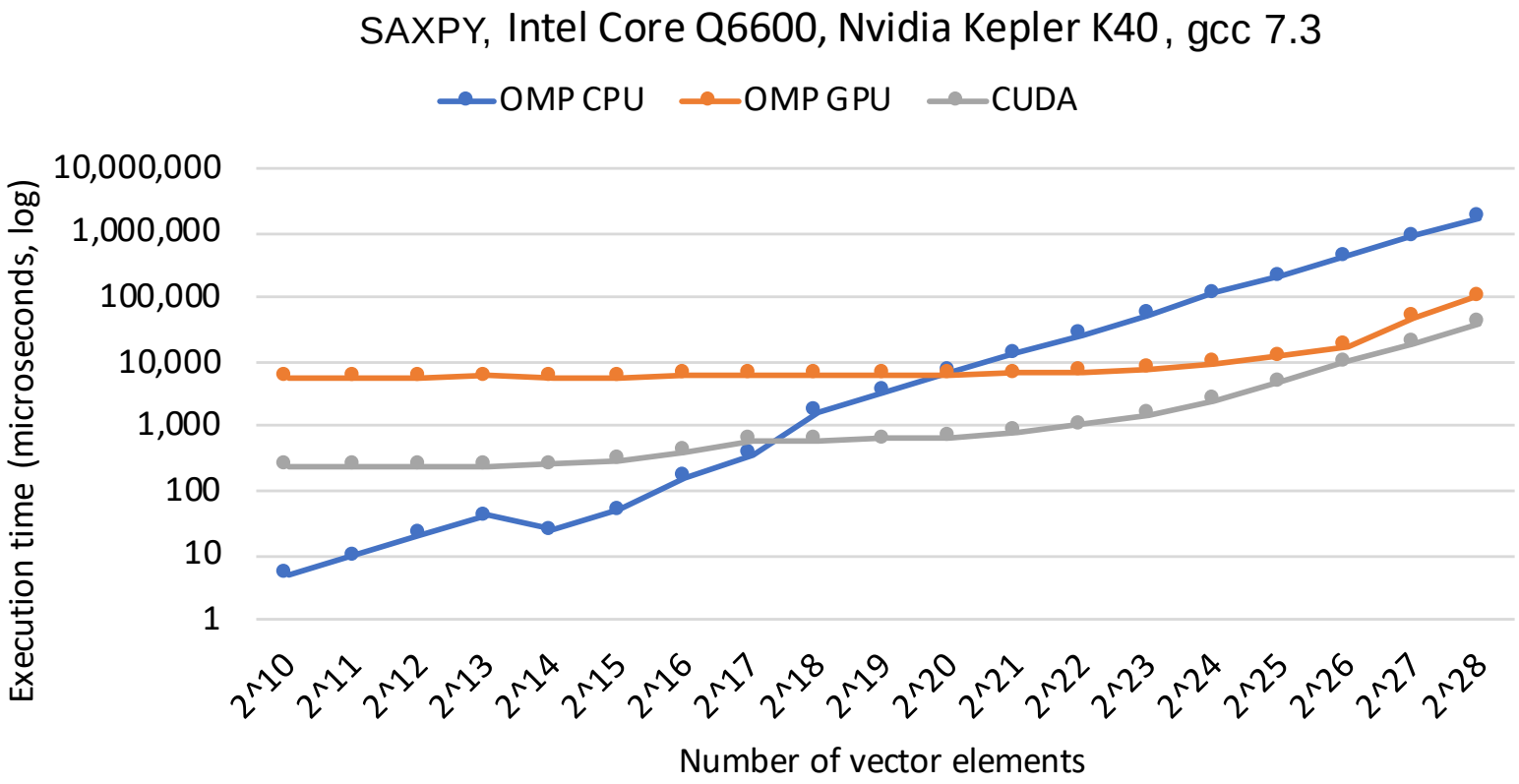
Compiler support

Compiler	OpenMP offload version	Device types
Gcc	4.5	Nvidia GPU, Xeon Phi
Clang	4.5	Nvidia GPU, AMD GPU
Flang	n/a	
icc	4.5	Xeon Phi
Cray cc	4.0	Nvidia GPU
IBM xl	4.5	Nvidia GPU
PGI	n/a	

Limitations (check for updates on OpenMP 5.0/5.1/5.2):
https://www.openmp.org/resources/openmp-compilers-tools/

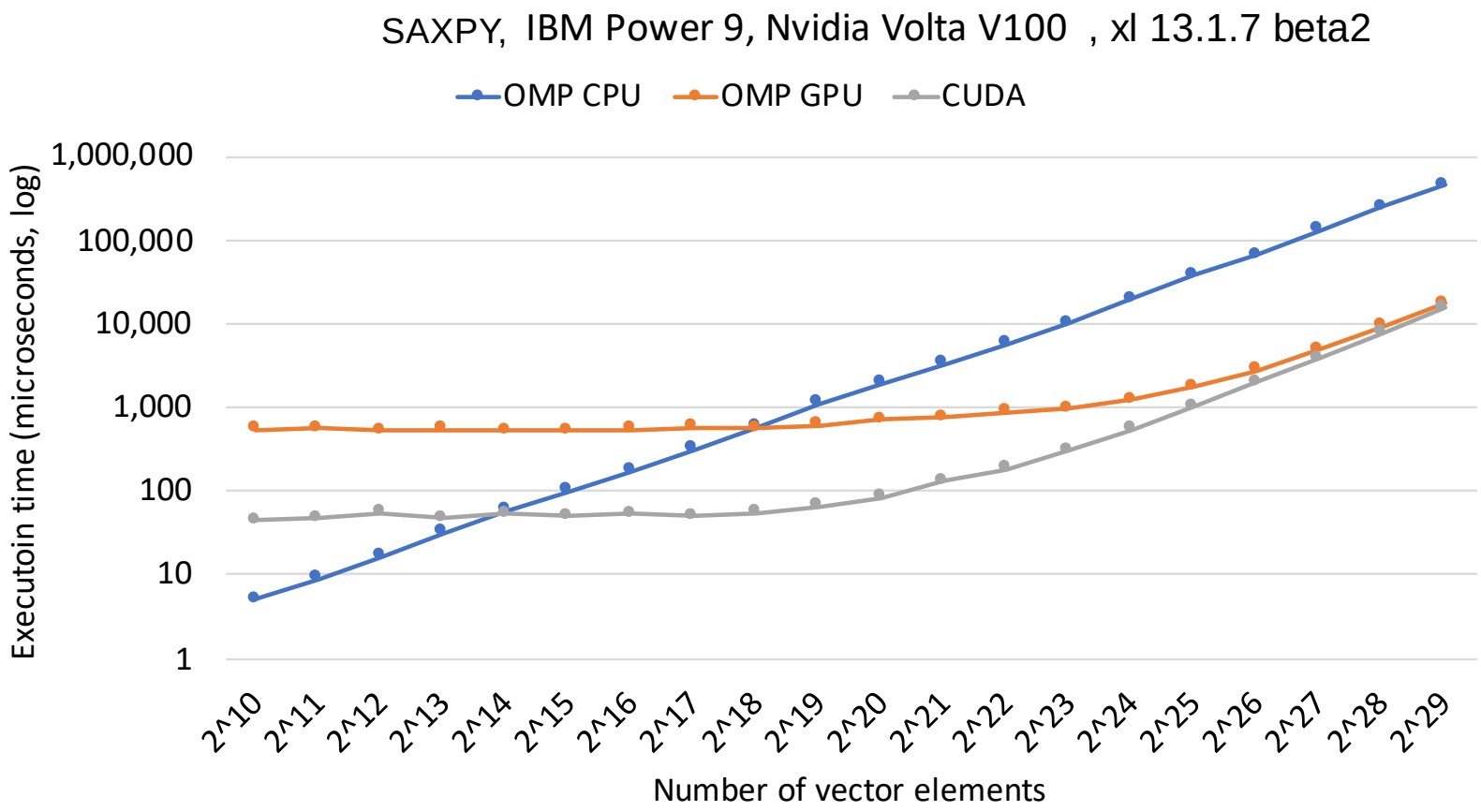
- Static linking only
- Recent linker
- No C++ exceptions
- Not all operations offloadable (e.g., I/O, network, …)

Performance results – K40



Source: https://charm.cs.illinois.edu/workshops/charmWorkshop2018/slides/CharmWorkshop2018_diener.pptx

Performance results – V100



Source: https://charm.cs.illinois.edu/workshops/charmWorkshop2018/slides/CharmWorkshop2018_diener.pptx

How do we exploit an accelerator in OpenMP? (1)

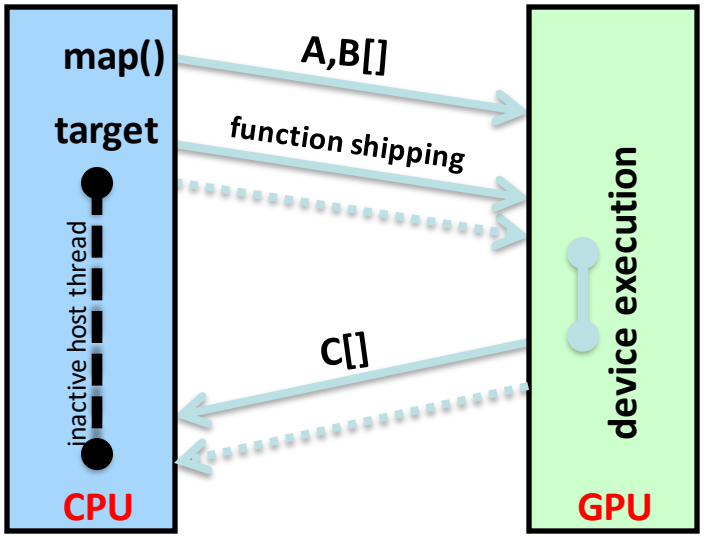
- Simply add a **target** construct around the computation to be offloaded to the accelerator
- **map** clauses are used to copy data; default is "map(to:/from:)"

```
#pragma omp target
    map(to: A, B) map(from: C)
{
    #pragma omp parallel for
    for(i=0; i<N; i++) {
        for(j=0; j<N; j++)
            for(k=0; k<N; k++)
                C[i, j] = A[I, k] * B[k, j];
    }
}
```

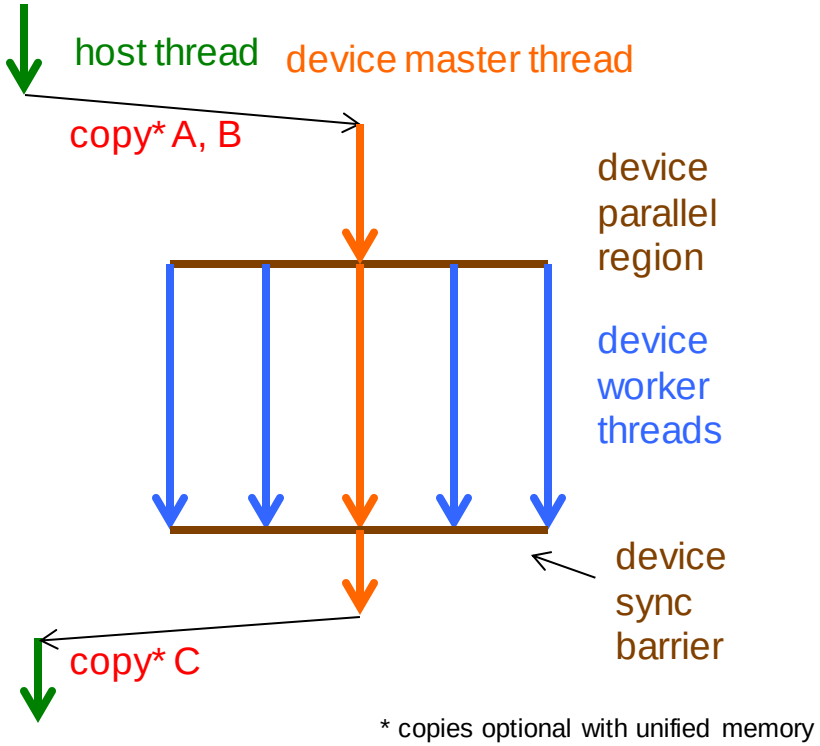
Source: https://www.ibm.com/developerworks/

How do we exploit an accelerator in OpenMP? (2)

- **target** transfer control of execution to a SINGLE device thread
  - Compiler packages the target region into a function
  - OpenMP runtime transfers execution of the function to the device




```
#pragma omp target map(to: A, B)
map(from: C)
{ … }
```

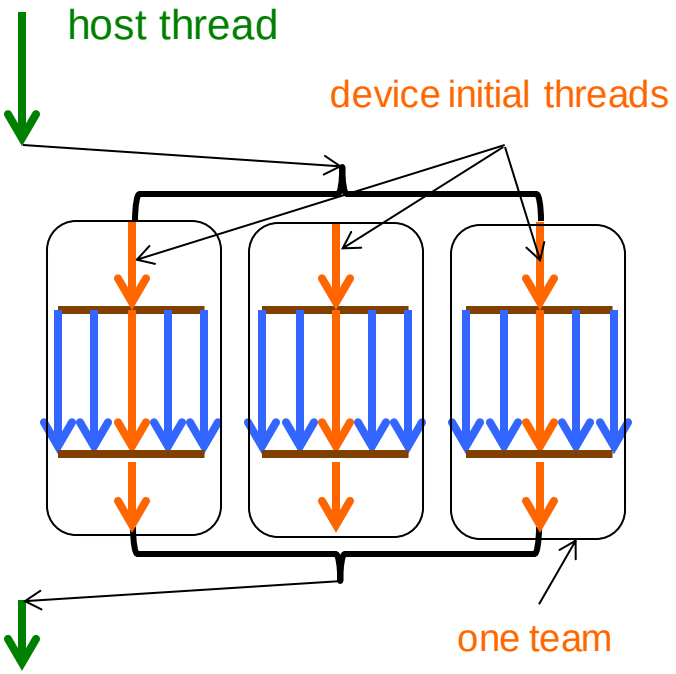


Source: https://www.ibm.com/developerworks/

OpenMP teams: Mapping teams to a thread blocks

- The OpenMP **teams** construct creates a ***league*** **of teams** that execute the target region ➔ each team maps to a Thread Block

```
#pragma omp target map(to: A, B)
map(from: C)
#pragma omp teams
            num_teams(num_teams)
            thread_limit (num_threads)
{
    #pragma omp parallel for
    for(i=0; i<N; i++) {
        for(j=0; j<N; j++)
            for(k=0; k<N; k++)
                C[i, j] += A[i, k] * B[k, j];
    }
}
```



- Each team executes the <u>*same*</u> code
- User can optionally control number of teams/threads via clauses.

OpenMP on GPU vs. CUDA (1)

OpenMP on GPUs…

- leverages every(*) OpenMP construct
- includes parallel regions, parallel loops, tasks, …
- includes fine grain and coarse grain synchronizations *within* one team
  - e.g. locks, critical regions, barriers…
- can have sequential and parallel code
- OpenMP on device: just like on the host (with small number of limitations)
- OpenMP**: programming as usual**

OpenMP on GPU vs. CUDA (2)

Traditional GPU programming models (CUDA, …)

- transfer control to a single “parallel loop”
- no sequential code (e.g. to initialize data serially on GPU)
- SPMD/SIMT programming model: every device thread executes the same program but operates on different data
- CUDA**: rewire everything from scratch**

Example: saxpy – CUDA

Computing z = ax + y with parallel loop

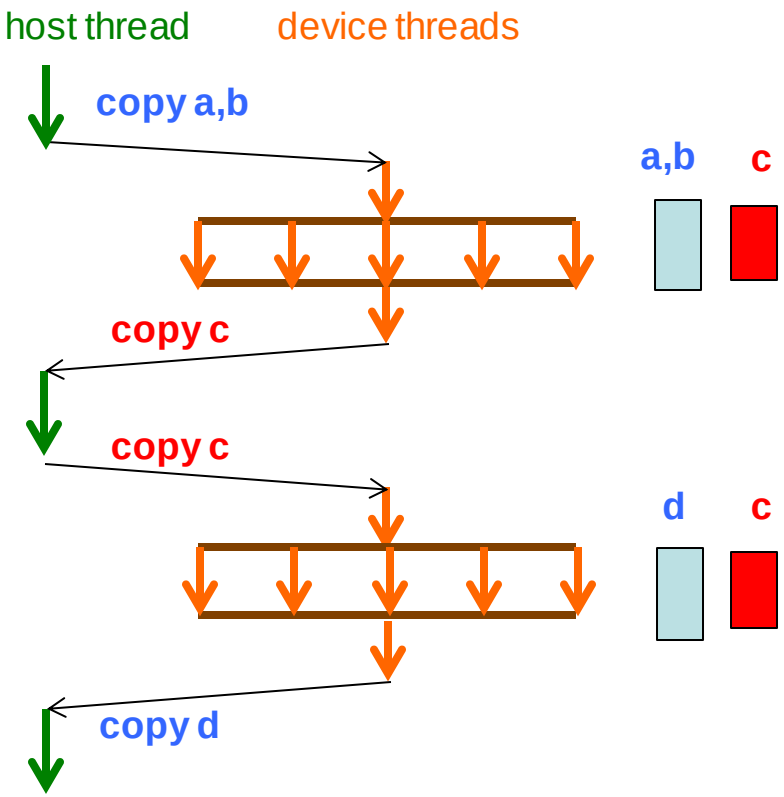
```
__global__
void saxpy_parallel(…)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i<n) z[i]= a * x[i] + y[i];
}
```

```
/* invoke parallel saxpy kernel with n threads */
/* organized in 256 threads per block */
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(…);
```

Controlling Data Movement (1)

- Map clauses control data transfers in/out of target regions
- Data transfer is host centric: map(**to**:…) transfer data to the device

```
#pragma omp target
   map(to: a, b) map(from: c)
{
      // define c in terms of a, b
}

 // sequential code

#pragma omp target
   map(from: d) map(to: c)
{
      // define d in terms of c
}
```



Controlling Data Movement (2)

- Data which is referenced in a target region and is not explicitly mapped is mapped by ***default***

```
int i, data[100];
#pragma omp target
{
      for(i=0; i<100; i++) {
        data[i] = 100;
}
```

- scalar "i" and array "data" are *implicitly* mapped
  - arrays are mapped tofrom: "**map(tofrom: data[100])**"
  - scalars become **firstprivate**: value not updated on exiting the target region (OpenMP 4.5 rules)

Controlling Data Movement (3)

- Each device has a single copy of each mapped variable (including the host device)
- The data required to execute a target region is mapped each time the region runs
  - This is often **undesirable** (data transfer can be expensive)
- **target [enter|exit] data** can be used to persist data on the target device
- map clauses are ***ignored*** when data is already in device scope (reference counted)

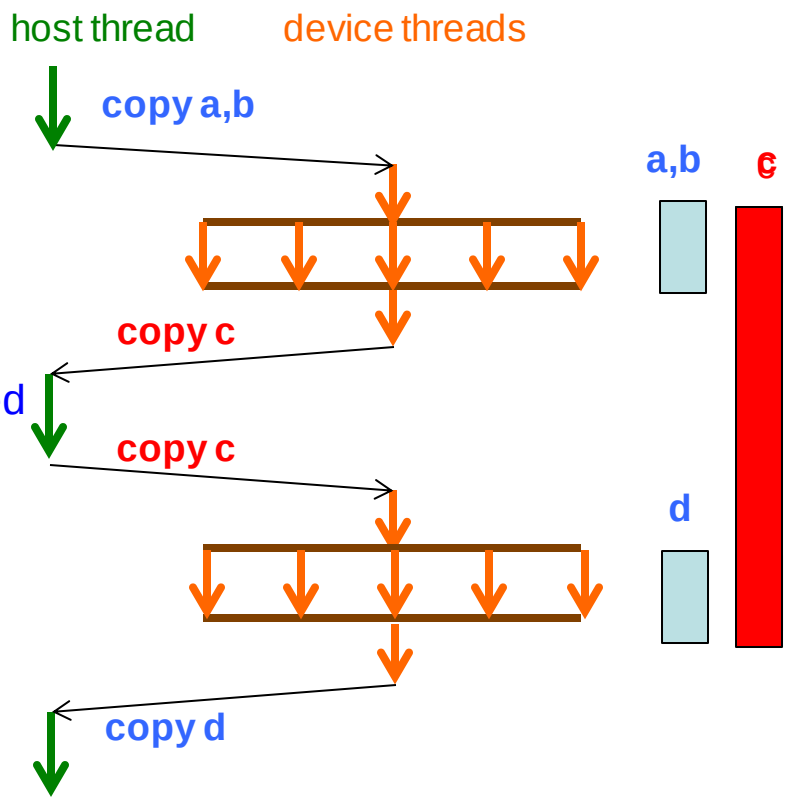
Controlling Data Movement (4)

- **target data** is used to limit unnecessary data transfers

Allocate c on the device

```
#pragma omp target data
map(alloc: c)
{
    #pragma omp target
      map(to: a, b) map(from: c)
    {
        // define c in terms of a, b
    }

    #pragma omp target
      map(from: d) map(to: c)
    {
        // define d in terms of c
    }
}
```

Ignored



Controlling Data Movement (5)

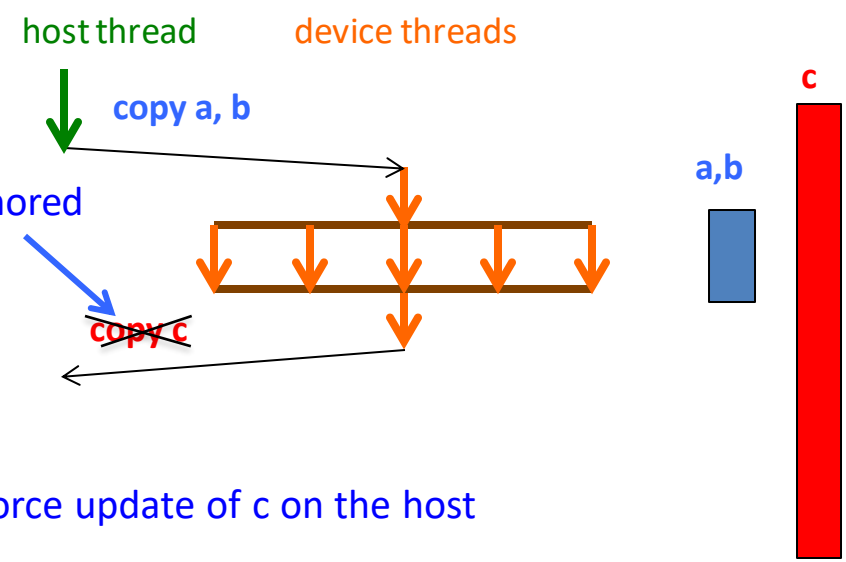
- **target update** is used to force data transfer to/from host
  - #pragma omp target update **from**(c) ➔ host copy updated
  - #pragma omp target update **to**(c) ➔ device copy updated

Allocate c on the device data environment

```
#pragma omp target data map(alloc: c)
{
    #pragma omp target
      map(to: a, b)  map(from:c)
    {
        // define c in terms of a, b
    }

    #pragma omp target update from(c)

    // use updated value of c on the host
}
```

Ignored

Force update of c on the host



Device Function and Data

- The **declare target** directive provides a way to declare functions on target devices and to install globally accessible data

```
#pragma omp declare target
int data[100];

void init(int N) {
    for(int  i=0; i < N; i++)
      data[i] = i;
}
#pragma omp end declare target

#pragma omp target
{
     init(100);
}
```

Allocate data on the device for the duration of the program

Create a function that can be called from either the host or the device execution environment

Call a device function from (within) a target region (i.e. a GPU kernel)

Recap – Managing data on the device (1)

- Control data transfer to/from a target region (i.e. GPU kernel):
  - **“#pragma omp target map(x) {…}”**
  - mapped data is valid for the execution of the target region
- Limit unnecessary data transfers, reuse data between target regions:
  - **“#pragma omp target data map(x) {…}”**
  - Maps data in the given structured scope
- Unstructured scopes: target enter/exit [OpenMP 4.5]
  - “**pragma omp target enter/exit data map(x)**”
  - Maps/Unmaps data at the point where the directive is encountered

Recap – Managing data on the device (2)

- Declare global data on the device
  - **“#pragma omp declare target to(x) {…}”**
  - Data available on the device for the duration of the program
  - User can update device/host copies via the **“target update”** directive

Device Execution (1)

- target regions are executed on the device (synchronously) by a single team
- teams directive can be used to execute target region on multiple teams
- User may control number of teams and threads per team via clauses
- Parallel regions can appear inside a target team region
- **How do we distribute the computation among teams?**

Device Execution (2)

- **How do we distribute the computation among teams?**

```
#pragma omp target map(to: A, B)
map(from: C)
#pragma omp teams
{
      // sequential code
    #pragma omp parallel for
    for(i=0; i<N; i++) {
        for(j=0; j<N; j++)
            for(k=0; k<N; k++)
                C[i, j] += A[i, k] * B[k, j];
    }
}
```

One thread per team (master) execute sequential region

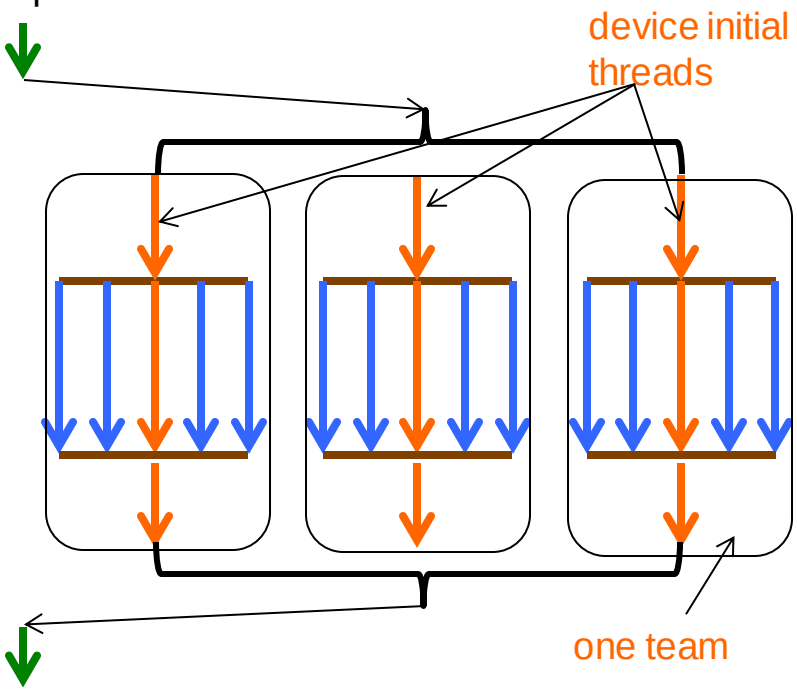
Parallel loop executed by parallel threads (in a team)

Parallel loop executed redundantly by every team!

Device Execution (3)

- The **"distribute"** directive can be used to assign loop iterations to teams

```
#pragma omp target teams
   map(to: a, b) map(from: c)
{
    #pragma omp distribute
    for(int i=0; i<n; i++) {
        #pragma omp parallel for
        for(int j=0; j<n; j++)
            for(k=0; k<n; k++)
                c[i, j] = a[I, k] * b[k, j];
    }
}
```



- the target region is executed by several teams, each team gets a subset of iteration space for the **i-loop**
- the **j-loop** iterations are distributed among the threads in a team
- distribute schedule controls size of iterations per team, **there is no synchronization between teams**

Optimization: omp distribute parallel for (1)

#pragma omp distribute parallel for

Programming model: OpenMP vs CUDA

- OpenMP uses a fork-join abstraction
- team regions start with one thread, and parallel threads are created as needed when a parallel region is found
- CUDA kernels are launched using a grid of blocks/threads (SPMD model)
- Orchestrating CUDA threads to fit the OpenMP programming model can have significant overhead (runtime manages state transitions)

Optimization: omp distribute parallel for (2)

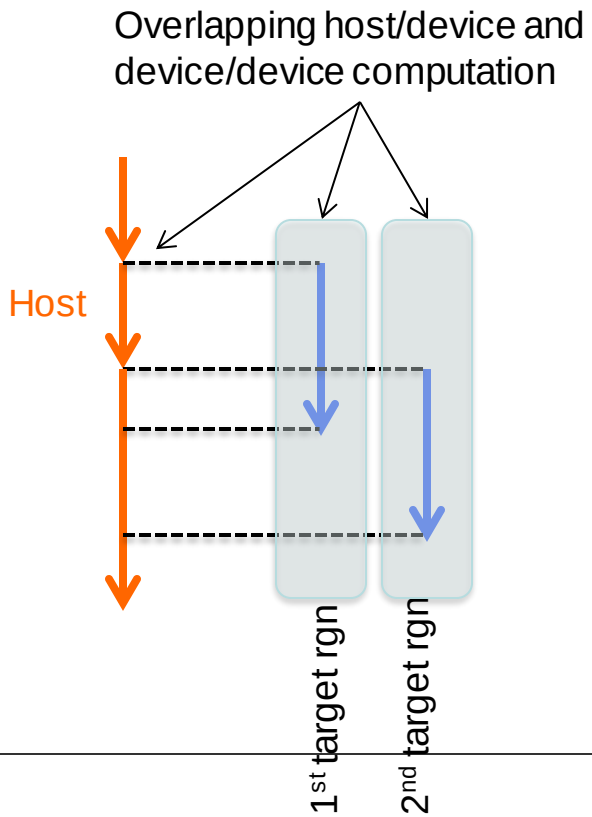
- However, OpenMP provides “SPMD-like” directives
- **distribute parallel for** directive can be used to distribute loop iterations among teams and then execute those iterations in parallel using the threads in each team
- Compiler can generate efficient GPU code for this construct (state transitions not required ➔ bypass OpenMP runtime system)
- Default schedule recommended to maximize performance portability
- HW coalescing on GPU, good cache locality on CPU

```
#pragma omp target map(from: z)
                   map(to:x,y)
#pragma omp teams
#pragma omp distribute parallel for
  for (i=0; i<N; i++)
    z[i] = a*x[i] + y[i];
```

Asynchronous target execution

- Target regions can be executed asynchronously with respect to the host (**nowait** clause)
- Target regions can be dispatched to different accelerators (**device** clause)

```
#pragma omp target device(1)
nowait
{
    …
}

// host computation

#pragma omp target device(2)
nowait
{
    …
}
```

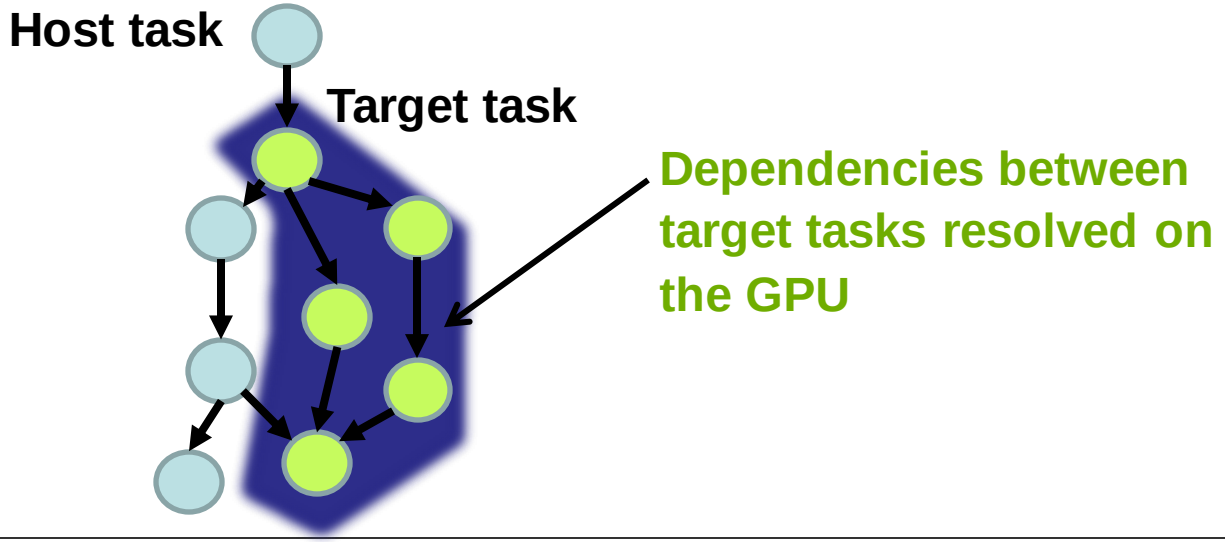


Concurrency in a node

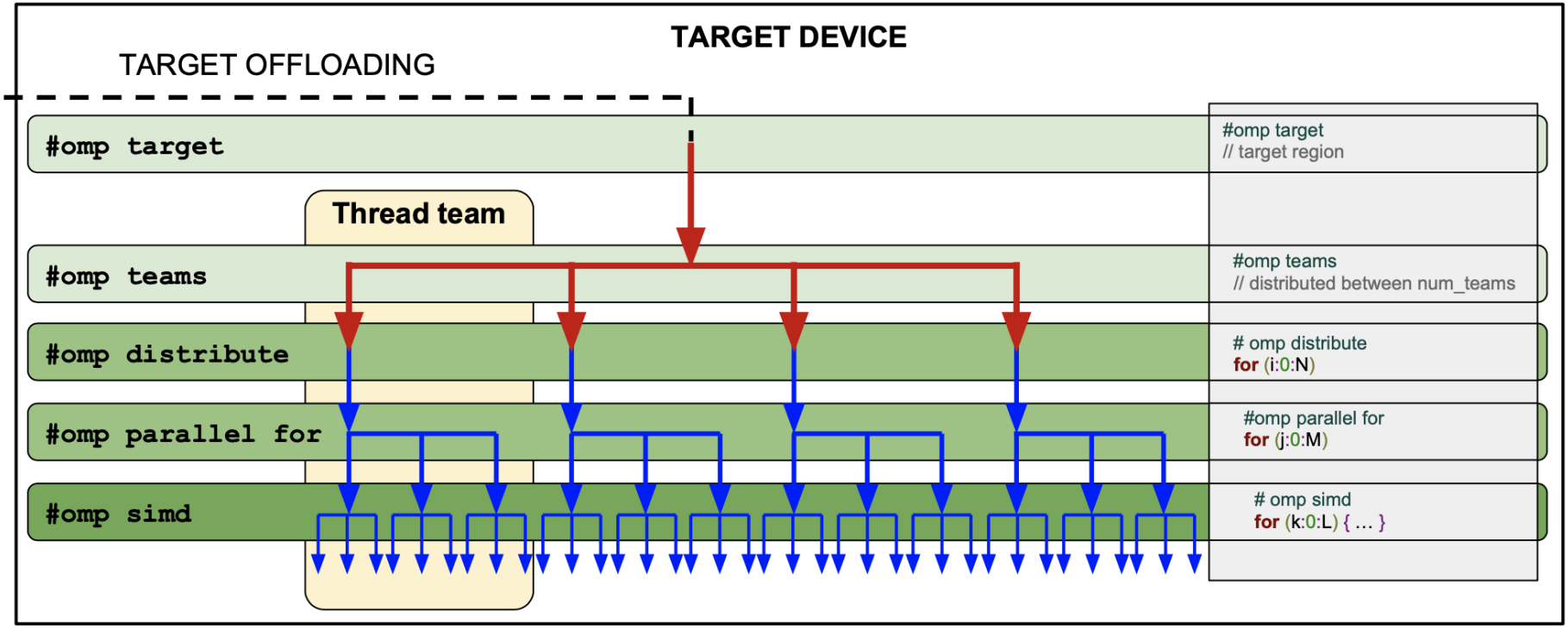
- Asynchronous target regions may have dependencies (**depend** clause)

```
#pragma omp target nowait depend(in:v1,v2) depend(out:p)
map(to:v1,v2) map( from: p)
```

- Can overlap host/device data transfers and execution
- Dependencies on a target region may be enforced on the device (no host intervention)



OpenMP execution model (Offloading)



Source: https://www.researchgate.net/

Example – Jacobi Iteration

Jacobi Iteration

```
while ( err > tol && iter < iter_max ) {
  err=0.0;


  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }


  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Convergence Loop

Calculate Next

Exchange Values

http://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf

Example – Jacobi Iteration

CPU-Parallelism

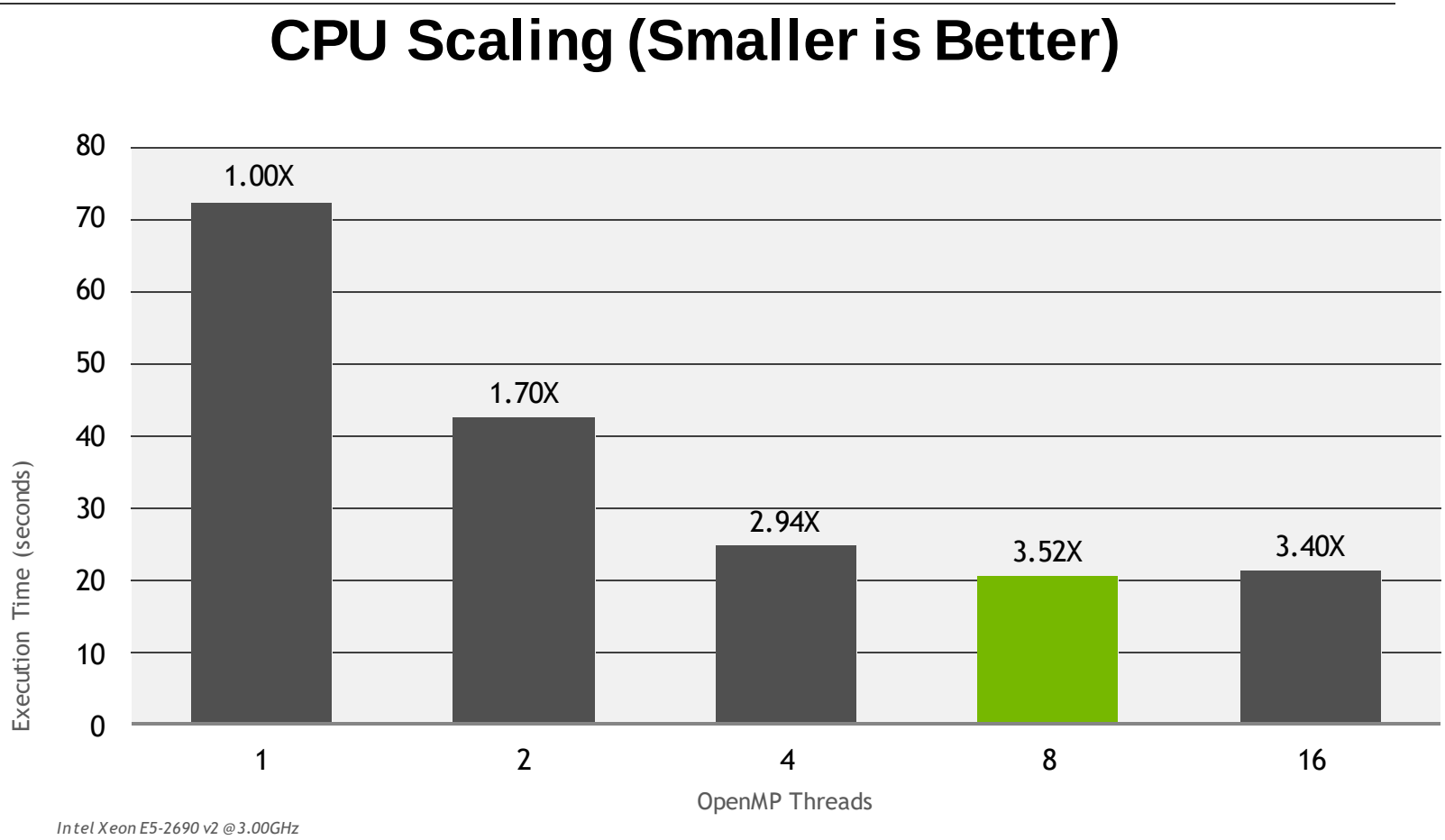
```
while ( error > tol && iter < iter_max )
{
    error = 0.0;

#pragma omp parallel
{
#pragma omp for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }
#pragma omp barrier
#pragma omp for
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Create a team of threads

Workshare this loop

Prevent threads from executing the second loop nest until the first completes

Example – Jacobi Iteration



Example – Jacobi Iteration

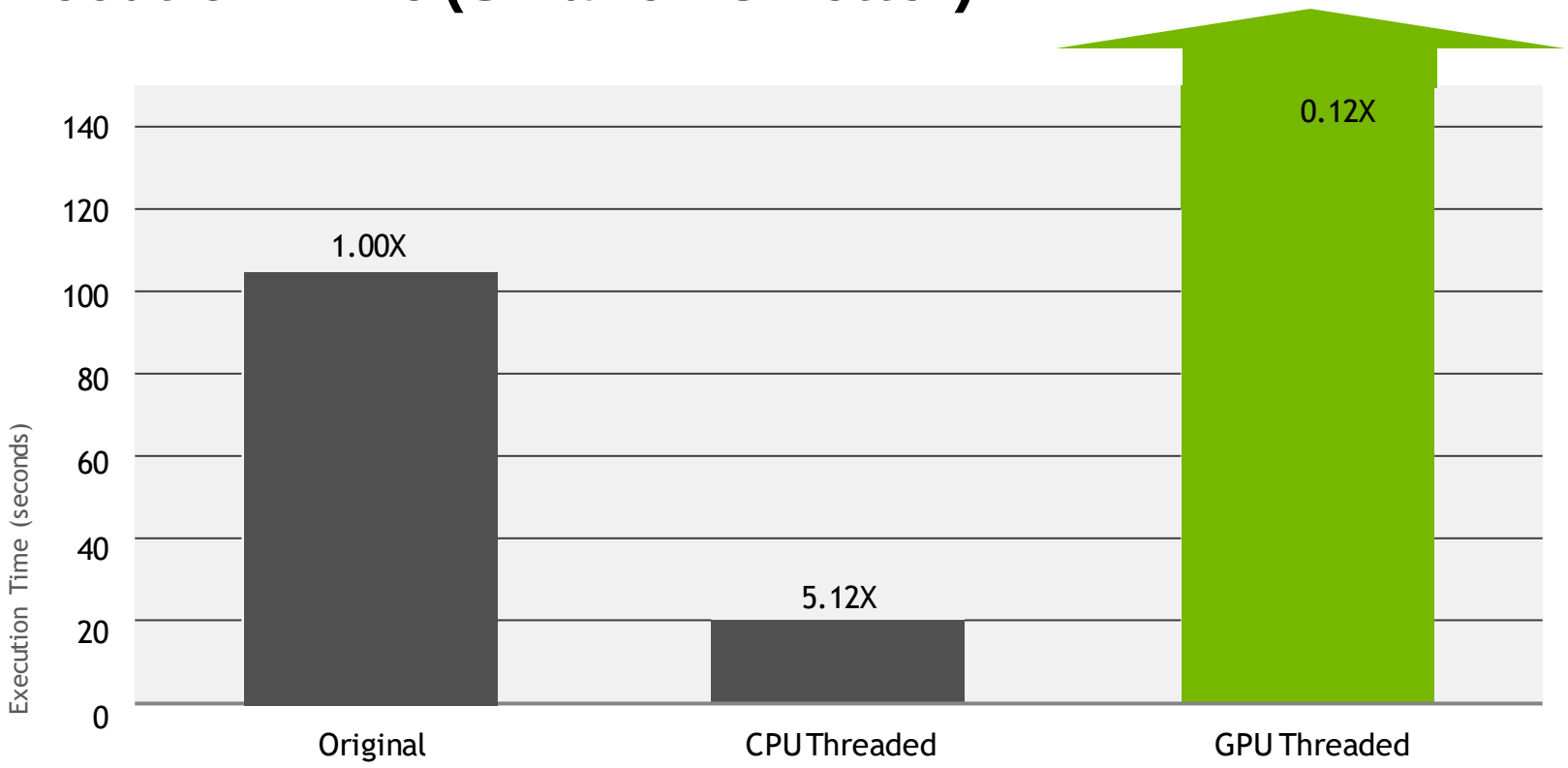
Target the GPU

```
while ( error > tol && iter < iter_max )
{
    error = 0.0;
#pragma omp target map(alloc:Anew[:n+2][:m+2]) map(tofrom:A[:n+2][:m+2])
{
#pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                  + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp parallel for
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Moves this region of code to the GPU and explicitly maps data.

Example – Jacobi Iteration

Execution Time (Smaller is Better)



NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @3.00GHz

Example – Jacobi Iteration

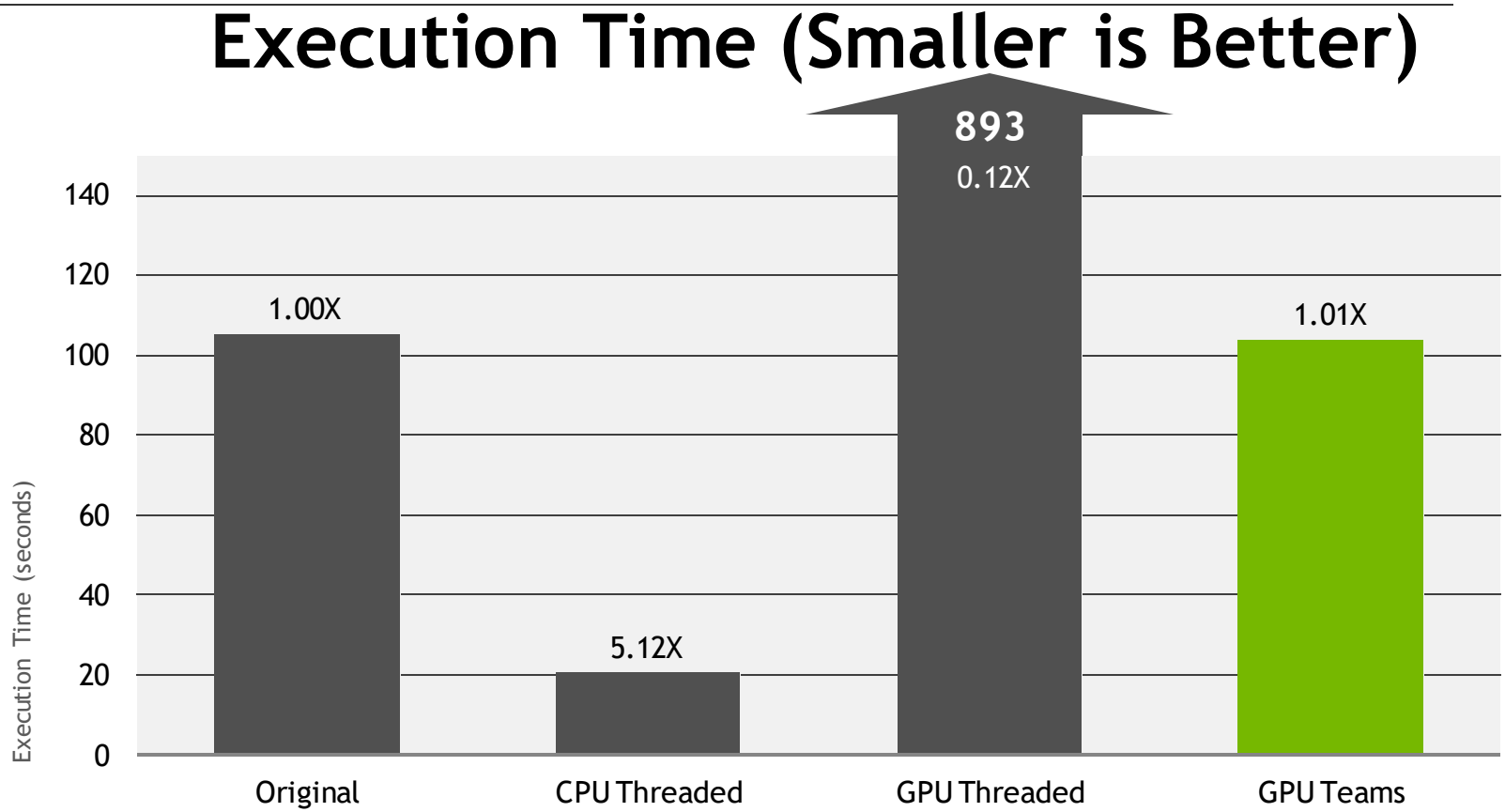
Teaming Up

```
#pragma omp target data map(alloc:Anew) map(A)
    while ( error > tol && iter < iter_max )
    {
        error = 0.0;

#pragma omp target teams distribute parallel for reduction(max:error)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute parallel for
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

        iter++;
    }
```

Explicitly maps arrays for the entire while loop.

- Spawns thread teams
- Distributes iterations to those teams
- Workshares within those teams.

Example – Jacobi Iteration



NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @3.00GHz

32

Example – Jacobi Iteration

Increasing Parallelism

Currently both our distributed and workshared parallelism comes from the same loop.

- We could move the PARALLEL to the inner loop
- We could collapse them together

The COLLAPSE(N) clause

- Turns the next N loops into one, linearized loop.
- This will give us more parallelism to distribute, if we so choose.

Example – Jacobi Iteration

Splitting Teams & Parallel

```
#pragma omp target teams distribute
        for( int j = 1; j < n-1; j++)
        {
#pragma omp parallel for reduction(max:error)
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute
        for( int j = 1; j < n-1; j++)
        {
#pragma omp parallel for
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

Distribute the “j” loop over teams.

Workshare the “i” loop over threads.

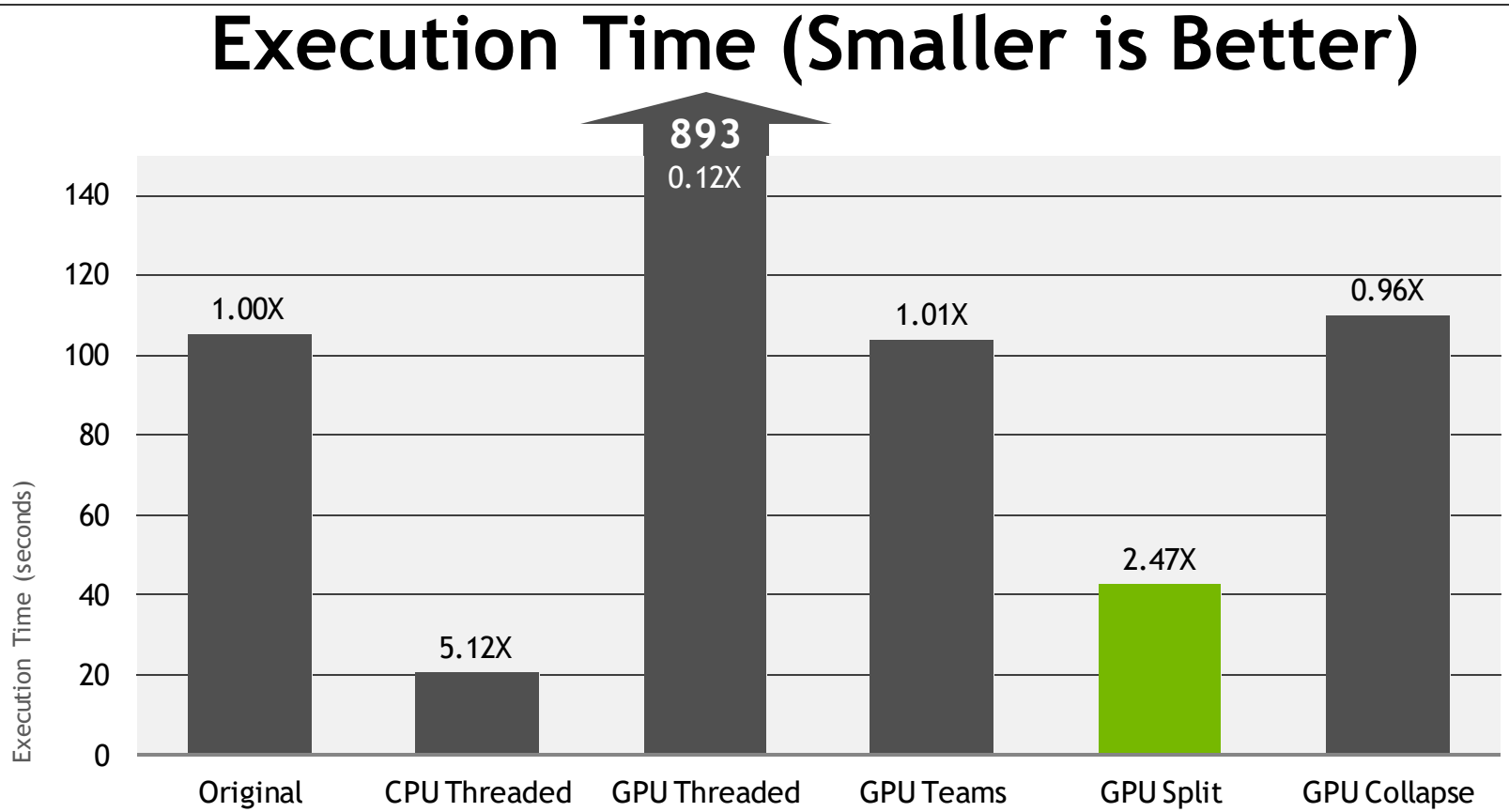
Example – Jacobi Iteration

Collapse

```
#pragma omp target teams distribute parallel for reduction(max:error) collapse(2)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute parallel for collapse(2)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

Collapse the two loops into one.

Example – Jacobi Iteration



NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @3.00GHz

Example – Jacobi Iteration

Improve Loop Scheduling

- Most OpenMP compilers will apply a static schedule to workshared loops, assigning iterations in *N / num_threads* chunks
  - Each thread will execute contiguous loop iterations, which is very cache & SIMD friendly
  - This is great on CPUs, but not efficient on GPUs
- The **SCHEDULE()** clause can be used to adjust how loop iterations are scheduled

39

Example – Jacobi Iteration

Improved Schedule (Split)

```
#pragma omp target teams distribute
        for( int j = 1; j < n-1; j++)
        {
#pragma omp parallel for reduction(max:error) schedule(static,1)
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute
        for( int j = 1; j < n-1; j++)
        {
#pragma omp parallel for schedule(static,1)
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

Assign adjacent threads adjacent loop iterations.

Example – Jacobi Iteration

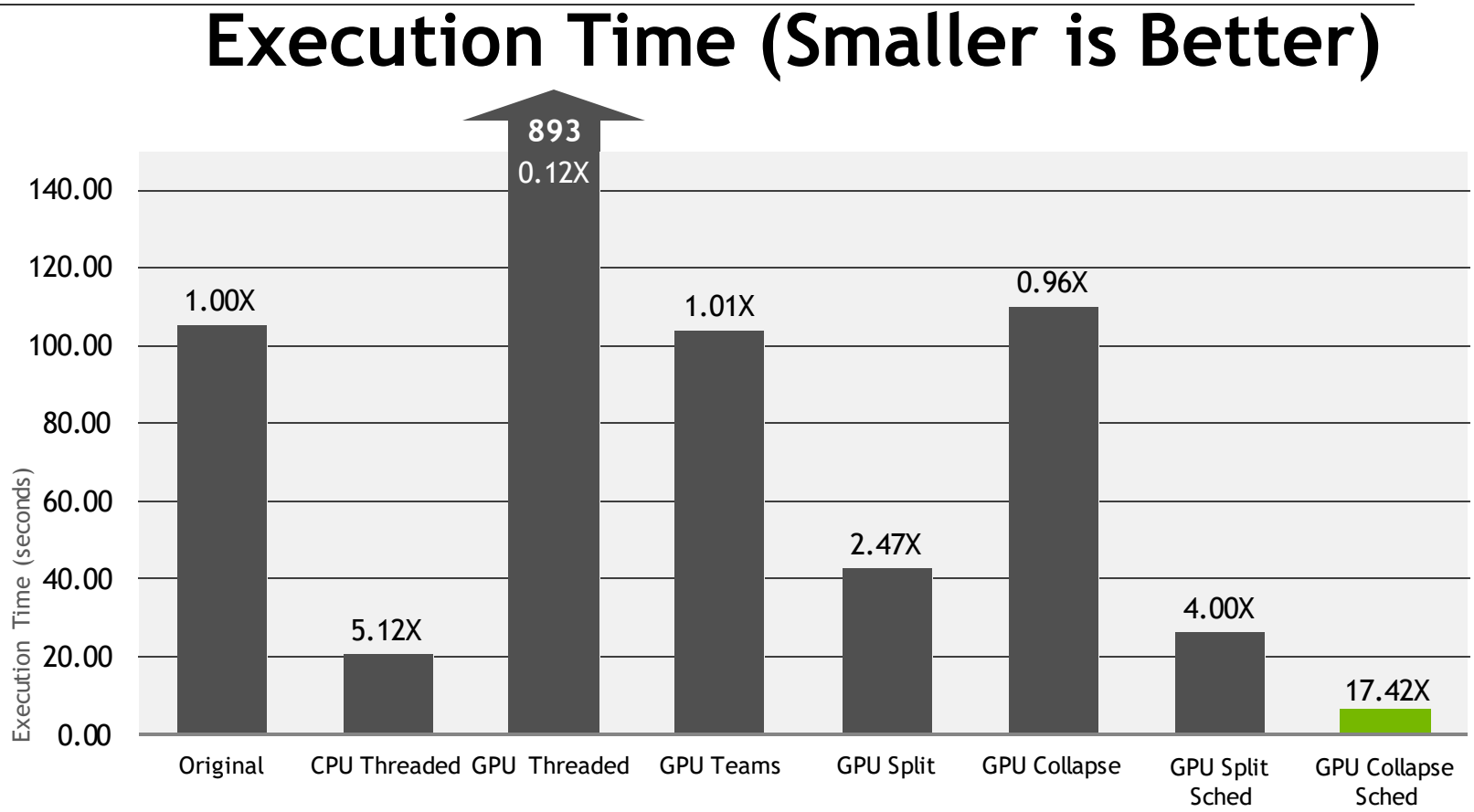
Improved Schedule (Collapse)

```
#pragma omp target teams distribute parallel for \
  reduction(max:error) collapse(2) schedule(static,1)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute parallel for \
  collapse(2) schedule(static,1)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

Assign adjacent threads adjacent loop iterations.

42

Example – Jacobi Iteration



NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @3.00GHz

43

OpenMP Target – Summary

Device:

- An implementation-defined (logical) execution unit (or accelerator)

- Device data environment
  - Storage associated with the device
- The execution model is host-centric (or initial device)
  - Host creates/destroys data environment on device(s)
  - Host maps data to the device(s) data environment
  - Host offloads OpenMP target regions to target device(s)
  - Host updates the data between the host and device(s)

OpenMP 4.5 Device Constructs – Summary

- Execute code on a target device
  - **omp target** *[clause[[,] clause],…]*

    structured-block
  - **omp declare target**

    [function-definitions-or-declarations]
- Manage the device data environment
  - **map** ([map-type:] list)

    map-type := alloc | tofrom | to | from | release | delete
  - **omp target data** [clause[[,] clause], …]

    structured-block
  - **omp target enter/exit data** [clause[[,] clause], …]
  - **omp target update** [clause[[,] clause],…]
  - **omp declare target** *[variable-definitions-or-declarations]*

OpenMP 4.5 Device Constructs – Summary

- Parallelism & Workshare for devices
  - **omp teams** *[clause[[,] clause],…]*

    structured-block
  - **omp distribute** *[clause[[,] clause],…]*

    for-loops
- Device Runtime Support
  - void **omp_set_default_device**(int dev_num)
  - int **omp_get_default_device**(void)
  - int **omp_get _num_devices**(void)
  - int **omp_get_team_num**(void)
  - int **omp_is_initial_device**(void)
  - …
- Environment variables
  - **OMP_DEFAULT_DEVICE**
  - **OMP_THREAD_LIMIT**

OpenMP 4.0 & 4.5 offloading features

Features	OpenMP 3.1 (in target region)	OpenMP 4.0	OpenMP 4.5
OpenMP Directive	***Parallel Construct*** • omp parallel • omp sections • *parallel workshare* ***Worksharing*** • parallel do/for • omp ordered • omp single ***Synchronization*** • omp master • omp critical • omp barrier • omp atomic • omp flush	***Device Constructs*** • omp target data • omp target • omp target update • omp declare target • omp teams • omp distribute • omp distribute parallel for • omp declare target • combined constructs ***SIMD Constructs*** • omp loop simd • Omp distribute parallel do simd • *omp simd* • *omp declare simd* • *omp distribute simd*	***Offloading Enhancements*** • First private, private, default map • map changes (4.5 semantics) • if clause for combined directives • implicit firstprivate (4.5) • omp target enter data • omp target exit data • omp target parallel • *target nowait & depend* • *omp target simd* **[*italic means in progress*]**

Check for updates: OpenMP 5.0/5.1/5.2 specification at www.openmp.org

Summary

- OpenMP 4.0 and 4.5 (and 5.0/5.1/5.2) have added support for offloading computation to target devices (e.g. accelerators)
- Specification is target architecture agnostic

  ➔ good program portability
- Allow good exploitation of NVIDIA GPUs compute capabilities
- Asynchronous execution model provides several opportunity for concurrency exploitation at the node level
- OpenMP 5.0/5.1/5.2 added new features such as deep copy,…
  - Deep copy: compiler correctly maps complex (pointer-based) data
- OpenMP 6.0 in Nov 2023

References

- https://waccpd.org/2017/wp-content/uploads/2017/11/2017-Evaluation-Asynchronous-Offloading-Models.pdf
- https://www.openmp.org/wp-content/uploads/SC17-OpenMPBooth_jlarkin.pdf
- https://www.exascaleproject.org/wp-content/uploads/2017/05/OpenMP-4.5-and-Beyond-SOLLVE-part-21.pdf
- https://www.openmp.org/wp-content/uploads/Rakesh_intel_cmplr_offload.pdf
- https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.Examples.pdf
- http://prace.it4i.cz/sites/prace.it4i.cz/files/files/phi-02-2018-06_openmp_0.pdf

Building GCC with support for NVIDIA PTX offloading

GCC COMPILER FOR OPENMP OFFLOADING

The installation script

```
#!/bin/sh
# Build GCC with support for offloading to NVIDIA GPUs.
work_dir=$HOME/offload/wrk
install_dir=$HOME/offload/install
# Location of the installed CUDA toolkit
cuda=/usr/local/cuda
# Build assembler and linking tools
mkdir -p $work_dir
cd $work_dir
git clone https://github.com/MentorEmbedded/nvptx-tools
cd nvptx-tools
./configure \
  --with-cuda-driver-include=$cuda/include \
  --with-cuda-driver-lib=$cuda/lib64 \
  --prefix=$install_dir
make
make install
cd ..
```

The installation script - continue

```
# Set up the GCC source tree
git clone https://github.com/MentorEmbedded/nvptx-newlib
svn co svn://gcc.gnu.org/svn/gcc/tags/gcc_7_2_0_release  gcc
cd gcc
contrib/download_prerequisites
ln -s ../nvptx-newlib/newlib newlib
cd ..
target=$(gcc/config.guess)
```

The installation script - continue

```
# Build nvptx GCC
mkdir build-nvptx-gcc
cd build-nvptx-gcc
../gcc/configure \
  --target=nvptx-none --with-build-time-tools=$install_dir/nvptx-none/bin \
  --enable-as-accelerator-for=$target \
  --disable-sjlj-exceptions \
  --enable-newlib-io-long-long \
  --enable-languages="c,c++,fortran,lto" \
  --prefix=$install_dir
make -j4
make install
cd ..
```

The installation script -conitnue

```
# Build host GCC
mkdir build-host-gcc
cd  build-host-gcc
../gcc/configure \
  --enable-offload-targets=nvptx-none \
  --with-cuda-driver-include=$cuda/include \
  --with-cuda-driver-lib=$cuda/lib64 \
  --disable-bootstrap \
  --disable-multilib \
  --enable-languages="c,c++,fortran,lto" \
  --prefix=$install_dir
make -j4
make install
cd ..
```

GPU server

Since it is downloaded in the server, we just need to do the following

```
module purge
module use /opt/local/spack-modules/lmod/linux-rhel7-x86_64/Core/
 module load gcc
 ml gcc
```

Compile

```
gcc -fopenmp -foffload=nvptx-none -o a.out a.c
```

Refernces

https://kristerw.blogspot.com/2017/04/building-gcc-with-support-for-nvidia.html