Programming your GPU with OpenMP

Tim Mattson

Tim demonstrates his new invention: Kayak snorkeling. Palawan Philippines, 2019

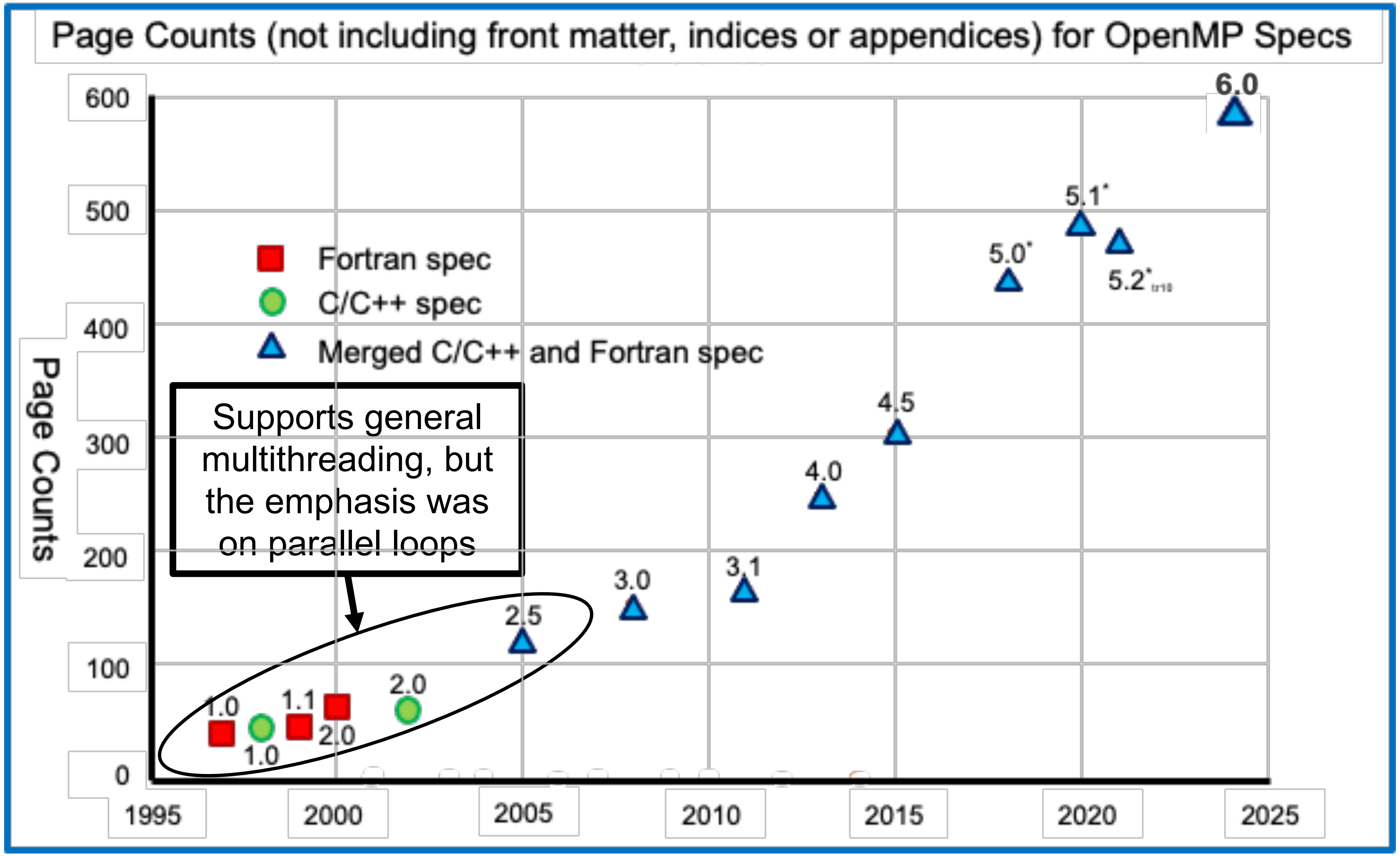
git clone https://github.com/tgmattso/ParProgForPhys.git

Much of this content was created with Tom Deakin and Simon McIntosh-Smith of the University of Bristol

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.

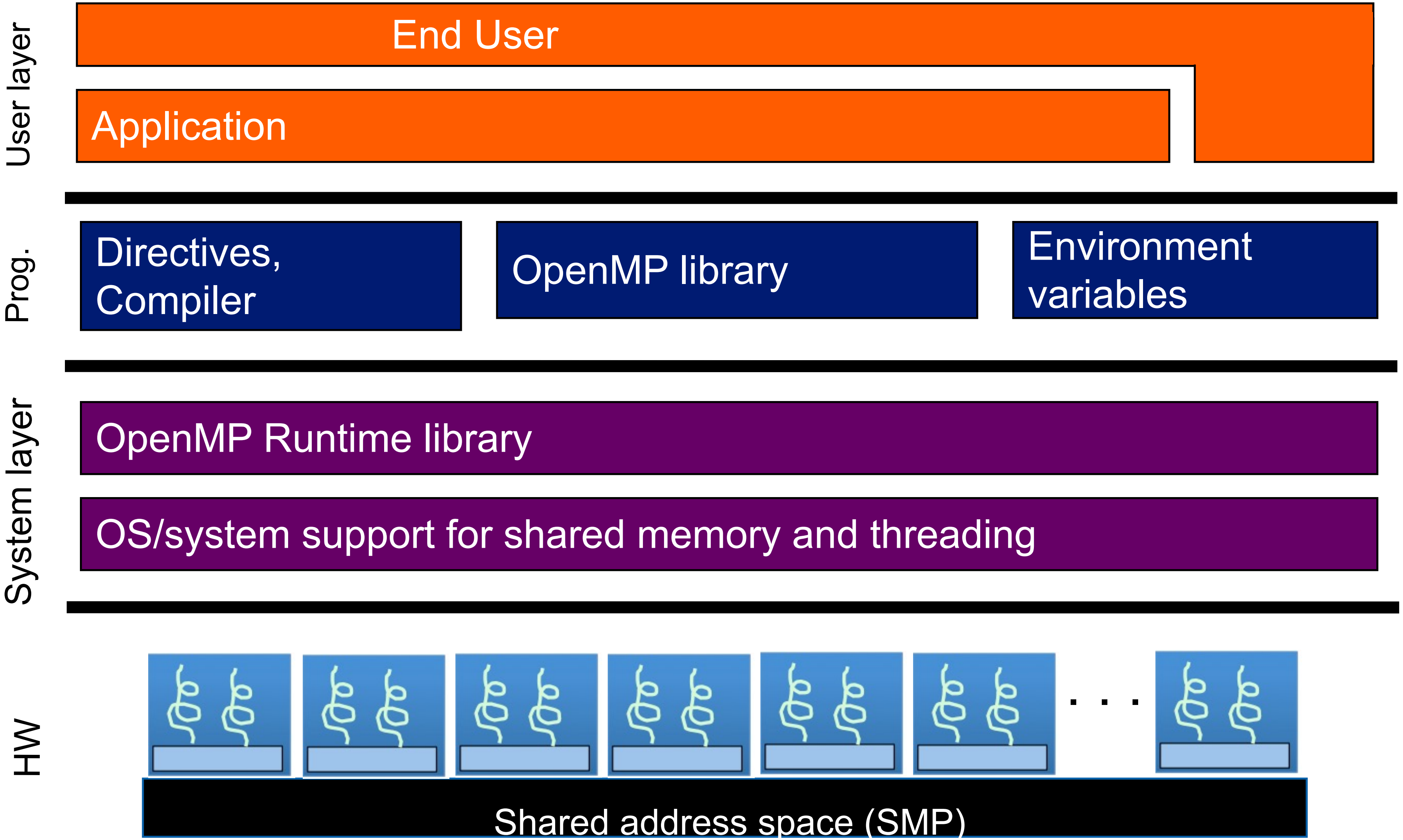
- The complexity has grown considerably over the years!



Page Counts (not including front matter, indices or appendices) for OpenMP Specs

Supports general multithreading, but the emphasis was on parallel loops

OpenMP Basic Definitions: Basic Solution Stack



User layer

End User

Application

Prog.

Directives,
Compiler

OpenMP library

Environment
variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

. . .

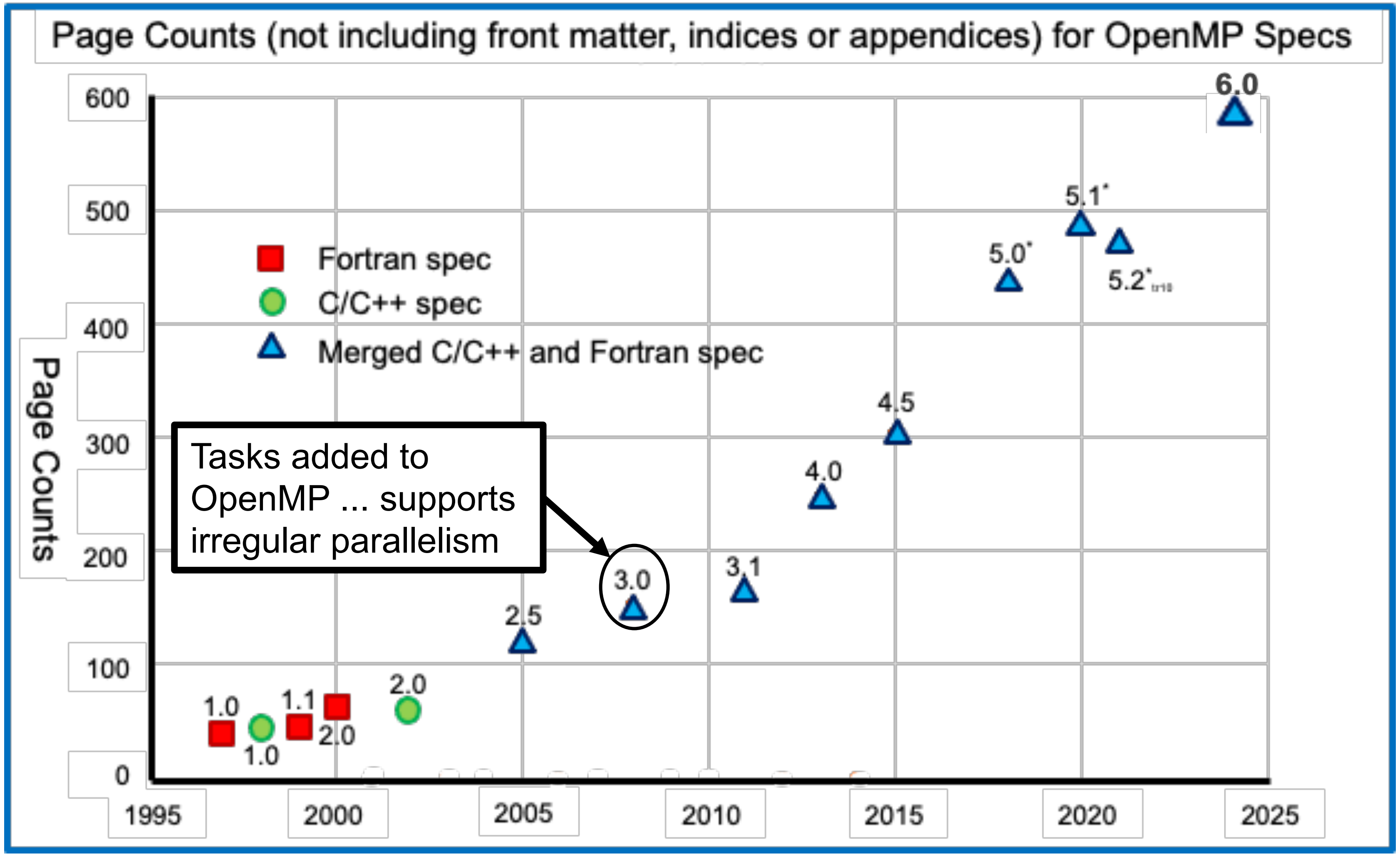
Shared address space (SMP)

For the OpenMP Common Core, we focus on Symmetric Multiprocessor Case ….
i.e., lots of threads with “equal cost access” to memory

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.

- The complexity has grown considerably over the years!



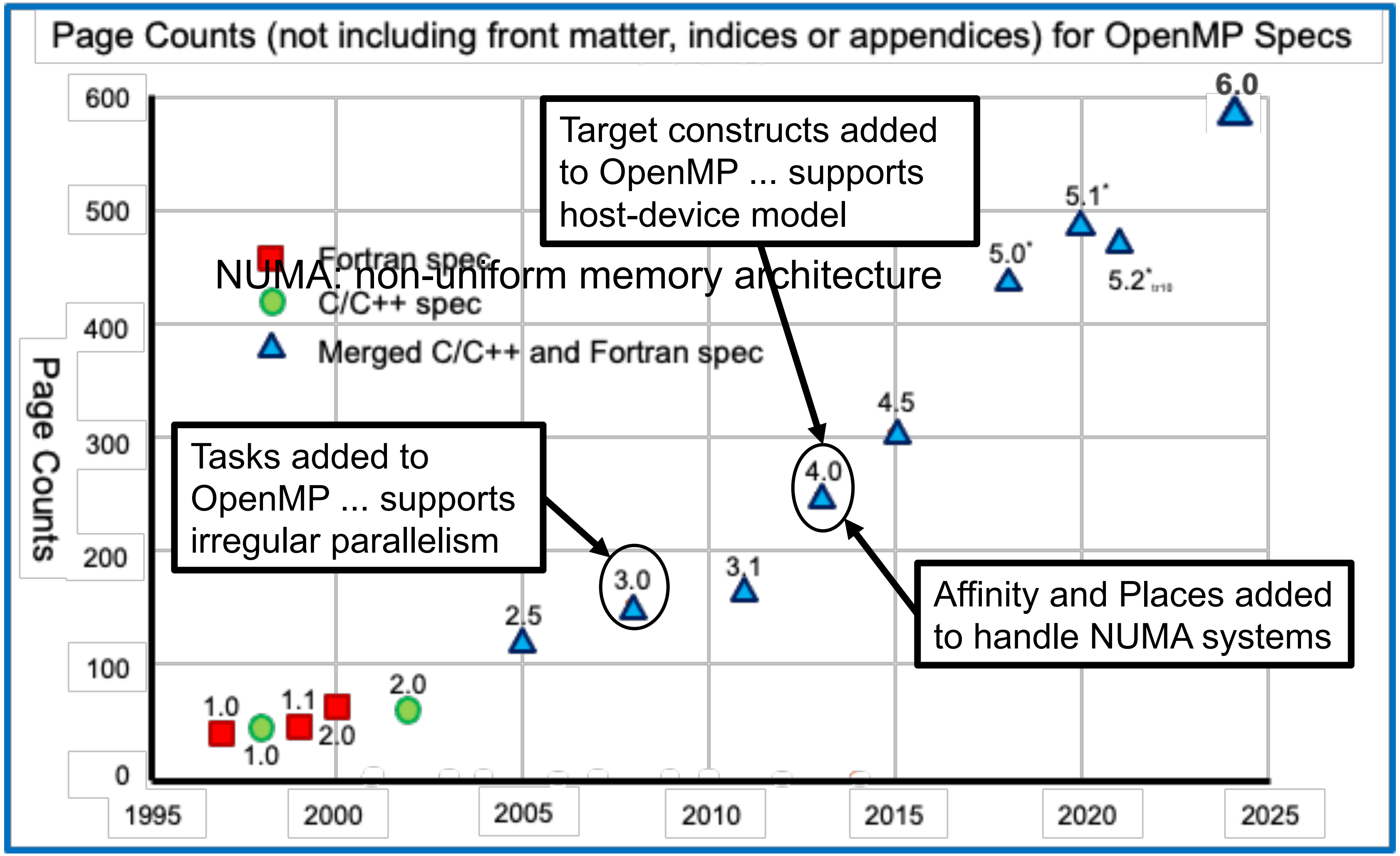
Page Counts (not including front matter, indices or appendices) for OpenMP Specs

Tasks added to OpenMP ... supports irregular parallelism

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.

- The complexity has grown considerably over the years!



Target constructs added to OpenMP ... supports host-device model

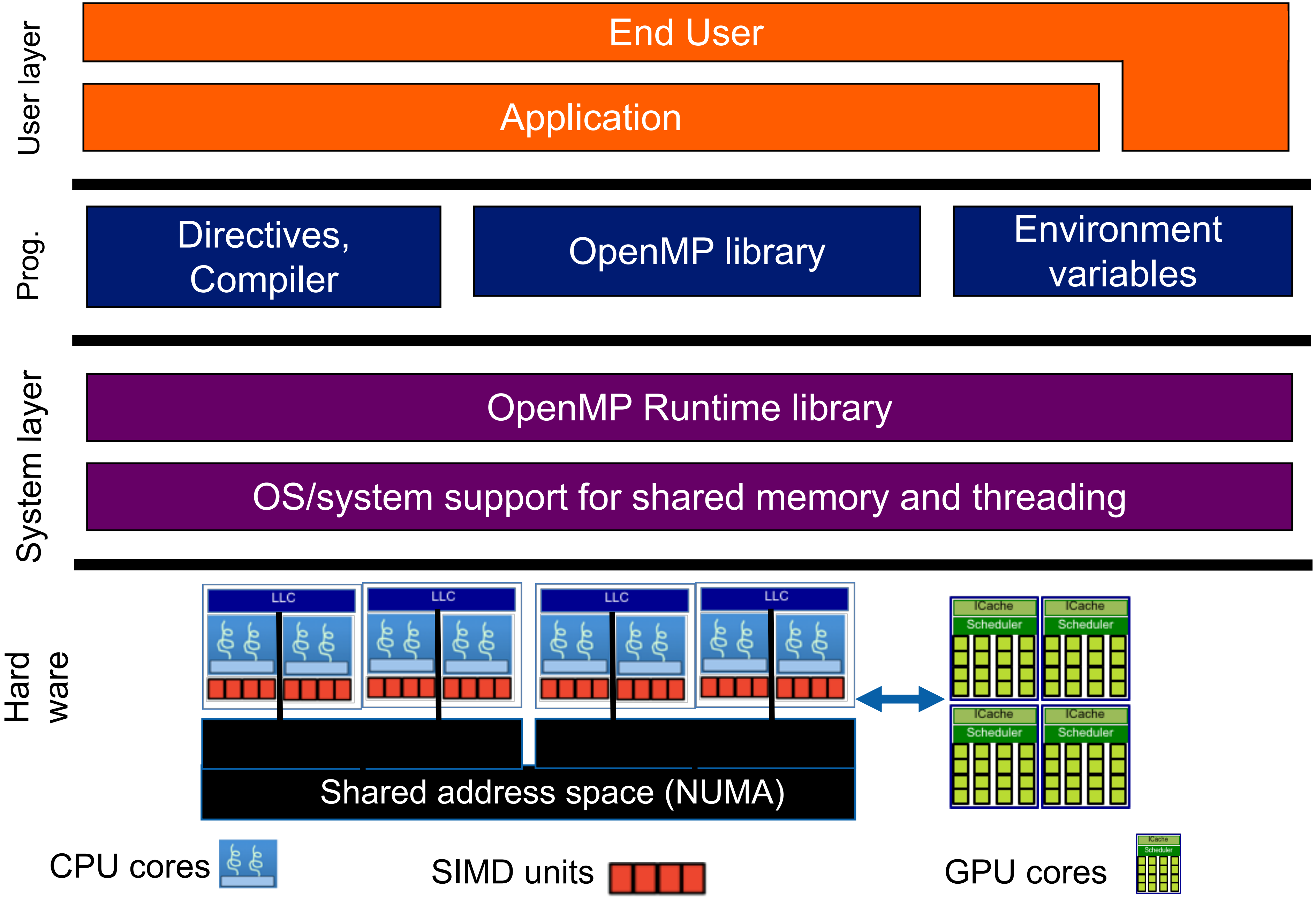
NUMA: non-uniform memory architecture

Tasks added to OpenMP ... supports irregular parallelism

Affinity and Places added to handle NUMA systems

NUMA: non-uniform memory architecture

OpenMP basic definitions: the solution stack



User layer

End User

Application

Prog.

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

Hard ware

Shared address space (NUMA)

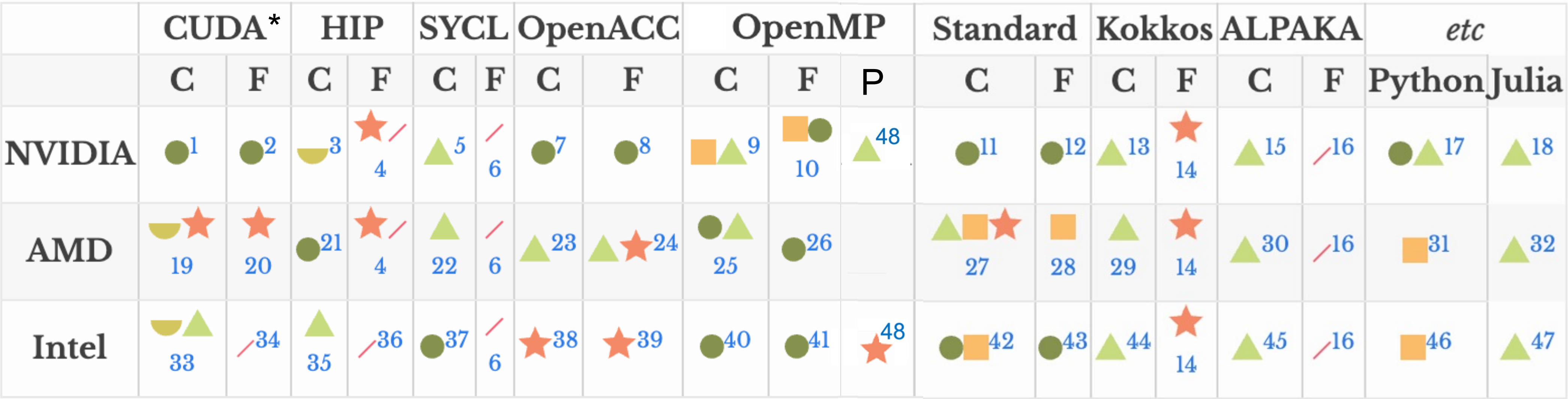
CPU cores

SIMD units

GPU cores

Why is OpenMP so important?

OpenMP is the **only** model with **full** support from **all** vendors for C/C++ and Fortran



	CUDA*		HIP		SYCL		OpenACC		OpenMP			Standard		Kokkos		ALPAKA		*etc*	
	C	F	C	F	C	F	C	F	C	F	P	C	F	C	F	C	F	Python	Julia
NVIDIA	● 1	● 2	◒ 3	★ ⟋ 4	▲ 5	⟋ 6	● 7	● 8	🟧 ▲ 9	🟧 ● 10	▲ 48	● 11	● 12	▲ 13	★ 14	▲ 15	⟋ 16	● ▲ 17	▲ 18
AMD	◒ ★ 19	★ 20	● 21	★ ⟋ 4	▲ 22	⟋ 6	▲ 23	▲ ★ 24	● ▲ 25	● 26		▲ 🟧 ★ 27	🟧 28	▲ 29	★ 14	▲ 30	⟋ 16	🟧 31	▲ 32
Intel	◒ ▲ 33	⟋ 34	▲ 35	⟋ 36	● 37	⟋ 6	★ 38	★ 39	● 40	● 41	★ 48	● 🟧 42	● 43	▲ 44	★ 14	▲ 45	⟋ 16	🟧 46	▲ 47

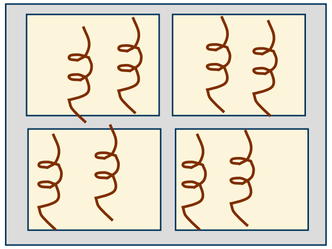
- ● Full vendor support
- ◒ Indirect, but comprehensive support, by vendor
- 🟧 Vendor support, but not (yet) entirely comprehensive
- ▲ Comprehensive support, but not by vendor
- ★ Limited, probably indirect support -- but at least some
- ⟋ No direct support available, but of course one could ISO-C-bind your way through it or directly link the libraries

C C++ (sometimes also C)
F Fortran
P Python

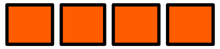
*While not shown in the table, CUDA supports Python as well through CuPy

Table updated Feb 11, 2025. The numbers in the table refer to notes about each listed option. The Table is available at the following URL: https://x-dev.pages.jsc.fz-juelich.de/models/
Table from https://doi.org/10.1145/3624062.3624178 “Many Cores, Many Models: GPU Programming Model vs. Vendor compatibility Overview”, Andreas Herten.
The table was modified by adding the python column for OpenMP (ref: **48**, PyOMP, https://ieeexplore.ieee.org/document/9658236)

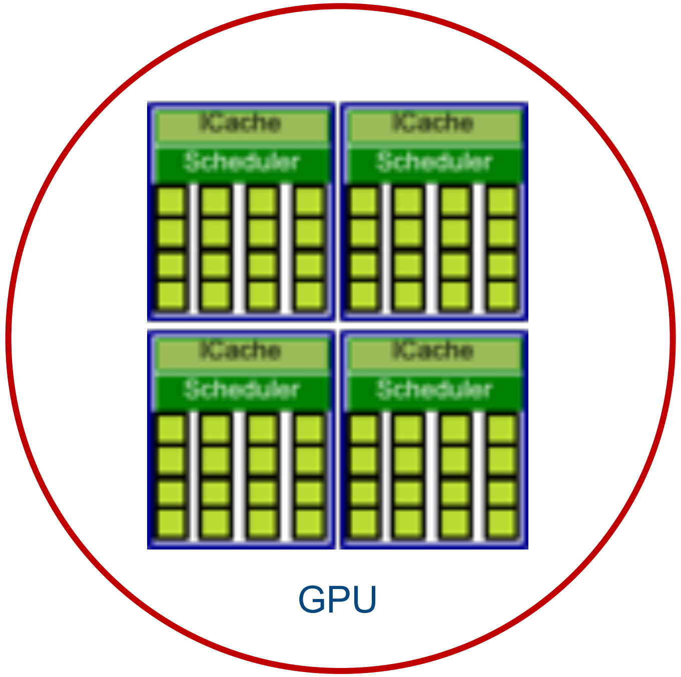
Hardware is diverse … and its only getting worse!!!



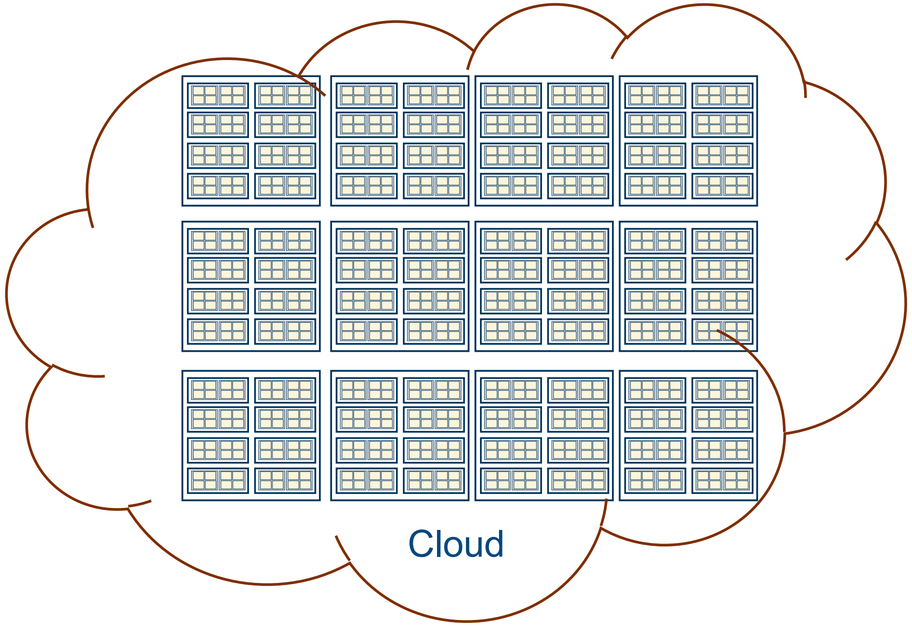
CPU



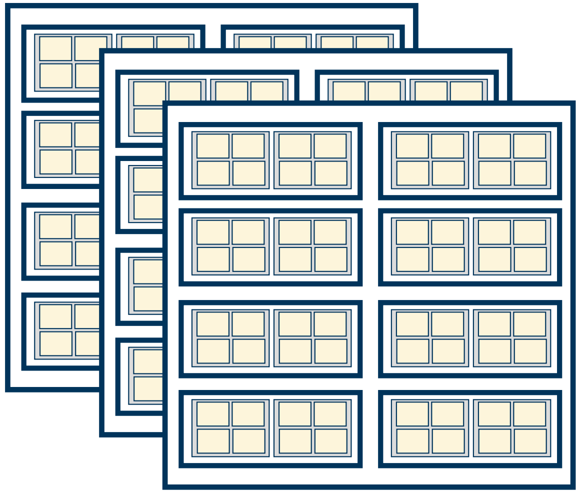
SIMD/Vector



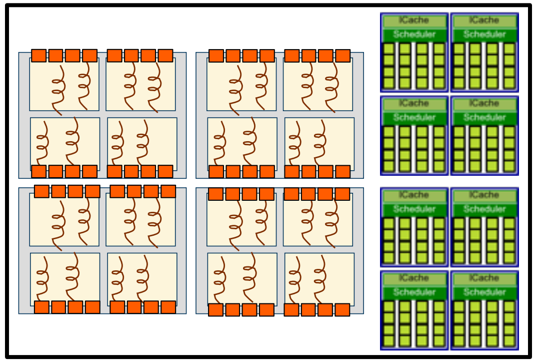
GPU



Cloud



Cluster



Heterogeneous node

The Big Three

- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.
  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP** … : GPU programming (use CUDA if you don't mind locking yourself to a single vendor … it is a really nice programming model)
- Even if you don't plan to spend much time programming with these systems … a well rounded HPC programmer should know what they are and how they work.

The “BIG idea” Behind GPU programming

Traditional Loop based vector addition (vadd)

```
int main() {
   int N = . . . ;
   float *a, *b, *c;

   a* =(float *) malloc(N * sizeof(float));

   // ... allocate other arrays (b and c)
   // and fill with data

   for (int i=0;i<N; i++)
      c[i] = a[i] + b[i];

}
```

Data Parallel vadd with CUDA

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

Assume a GPU with unified shared memory … allocate on host, visible on device too

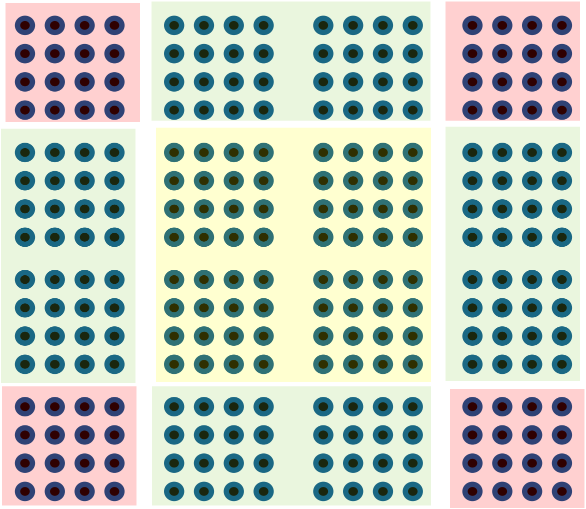
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Turn kernel code into a scalar work-item

```
// Compute sum of order-N matrices: C = A + B
void __global__
matAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j<N) c[i][j] == a[i][j] + b[i][j];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // define threadBlocks and the Grid
   dim3 dimBlock(4,4);
   dim3 dimGrid(4,4);

  // Launch kernel on Grid
    matAdd <<< digGrid,dimBlock>>> (a, b, c, N);
}
```

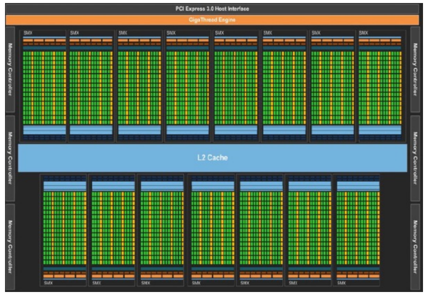
This is CUDA code

2. Map work-items onto an N dim index space.



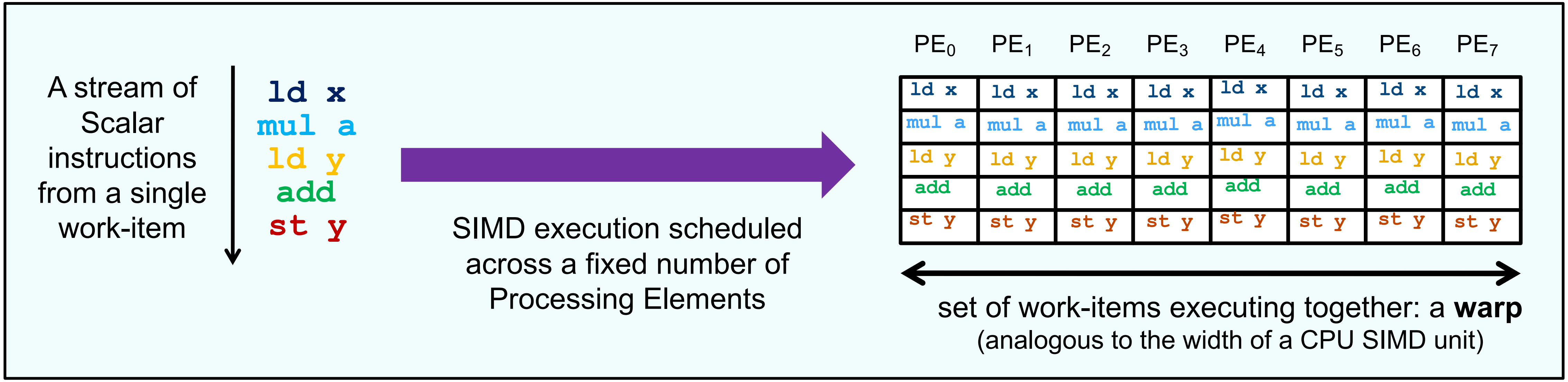
3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model



SIMT: One instruction stream maps onto many Processing Elements

- SIMT model: Individual scalar instruction streams are grouped together for SIMD execution on hardware



A stream of Scalar instructions from a single work-item

```
ld x
mul a
ld y
add
st y
```

SIMD execution scheduled across a fixed number of Processing Elements

PE_0	PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	PE_7
ld x	ld x	ld x	ld x	ld x	ld x	ld x	ld x
mul a	mul a	mul a	mul a	mul a	mul a	mul a	mul a
ld y	ld y	ld y	ld y	ld y	ld y	ld y	ld y
add	add	add	add	add	add	add	add
st y	st y	st y	st y	st y	st y	st y	st y

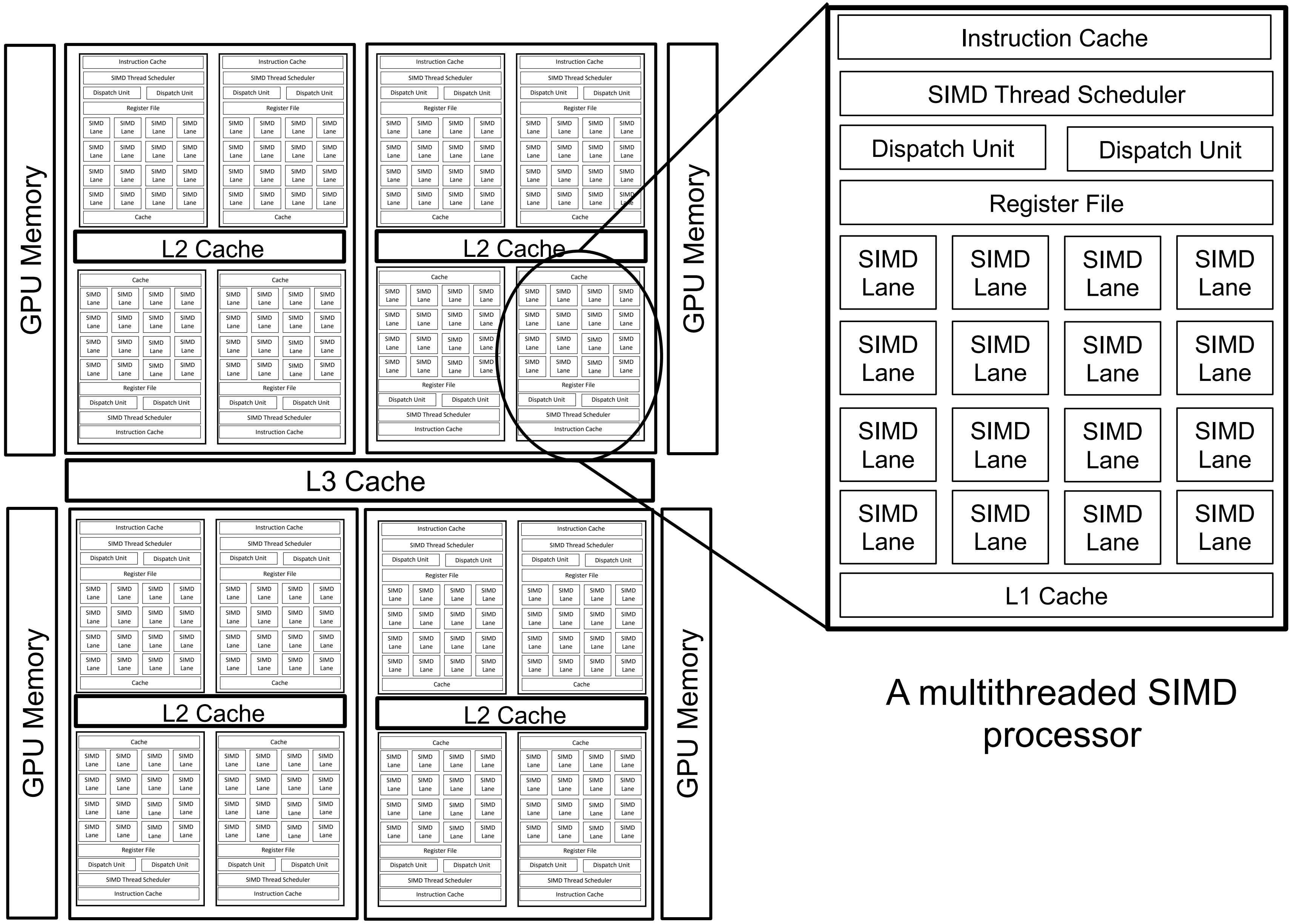
set of work-items executing together: a **warp**
(analogous to the width of a CPU SIMD unit)

GPU nomenclature is really messed up. (sorry about that … we tried to unify around OpenCL but failed).

Instruction stream at finest grain	**Work-item**, CUDA Thread
Blocks for scheduling work-items	**work-group**, thread block
Execution width for work-items	Subgroup, **warp**
Finest grained processing element (PE) in a GPU	SIMD Lane, **Processing Element**, CUDA Core
Block of PEs driven by a single Instruction sequencer	multithreaded SIMD processor, compute unit, **Streaming multiprocessor**

These names are particularly awful since they conflict with established names from CPU Computing.

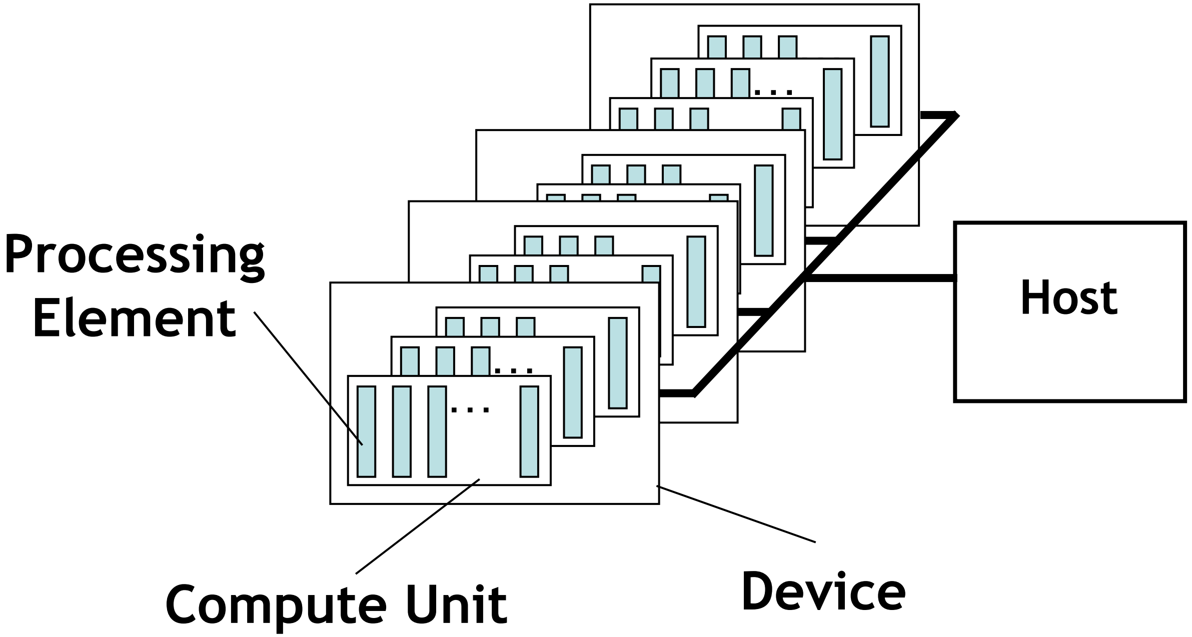
A Generic GPU (following Hennessey and Patterson)



A multithreaded SIMD processor

Computer Architecture: A Quantitative Approach, John L. Hennessy and David A. Patterson.

A Generic Host/Device Platform Model



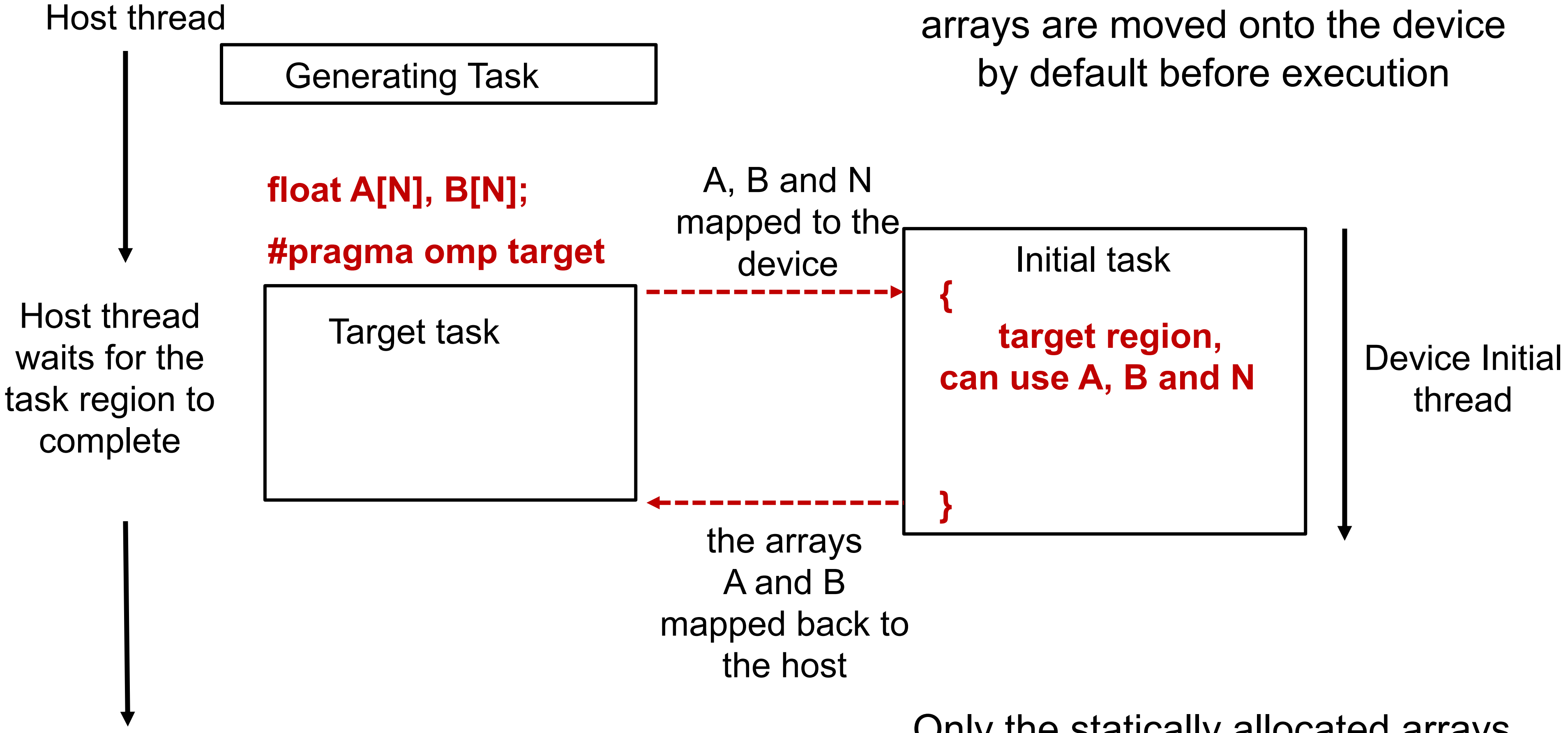
- One ***Host*** and one or more ***Devices***
  - Each Device is composed of one or more Compute Units
  - Each Compute Unit is divided into one or more ***Processing Elements***

- Memory divided into ***host memory*** and ***device memory***

Running code on the GPU:

The target construct and default data movement

Scalars and statically allocated
arrays are moved onto the device
by default before execution



Host thread

Generating Task

float A[N], B[N];

#pragma omp target

Host thread
waits for the
task region to
complete

Target task

A, B and N
mapped to the
device

Initial task

{

target region,
can use A, B and N

}

Device Initial
thread

the arrays
A and B
mapped back to
the host

Only the statically allocated arrays
are moved back to the host after
the target region completes

Based on figure 6.4 in **Using OpenMP – The Next Step** by van der Pas, Stotzer and Terboven, MIT Press, 2017

Default Data Sharing: example

```
int main(void) {
  int N = 1024;
  double A[N], B[N];

  #pragma omp target
  {

    for (int ii = 0; ii < N; ++ii) {

      A[ii] = A[ii] + B[ii];

    }

  } // end of target region
}
```

1. Variables created in host memory.
2. Scalar **N** and stack arrays **A** and **B** are copied *to* device memory. Execution transferred to device.
3. **ii** is **private** on the device as it’s declared within the target region
4. Execution on the device.
5. stack arrays **A** and **B** are copied *from* device memory back to the host. Host resumes execution.

Now let’s run code in parallel on the device

```
int main(void) {
  int N = 1024;
  double A[N], B[N];

  #pragma omp target
  {
    #pragma omp loop
    for (int ii = 0; ii < N; ++ii) {

      A[ii] = A[ii] + B[ii];

    }

  } // end of target region
}
```

The loop construct tells the compiler:

> *“this loop will execute correctly if the loop iterations run in any order. You can safely run them concurrently. And the loop-body doesn’t contain any OpenMP constructs. So do whatever you can to make the code run fast”*

The loop construct is a declarative construct. You tell the compiler what you want done but you DO NOT tell it how to “do it”. This is new for OpenMP

Exercise: Parallel vector addition on a GPU

- Make a copy of your parallel vadd.c program for a CPU (i.e. save the CPU version)
  - vadd.c Adds together two arrays, element by element: `for(i=0;i<N;i++) c[i]=a[i]+b[i];`
- Parallelize your vadd program for a GPU.
  - `module load nvhpc/24.5`
  - `nvc -O3 -mp=gpu  vadd.c`
- Time it for large N and save the result. How does it compare to the CPU version?
  - `double omp_get_wtime();`
  - `#pragma omp target`
  - `#pragma omp loop`
  - `sbatch gpu_job.sh`

For tiny little programs, OpenMP may opt to run the code on the host. You can force the OpenMP runtime to use the GPU by setting the OMP_TARGET_OFFLOAD environment variable

```
> OMP_TARGET_OFFLOAD=MANDATORY ./a.out
```

gpu_job.sh

```
#!/bin/bash
#SBATCH -p sched_mit_psfc_gpu_r8
#SBATCH --reservation=cpsfr_2025
#SBATCH -o gpu-job.log-%j
#SBATCH --gres=gpu:1
module load nvhpc/24.5
OMP_TARGET_OFFLOAD=MANDATORY ./vadd
```

git clone https://github.com/tgmattso/ParProgForPhys.git

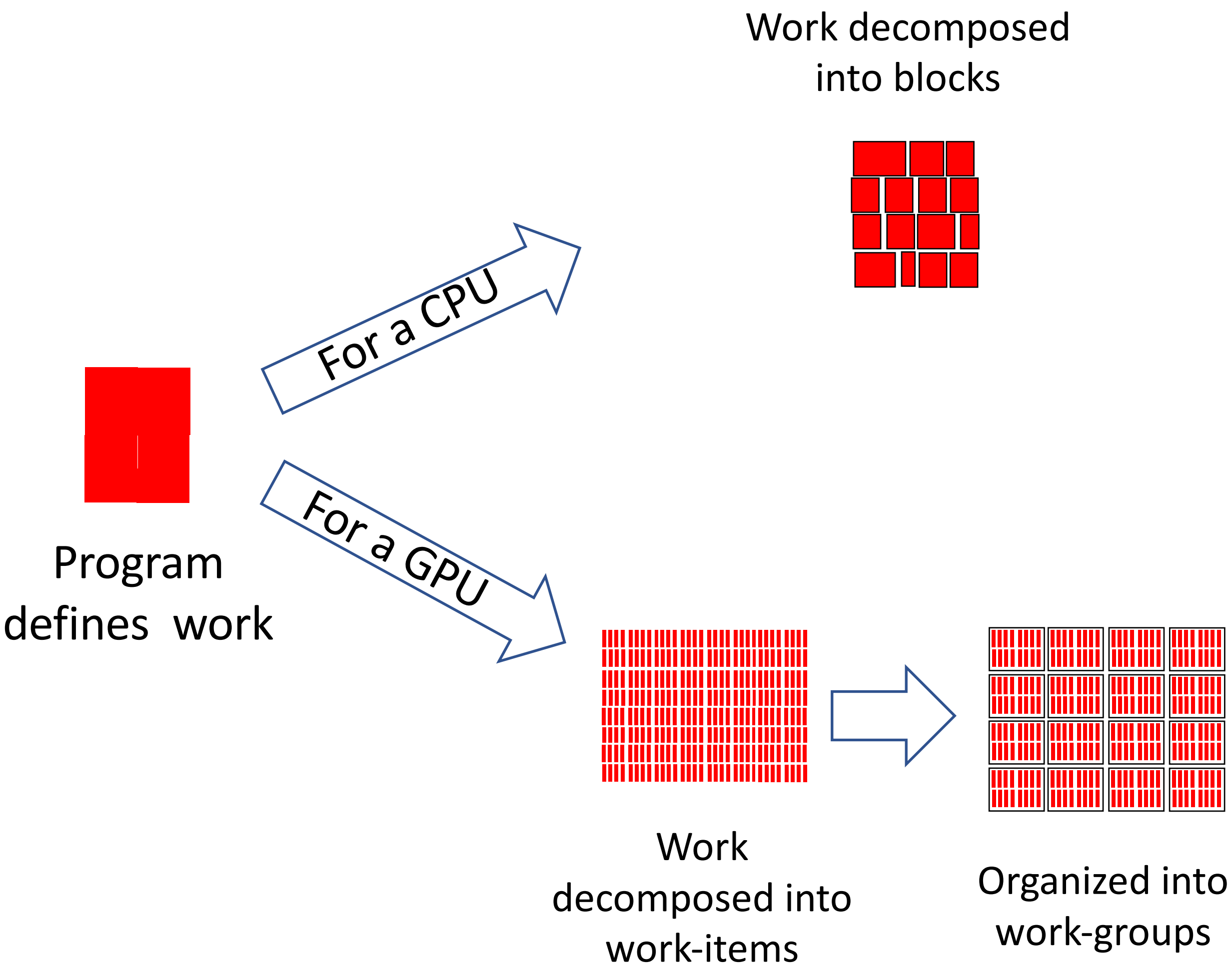
Solution: Simple vector add in OpenMP on GPU

```c
int main()
{
     float a[N], b[N], c[N], res[N];
     int err=0;

    // fill the arrays
    #pragma omp parallel for
    for (int i=0; i<N; i++){
       a[i] = (float)i;
       b[i] = 2.0*(float)i;
       c[i] = 0.0;
       res[i] = i + 2*i;
    }

    // add two vectors
    #pragma omp target
    #pragma omp loop
    for (int i=0; i<N; i++){
       c[i] = a[i] + b[i];
    }
```

```c
  // test results
  #pragma omp parallel for reduction(+:err)
  for(int i=0;i<N;i++){
     float val = c[i] - res[i];
     val = val*val;
     if(val>TOL) err++;
  }
  printf("vectors added with %d errors\n", err);
  return 0;
}
```

Let’s compare/contrast concurrency on a CPU and a GPU

Executing a program on CPUs and GPUs

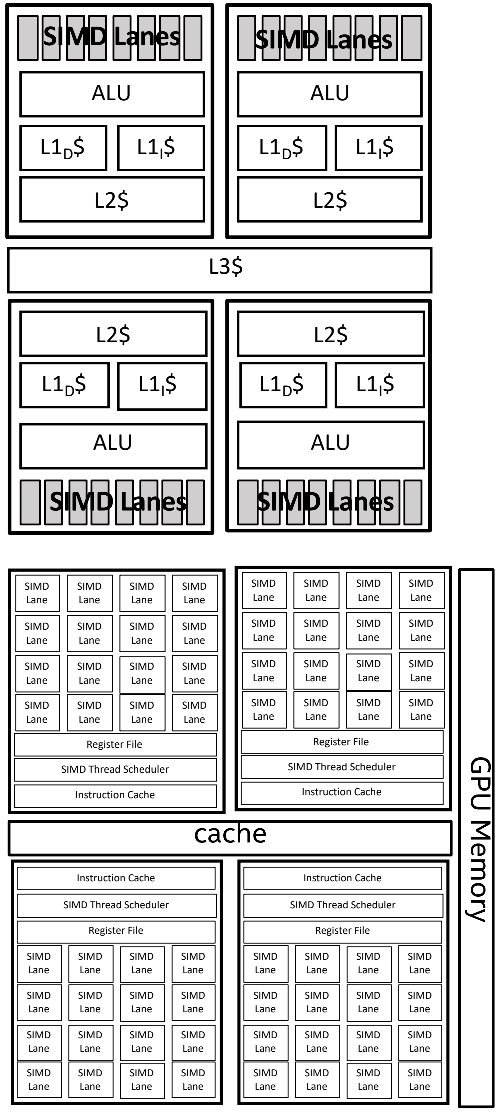


Program defines work

Work decomposed into blocks

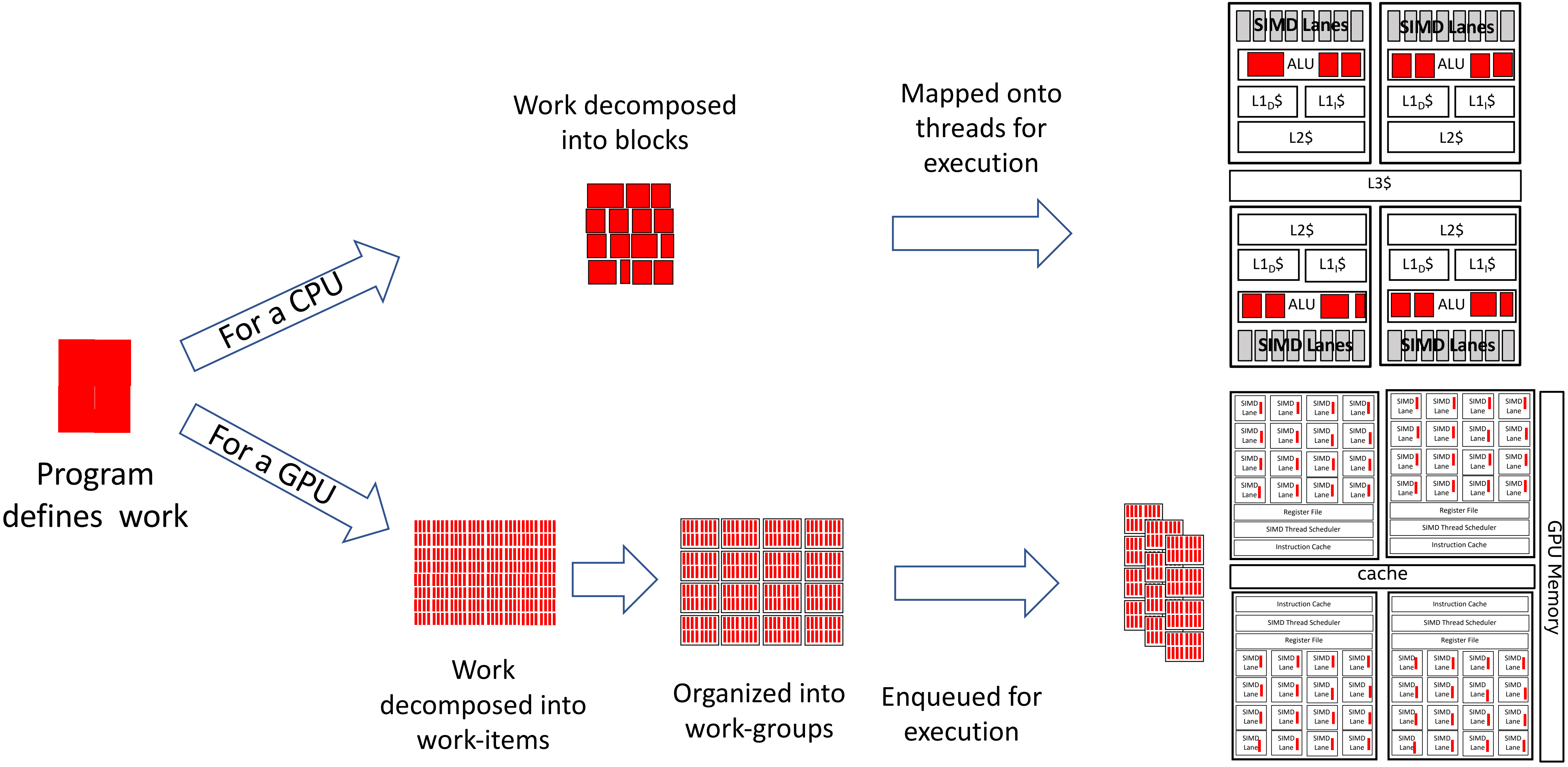
Work decomposed into work-items

Organized into work-groups



One work-group per compute-unit executing

Executing a program on CPUs and GPUs



Program defines work

For a CPU

Work decomposed into blocks

Mapped onto threads for execution

For a GPU

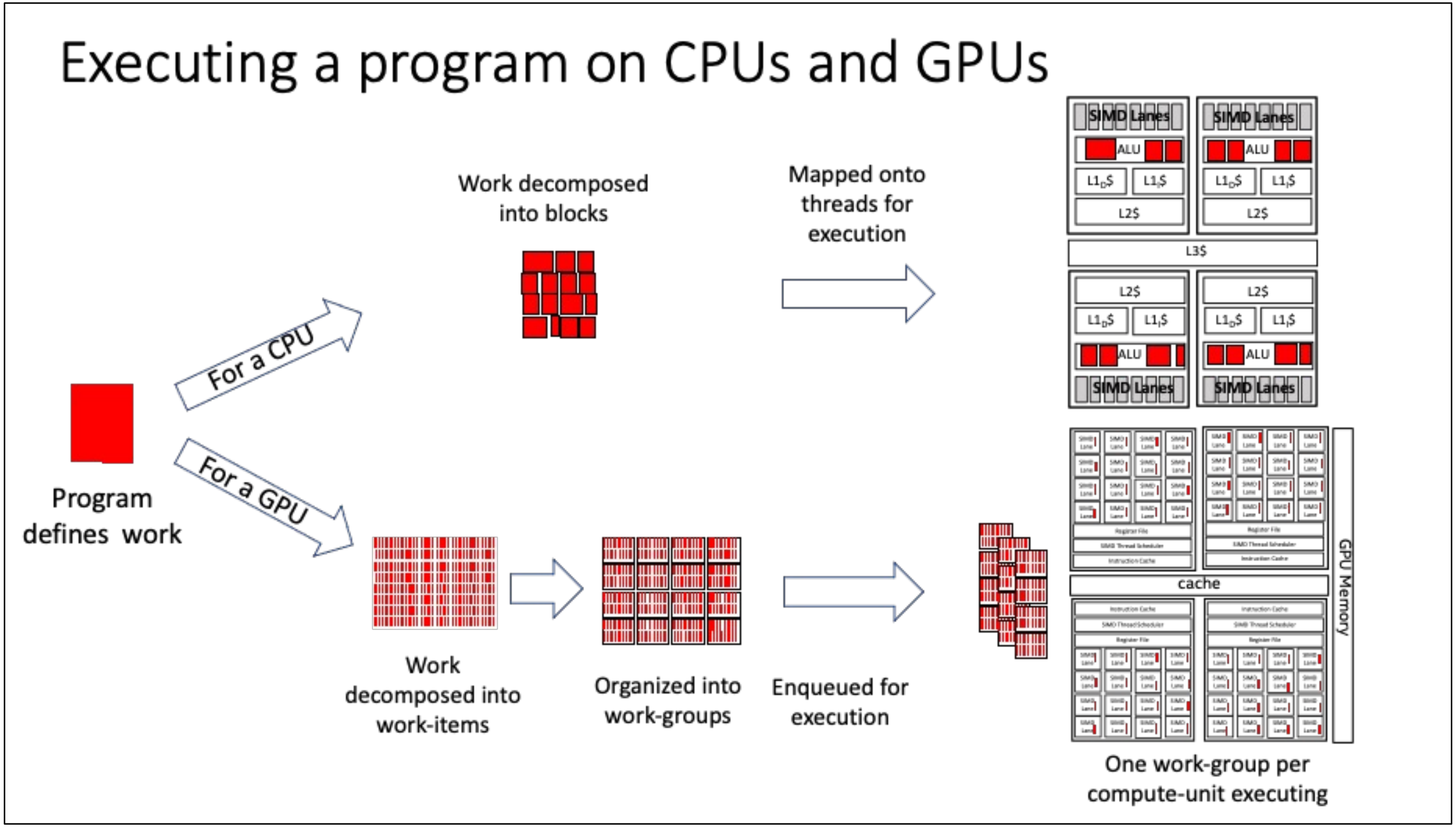
Work decomposed into work-items

Organized into work-groups

Enqueued for execution

One work-group per compute-unit executing

CPU/GPU execution modesl



Executing a program on CPUs and GPUs

Program defines work

For a CPU

Work decomposed into blocks

Mapped onto threads for execution

For a GPU

Work decomposed into work-items

Organized into work-groups

Enqueued for execution

One work-group per compute-unit executing

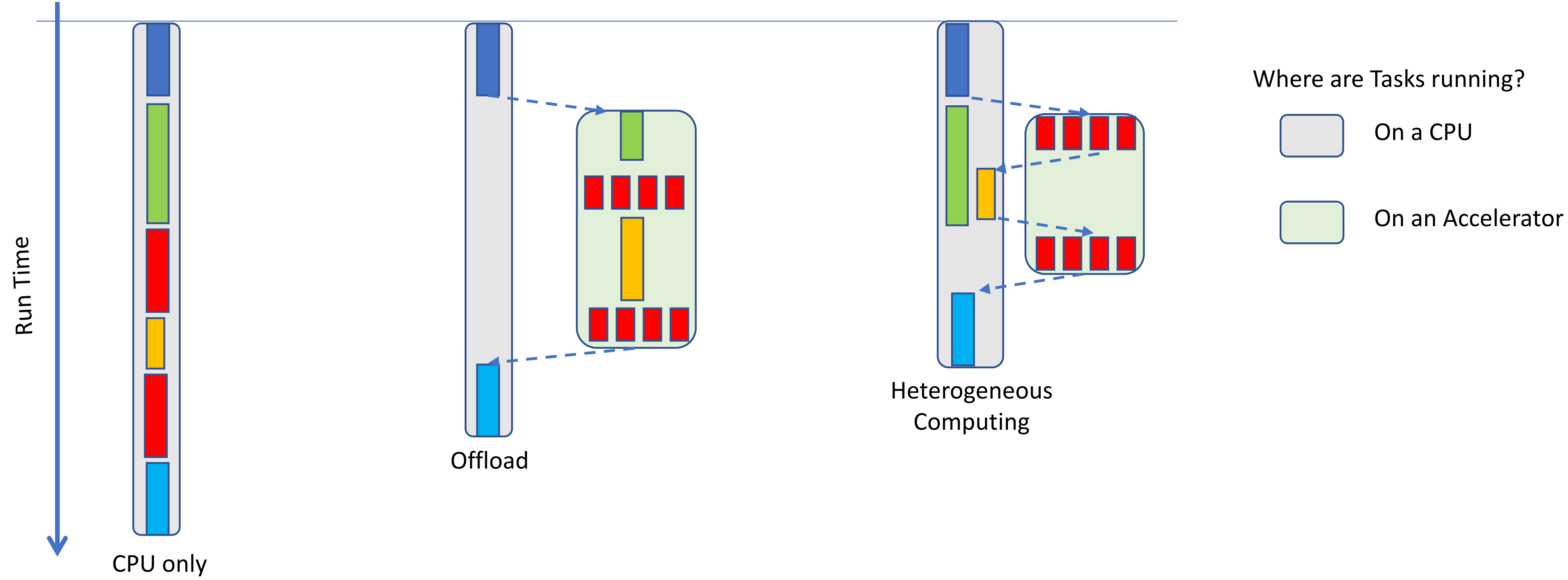
For a CPU, the threads are all active and able to make forward progress.

For a GPU, any given work-group might be in the queue waiting to execute.

No single processor is best at everything

- The idea that you should move everything to the GPU makes no sense

- **Heterogeneous Computing**: Run sub-problems in parallel on the hardware best suited to them.



Implicit data movement covers a small subset of the cases you need in a real program.

To be more general … we need to manage data movement explicitly

Explicit data movement

- Previously, we described the rules for *implicit* data movement.

- We can *explicitly* control the movement of data using the **map** clause.

- **Data allocated on the heap needs to be explicitly copied to/from the device**:

```
int main(void) {
   int  ii=0, N = 1024;
   int* A = (int *)malloc(sizeof(int)*N);

   #pragma omp target
   {
     // N, ii and A all exist here
     // The data that A points to (*A , A[ii]) DOES NOT exist here!
   }
}
```

Moving data with the map clause

```
int main(void) {
  int  N = 1024;
  int* A = malloc(sizeof(int)*N);

  #pragma omp target map(A[0:N])
  {
   // N, ii and A all exist here
   // The data that A points to DOES exist here!
  }
}
```

Default mapping
map(tofrom: A[0:N])

Copy at start and end of **target** region.

Controlling data movement

```
int i, a[N], b[N], c[N];
#pragma omp target map(to:a,b) map(tofrom:c)
```

Data movement defined from the *host* perspective.

- The various forms of the map clause
  - **map(to:list)**: On entering the region, variables in the list are initialized on the device using the original values from the host (host to device copy).
  - **map(from:list)**: At the end of the target region, the values from variables in the list are copied into the original variables on the host (device to host copy). On entering the region, the initial value of the variables on the device is not initialized.
  - **map(tofrom:list)**: the effect of both a map-to and a map-from (host to device copy at start of region, device to host copy at end).
  - **map(alloc:list)**: On entering the region, data is allocated and uninitialized on the device.
  - **map(list)**: equivalent to **map(tofrom:list)**.

Vector add with dynamic memory on GPU

```
int main()
{
     float *a   = (float *) malloc(sizeof(float) * N);
     float *b   = (float *) malloc(sizeof(float) * N);
     float *c   = (float *) malloc(sizeof(float) * N);
     float *res = (float *) malloc(sizeof(float) * N);
     int err=0;

    // fill the arrays <<<code not shown>>>>

    // add two vectors
    #pragma omp target map(to: a[0:N],b[0:N]) map (tofrom: c[0:N])
    #pragma omp loop
    for (int i=0; i<N; i++){
       c[i] = a[i] + b[i];
    }

   // test results <<<code not shown>>>>

  printf("vectors added with %d errors\n", err);
  return 0;
}
```

Commonly used clauses on target and loop constructs

- The basic construct* is:

  #pragma omp target *[clause[[,]clause]...]*

  #pragma omp loop *[clause[[,]clause]...]*

  for-loops

- The most commonly used clauses are:
  - **map(to | from | tofrom list)** ← default is tofrom
  - **private(***list***) firstprivate(***list***) lastprivate(***list***) shared(***list***)**
    - behave as data environment clauses in the rest of OpenMP, but note values are only created or copied into the region, not back out "at the end".
  - **reduction(***reduction-identifier* **:** *list***)**
    - behaves as in the rest of OpenMP
  - **collapse(***n***)**
    - Combines loops before the distribute directive splits up the iterations between teams

Going beyond simple vector addition …

Using OpenMP for GPU application programming … the heat diffusion problem

5-point stencil: the heat program

- The heat equation models changes in temperature over time.

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$

- We'll solve this numerically on a computer using an explicit **finite difference** discretisation.
- $u = u(t, x, y)$ is a function of space and time.
- Partial differentials are approximated using diamond difference formulae:

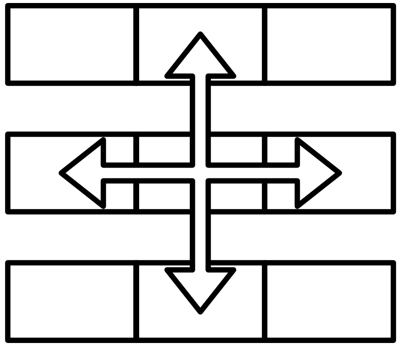
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t, x+1, y) - 2u(t, x, y) + u(t, x-1, y)}{dx^2}$$

  - Forward finite difference in time, central finite difference in space.

5-point stencil: the heat program

- Given an initial value of u, and any boundary conditions, we can calculate the value of u at time t+1 given the value at time t.
- Each update requires values from the north, south, east and west neighbours only:



- Computation is essentially a weighted average of each cell and its neighbouring cells.
- If on a boundary, look up a boundary condition instead.

Method of Manufactured Solution

- Stencil codes are notoriously difficult to **know** if the answer is “correct”.

- Analytic solutions hard to come by:
  - It’s why you’re using a computer to solve the equation approximately after all!

- Method of Manufactured Solution (MMS) is a way to help determine if the code does the correct thing.
- An approach often used to find errors in CFD codes and check convergence properties.

Method of Manufactured Solution

- **Choose** a function for $u(t, x, y)$, substitute into the equation and work through the algebra.

- Ideally like the equation to evaluate to zero so don’t need to consider a right-hand side to the equation.

- $u(0, x, y)$ gives the initial conditions.
- Can evaluate boundary conditions, e.g. bottom boundary $u(0,0, y)$

- Because u is **known** for all timesteps (it was chosen!), the exact solution is **known**.

- Compare the **computed** solution to the known u to compute an error.
- Any differences come from approximations in the method, or a bug in your code.

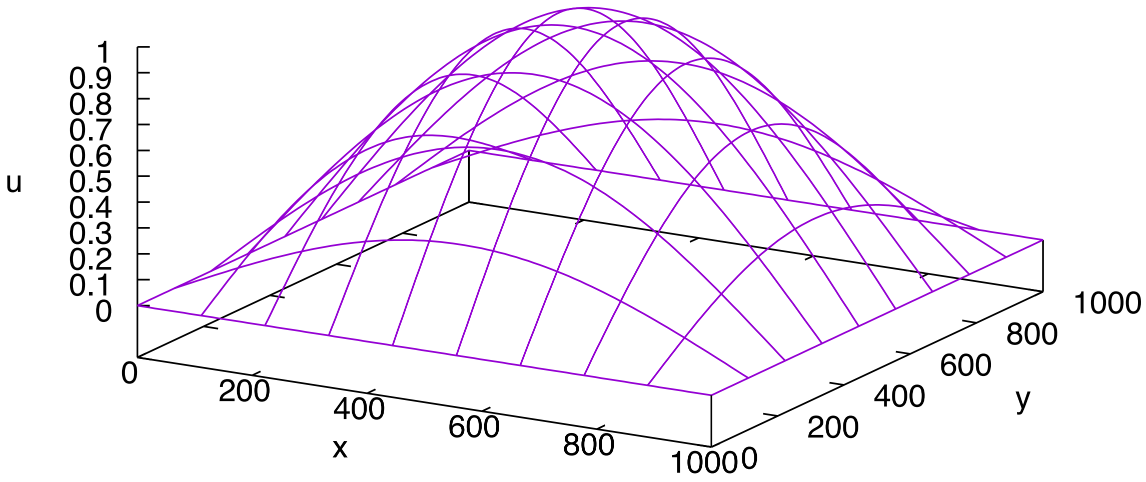
Method of Manufactured Solution

- For the problem of length l, choose u:

$$u(t,x,y) = e^{\frac{-2\alpha\pi^2 t}{l^2}} \sin\frac{\pi x}{l} \sin\frac{\pi y}{l}$$

- Boundary conditions: u is always zero on the boundaries

- Initial value of grid is then $u(0,x,y) = \sin\frac{\pi x}{l} \sin\frac{\pi y}{l}$



Heat program …

```
// Loop over time steps
for (int t = 0; t < nsteps; ++t) {

  // solve over spatial domain for step t
  solve(n, alpha, dx, dt, u, u_tmp);

  // Pointer swap to get ready for next step
  tmp = u;
  u = u_tmp;
  u_tmp = tmp;
}
```

Note: Swapping pointers on the host before entering the target region on the next iteration works on a GPU.

When you map pointers between the host and the device, OpenMP remembers the address.

Swapped addresses on the hosts swaps addresses on the device

- Takes two optional command line arguments: <ncells> <nsteps>
  - E.g. ./heat 1000 10
  - 1000x1000 cells, 10 timesteps (the default problem size).
- If no command line arguments are provided, it uses a default:
  - These two commands both run the default problem size of 1000x1000 cells, 10 timesteps.
  - ./heat
  - ./heat 1000 10
- A sensible bigger problem is 8000 x 8000 cells and 10 timesteps.

Heat diffusion problem: 5-point stencil code

Serial CPU code

```
 const double r = alpha * dt / (dx * dx);
  const double r2 = 1.0 - 4.0*r;

  // malloc and initialize u_tmp and u (code not shown)

  for (int t = 0; t < nsteps; ++t) {

     for (int i = 0; i < n; ++i) {
       for (int j = 0; j < n; ++j) {
         u_tmp[i+j*n] =  r2 * u[i+j*n]          +
             r * ((i < n-1) ? u[i+1+j*n]    : 0.0) +
             r * ((i > 0)   ? u[i-1+j*n]    : 0.0) +
             r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
             r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
         }
       }
       // Pointer swap to get ready for next step
       tmp = u;
       u = u_tmp;
       u_tmp = tmp;
  }
}
```

Loop over time steps

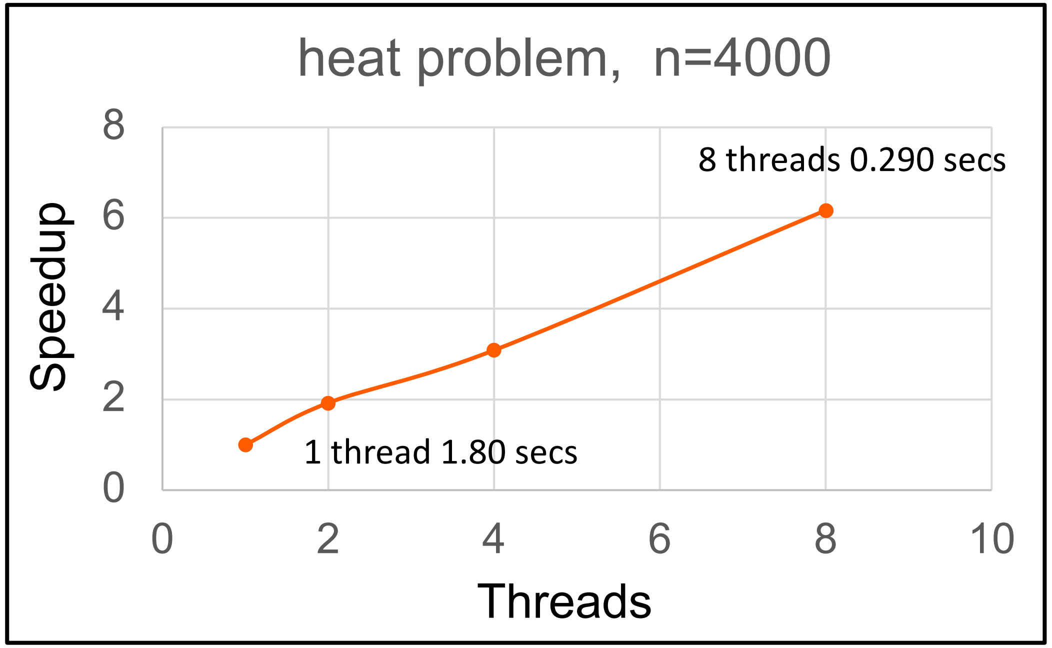
Loop over NxN spatial domain

Update the 5-point stencil. Boundary conditions on the edges of the domain are fixed at zero.

Parallel CPU code, n=4000

Heat diffusion problem: 5-point stencil code

```
  const double r = alpha * dt / (dx * dx);
  const double r2 = 1.0 - 4.0*r;

  // malloc and initialize u_tmp and u (code not shown)

  for (int t = 0; t < nsteps; ++t) {

     #pragma omp parallel for collapse(2)
     for (int i = 0; i < n; ++i) {
       for (int j = 0; j < n; ++j) {
         u_tmp[i+j*n] =  r2 * u[i+j*n]         +
             r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
             r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
             r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
             r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
        }
      }
      // Pointer swap to get ready for next step
      tmp = u;
      u = u_tmp;
      u_tmp = tmp;
  }
}
```



Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.   Nvidia HPC Toolkit compiler
nvc –fast –fopenmp heat.c

Exercise: parallel stencil (heat)

- Take the provided heat stencil code (heat.c)
- Add OpenMP directives to parallelize the loops on the GPU
- Most of the runtime occurs in the solve() routine. Focus on that function. The rest of the code is there to just support the work inside solve.
  - double omp_get_wtime();
  - #pragma omp parallel
  - #pragma omp for
  - #pragma omp parallel for
  - #pragma omp task
  - #pragma omp taskwait
  - #pragma single
  - #pragma omp target
  - #pragma omp loop
  - Plus the clauses
    - private(), firstprivate(), reduction(+:var), collapse(n)
    - map(to:vptr[Lower:Count]) map(from:vptr[Lower:Count]) map(tofrom:vptr[Lower:Count])

Default is tofrom: map(vptr[Lower:Count])

```
module load nvhpc/24.5
nvc -O3 -mp=gpu  heat.c –o heat
sbatch job_gpu.sh
```

job_gpu.sh

```
#!/bin/bash
#SBATCH -p sched_mit_psfc_gpu_r8
#SBATCH --reservation=cpsfr_2025
#SBATCH -o gpu-job.log-%j
#SBATCH --gres=gpu:1
module load nvhpc/24.5
OMP_TARGET_OFFLOAD=MANDATORY ./heat
```

Heat diffusion problem: 5-point stencil code

Parallel GPU code, n=4000

```
  const double r = alpha * dt / (dx * dx);
  const double r2 = 1.0 - 4.0*r;

  // malloc and initialize u_tmp and u (code not shown)

  for (int t = 0; t < nsteps; ++t) {
     #pragma omp target map(tofrom: u[0:n*n], u_tmp[0:n*n])
     #pragma omp loop
     for (int i = 0; i < n; ++i) {
       for (int j = 0; j < n; ++j) {
         u_tmp[i+j*n] =  r2 * u[i+j*n]         +
             r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
             r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
             r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
             r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
       }
      }
      // Pointer swap to get ready for next step
      tmp = u;
      u = u_tmp;
      u_tmp = tmp;
  }
}
```

When you map pointers between the host and the device, OpenMP remembers the address.

Swapped addresses on the hosts swaps addresses on the device

```
GPU Solver time = 1.40 secs
```

This isn't much better than the runtime for a single CPU (1.8 secs) and worse than 8 cores on a CPU (0.29 secs).

Why is the performance so bad?

```
NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
Nvidia HPC Toolkit compiler
nvc –fast –mp=gpu –gpu=cc75 heat.c
```

Parallel GPU code, n=4000

Heat diffusion problem: 5-point stencil code

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {
   #pragma omp target map(tofrom: u[0:n*n], u_tmp[0:n*n])
   #pragma omp loop
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
  }
}
```

With a runtime of 1.4 secs (worse than the CPU time) we see that Data Movement dominates performance.

At the beginning of each iteration, copy (2*N^2)*sizeof(TYPE) bytes to the device

At the end of each iteration, copy (2*N^2)*sizeof(TYPE) bytes from the device

We need to create a **data region** on the GPU that is distinct from the target region.

That way, we can keep the data on the device between target constructs

Finer control over data movement

- Recall that data is mapped to/from the device at start/end of target region

  -
    ```
    #pragma omp target map(tofrom: A[0:N])
    {
       …
    }
    ```

- Inefficient to move data around all the time

- We need to keep data resident on the device *between* target regions.

- We need to manage the device data environment independently from the target regions that launch kernels.

Target enter/exit data constructs

- Create a data region on the target device (a device data environment) with two standalone directives:

```
#pragma omp target enter data map(…)
#pragma omp target exit data map(…)
```

- The **target enter data** maps variables to the device data environment.
- The **target exit data** unmaps variables from the device data environment.
- Once created, subsequent **target** regions inherit the existing data environment.

Target enter/exit data example

```
void init_array(int *A, int *B, int N) {
   for (int i = 0; i < N; ++i) { A[i] = i;  B[i]=2*I;}

  #pragma omp target enter data map(to: A[0:N], B[0:N])
}


int main(void) {

   int N = 1024;
   int *A = malloc(sizeof(int) * N);
   int *B = malloc(sizeof(int) * N);
   init_array(A, B, N);

   #pragma omp target
   #pragma omp loop
   for (int i = 0; i < N; ++i)
      A[i] = A[i] * B[i];

 #pragma omp target exit data map(from: A[0:N])
}
```

Exercise

- Modify your parallel heat code from the last exercise.
- Use the ‘target data’ constructs to control the device data environment.
- Think about the map clauses and how to use them to minimize data movement.
- Question … will the pointer swap on the host still work?
    - #pragma omp target
    - #pragma omp target enter data map(to: list)
    - #pragma omp target exit data map(from: list)
    - #map(to:list) map(from:list) map(tofrom:list)
    - #pragma omp target teams loop

```
module load nvhpc/24.5
nvc -O3 -mp=gpu  heat.c –o heat
sbatch job_gpu.sh
```

```
#!/bin/bash
#SBATCH -p sched_mit_psfc_gpu_r8
#SBATCH --reservation=cpsfr_2025
#SBATCH -o gpu-job.log-%j
#SBATCH --gres=gpu:1
module load nvhpc/24.5
OMP_TARGET_OFFLOAD=MANDATORY ./heat
```

git clone https://github.com/tgmattso/ParProgForPhys.git

Parallel GPU code, n=4000

Heat diffusion problem: 5-point stencil code

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;
// malloc and initialize u_tmp and u (code not shown)
#pragma omp target enter data map(to: u[0:n*n], u_tmp[0:n*n])

for (int t = 0; t < nsteps; ++t) {
   #pragma omp target
   #pragma omp loop
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
     }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
 }
#pragma omp target exit data map(from: u[0:n*n])
}
```

Create a data region and map indicated data on entry

GPU Solver time* = **0.42** secs

This is a general principal …
if you want performance, you must optimize data movement.

*includes time for target enter/exit data

Exit the data region and map indicated data

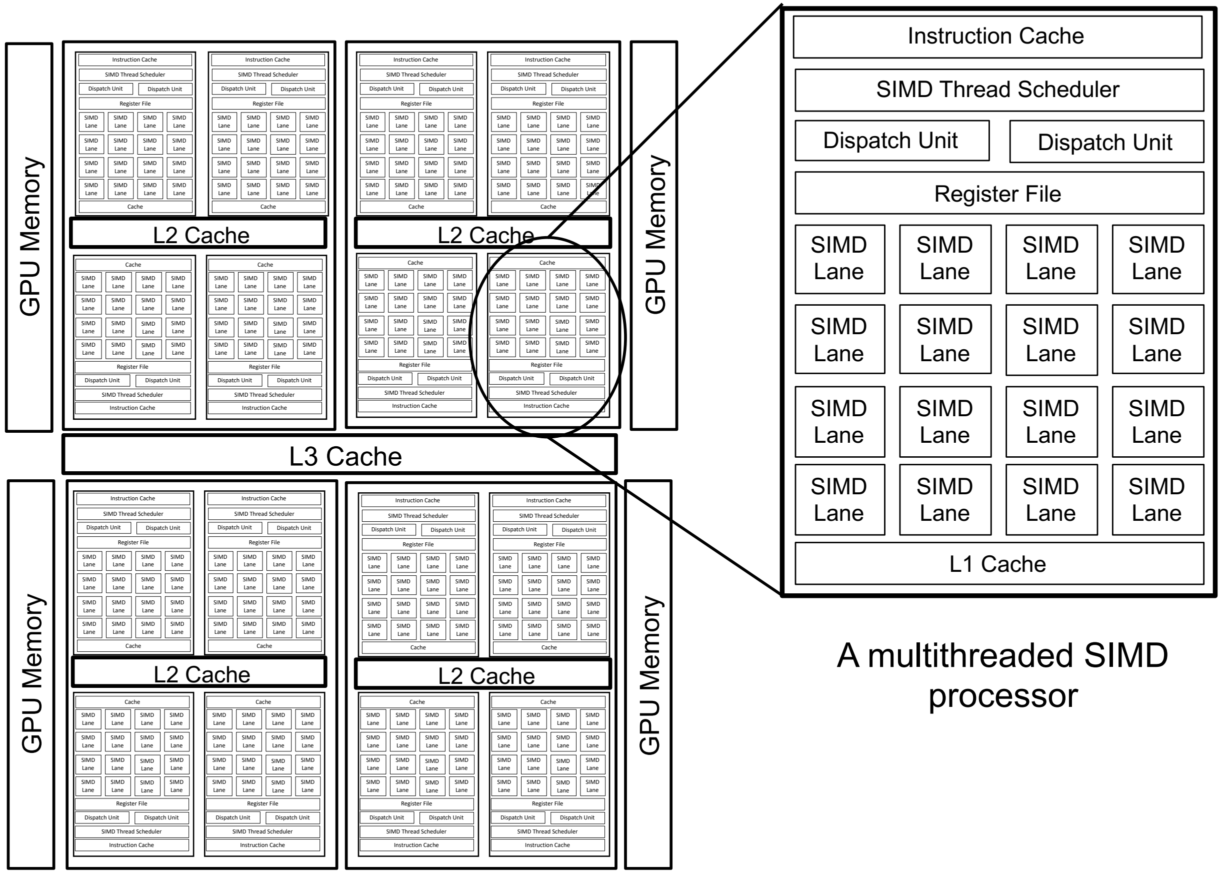
```
NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
Nvidia HPC Toolkit compiler
nvc –fast –mp=gpu –gpu=cc75 heat.c
```

Getting the data movement between host memory and device memory is key.

What are the other major issues to consider when optimizing performance?

Occupancy: Keep all the GPU resources busy

- In our “GPU cartoon” we have 16 multithreaded SIMD processors each with 16 SIMD lanes …. For a total of 16^2=256 processing elements.
- You want all resources busy at all times. You do that by keeping excess work for the multithreaded SIMD processors … if they are other busy on some high latency operation, you want a new work-group is ready to be scheduled for execution.
- *Occupancy* having enough work-groups to keep the GPU busy.   To support high occupancy, you need many more work-items than SIMD-lanes.



A multithreaded SIMD processor

```
#pragma omp parallel for
for(int i=0;i<N;i++)
   for(int j=0;j<N;j++)
      for(int k=0;k<N;k++)
          *(C+(i*N+j)) += *(A+(i* N +k)) *  *(B+(k* N +j));
```

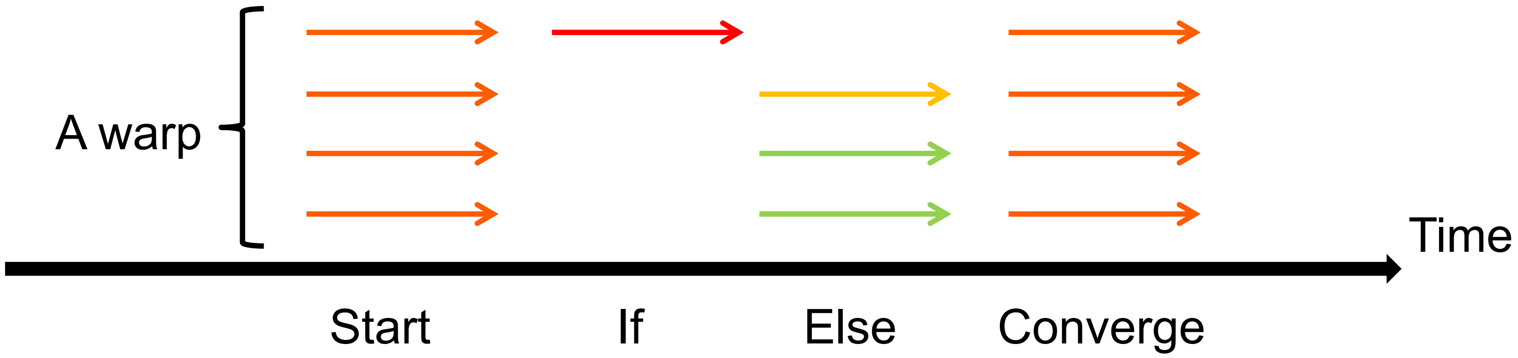
Parallelize i-loop parallelism O(N)

```
#pragma omp parallel for collapse(2)
for(int i=0;i<N;i++)
   for(int j=0;j<N;j++)
      for(int k=0;k<N;k++)
          *(C+(i*N+j)) += *(A+(i* N +k)) *  *(B+(k* N +j));
```

Parallelize combined i/j-loops parallelism O(N^2)

Converged Execution: Single Instruction Multiple Data

- Individual work-items of a warp start together at the same program address
- Each work-item has its own instruction address counter and register state
  - Each work-item is free to branch and execute independently
  - Supports the Single Program Multiple Data (SPMD) pattern.
- Branch behavior
  - Each branch will be executed serially
  - Work-items not following the current branch will be disabled



Converged Execution: Branching

- GPUs tend not to support speculative execution, which means that branch instructions have high latency
- This latency can be hidden by switching to alternative work-items/work-groups, but avoiding branches where possible is still a good idea to improve performance
- When different work-items executing within the same SIMD ALU array take different paths through conditional control flow, we have ***divergent branches*** (*vs.* ***uniform branches***)
- Divergent branches are bad news: some work-items will stall while waiting for the others to complete
- We can use predication, selection and masking to convert conditional control flow into straight line code and significantly improve the performance of code that has lots of conditional branches

Branching

Conditional execution

```
// Only evaluate expression
// if condition is met
if (a > b)
{
  acc += (a - b*c);
}
```

Selection and masking

```
// Always evaluate expression
// and mask result
temp = (a - b*c);
mask = (a > b ? 1.f : 0.f);
acc += (mask * temp);
```

Coalesced memory accesses

- ***Coalesced memory accesses*** are key for high performance code, especially on GPUs
- In principle, it’s very simple, but frequently requires transposing or transforming data on the host before sending it to the GPU
- Sometimes this is an issue of Array of Structures vs. Structure of Arrays (AoS vs. SoA)

Memory layout is critical to performance

- Structure of Arrays vs. Array of Structures
  - Array of Structures (AoS) more natural to code:

```
struct Point{ float x, y, z, a; };
Point *Points;
```

x	y	z	a	…	x	y	z	a	…	x	y	z	a	…	x	y	z	a	…

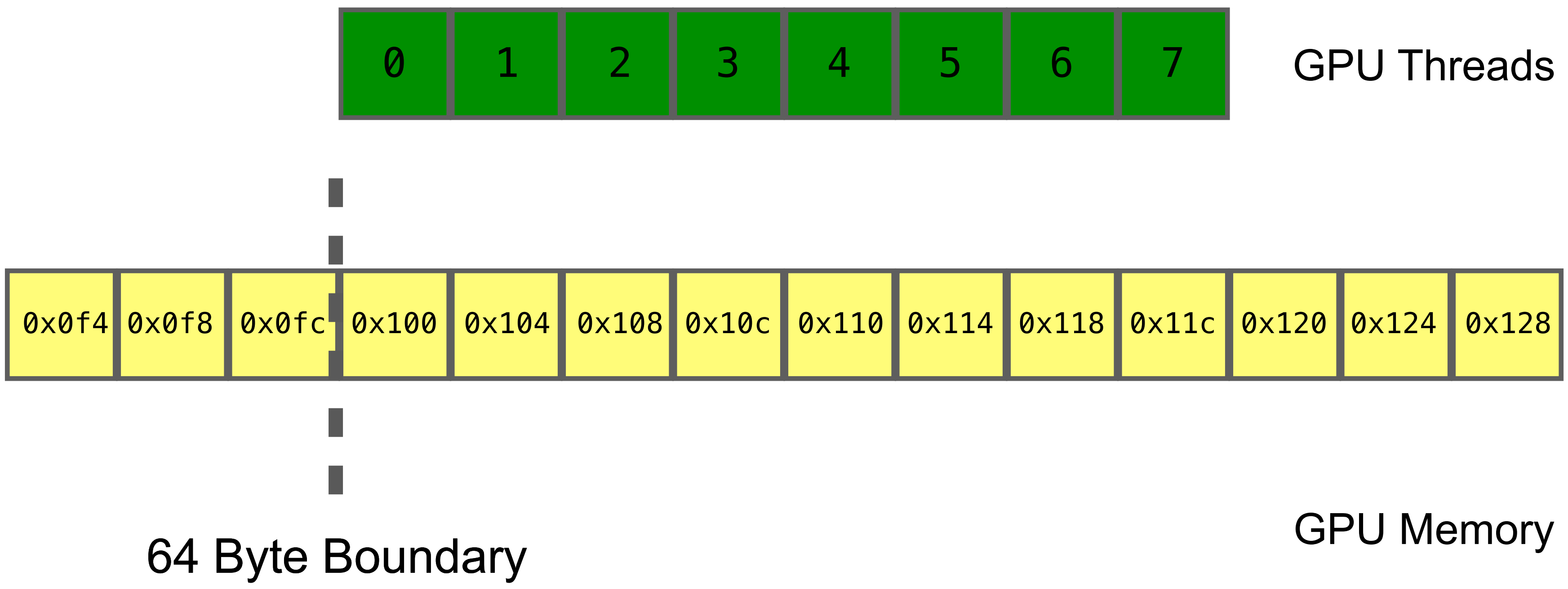
  - Structure of Arrays (SoA) suits memory coalescence in vector units

```
struct { float *x, *y, *z, *a; } Points;
```

x	x	x	x	…	y	y	y	y	…	z	z	z	z	…	a	a	a	a	…

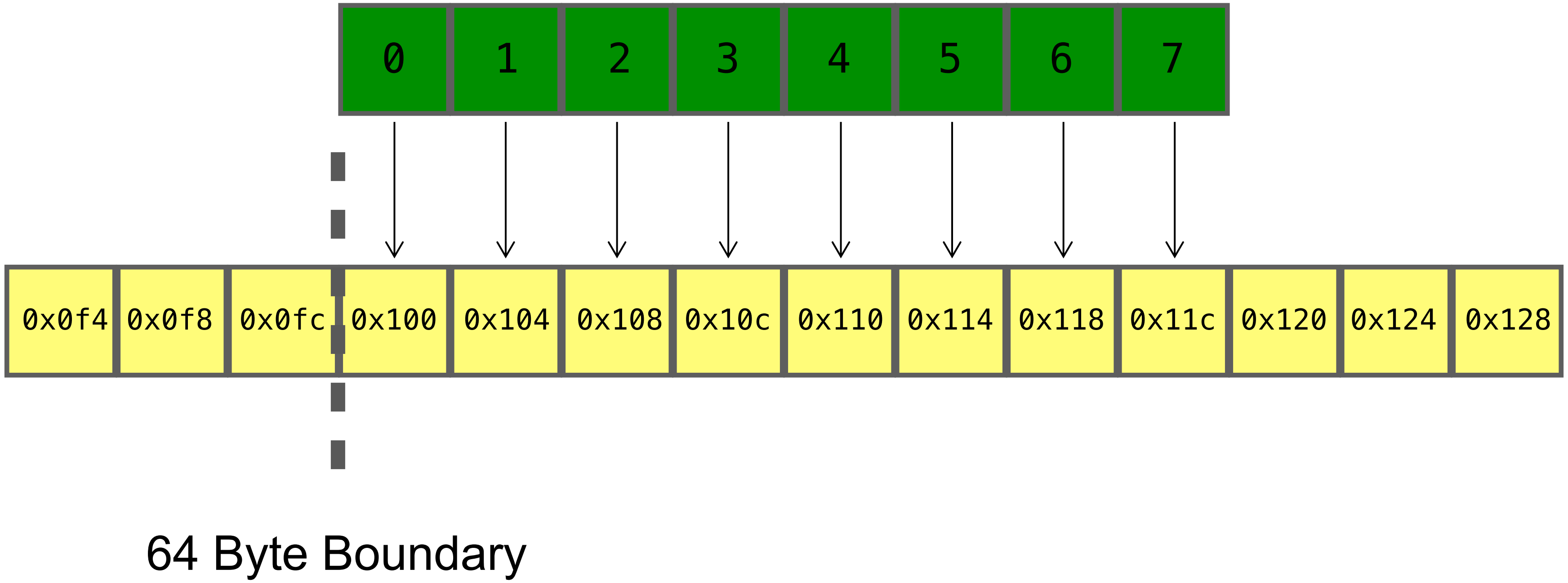
Adjacent work-items/vector-lanes like to access adjacent memory locations

Memory access patterns



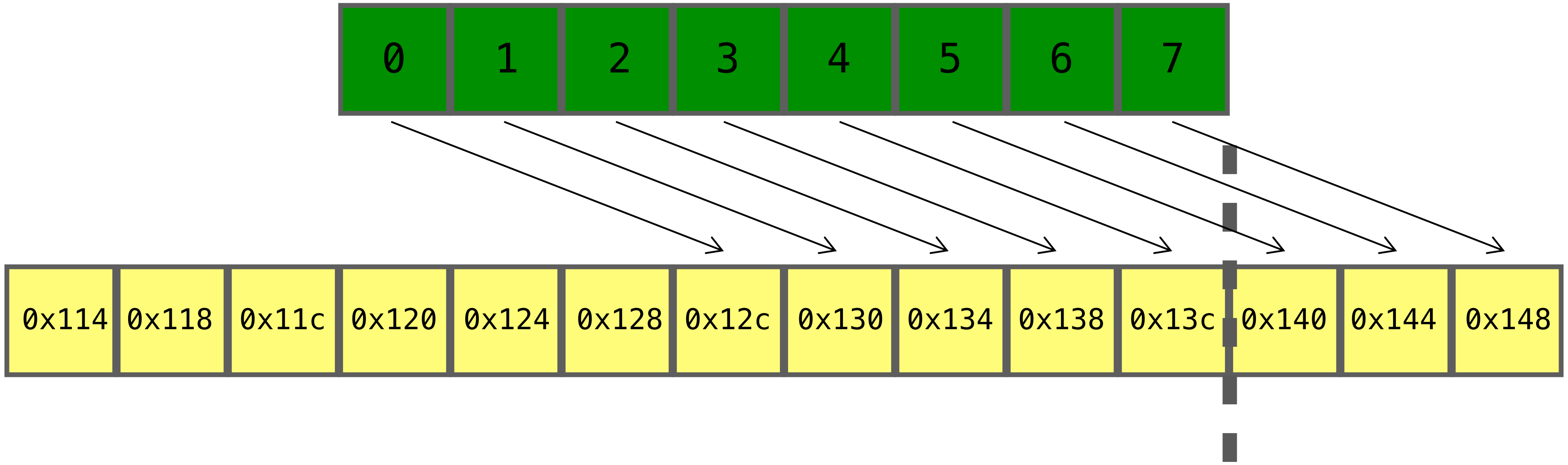
Memory access patterns

```
float val1 = memA[id];
```



Memory access patterns

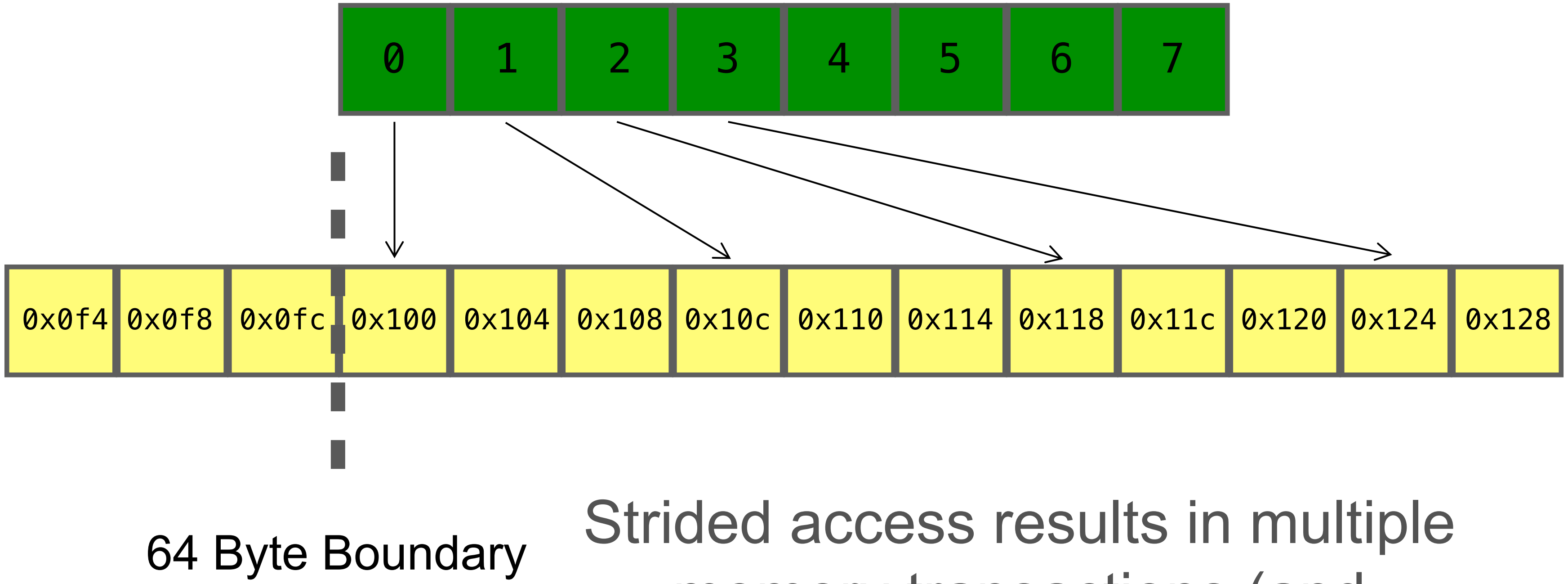
```
const int c = 3;
float val2 = memA[id + c];
```



64 Byte Boundary

Memory access patterns

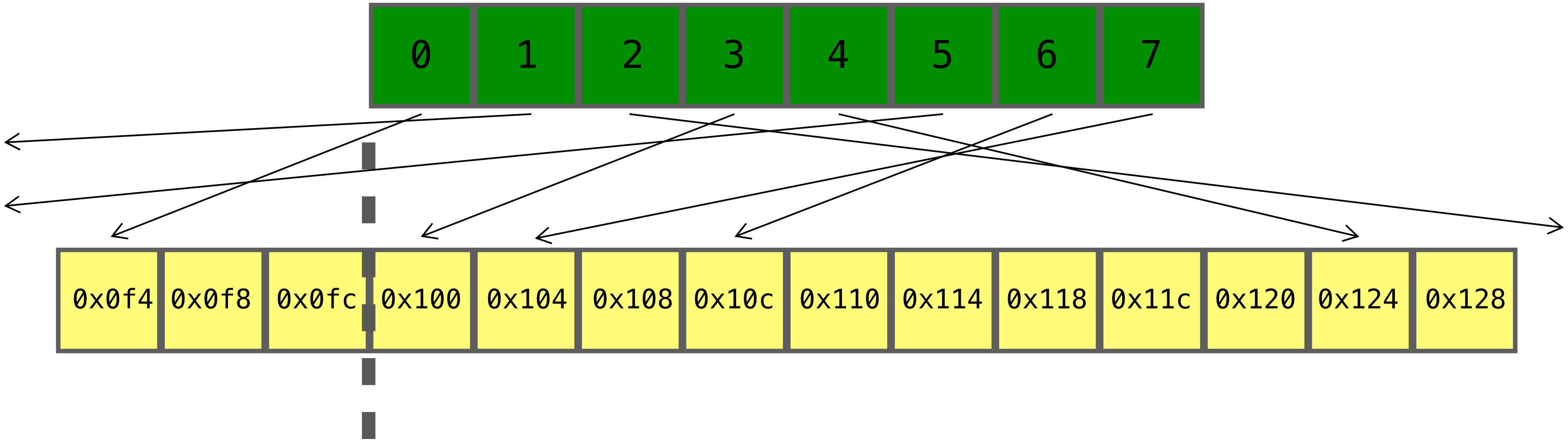
```
float val3 = memA[3*id];
```



Strided access results in multiple memory transactions (and kills throughput)

Memory access patterns

```
const int loc =
  some_strange_func(id);

float val4 = memA[loc];
```



64 Byte Boundary

Coalescence

- **Coalesce** - to combine into one
- Coalesced memory accesses are key for high bandwidth
- Simply, it means, if thread *i* accesses memory location *n* then thread *i+1* accesses memory location *n+1*
- In practice, it’s not quite as strict…

```
for (int id = 0; id < size; id++)
{
  // ideal
     float val1 = memA[id];

  // still pretty good
     const int c = 3;
     float val2 = memA[id + c];

  // stride size is not so good
     float val3 = memA[c*id];

  // terrible
     const int loc =
       some_strange_func(id);

     float val4 = memA[loc];
}
```

Exercise

- Optimize the stencil ‘solve’ kernel.
- Start with your code with optimized memory movement from the last exercise.
- Experiment with the optimizations we’ve discussed.
- Focus on the memory access pattern.
- Try different input sizes to see the effect of the optimizations.
- Keep an eye on the solve time as reported by the application.

- #pragma omp target
- #pragma omp target enter data map(to: list)
- #pragma omp target exit data map(from: list)
- #map(to:list) map(from:list) map(tofrom:list)
- #pragma omp target teams loop

```
module load nvhpc/24.5
nvc -O3 -mp=gpu  heat.c –o heat
sbatch job_gpu.sh
```

```
#!/bin/bash
#SBATCH -p sched_mit_psfc_gpu_r8
#SBATCH --reservation=cpsfr_2025
#SBATCH -o gpu-job.log-%j
#SBATCH --gres=gpu:1
module load nvhpc/24.5
OMP_TARGET_OFFLOAD=MANDATORY ./heat
```

Heat diffusion problem: 5-point stencil code

Parallel CPU and GPU results, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;
// malloc and initialize u_tmp and u (code not shown)


for (int t = 0; t < nsteps; ++t) {

   #pragma omp parallel for
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
     }
   }
   // Pointer swap for  next step
   tmp = u;
   u = u_tmp;
   u_tmp = tmp;
 }
}
```

This is the ij loop order. Swap these loops to get the ji order.

What if I optimize the CPU code?

… just swap the i and j loops. Using collapse(2) did not help on the GPU or the CPU

CPU

Num threads	ij loop order	ji loop order
1	1.512849	0.262260
2	0.776229	0.132453
4	0.400822	0.064220
8	0.227317	0.046586

Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.
Nvidia HPC Toolkit compiler nvc –fast –fopenmp heat.c

All times in seconds

GPU

ij without timing enter and exit data	ij loop order	ji without timing enter and exit data	ji loop order
0.056830	0.417887	0.020123	0.358905

This GPU code used the target enter data and target exit data

NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
Nvidia HPC Toolkit compiler. nvc -fast -mp=gpu heat.c

Solution: collapse + swap loop order

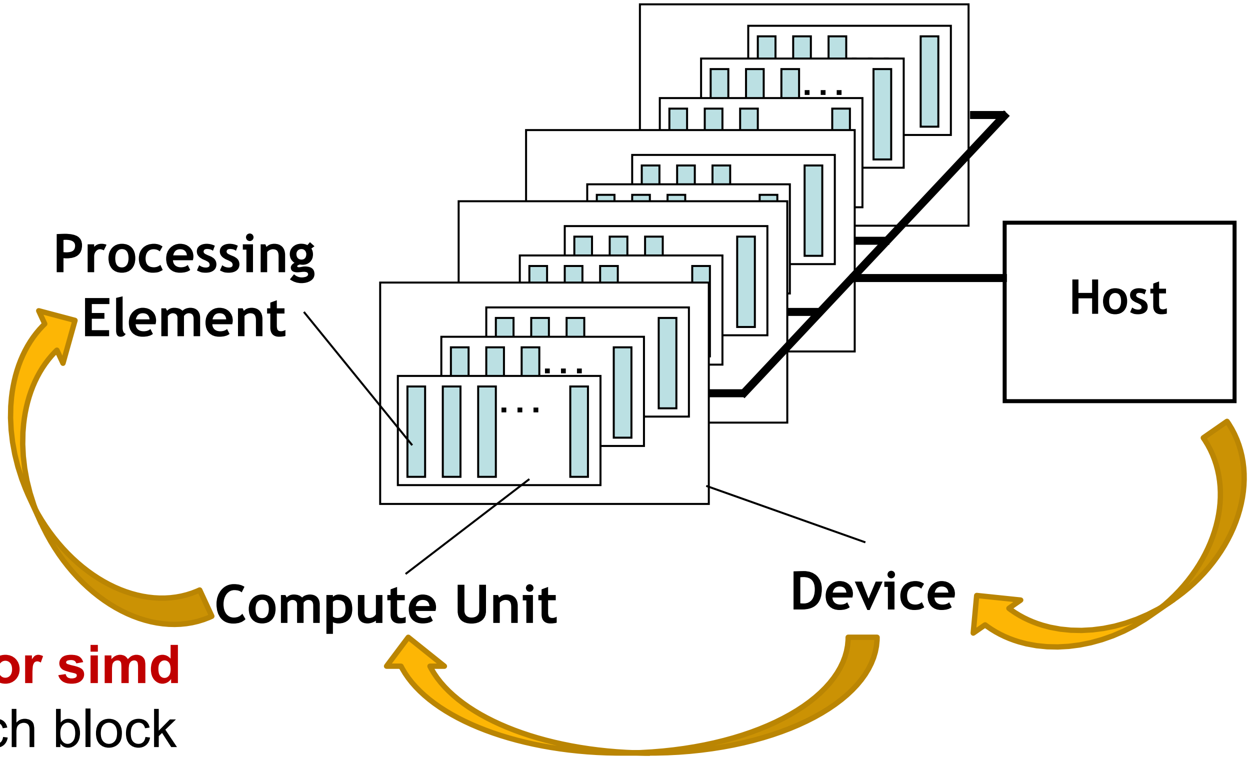
```
// Compute the next timestep, given the current timestep
void solve(const int n, const double alpha, const double dx, const double dt, const double * restrict u,
double * restrict u_tmp) {
  // Finite difference constant multiplier
  const double r = alpha * dt / (dx * dx);
  const double r2 = 1.0 - 4.0*r;

  // Loop over the nxn grid
  #pragma omp target
  #pragma omp loop collapse(2)
  for (int j = 0; j < n; ++j) {
    for (int i = 0; i < n; ++i) {


      // Update the 5-point stencil, using boundary conditions on the edges of the domain.
      // Boundaries are zero because the MMS solution is zero there.
      u_tmp[i+j*n] =  r2 * u[i+j*n] +
      r * ((i < n-1) ? u[i+1+j*n] : 0.0) +
      r * ((i > 0)   ? u[i-1+j*n] : 0.0) +
      r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
      r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
}}}
```

Create more work … to better fill the processing elements of the GPU

Swap the i and j loops so that the i+j*n memory accesses are contiguous

The loop construct is great, but sometimes you want more control.

Our host/device Platform Model and OpenMP



Parallel for simd to run each block of loop iterations on the processing elements

Teams construct to create a league of teams with one team of threads on each compute unit.

Distribute construct to assign blocks of loop iterations to teams.

Target construct to get onto a device

teams and distribute constructs

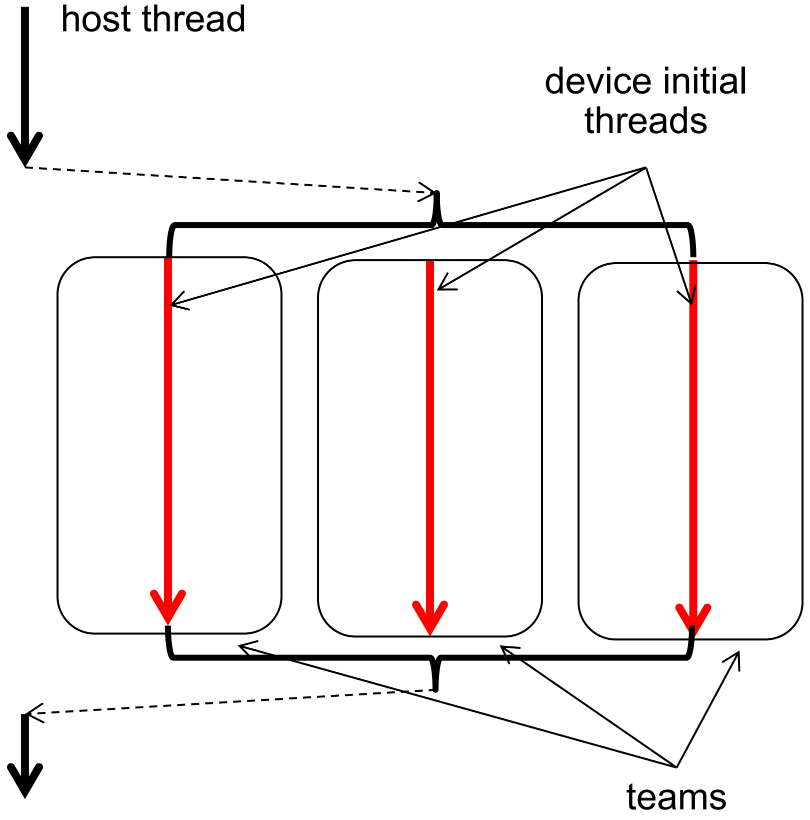
- The **teams** construct
  - Similar to the **parallel** construct
  - It starts a league of thread teams
  - Each team in the league starts as one initial thread – a team of one
  - Threads in different teams cannot synchronize with each other
  - The construct must be “perfectly” nested in a **target** construct

- The **distribute** construct
  - Similar to the **for** construct
  - Loop iterations are workshared across the initial threads in a league
  - No implicit barrier at the end of the construct
  - **dist_schedule(*kind[, chunk_size]*)**
    - If specified, scheduling kind must be static
    - Chunks are distributed in round-robin fashion in chunks of size ***chunk_size***
    - If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

Create a league of teams and distribute a loop among them

- teams construct
- distribute construct

```
#pragma omp target
#pragma omp teams
#pragma omp distribute
 for (i=0;i<N;i++)
   …
```

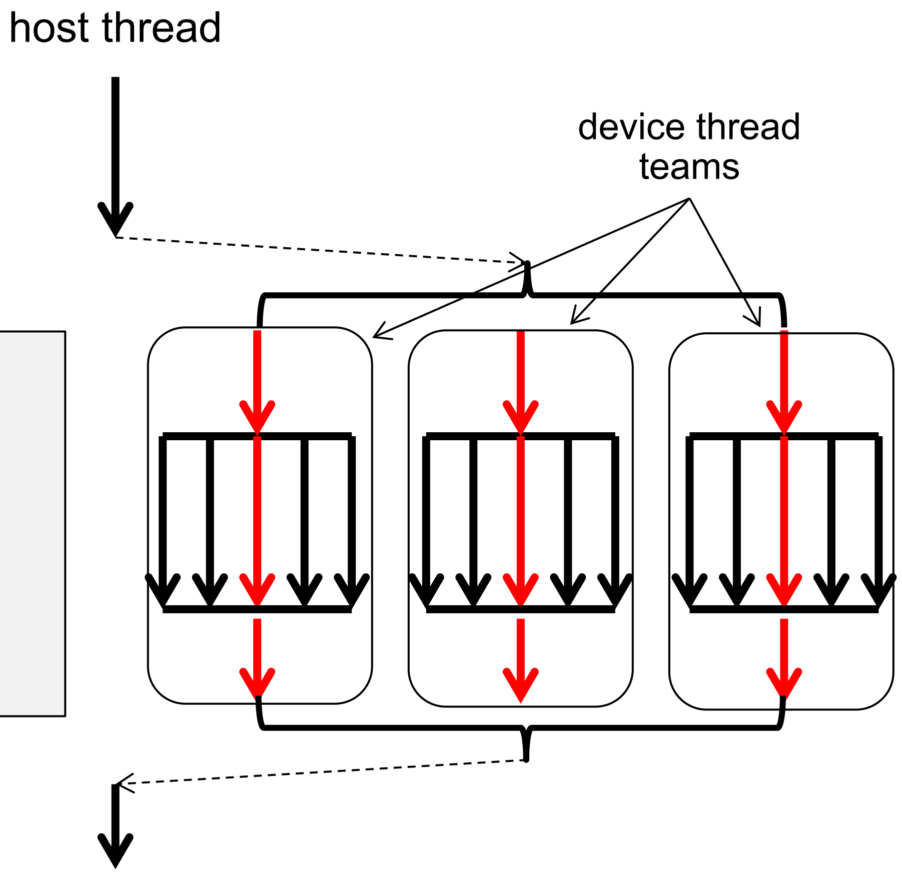


- Transfer execution control to MULTIPLE device initial threads
- Workshare loop iterations across the initial threads.

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for

```
#pragma omp target
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
 for (i=0;i<N;i++)
   ...
```

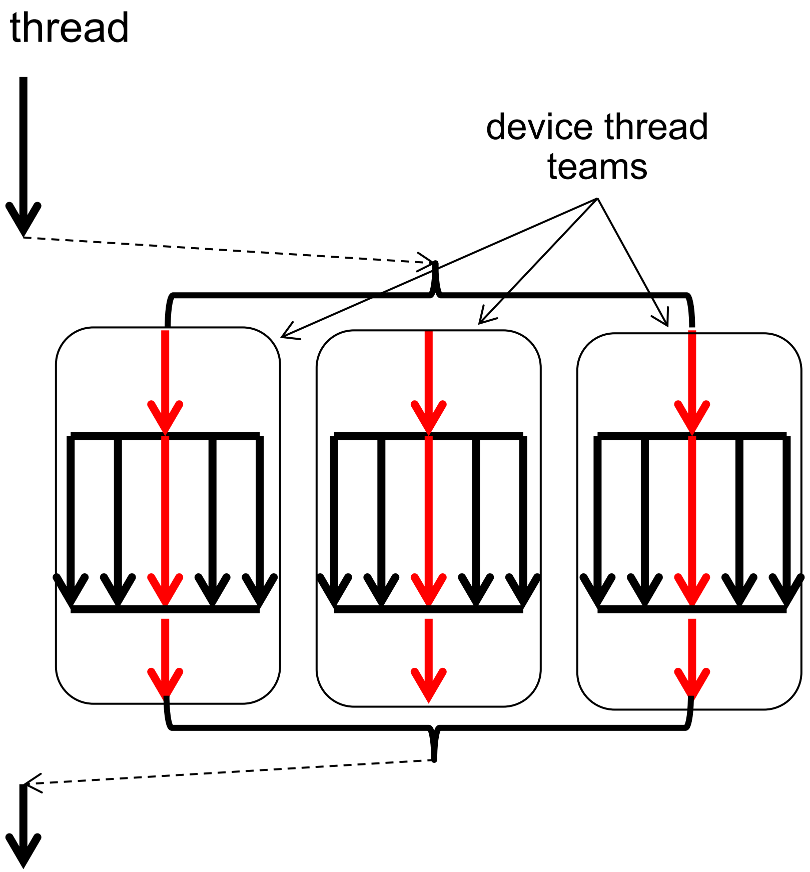


- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- loop

```
#pragma omp target
#pragma omp teams
#pragma omp loop
 for (i=0;i<N;i++)
   …
```



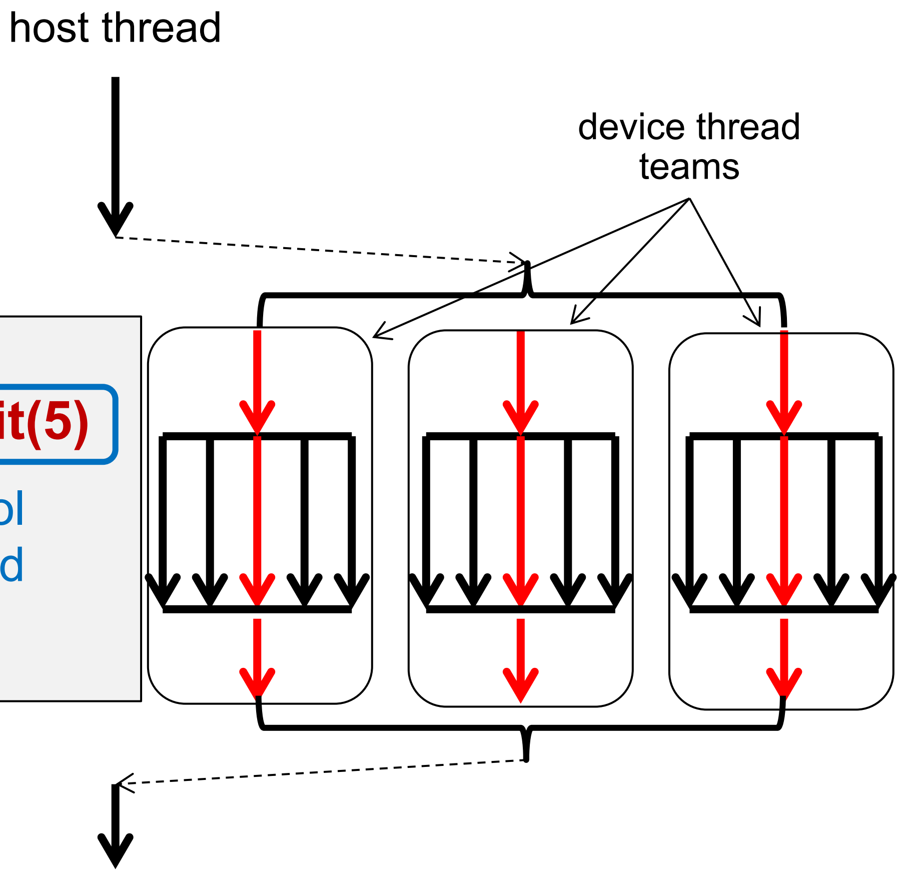
- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for simd

```
#pragma omp target
#pragma omp teams num_teams(3) thread_limit(5)
#pragma omp distribute
#pragma omp parallel for
 for (i=0;i<N;i++)
   …
```

Explicit control of number and size of teams

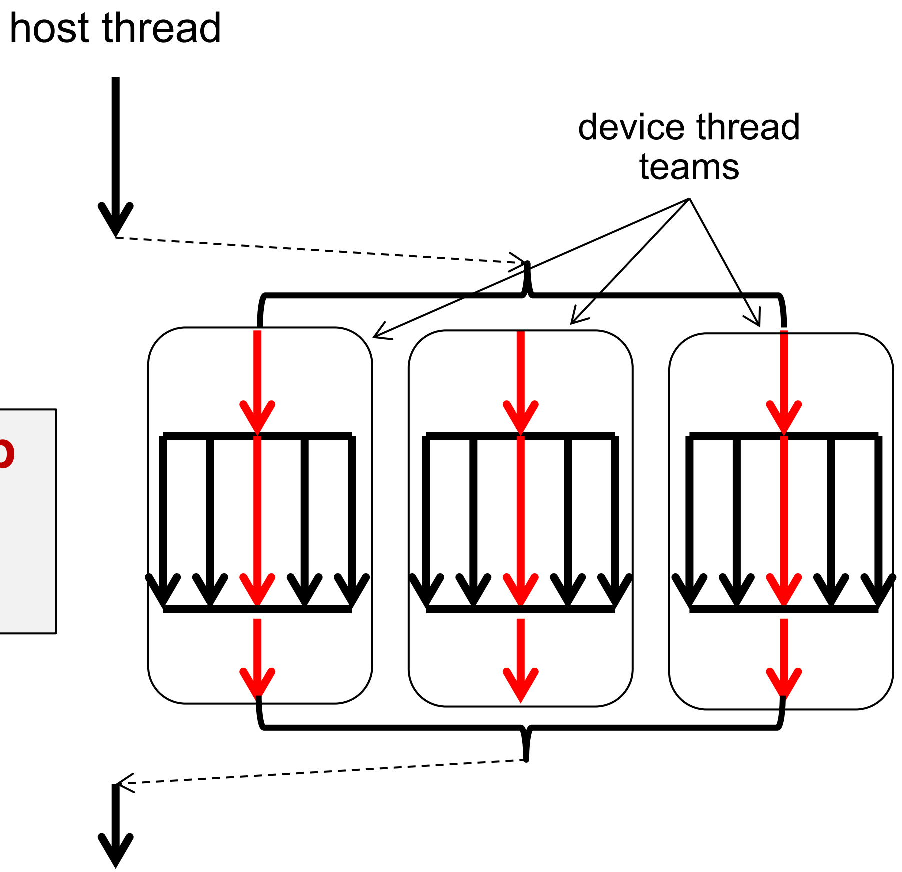


- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- Combined construct

```
#pragma omp target  teams loop
for (i=0;i<N;i++)
   …
```



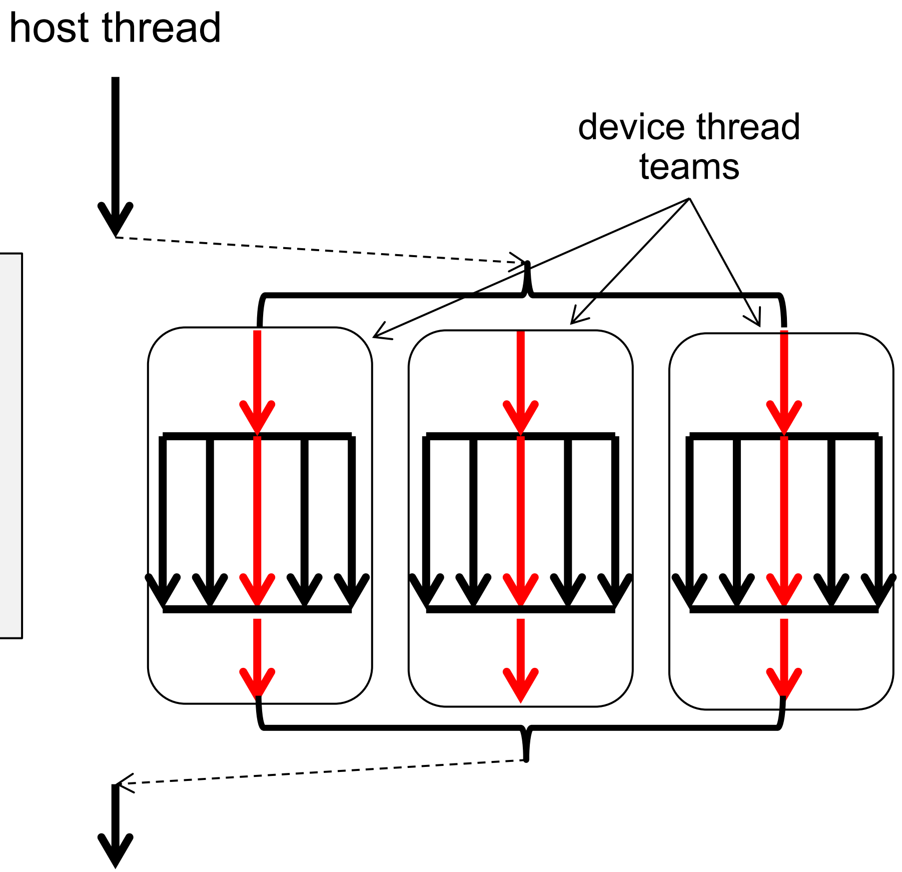
- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for simd

Works with
nested loops
as well

```
#pragma omp target
#pragma omp teams distribute
 for (i=0;i<N;i++)
 #pragma omp parallel for
 for (j=0;j<M;i++)
   ...
```



- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

There is MUCH more … beyond what have time to cover

- Do as much as you can with a simple loop construct. It's portable and as compilers improve over time, it will keep up with compiler driven performance improvements.
- But sometimes you need more:
  - Control over number of teams in a league and the size of the teams
  - Explicit scheduling of loop iterations onto the the teams
  - Management of data movement across the memory hierarchy: global vs. shared vs. private …
  - Calling optimized math libraries (such as cuBLAS)
  - Multi-device programming
  - Asynchrony
- Ultimately, you may need to master all those advanced features of GPU programming. But start with loop. Start with how data on the host maps onto the device (i.e. the GPU). Master that level of GPU programming before worrying about the complex stuff.

The book, "Programming your GPU with OpenMP", Tom Deakin and Tim Mattson, MIT Press, 2023 covers these additional topics.

This is the end … well almost the end.

Let’s wrap up with a few high-level comments about the state of GPU programming more generally

SIMT Programming models: it’s more than just OpenMP

- CUDA:
  - Released ~2006. Made GPGPU programming “mainstream” and continues to drive innovation in SIMT programming.
    - Downside: proprietary to NVIDIA

- OpenCL:
  - Open Standard for SIMIT programming created by Apple, Intel, NVIDIA, AMD, and others. 1st release in 2009.
  - Supports CPUs, GPUs, FPGAs, and DSP chips. The leading cross platform SIMT model.
    - Downside: extreme portability means verbose API. Painfully low level especially for the host-program.

- Sycl:
  - C++ abstraction layer implements SIMT model with kernels as lambdas. Closely aligned with OpenCL. 1st release 2014
    - Downside: Cross platform implementations only emerging recently.

- Directive driven programming models:
  - **OpenACC**: they split from an OpenMP working group to create a competing directive driven API emphasizing descriptive (rather than prescriptive) semantics.
    - Downside: NOT an Open Standard. Controlled by NVIDIA.
  - **OpenMP**: Mixes multithreading and SIMT. Semantics are prescriptive which makes it more verbose. A truly Open standard supported by all the key GPU players.
    - Downside: Poor compiler support so far … but that will change over the next couple years.

Vector addition with CUDA

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
   / ... allocate other arrays (b and c), fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

CUDA kernel as function

Unified shared memory … allocate on host, visible on device too

Enqueue the kernel to execute on the Grid

Note: Not all GPUs support a unified shared memory. We assume its available hear to we don't need to clutter the code with explicit data movement.

Vector addition with SYCL

```
// Compute sum of length-N vectors: C = A + B
#include <CL/sycl.hpp>

int main () {
    int N = ... ;
    float *a, *b, *c;
    sycl::queue q;
    *a = (float *)sycl::malloc_shared(N * sizeof(float), q);
  // ... allocate other arrays (b and c), fill with data

        q.parallel_for(sycl::range<1>{N},
                 [=](sycl::id<1> i) {
                   c[i] = a[i] + b[i];
                 });
         q.wait();
}
```

Create a queue for SYCL commands

Unified shared memory … allocate on host, visible on device too

Kernel as a C++ Lambda function

[=] means capture external variables by value.

Note: Not all GPUs support a unified shared memory. We assume its available hear to we don’t need to clutter the code with explicit data movement.

Vector addition with OpenACC

- Let’s add two vectors together …. C = A + B

Host waits here until the kernel is done. Then the output array c is copied back to the host.

```
void vadd(int n,
          const float *a,
          const float *b,
          float *restrict c)
{
  int i;
 #pragma acc parallel loop
  for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
}
int main(){
float *a, *b, *c;  int n = 10000;
// allocate and fill a and b

    vadd(n, a, b, c);

}
```

Assure the compiler that c is not aliased with other pointers

Turn the loop into a kernel, move data to a device, and launch the kernel.

Why so many ways to do the same thing?

- The parallel programming model people have failed you …
  - It's more fun to create something new in your own closed-community that work across vendors to create a portable API

- The hardware vendors have failed you …
  - Don't you love my "walled garden"? It's so nice here, programmers, just don't even think of going to some other platform since your code is not portable.

- The standards community has failed you …
  - Standards are great, but they move too slow. OpenACC stabbed OpenMP in the back and I'm pissed, but their comments at the time were spot-on (OpenMP was moving so slow … they just couldn't wait).

- The applications community failed themselves …
  - If you don't commit to a standard and use "the next cool thing" you end up with the diversity of overlapping options we have today. Think about what happened with OpenMP and MPI.

Summary

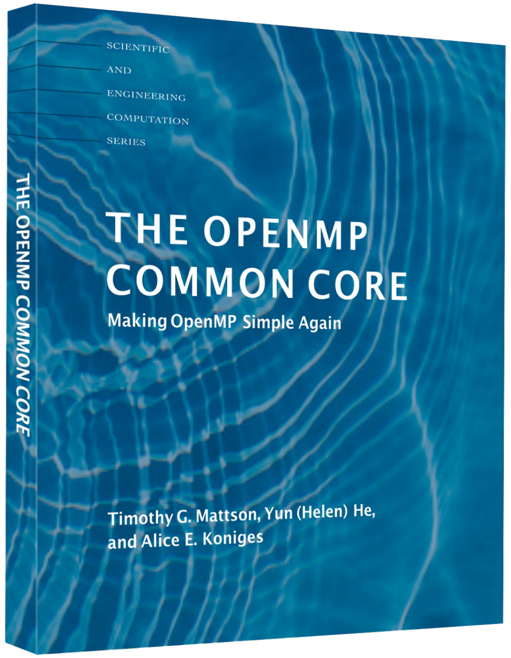
- Parallel computing is fun … but it can be hard.
- Fortunately, if you stick to the Big-3 and the core patterns of parallel computing for HPC, it’s not too overwhelming
  - The big 3: MPI, OpenMP, and “a GPU programming model”
  - Key Patterns: SPMD, loop level parallelism, geometric decomposition, divide and conquer, and SIMT

- Some day we’ll automate the hard-parts with Machine Programming, but that may be 10 or 20 years!!!!

To learn more about OpenMP

The OpenMP web site has a great deal of material to help you with OpenMP www.openmp.org
Reading the spec is painful … but each spec has a collection of examples. Study the examples, don't try to read the specs

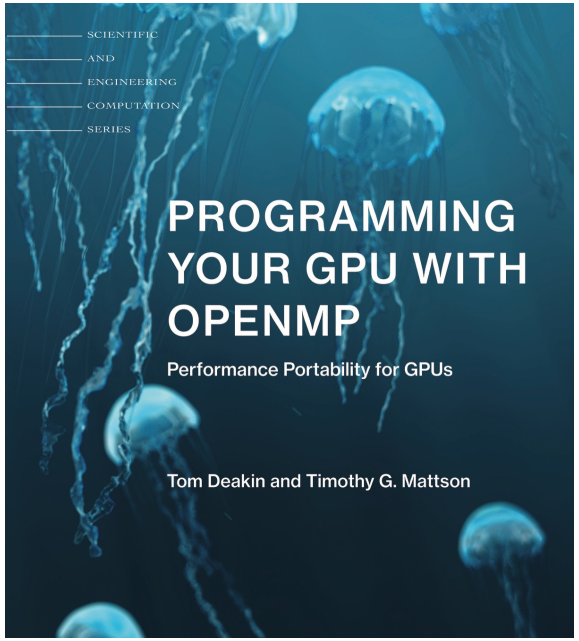
Since the specs are written ONLY for implementors … programmers need the OpenMP Books to master OpenMP.



Start here … learn the basics and build a foundation for the future

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

Learn advanced features in OpenMP including tasking and GPU programming (up to version 4.5)


SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES
PROGRAMMING YOUR GPU WITH OPENMP
Performance Portability for GPUs
Tom Deakin and Timothy G. Mattson

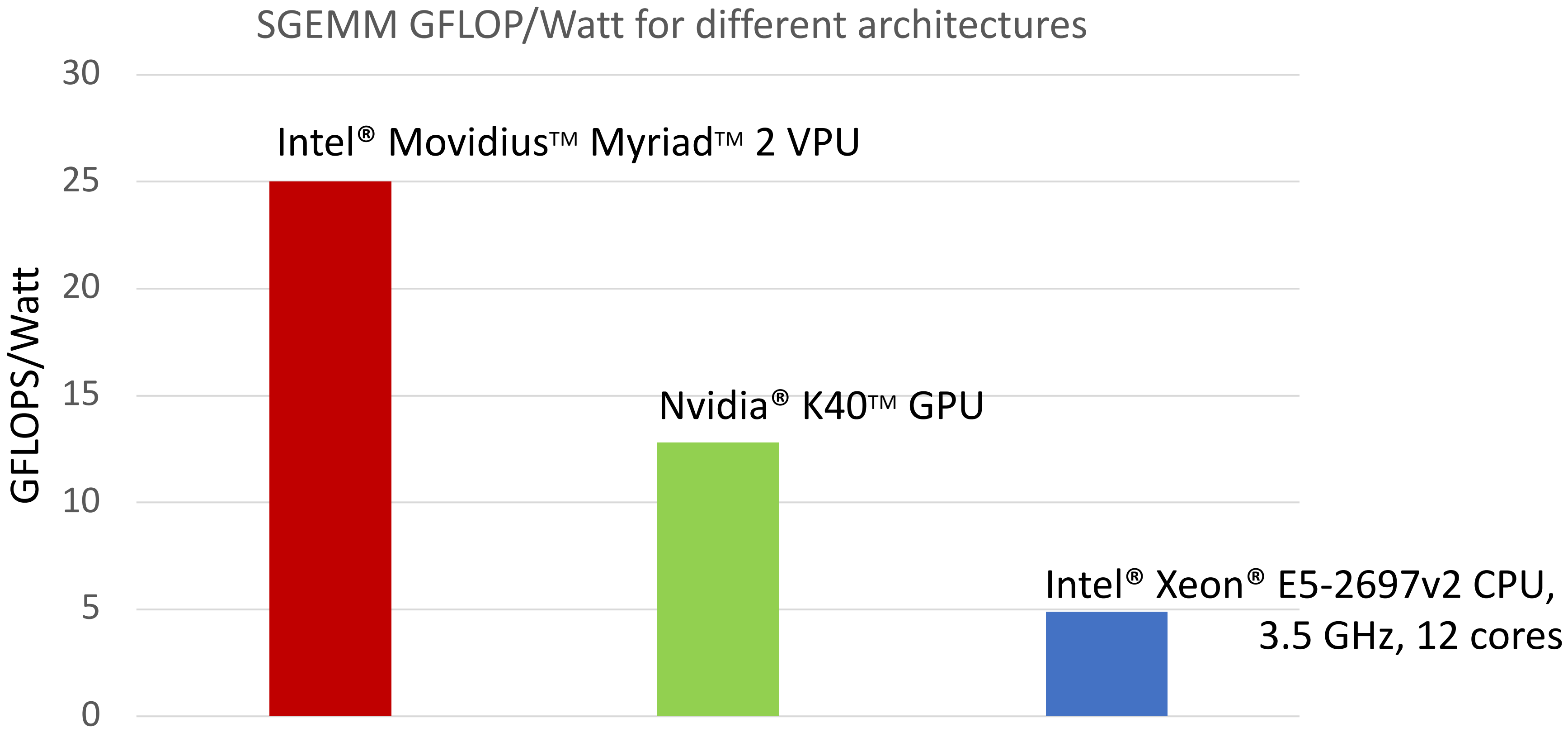
Learn all the details of GPU programming with OpenMP
(up to version 5.2)
Released in November 2023

Backup … and a bit of extra content

- Heterogeneous computing
- The Jacobi solver case study
- Writing functions to call from inside a kernel

If you care about power, the world is heterogeneous?

Specialized processors doing operations suited to their architecture are more efficient than general purpose processors.

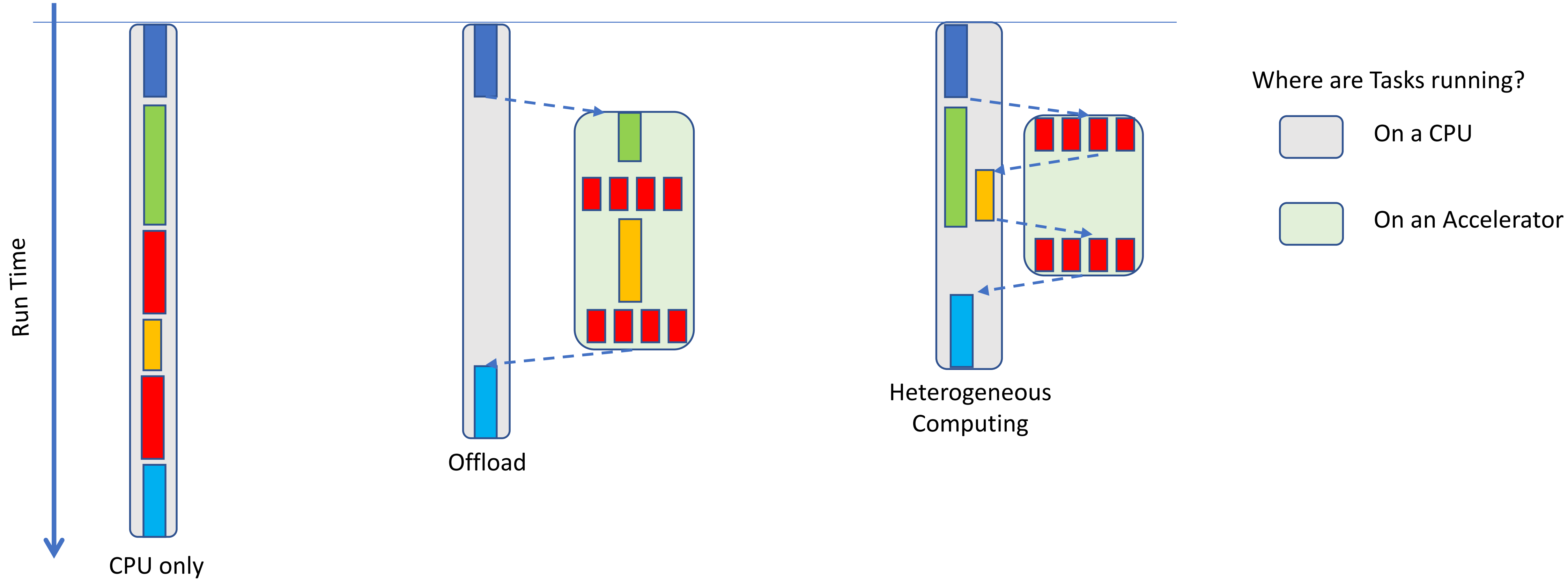


Hence, future systems will be increasingly heterogeneous … GPUs, CPUs, FPGAs, and a wide range of accelerators

Source: Suyash Bakshi and Lennart Johnsson, “A Highly Efficient SGEMM Implementation using DMA on the Intel/Movidius Myriad-2. IEEE International Symposium on Computer Architecture and High Performance Computing, 2020

Offload vs. Heterogeneous computing

- **Offload**: The CPU moves work to an accelerator and waits for the answer.
- **Heterogeneous Computing**: Run sub-problems in parallel on the hardware best suited to them.



Example: Single-cell RNA-Seq benchmark (SCANPY)

- SCANPY … a widely used tool for studying gene expression. All data are elapsed time in seconds
- We started with results from an Nvidia blog (Example 2 from link), optimized code for one socket of Intel® Xeon® 8380 CPU and then "simulated" heterogeneous computing result by taking the faster of CPU and GPU execution times.

Pipeline stages	64 vCPUs n1-highmem-64 (off-the-shelf Python)	A100 40Gb (Clara Parabricks)	ICX-1s, 40 cores (**optimized by Intel**)	[redacted]
Data Loading & Preprocessing	1120	475	15.7	[redacted]
PCA	44	17.8	5.0	[redacted]
T-SNE	6509	37	205.6	[redacted]
K-means (single iteration)	148	2	7.1	[redacted]
KNN	154	62	59.8	[redacted]
UMAP	2571	21	84.5	[redacted]
Louvain clustering	1153	2.4	6.0	[redacted]
Leiden clustering	6345	1.7	28.4	[redacted]
Reanalysis of subgroup	255	17.9	22.5	[redacted]
Rest	39	49.2	49.0	[redacted]
End-to-End runtime	**18338**	**686**	**483.6**	[redacted]

Redacted

Imagine mixing the best of the CPU and GPU numbers. What would the performance look like?

https://github.com/clara-parabricks/rapids-single-cell-examples

github repository as of Dec 16, 2020

Clara Parabricks: Nvidia solution stack built on RAPIDS for healthcare applications

This column shows the potential of heterogenous computing. We ignored extra communication and synchronization overhead, so actual runtimes would be slightly greater.

Lessons learned:

- Be careful comparing unoptimized python to hand-tuned CUDA code
- GPUs are great. So are CPUs if you fully utilize all the cores and vector units.
- What you really want is the best of both worlds. **You want heterogeneous computing!**

See Backup for workloads and configurations. Results may vary.

Source: Github repository as of Dec 16, 2020 - Example 2: Single-cell RNA-seq of 1.3 Million Mouse Brain Cells comparing CPU (n1-highmem-64 64 vCPUs) vs GPU (n1-highmem-16. https://github.com/clara-parabricks/rapids-single-cell-examples. Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.
1S Ice Lake: See Backup for workloads and configurations. Results may vary.

SCANPY workload details and system configuration

ame	Intel® Xeon® Platinum 8380
Time	Jan 20, 2022
Manufacturer	Intel Corporation
Product Name	Intel® Xeon® Platinum 8380
BIOS Version	SE5C6200.86B.0020.P23.2103261309
OS	Rocky Linux release 8.5 (Green Obsidian)
Kernel	4.18.0-240.22.1.el8_3.crt6.x86_64
Microcode	0xd000270
IRQ Balance	enabled
CPU Model	Intel(R) Xeon(R) Platinum 8380 CPU @ 2.30GHz
Base Frequency	2.3GHz
Maximum Frequency	3.4GHz
All-core Maximum Frequency	2.5GHz
CPU(s)	40
Thread(s) per Core	2
Core(s) per Socket	40

Socket(s)	1
NUMA Node(s)	1
Prefetchers	
Turbo	Enabled
PPIN(s)	
Power & Perf Policy	Performance
TDP	270 watts
Frequency Driver	
Frequency Governer	Performance
Frequency (MHz)	
Max C-State	
Installed	Intel® Xeon® Platinum 8380 40c D1 DDR4 16*16GB@3200MHz - Mellanox HDR
Huge Pages Size	2048 kB
Transparent Huge Pages	Always
Automatic NUMA Balancing	Enabled

- The following was done to optimize the SCANPY benchmark
    - Data preprocessing - used warm file cache and multi-threaded using Numba JIT
    - PCA, K-means, KNN – Used the Intel extension for scikit-learn.
    - t-SNE - Used optimized version from Intel's oneDAL Library.
    - Parallelized quadtree building, sorting and summarization steps using Morton codes.
    - UMAP - optimized the UMAP code using AVX512/AVX2. Used MKL for eigenvalue computation.
    - Louvain and Leiden algorithms – collaborated with Katana Graph to get well optimized versions and integrated them into SCANPY.

Backup ... and a bit of extra content

- Heterogeneous computing
- ➡ The Jacobi solver case study
- Writing functions to call from inside a kernel

Our running example: Jacobi solver

- An iterative method to solve a system of linear equations
  - Given a matrix A and a vector b find the vector x such that $Ax=b$
- The basic algorithm:
  - Write A as a lower triangular (L), upper triangular (U) and diagonal matrix

$$Ax = (L+D+U)x = b$$

  - Carry out multiplications and rearrange

$$Dx=b-(L+U)x \quad \rightarrow \quad x = (b-(L+U)x)/D$$

  - Iteratively compute a new x using the x from the previous iteration

$$X_{new} = (b-(L+U)x_{old})/D$$

- Advantage: we can easily test if the answer is correct by multiplying our final x by A and comparing to b
- Disadvantage: It takes many iterations and only works for diagonally dominant matrices

Jacobi Solver

Iteratively update xnew until the value stabilizes (i.e. change less than a preset TOL)

```
<<< allocate and initialize the matrix A >>>
<<< and vectors x1, x2 and b             >>>

while((conv > TOL) && (iters<MAX_ITERS))
  {
   iters++;

   for (i=0; i<Ndim; i++){
      xnew[i] = (TYPE) 0.0;
      for (j=0; j<Ndim;j++){
         if(i!=j)
           xnew[i]+= A[i*Ndim + j]*xold[j];
      }
      xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
   }
```

```
    // test convergence
    conv = 0.0;
    for (i=0; i<Ndim; i++){
       tmp  = xnew[i]-xold[i];
       conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

   // swap pointers for next
   // iteration
   TYPE* tmp = xold;
   xold = xnew;
   xnew = tmp;

} // end while loop
```

Jacobi Solver (Parallel Target/loop, 1/2)

```
 while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
#pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                        map(to:A[0:Ndim*Ndim], b[0:Ndim])
#pragma omp loop
for (i=0; i<Ndim; i++){
         xnew[i] = (TYPE) 0.0;
         for (j=0; j<Ndim;j++){
             if(i!=j)
               xnew[i]+= A[i*Ndim + j]*xold[j];
         }
         xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (Parallel Target/loop, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
 #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                                    map(tofrom:conv)
#pragma omp loop private(i,tmp) reduction(+:conv)
for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);
   TYPE* tmp = xold;
   xold = xnew;
   xnew = tmp;
} // end while loop
```

This worked but the performance was awful. Why?

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs

Cray® XC40™ Supercomputer running Cray® Compiling Environment 8.5.3.
Intel® Xeon ® CPU E5-2697 v2 @ 2.70GHz with 32 GB DDR3. NVIDIA® Tesla® K20X, 6GB.

Data movement dominates!!!

```
while((conv > TOLERANCE) && (iters<MAX_ITERS))
 { iters++;
  xnew = iters % s ? x2 : x1;
  xold   = iters % s ? x1 : x2;

    #pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                        map(to:A[0:Ndim*Ndim], b[0:Ndim] )
     #pragma omp loop private(i,j)
     for (i=0; i<Ndim; i++){
         xnew[i] = (TYPE) 0.0;
         for (j=0; j<Ndim;j++){
             if(i!=j)
               xnew[i]+= A[i*Ndim + j]*xold[j];
         }
         xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
     }
 // test convergence
  conv = 0.0;
    #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                              map(tofrom:conv)
      #pragma loop reduction(+:conv)
      for (i=0; i<Ndim; i++){
          tmp  = xnew[i]-xold[i];
          conv += tmp*tmp;
      }
  conv = sqrt((double)conv);
 }
```

Typically over 4000 iterations!

For each iteration, copy to device $(3*Ndim+Ndim^2)*sizeof(TYPE)$ bytes

For each iteration, copy from device 2*Ndim*sizeof(TYPE) bytes

For each iteration, copy to device 2*Ndim*sizeof(TYPE) bytes

Target data directive

- The **target data** construct creates a target data region
  … use **map** clauses for explicit data management

Data is mapped onto the device at the beginning of the construct

```
#pragma omp target data map(to:A, B) map(from: C)
{
   #pragma omp target
      {do lots of stuff with A, B and C}

   {do something on the host}

   #pragma omp target
      {do lots of stuff with A, B, and C}
}
```

one or more **target regions** work within the **target data region**

Data is mapped back to the host at the end of the target data region

Jacobi Solver (Par Target Data, 1/2)

```
 #pragma omp target data map(tofrom:xold[0:Ndim],xnew[0:Ndim]) \
                      map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;

#pragma omp target
#pragma omp loop private(j) firstprivate(xnew,xold)
     for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
           if(i!=j)
             xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
     }
```

Jacobi Solver (Par Target Data, 2/2)

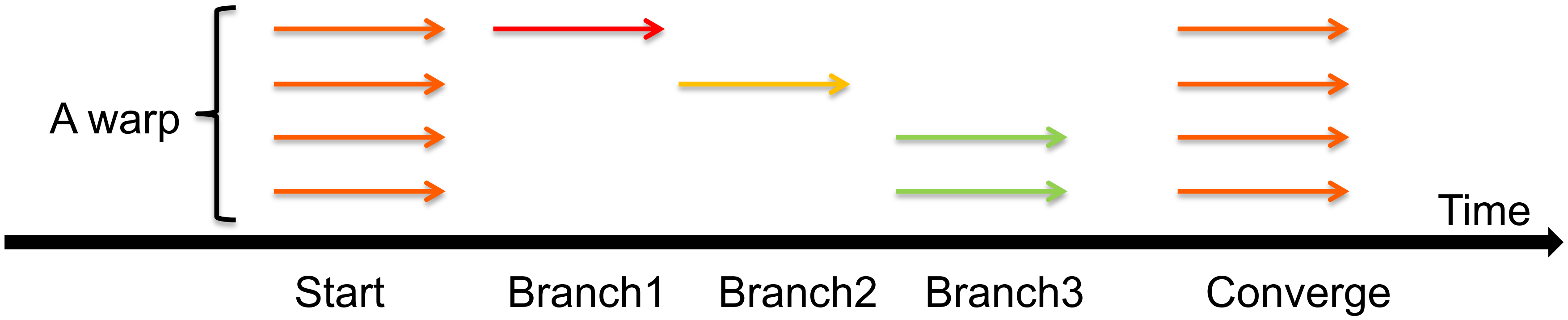
```
// test convergence
conv = 0.0;
#pragma omp target map(tofrom: conv)
#pragma omp loop private(tmp) firstprivate(xnew,xold)  reduction(+:conv)

     for (i=0; i<Ndim; i++){
         tmp  = xnew[i]-xold[i];
         conv += tmp*tmp;
     }
// end target region
 conv = sqrt((double)conv);

    TYPE* tmp = xold;
    xold = xnew;
    xnew = tmp;
} // end while loop
```

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs

Third party names are the property of their owners.

Single Instruction Multiple Data

- Individual work-items of a warp start together at the same program address
- Each work-item has its own instruction address counter and register state
  - Each work-item is free to branch and execute independently
  - Supports the SPMD pattern.
- Branch behavior
  - Each branch will be executed serially
  - Work-items not following the current branch will be disabled



Branching

Conditional execution

```
// Only evaluate expression
// if condition is met
if (a > b)
{
  acc += (a - b*c);
}
```

Selection and masking

```
// Always evaluate expression
// and mask result
temp = (a - b*c);
mask = (a > b ? 1.f : 0.f);
acc += (mask * temp);
```

Coalescence

- Coalesce - to combine into one
- Coalesced memory accesses are key for high bandwidth
- Simply, it means, if thread *i* accesses memory location *n* then thread *i+1* accesses memory location *n+1*
- In practice, it’s not quite as strict…

```
for (int id = 0; id < size; id++)
{
  // ideal
     float val1 = memA[id];

  // still pretty good
     const int c = 3;
     float val2 = memA[id + c];

  // stride size is not so good
     float val3 = memA[c*id];

  // terrible
     const int loc =
       some_strange_func(id);

     float val4 = memA[loc];
}
```

Jacobi Solver (Target Data/branchless/coalesced mem, 1/2)

```
  #pragma omp target data map(tofrom:x1[0:Ndim],x2[0:Ndim]) \
                     map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
#pragma omp target
      #pragma omp loop private(j)
     for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
             xnew[i]+= (A[j*Ndim + i]*xold[j])*((TYPE)(i != j));
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
     }
```

We replaced the original code with a poor memory access pattern

`xnew[i]+= (A[i*Ndim + j]*xold[j])`

With the more efficient

`xnew[i]+= (A[j*Ndim + i]*xold[j])`

Jacobi Solver (Target Data/branchless/coalesced mem, 2/2)

```
      //
      // test convergence
      conv = 0.0;
#pragma omp target map(tofrom: conv)
 #pragma omp loop private(tmp) reduction(+:conv)
      for (i=0; i<Ndim; i++){
         tmp  = xnew[i]-xold[i];
         conv += tmp*tmp;
      }
conv = sqrt((double)conv);
    TYPE* tmp = xold;
    xold = xnew;
    xnew = tmp;
} // end while loop
```

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs
	Above plus reduced branching	13.74 secs
	Above plus improved mem access	7.64 secs

Cray® XC40™ Supercomputer running Cray® Compiling Environment 8.5.3.
Intel® Xeon ® CPU E5-2697 v2 @ 2.70GHz with 32 GB DDR3. NVIDIA® Tesla® K20X, 6GB.

A more complicated example:

Jacobi iteration: OpenACC (GPU)

```
#pragma acc data copy(A), create(Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma acc parallel loop reduction(max:err)
   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]));
       }
    }
    #pragma acc parallel loop
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the GPU. Copy A once onto the GPU, and create Anew on the device (no copy from host)

Copy A back out to host … but only once

Source: based on Mark Harris of NVIDIA®, “Getting Started with OpenACC”, GPU technology Conf., 2012

A more complicated example:

Jacobi iteration: OpenMP target directives

```
#pragma omp target data map(A) map(alloc:Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma target
   #pragma omp teams loop reduction(max:err)
   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]));
       }
    }
    #pragma omp target
    #pragma omp teams loop
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the GPU. Map A and Anew onto the target device

Copy A back out to host … but only once

Backup ... and a bit of extra content

- Heterogeneous computing
- The Jacobi solver case study
- Writing functions to call from inside a kernel

Defining a function to be called from inside a kernel

Tell OpenMP to compile this function for the GPU (and the CPU)

Call inside a target region, and the GPU version is called.

Call on the host, and a CPU version is called

```c
#include mm_utils.h

#pragma omp declare target
void ddot(double *C, double *A, double *B, int i, int j, int Mdim, int Pdim){
   for(int k=0;k<Pdim;k++){
      /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
      *(C+(i*Mdim+j)) += *(A+(i*Pdim+k)) *  *(B+(k*Mdim+j));
   }
}
#pragma omp end declare target


void mm_gpu(int Ndim, int Mdim, int Pdim, TYPE *A, TYPE *B, TYPE *C){
  int i, j, k;

#pragma omp target teams map(tofrom:C[0:Ndim*Mdim]) map(to:B[0:Pdim*Mdim],A[0:Ndim*Pdim])
#pragma omp loop  collapse(2)
  for (i=0; i<Ndim; i++){
     for (j=0; j<Mdim; j++){
        ddot(C, A, B, i, j, Mdim, Pdim);

     }
```

This is in my file mm_gpu.c