The Parallel Programming world beyond OpenMP

Tim Mattson

Intel Corp.

timothy.g.mattson@ intel.com

Legal Disclaimer & Optimization Notice

- INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
-

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Disclaimer

- The views expressed in this talk are those of the speaker and not his employer.

- If I say something "smart" or worthwhile:
  - Credit goes to the many smart people I work with.

- If I say something stupid…
  - It's my own fault

I work in Intel's research labs. I don't build products. Instead, I get to poke into dark corners and think silly thoughts… just to make sure we don't miss any great ideas.

Hence, my views are by design far "off the roadmap".

The Big “Three”

- In HPC, “three” programming environments dominate … covering the major classes of hardware.

1. **OpenMP**: Shared memory systems … more recently, GPUs too.

2. **MPI**: distributed memory systems … though it works nicely on shared memory computers.

3. **CUDA**, **OpenACC**, **OpenCL**, **OpenMP/Target**, **Sycl**: GPU programming

   (use CUDA or OpenACC if you don’t mind locking yourself to a single vendor)

- 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 "Three"

- In HPC, "three" programming environments dominate … covering the major classes of hardware.

1. **OpenMP**: Shared memory systems … more recently, GPUs too.
2. **MPI**: distributed memory systems … though it works nicely on shared memory computers.
3. **CUDA**, **OpenACC**, **OpenCL**, **OpenMP/Target**, **Sycl**: GPU programming

   (use CUDA or OpenACC if you don't mind locking yourself to a single vendor)

These all are expressions of the same execution model … so I lump them together.

- 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 "Three"

- In HPC, "three" programming environments dominate … covering the major classes of hardware.

1. **OpenMP**: Shared memory systems … more recently, GPUs too.

You are all OpenMP experts and know all about multithreading

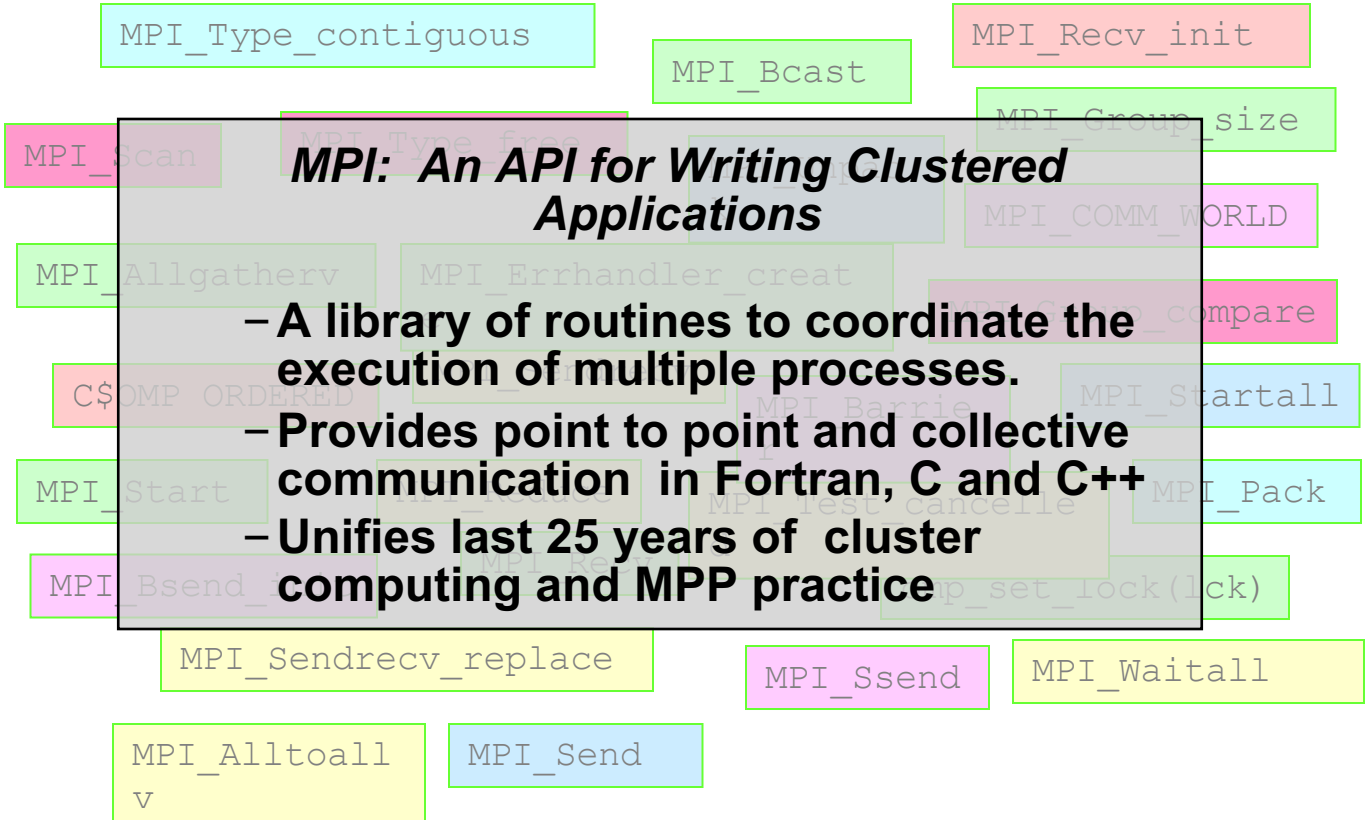
2. **MPI**: distributed memory systems … though it works nicely on shared memory computers.

3. **CUDA**, **OpenACC**, **OpenCL**, **OpenMP/Target**, **Sycl**: GPU programming

(use CUDA or OpenACC if you don't mind locking yourself to a single vendor)

- 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.

Parallel API's: MPI
the Message Passing Interface

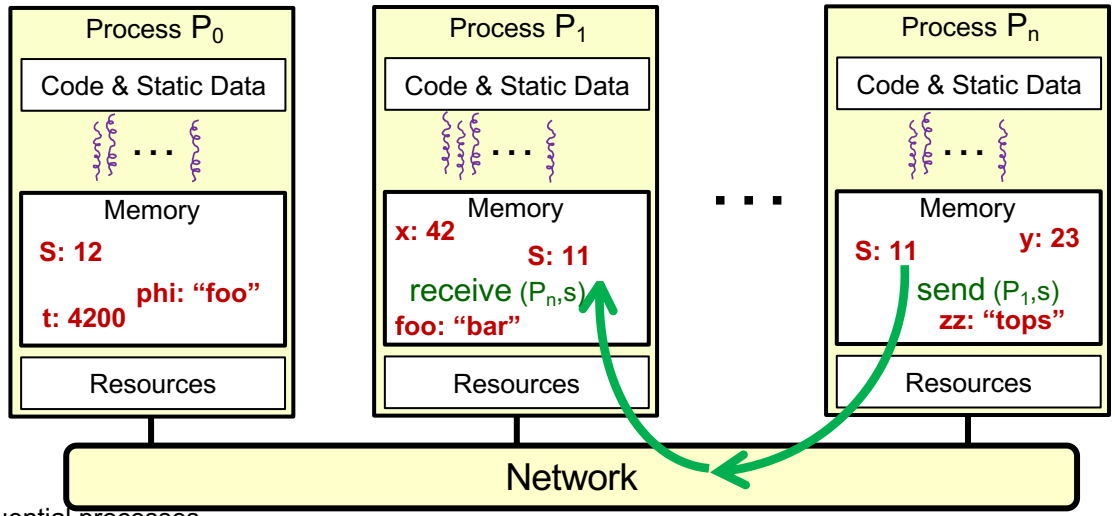


MPI: An API for Writing Clustered Applications

- **A library of routines to coordinate the execution of multiple processes.**
- **Provides point to point and collective communication in Fortran, C and C++**
- **Unifies last 25 years of cluster computing and MPP practice**

Execution Model: Distributed memory, CSP*

- Program consists of a collection of **named** processes.
  - Number of processes almost always fixed at program startup time
  - Local address space per node -- NO physically shared memory.
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is the most commonly used API

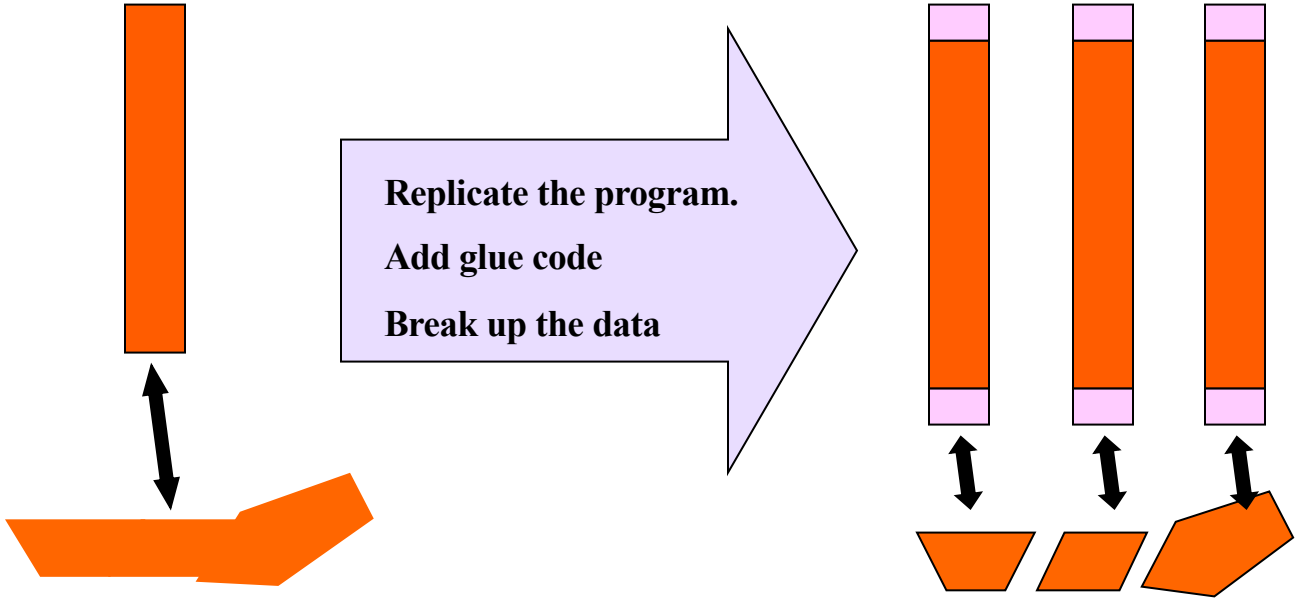


*CSP: communicating sequential processes

How do people use MPI?
The SPMD Design Pattern

- A single program working on a decomposed data set.
- Use process ID and numb of processes to split up work between processes
- Coordination by passing messages.

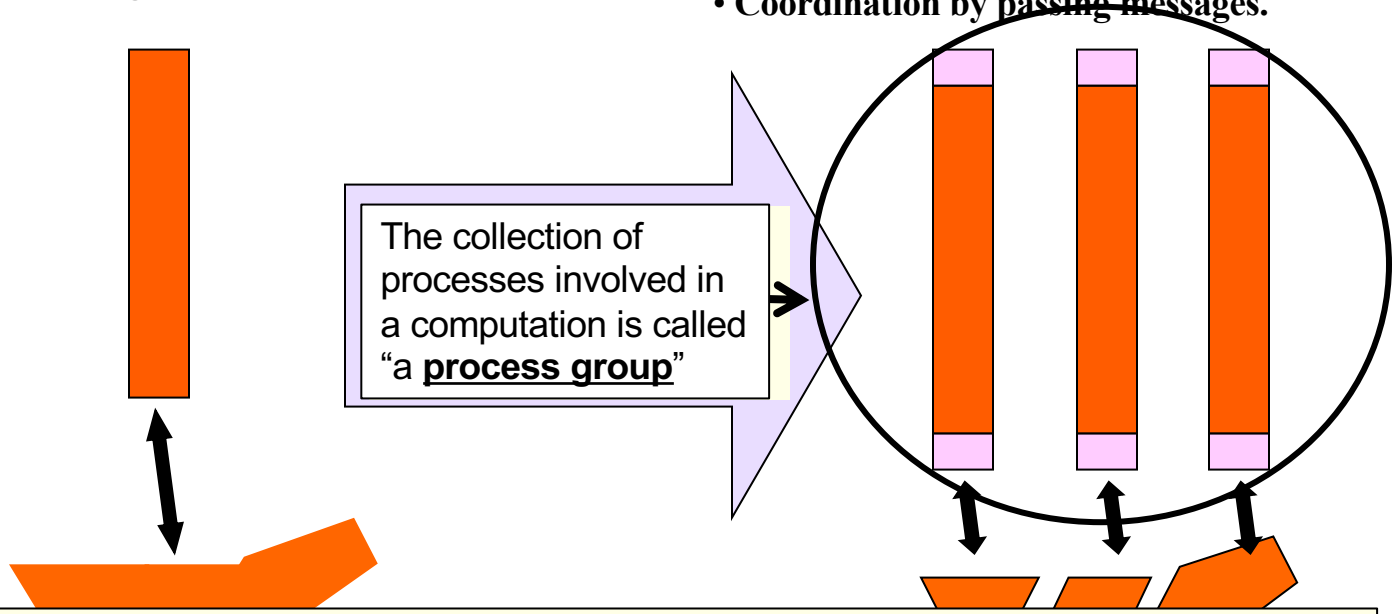
A sequential program working on a data set



How do people use MPI?
The SPMD Design Pattern

- A single program working on a decomposed data set.
- Use process ID and numb of processes to split up work between processes
- Coordination by passing messages.

A sequential program working on a data set



MPI functions work within a "**context**": MPI actions occurring in different contexts, even if they share a process group, cannot interfere with each other.

MPI Hello World

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                    rank, size );
    MPI_Finalize();
    return 0;
}
```

Initializing and finalizing MPI

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                rank, size );
    MPI_Finalize();
    return 0;
}
```

`int MPI_Init (int* argc, char* argv[])`

- Initializes the MPI library … called before any other MPI functions.
- agrc and argv are the command line args passed from main()

`int MPI_Finalize (void)`

- Frees memory allocated by the MPI library … close every MPI program with a call to MPI_Finalize

How many processes are involved?

`int MPI_Comm_size (MPI_Comm comm, int* size)`

- **`MPI_Comm`**, an *opaque data type called a communicator. D*efault context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_size`** returns the number of processes in the process group associated with the communicator

```
#inclu
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                rank, size );
    MPI_Finalize();
    return 0;
}
```

Communicators consist of two parts, a **context** and a **process group**.

The communicator lets one control how groups of messages interact.

Communicators support modular SW … i.e. I can give a library module its own communicator and know that it's messages can't collide with messages originating from outside the module

Which process “am I” (the rank)

`int MPI_Comm_rank (MPI_Comm comm, int* rank)`

- **`MPI_Comm`**, an *opaque data type,* **`a`** communicator. Default context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_rank`** An integer ranging from 0 to “(num of procs)-1”

```
#inclu
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                rank, size );
    MPI_Finalize();
    return 0;
}
```

Note that other than init() and finalize(), every MPI function has a communicator.

This makes sense .. You need a context and group of processes that the MPI functions impact … and those come from the communicator.

Running the program

- On a 4 node cluster, I'd run this program (hello) as:

  > mpiexec –np 4 –hostfile hostf hello

- Where "hostf" is a file with the names of the cluster nodes, one to a line.
- What would this program would output?

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                rank, size );
    MPI_Finalize();
    return 0;
}
```

Running the program

- On a 4 node cluster, I'd run this program (hello) as:

  > mpiexec –np 4 –hostfile hostf hello

  Hello from process 1 of 4

  Hello from process 2 of 4

  Hello from process 0 of 4

  Hello from process 3 of 4

- Where "hostf" is a file with the names of the cluster nodes, one to a line.

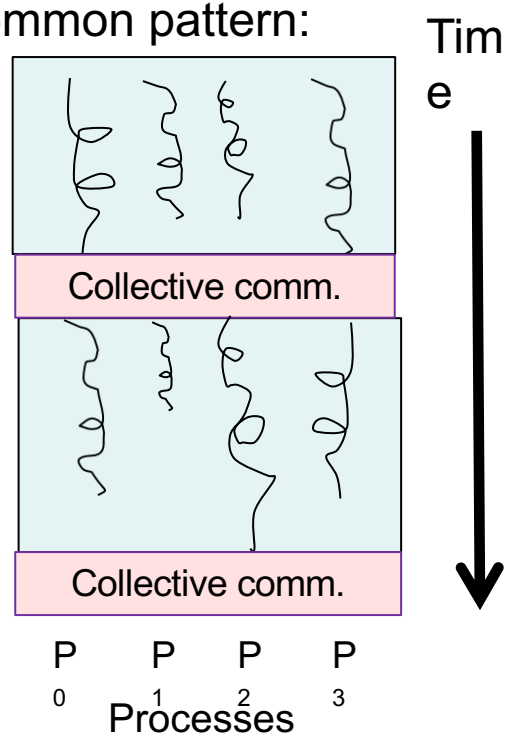
```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **ar
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                rank, size );
    MPI_Finalize();
    return 0;
}
```

Bulk Synchronous Programming (BSP):

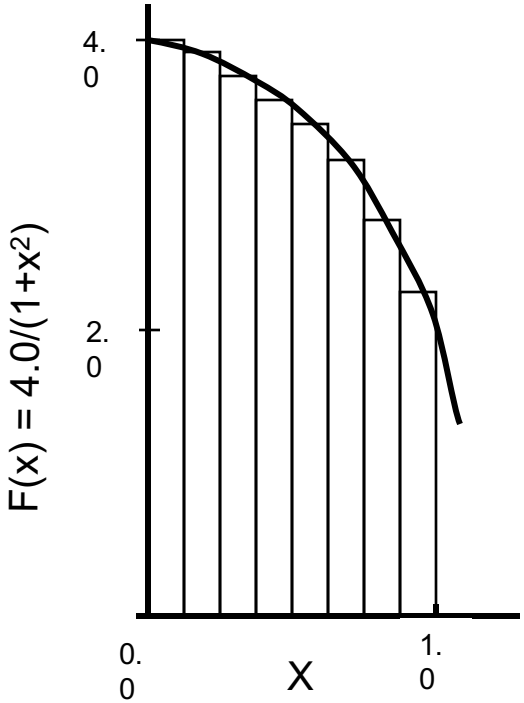
A common design pattern used with MPI Programs

- Many MPI applications have few (if any) sends and receives. They use the following very common pattern:
  - Use the Single Program Multiple Data pattern
  - Each process maintains a local view of the global data
  - A problem broken down into phases each of which is composed of two subphases:
    - Compute on local view of data
    - Communicate to update global view on all processes (collective communication).
  - Continue phases until complete

BSP is a subset of the SPMD pattern.



Example Problem: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

PI Program: an example

```
static long num_steps = 100000;
double step;
void main ()
{       int i;       double x, pi, sum = 0.0;

        step = 1.0/(double) num_steps;
           x = 0.5 * step;
        for (i=0;i<= num_steps; i++){
              x+=step;
              sum += 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

Pi program in MPI … using the BSP pattern

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
            int istart = my_id*my_steps;
            int iend = (my_id+1)*my_steps;
            if (my_id = numprocs-1) iend = num_steps;

        for (i=istarts; i<iend; i++){
                    x = (i+0.5)*step;
                    sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD) ;
            MPI_finalize();
}
```

Sum values in “sum” from each process and place it in “pi” on process 0

Reduction

```
int MPI_Reduce (void* sendbuf,
    void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op,
    int root, MPI_Comm comm)
```

- **`MPI_Reduce`** performs specified reduction operation on specified data from all processes in communicator, places result in process “root” only.
- **`MPI_Allreduce`** places result in all processes (avoid unless necessary)

Operation	Function
MPI_SUM	Summation
MPI_PROD	Product
MPI_MIN	Minimum value
MPI_MINLOC	Minimum value and location
MPI_MAX	Maximum value
MPI_MAXLOC	Maximum value and location
MPI_LAND	Logical AND

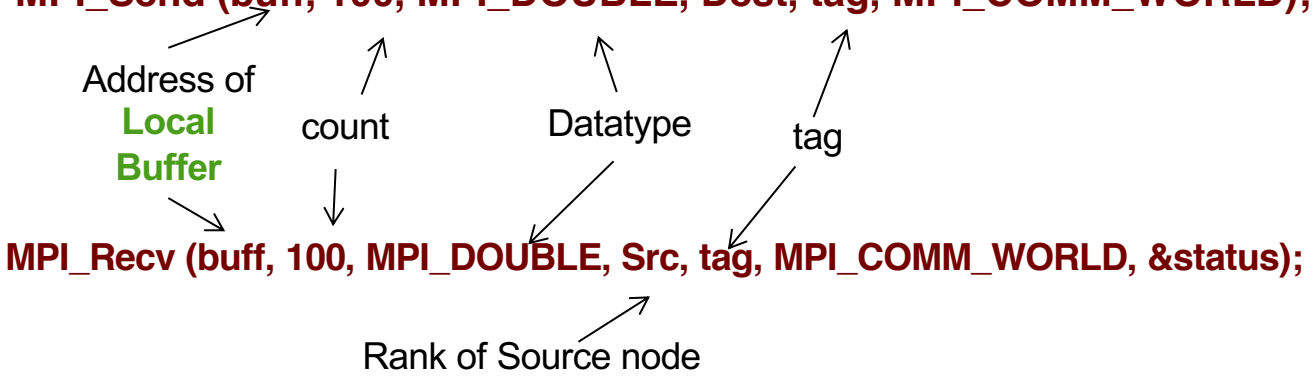
Operation	Function
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
User-defined	It is possible to define new reduction operations

Sending and receiving messages

- Pass a buffer which holds “count” values of MPI_TYPE
- The data in a message to send or receive is described by a triple:
  - **(address, count, datatype)**

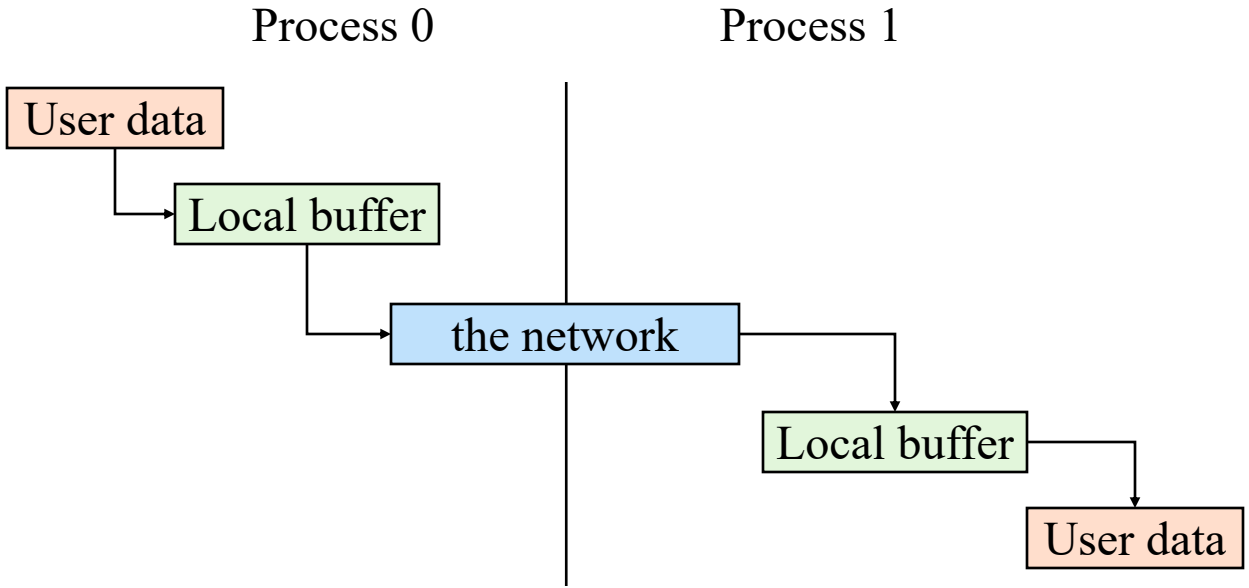
- The receiving process identifies messages with the double :
  - **(source, tag)**
- Where:
  - Source is the rank of the sending process
  - Tag is a user-defined integer to help the receiver keep track of different messages from a single source

MPI_Send (buff, 100, MPI_DOUBLE, Dest, tag, MPI_COMM_WORLD);



Buffers

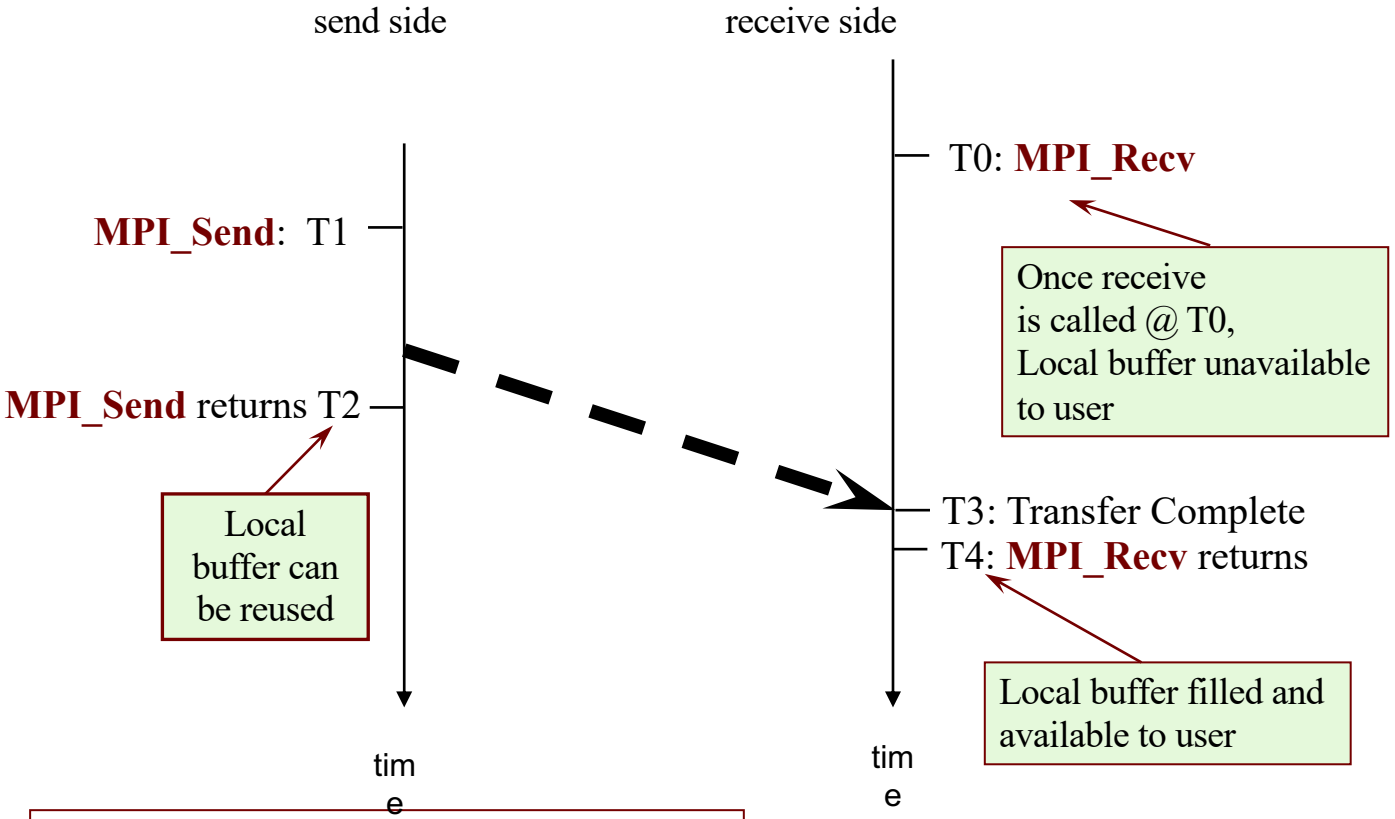
- Message passing has a small set of primitives, but there are subtleties
  - Buffering and deadlock
  - Deterministic execution
  - Performance
- When you send data, where does it go? One possibility is:



Derived from: Bill Gropp, UIUC

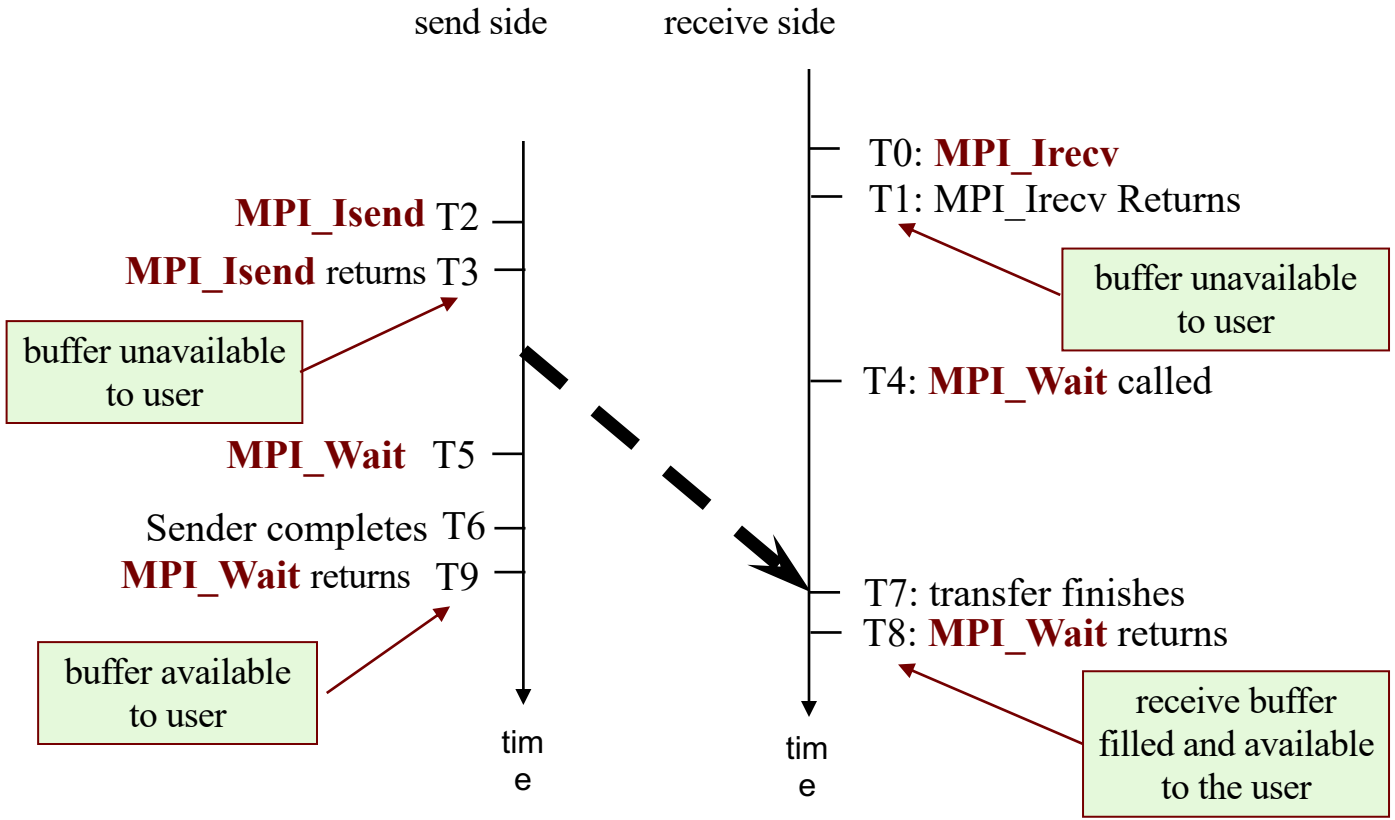
Blocking Send-Receive Timing Diagram

(MPI functions return when local buffer can be used again)



Non-Blocking Send-Receive Diagram

(MPI functions return immediately)



Example: finite difference methods

- Solve the heat diffusion equation in 1 D:
  - u(x,t) describes the temperature field
  - We set the heat diffusion constant to one
  - Boundary conditions, constant u at endpoints.

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}$$

- map onto a mesh with stepsize h and k

$$x_i = x_0 + ih \qquad t_i = t_0 + ik$$

- Central difference approximation for spatial derivative (at fixed time)

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2}$$

- Time derivative at $t = t^{n+1}$

$$\frac{du}{dt} = \frac{u^{n+1} - u^n}{k}$$

Example: Explicit finite differences

- Combining time derivative expression using spatial derivative at t = t^n

$$\frac{u_j^{n+1} - u_j^n}{k} = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2}$$

  - Solve for u at time n+1 and step j

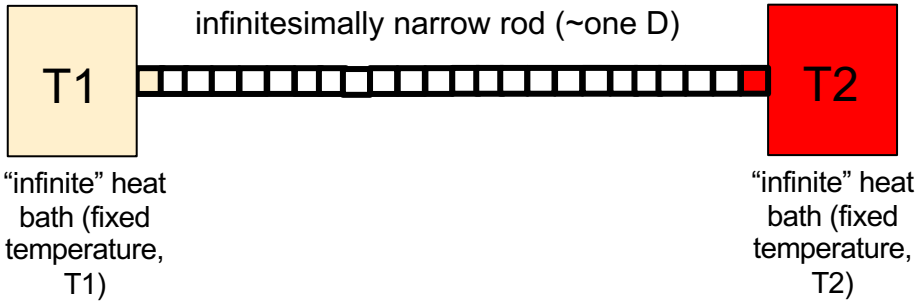
$$u_j^{n+1} = (1-2r)u_j^n + ru_{j-1}^n + ru_{j+1}^n \qquad r = k/h^2$$

  - The solution at t = t_{n+1} is determined explicitly from the solution at t = t_n (assume u[t][0] = u[t][N] = Constant for all t).

```
for (int t = 0; t < N_STEPS-1; ++t)
   for (int x = 1; x < N-1; ++x)
         u[t+1][x] = u[t][x] + r*(u[t][x+1] - 2*u[t][x] + u[t][x-1]);
```

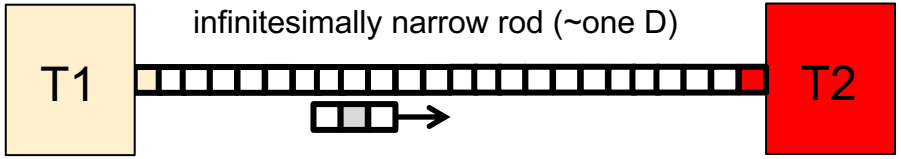
  - Explicit methods are easy to compute … each point updated based on nearest neighbors. Converges for r<1/2.

Heat Diffusion equation



28

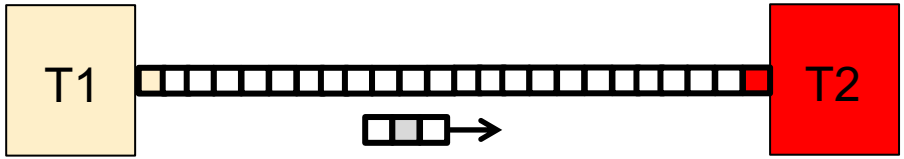
Heat Diffusion equation



Pictorially, you are sliding a three point "stencil" across the domain (u) and updating the center point at each stop.

29

Heat Diffusion equation

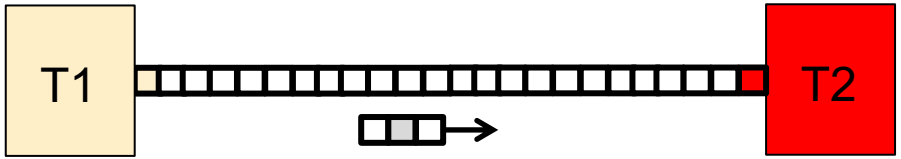



```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   initialize_data(uk, ukp1, N, P); // init to zero, set end temperatures
   for (int t = 0; t < N_STEPS; ++t){
      for (int x = 1; x < N-1; ++x)
          up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);

     temp = up1; up1 = u; u = temp;
    }
return 0;
```

Note: I don't need the intermediate "u[t]" values hence "u" is just indexed by x.

A well known trick with 2 arrays so I don't overwrite values from step k-1 as I fill in for step k

Heat Diffusion equation

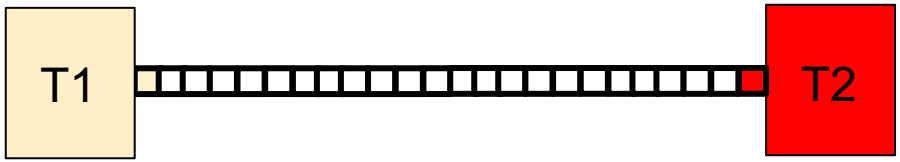



```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   initialize_data(uk, ukp1, N, P); // init to zero, set end temperatures
   for (int t = 0; t < N_STEPS; ++t){
      for (int x = 1; x < N-1; ++x)
          up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);

      temp = up1; up1 = u; u = temp;
    }
return 0;
```

How would you parallelize this program?

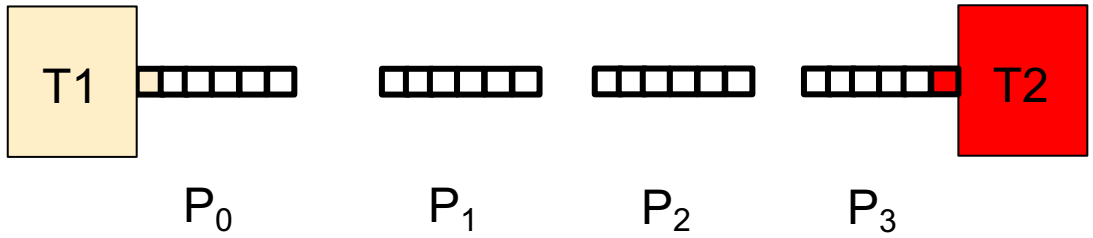
Heat Diffusion equation

- Start with our original picture of the problem … a one dimensional domain with end points set at a fixed temperature.



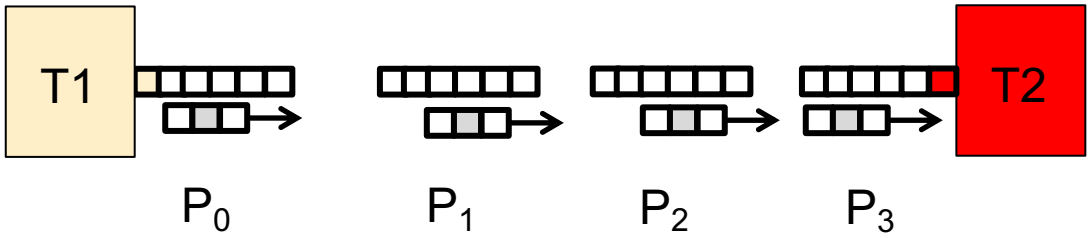
Heat Diffusion equation

- Break it into chunks assigning one chunk to each process.



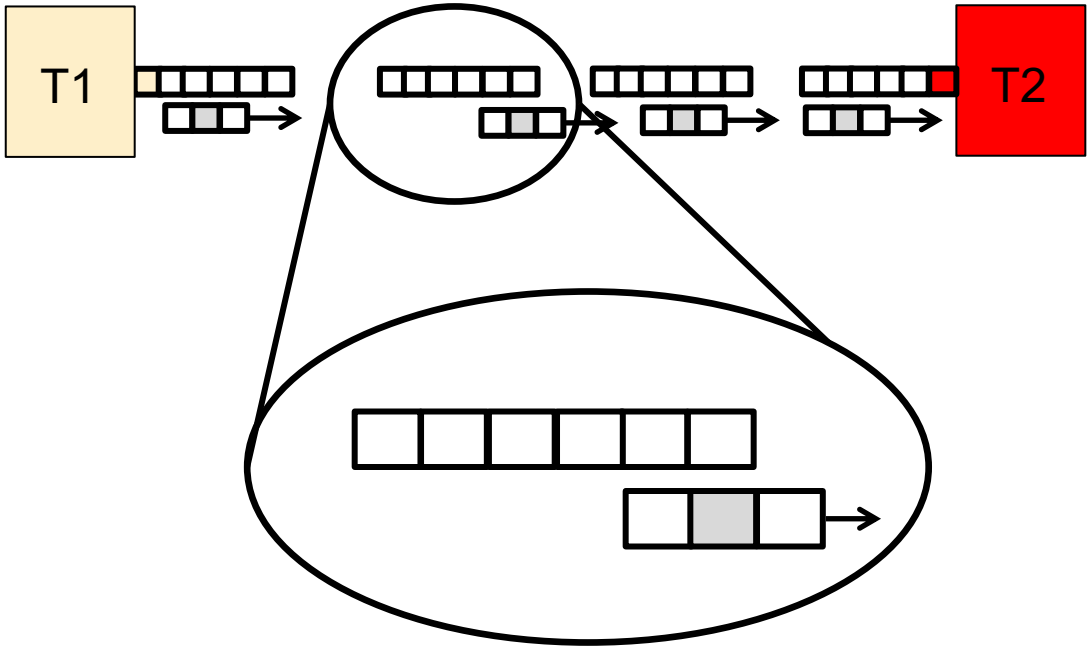
Heat Diffusion equation

- Each process works on it's own chunk … sliding the stencil across the domain to updates its own data.



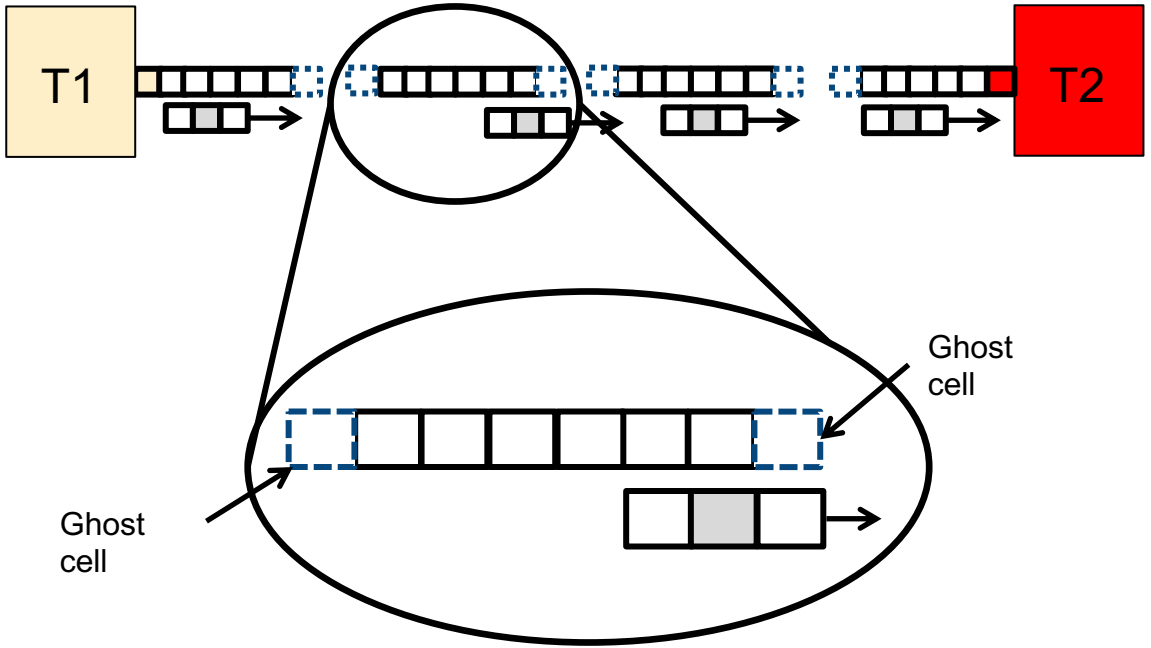
Heat Diffusion equation

- What about the ends of each chunk … where the stencil will run off the end and hence have missing values for the computation?



Heat Diffusion equation

- We add ghost cells to the ends of each chunk, update them with the required values from neighbor chunks at each time step … hence giving the stencil everything it needs on any given chunk to update all of its values.

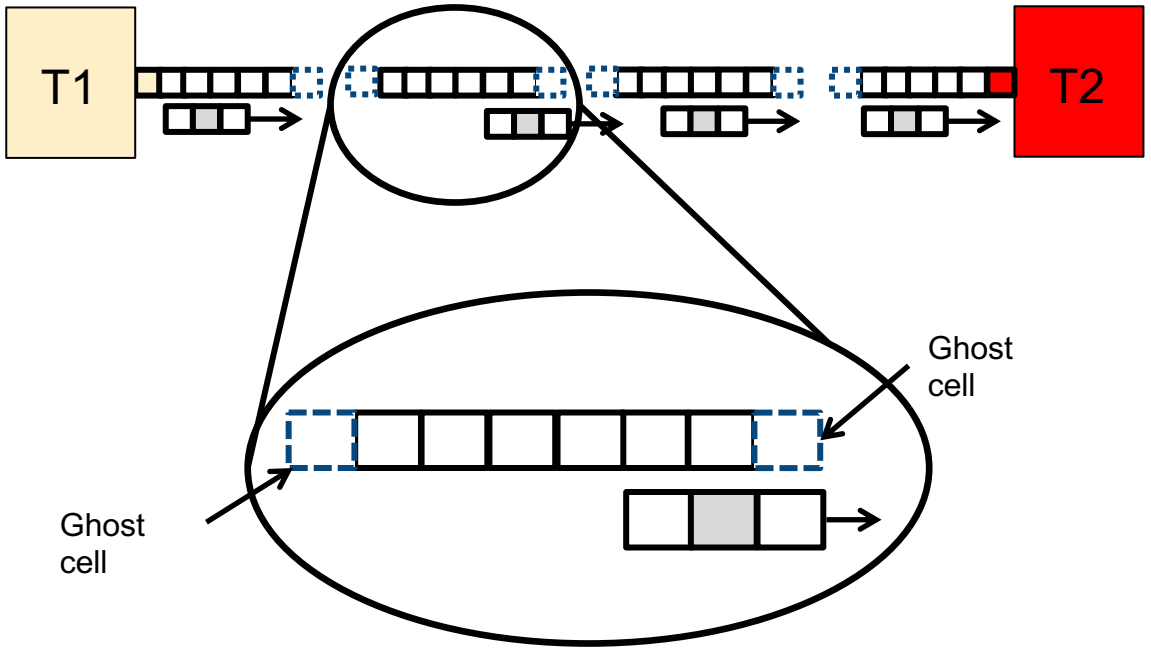


Design Pattern: Geometric Decomposition

- Use when:
  - The problem is organized around a central data structure that can be decomposed into smaller segments (chunks) that can be updated concurrently.
- Solution
  - Typically, the data structure is updated iteratively where a new value for one chunk depends on neighboring chunks.
  - The computation breaks down into three components: (1) exchange boundary data, (2) update the interiors or each chunk, and (3) update boundary regions. The optimal size of the chunks is dictated by the properties of the memory hierarchy.
- Note:
  - This pattern is often used with the Structured Mesh and linear algebra computational strategy pattern.

The Geometric Decomposition Pattern

- This is an instance of a very important design pattern … the Geometric decomposition pattern.



Heat Diffusion MPI Example

```
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &myID);
double *u   = malloc (sizeof(double) * (2 + N/P))  // include "Ghost Cells"
double *up1 = malloc (sizeof(double) * (2 + N/P)); // to hold values
                                                   // from my neighbors
initialize_data(uk, ukp1, N, P);
for (int t = 0; t < N_STEPS; ++t){
  if (myID != 0)  MPI_Send (&u[1], 1, MPI_DOUBLE, myID-1, 0, MPI_COMM_WORLD);
  if (myID != P-1) MPI_Recv (&u[N/P+1], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD, &status);
  if (myID != P-1) MPI_Send (&u[N/P], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD);
  if (myID != 0)   MPI_Recv (&u[0], 1, MPI_DOUBLE, myID-1, 0,MPI_COMM_WORLD, &status);

  for (int x = 2; x <= N/P; ++x)
    up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);
  if (myID != 0)
    up1[1] = u[1] + (k / (h*h)) * (u[1+1] - 2*u[1] + u[1-1]);
  if (myID != P-1)
    up1[N/P] = u[N/P] + (k/(h*h)) * (u[N/P+1] - 2*u[N/P] + u[N/P-1]);
  temp = up1; up1 = u; u = temp;

} // End of for (int t ...) loop

MPI_Finalize();
return 0;
```

We write/explain this part first and then address the communication and data structures

Heat Diffusion MPI Example

```
/* continued from previous slide */

  for (int x = 2; x <= N/P; ++x)
    up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);

  if (myID != 0)
    up1[1] = u[1] + (k / (h*h)) * (u[1+1] - 2*u[1] + u[1-1]);

  if (myID != P-1)
    up1[N/P] = u[N/P] + (k/(h*h)) * (u[N/P+1] - 2*u[N/P] + u[N/P-1]);

  temp = up1; up1 = u; u = temp;

} // End of for (int t ...) loop

MPI_Finalize();
return 0;
```

Temperature fields using local data and values from ghost cells.

u[0] and u[N/P+1] are the ghost cells

Note I was lazy and assume N was evenly divided by P. Clearly, I'd never do this in a "real" program.

We don't update up1[1] on node 0 or up1[N/P] on node (P-1) since the boundary conditions stipulate that the end points have a fixed temperature

Heat Diffusion MPI Example

1D PDE solver … the simplest "real" message passing code I can think of. Note: edges of domain are held at a fixed temperature

```
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &myID);
double *u   = malloc (sizeof(double) * (2 + N/P))  // include "Ghost Cells"
double *up1 = malloc (sizeof(double) * (2 + N/P)); // to hold values
                                                   // from my neighbors
initialize_data(uk, ukp1, N, P);
for (int t = 0; t < N_STEPS; ++t){
  if (myID != 0)
    MPI_Send (&u[1], 1, MPI_DOUBLE, myID-1, 0, MPI_COMM_WORLD);

  if (myID != P-1)
    MPI_Recv (&u[N/P+1], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD, &status);

  if (myID != P-1)
    MPI_Send (&u[N/P], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD);

  if (myID != 0)
    MPI_Recv (&u[0], 1, MPI_DOUBLE, myID-1, 0,MPI_COMM_WORLD, &status);
/* continued on next slide */
```

Send my "right" boundary value to my "right' neighbor

Receive my "left" ghost cell from my "left' neighbor

Send my "left" boundary value to my "left' neighbor

Receive my "right" ghost cell from my "right' neighbor

MPI is huge!!!

- MPI has over 430 functions!!!
  - Many forms of message passing
  - Full range of collectives (such as reduction)
  - dynamic process management
  - Shared memory
  - and much more

- Most programs, however use around a dozen different constructs … so it's not as hard to learn as it may seem.

Management/time

- MPI_Init
- MPI_Finish
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Wtime

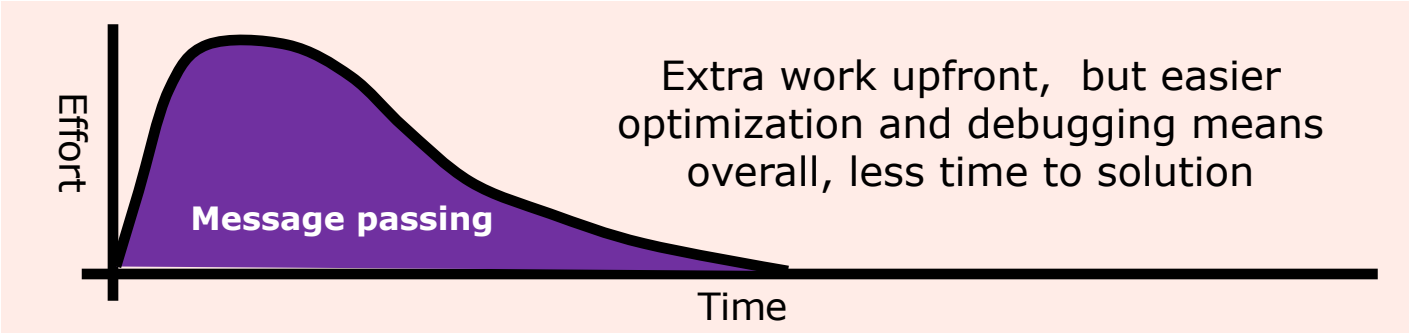
Message Passing

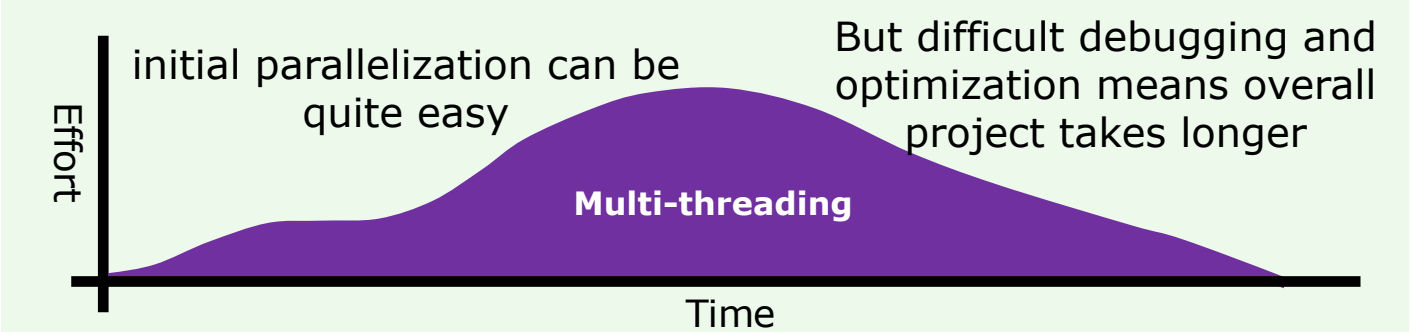
- MPI_Send
- MPI_Recv
- MPI_Isend
- MPI_Irecv
- MPI_Wait

Collective Comm.

- MPI_Reduce
- MPI_Bcast

Does a shared address space make programming easier?





Proving that a shared address space program using semaphores is race free is an NP-complete problem*

*P. N. Klein, H. Lu, and R. H. B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35 pp. 321–345,

The Big "Three"

- In HPC, "three" programming environments dominate … covering the major classes of hardware.

1. **OpenMP**: Shared memory systems … more recently, GPUs too.
2. **MPI**: distributed memory systems … though it works nicely on shared memory computers.
3. **CUDA**, **OpenACC**, **OpenCL**, **OpenMP/Target**, **Sycl**: GPU programming

(use CUDA or OpenACC if you don't mind locking yourself to a single vendor)

You are all OpenMP experts and know all about multithreading

You aren't an expert, but you now hopefully grok* MPI

- 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.

*grok: a Martian word meaning to understand something deeply; to merge with it and for it to merge with you.

Single Instruction Multiple Thread (SIMT)

- SIMT:
  - Implement data parallel problems:
    - Define an abstract index space that spans the problem domain.
    - Data structures in the problem are aligned to this index space.
    - Run an instance of a kernel at each point in that space.

- This approach was popularized for graphics applications where the index space mapped onto the pixels in an image. Since 2006, It's been extended to General Purpose GPU (GPGPU) programming.

Note: This is closely related the SPMD pattern.

The BIG idea behind SIMT

- Execution model … execute an instance of a kernel at each point in a problem domain.
  - E.g., process a 1024 x 1024 image with one instance of a kernel per pixel or 1024 x 1024 = 1,048,576 kernel instances

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
}
```

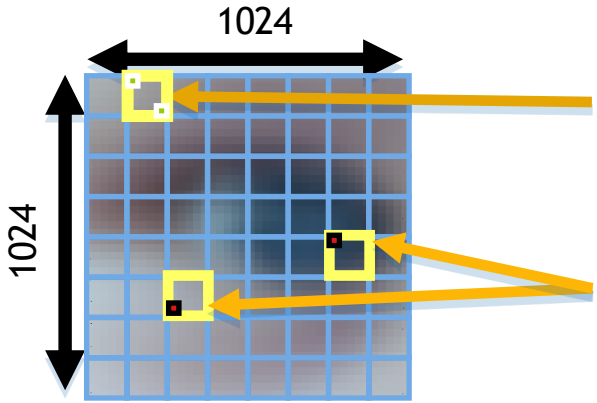
Data Parallel … OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] + b[id];

} // execute over "n" work-items
```

An N-dimensional range of work-items

OpenCL

- SIMT execution model … execute an instance of a kernel at each point in a problem domain.
  - E.g., process a 1024 x 1024 image with one instance of a kernel per pixel or 1024 x 1024 = 1,048,576 kernel instances
    - Global Dimensions: (NDRange) 1024x1024 (whole problem space)
    - Local Dimensions: 128x128 (work-group, executes together)



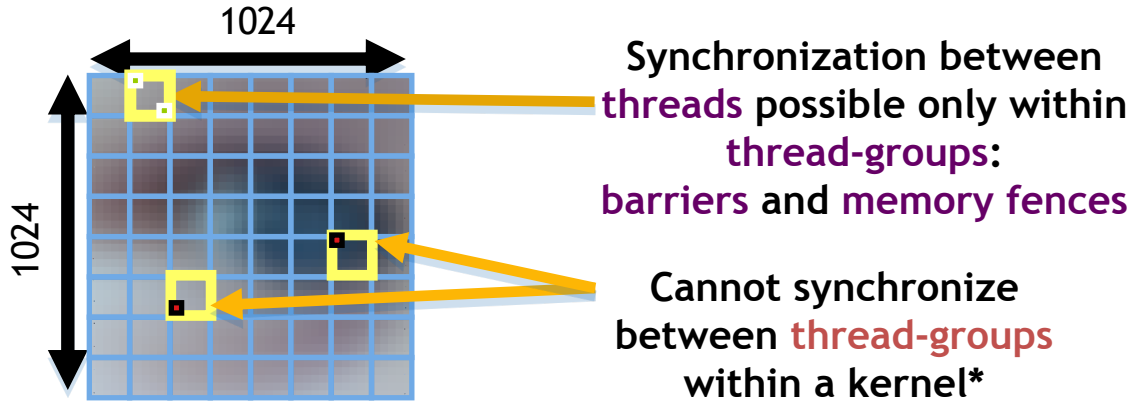
Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel

An N-dimensional grid of threads

NVIDIA CUDA

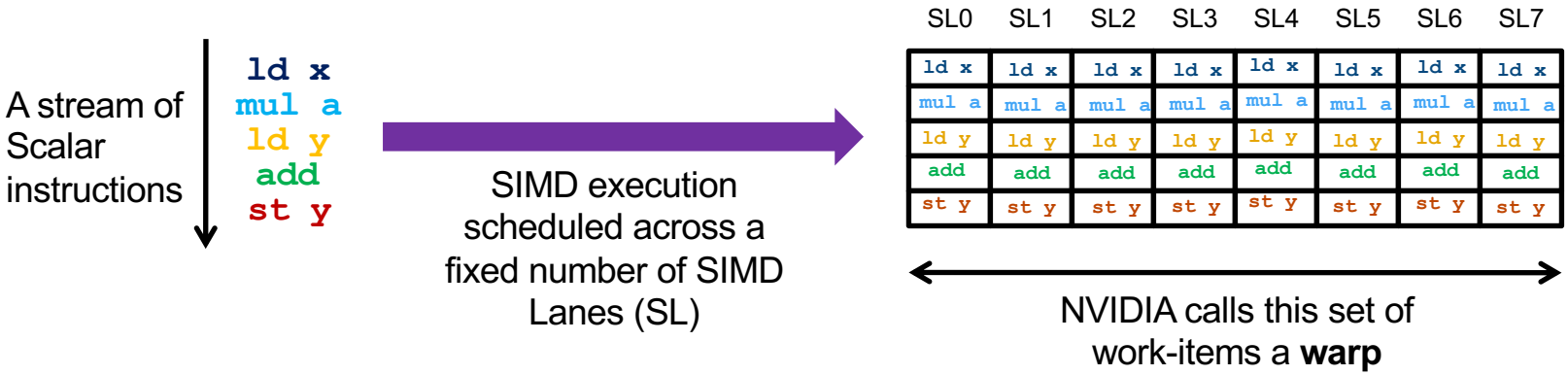
- SIMT execution model … execute an instance of a kernel at each point in a problem domain.
  - E.g., process a 1024 x 1024 image with one instance of a kernel per pixel or 1024 x 1024 = 1,048,576 kernel instances
    - Global Dimensions: (Grid) 1024x1024 (whole problem space)
    - Local Dimensions: 128x128 (thread-group, executes together)



*CUDA since 9.X has relaxed this constraint

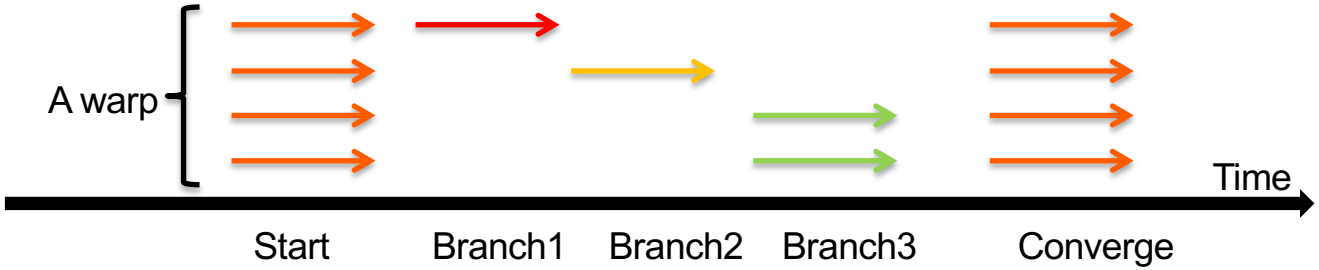
SIMT: Single Instruction, Multiple Thread

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



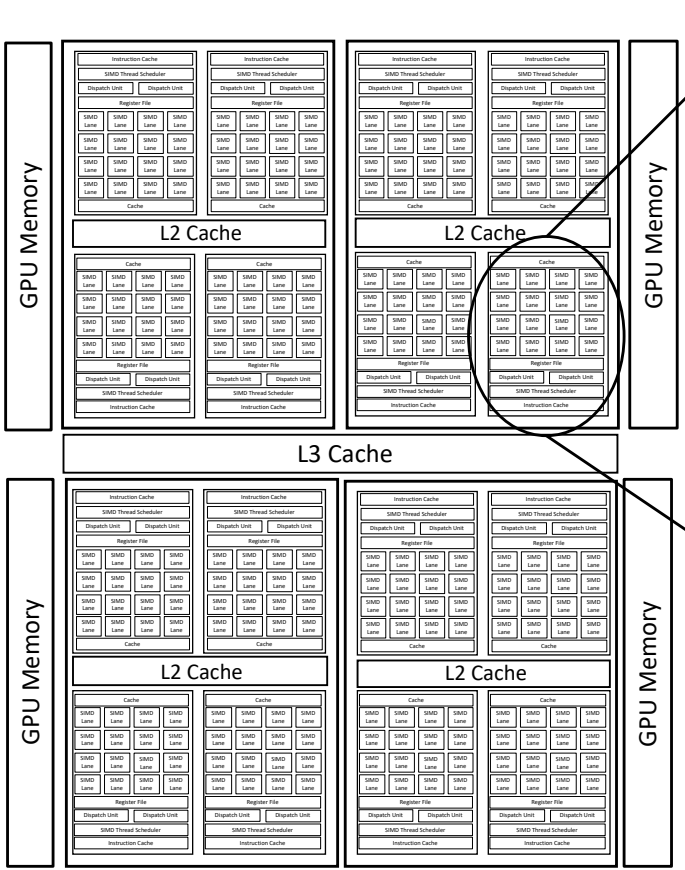
Single Instruction Multiple Data

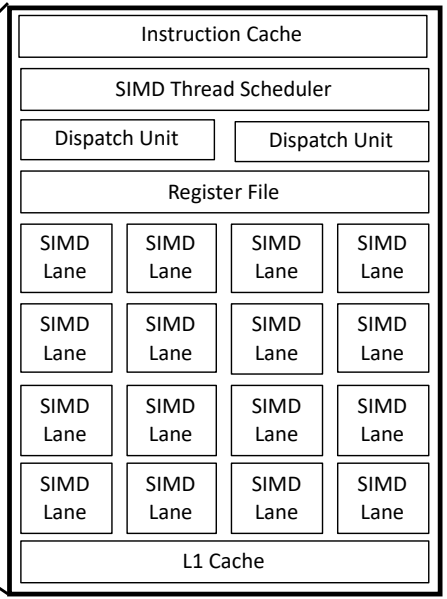
- Individual work-items of a warp start together at the same program address
- Each work-item has its own instruction address counter and register state
  - Each work-item is free to branch and execute independently
  - Supports the SPMD pattern.
- Branch behavior
  - Each branch will be executed serially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



GPU Architecture:

A generic GPU with 16 Streaming Multiprocessors each with with 16 SIMD Lanes



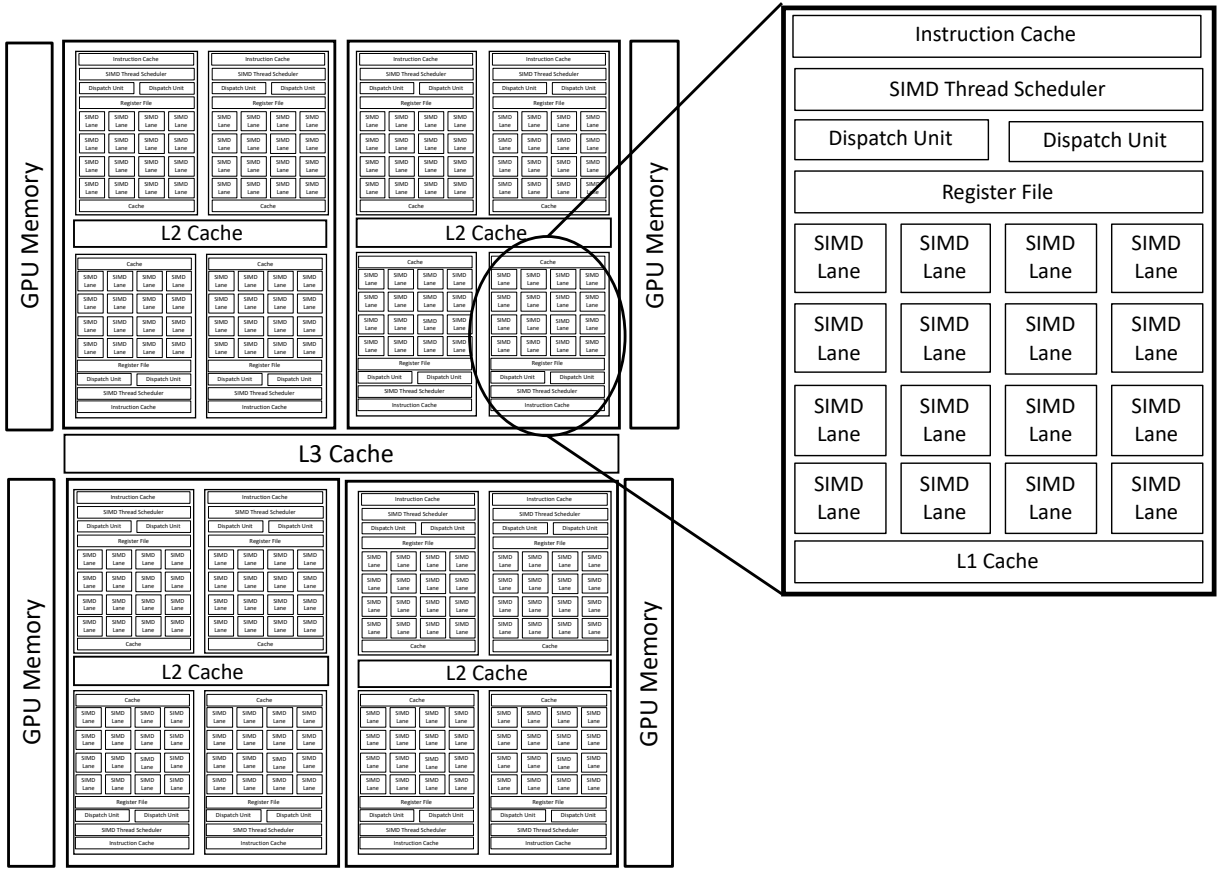


A modern GPU also has load-store, floating point, and tensor processing units spread around the SM. They are accessed through a fixed API and are not directly programmed through a GPGPU programing model … hence we do not include them here.

[#] Frigure from “Programming your GPU with OpenMP”, but Tom Deakin and Tim Mattson, MIT Press, 2022

GPU Architecture: Nvidia Nomenclature

- **Thread Groups:**
  - Each Thread Group is scheduled onto a Streaming SIMD processor
  - Peak efficiency requires multiple work groups per Streaming SIMD processor
- **Warps:**
  - A thread group is broken down into warps that execute together.
  - A SIMD instruction acts on a “warp”
  - The NVIDIA Warp width is 32 elements: LOGICAL SIMD width (though the device on this page has a SIMD width of 16)
- **CUDA threads:**
  - each thread is a SIMD vector lane and runs on the processing element within a Streaming SIMD processor



[#] Frigure from “Programming your GPU with OpenMP”, but Tom Deakin and Tim Mattson, MIT Press, 2022

GPU terminology is really messed up

Table 1.1[#] **GPU Terminology:** – GPU Terminology from Hennesy and Patterson,[1] CUDA, and OpenCL. The one uncertain match is the last row; a sub-group usually corresponds to a warp, but a conforming implementation of OpenCL may use a sub-group to refer to smaller blocks of work-items that make up a warp.

Hennessy and Patterson	CUDA	OpenCL
Multithreaded SIMD Processor	Streaming multiprocessor	Compute Unit
SIMD Thead Scheduler	Warp Scheduler	Work-group scheduler
SIMD Lane	CUDA Core	Processing Element
GPU Memory	Global Memory	Global Memory
Private Memory	Local Memory	Private Memory
Local Memory	Shared Memory	Local Memory
Vectorizable Loop	Grid	NDRange
Sequence of SIMD Lane operations	CUDA Thread	work-item
A thread of SIMD instructions	Warp	sub-group

[1] Computer Architecture: A Quantitative Approach, Sixth Edition, John L. Hennessy and David A. Patterson, Morgan Kaufmann, 2019.

[#] Table from “Programming your GPU with OpenMP”, but Tom Deakin and Tim Mattson, MIT Press, 2022

SIMT Programming models

- 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

Vector addition with SYCL

```
// Compute sum of length-N vectors: C = A + B
#include <CL/sycl.hpp>
constexpr int N = 8;
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.

Vector addition with OpenACC

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

```
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.

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

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:

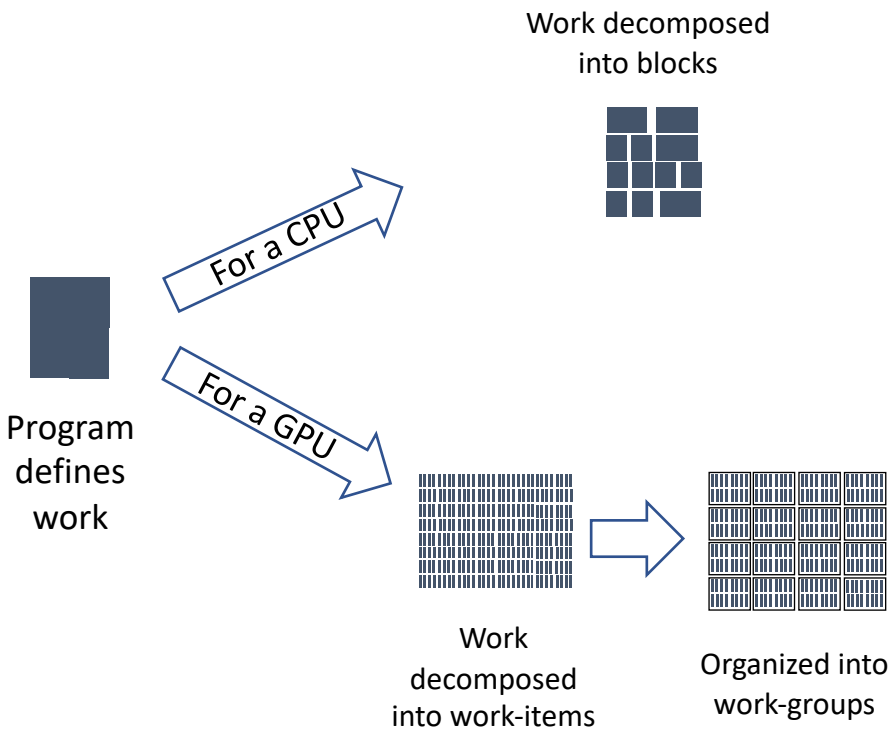
Jacobi iteration: OpenMP target directives

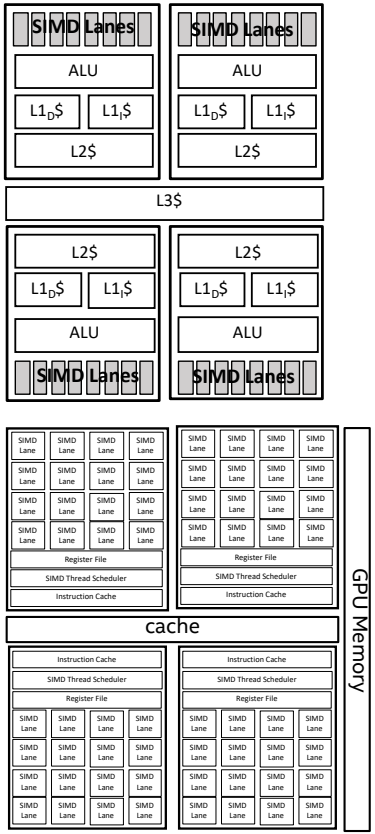
```
#pragma omp target data map(A) map(alloc:Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma target
   #pragma omp teams distribute parallel for reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
       }
    }
    #pragma omp target
    #pragma omp teams 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];
       }
    }
    iter ++;
}
```

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

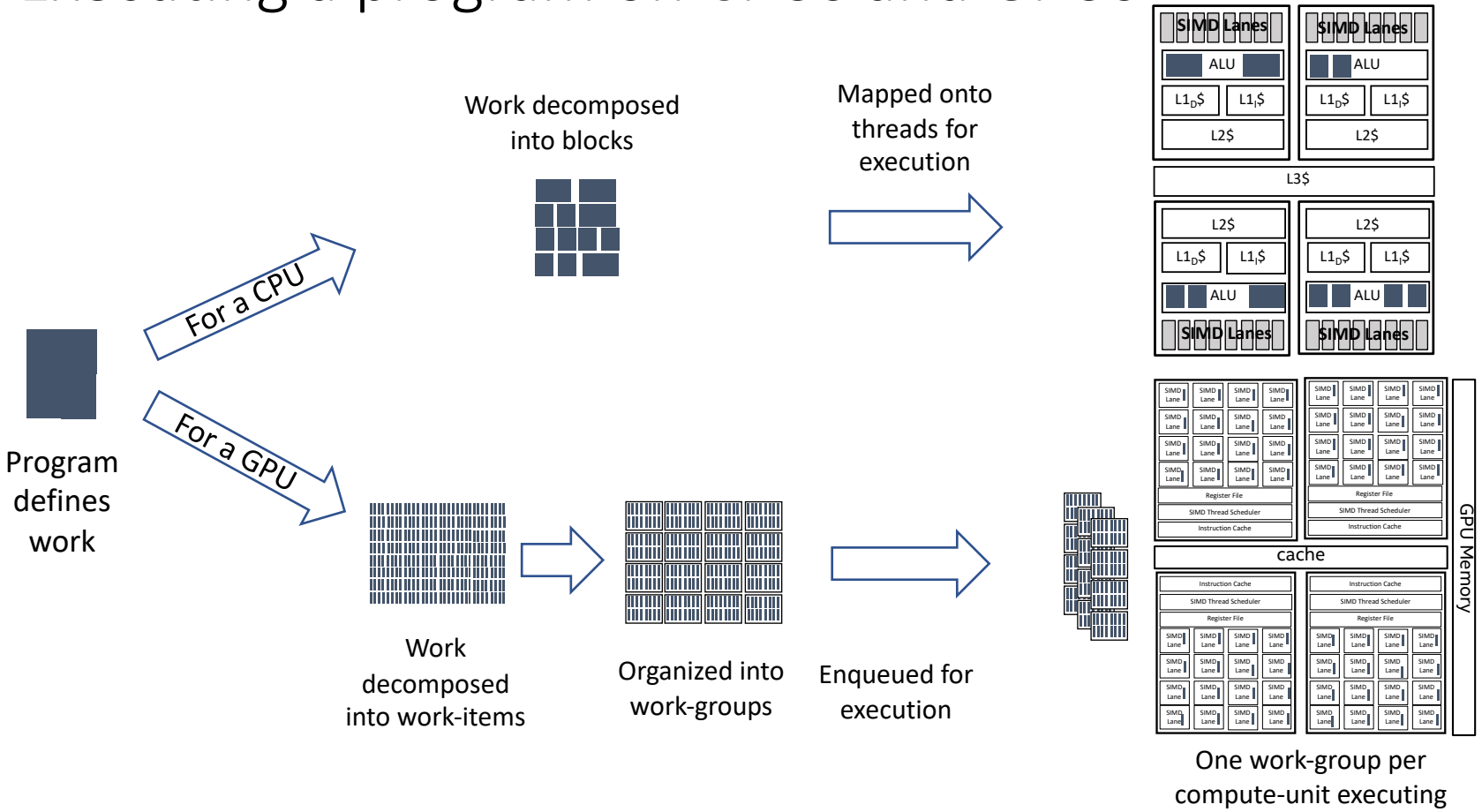
Copy A back out to host … but only once

Executing a program on CPUs and GPUs

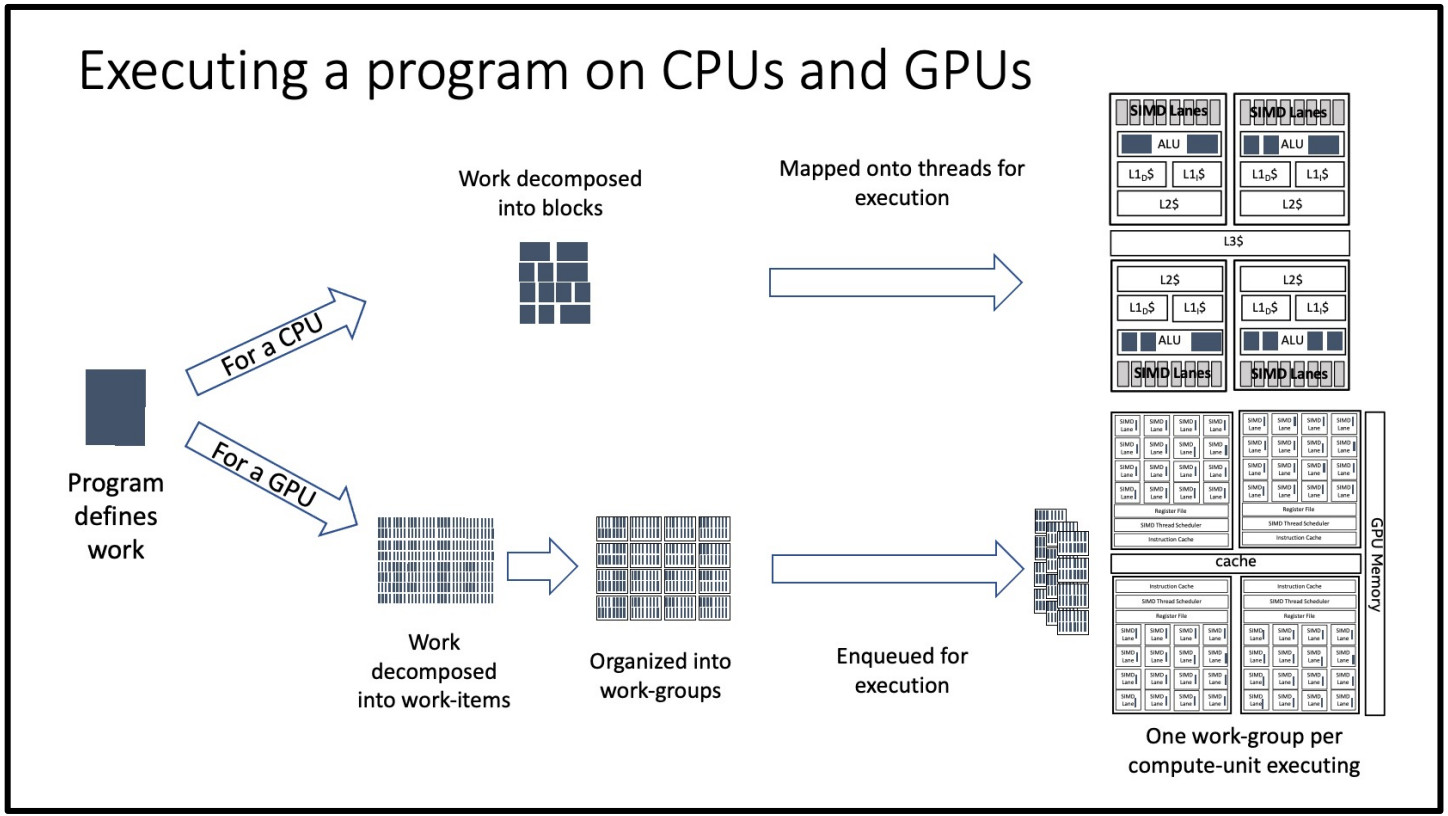




Executing a program on CPUs and GPUs



CPU/GPU execution and the idea of forward progress



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

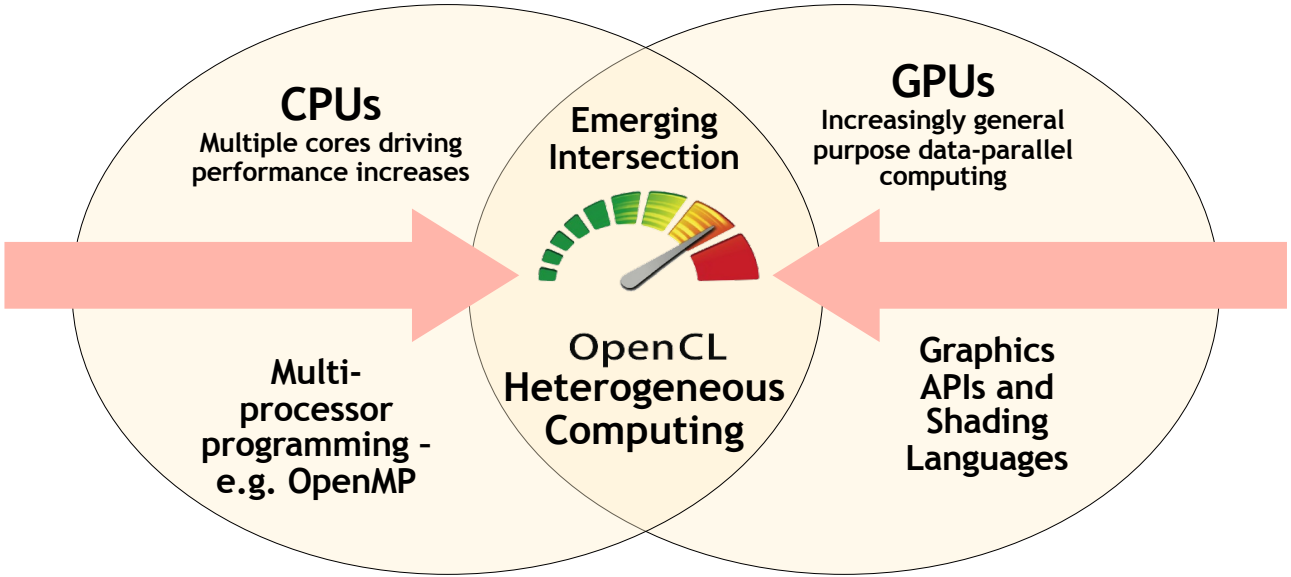
For a GPU, any given work-group might be in the queue waiting to execute.

Backup Content

(or what I'd cover if we had a couple more hours)

- Developing a complex kernel in OpenCL; the dream of performance portability
- The history of General Purpose GPU programming (GPGPU)
- Debunking the 100X GPU vs. CPU Myth

Industry Standards for Programming Heterogeneous Platforms

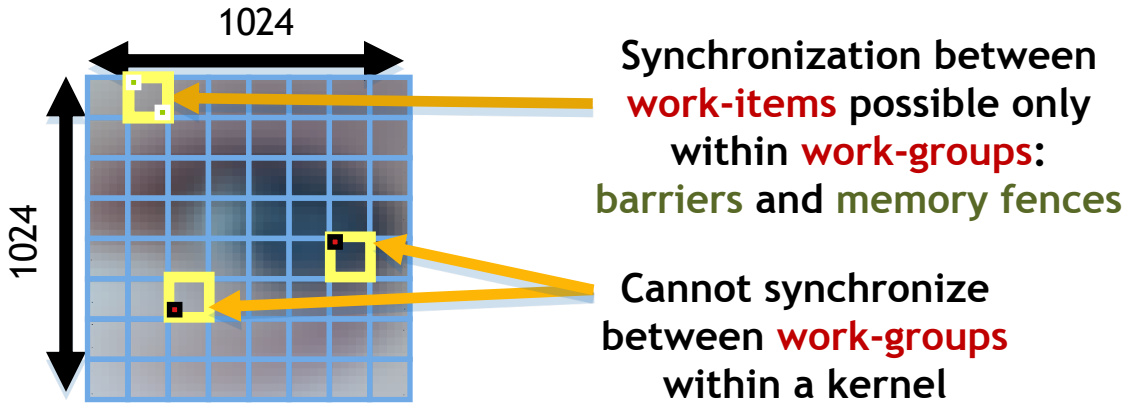


OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

An N-dimensional domain of work-items

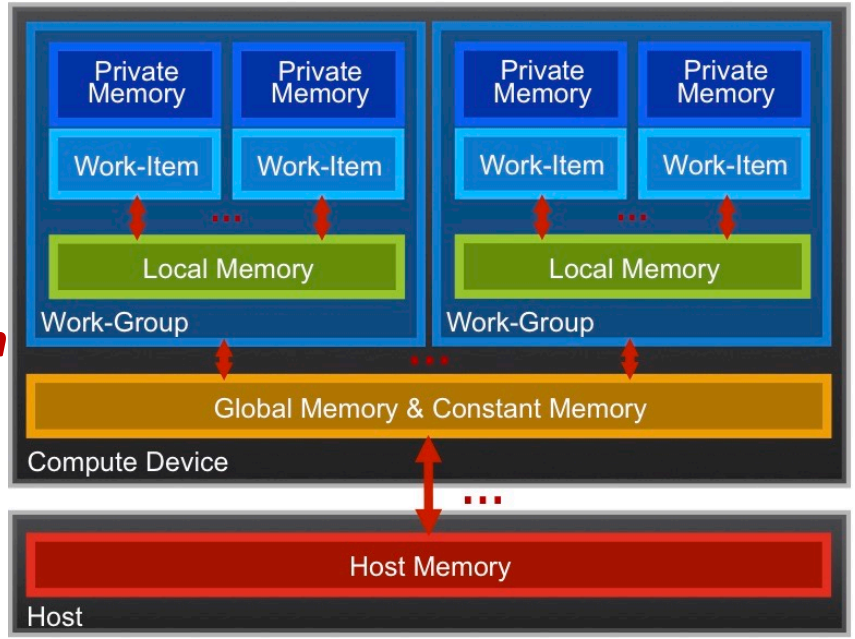
- **Global** Dimensions:
  - 1024x1024 (whole problem space)
- **Local** Dimensions:
  - 128x128 (**work-group**, executes together)



- Choose the dimensions (1, 2, or 3) that are "best" for your algorithm

OpenCL Memory model

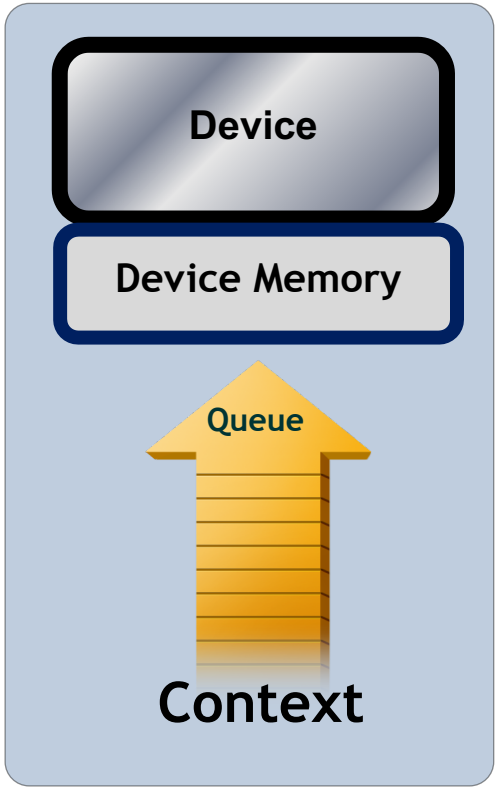
- ***Private Memory***
  - Per work-item
- ***Local Memory***
  - Shared within a work-group
- ***Global Memory /Constan Memory***
  - Visible to all work-groups
- ***Host memory***
  - On the CPU



Memory management is explicit:
You are responsible for moving data from
host → global → local *and* back

Context and Command-Queues

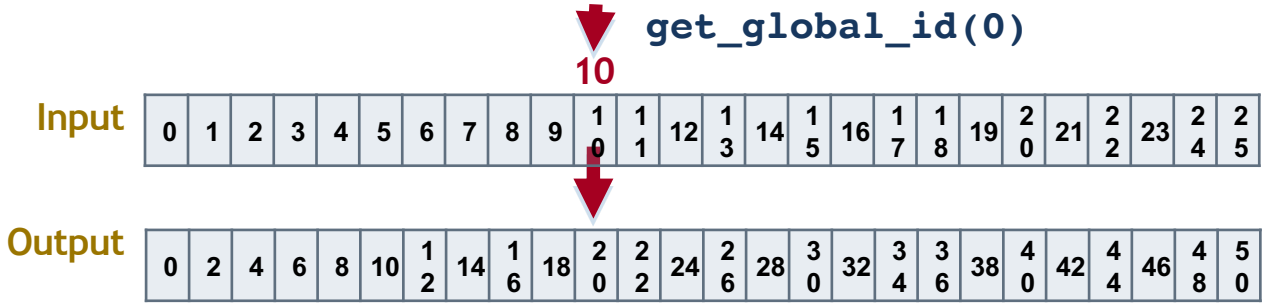
- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, and memory operations) are submitted through a ***command-queue***.
- Each ***command-queue*** points to a single device within a context.



Execution model (kernels)

- OpenCL execution model … define a problem domain and execute an instance of a **kernel** for each point in the domain

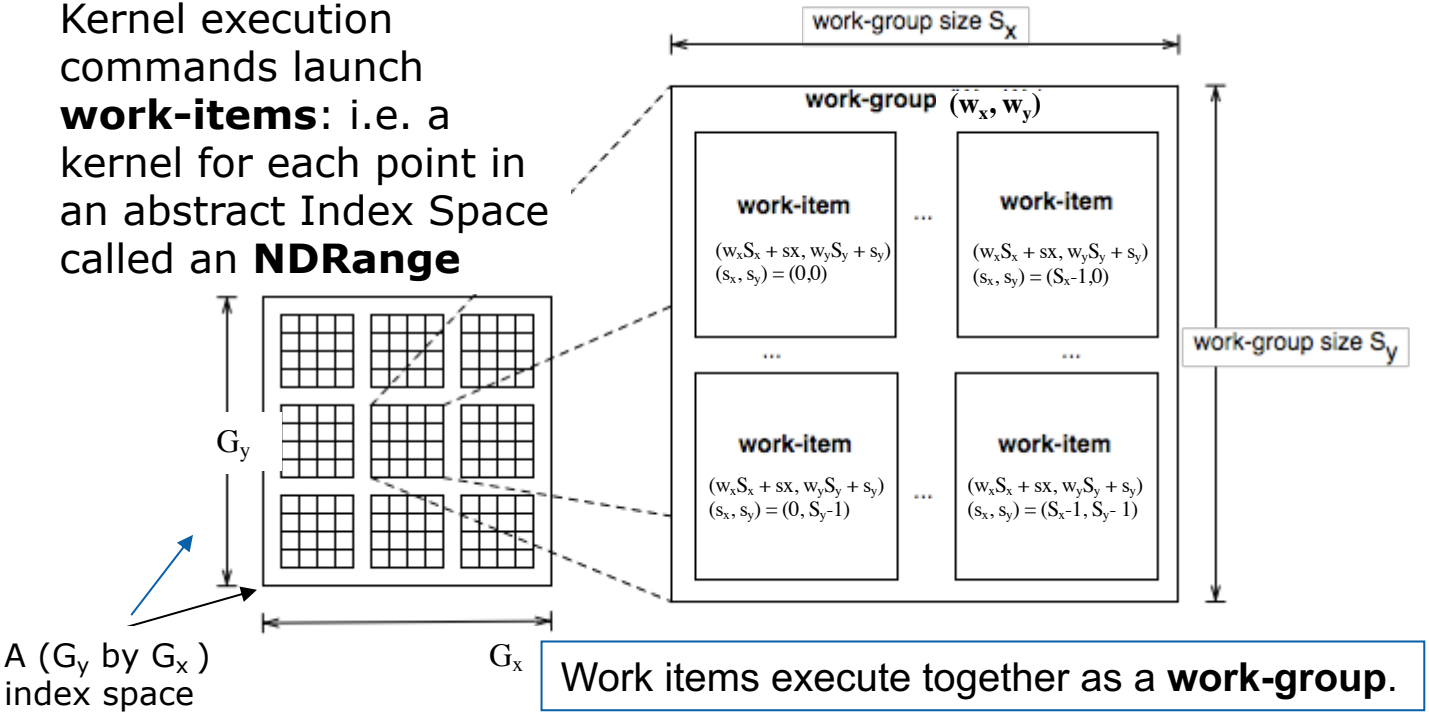
```
__kernel void times_two(
    __global float* input,
    __global float* output)
{
   int i = get_global_id(0);
   output[i] = 2.0f * input[i];
}
```



The OpenCL Execution Model

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**



Work items execute together as a **work-group**.

OpenCL vs. CUDA Terminology

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

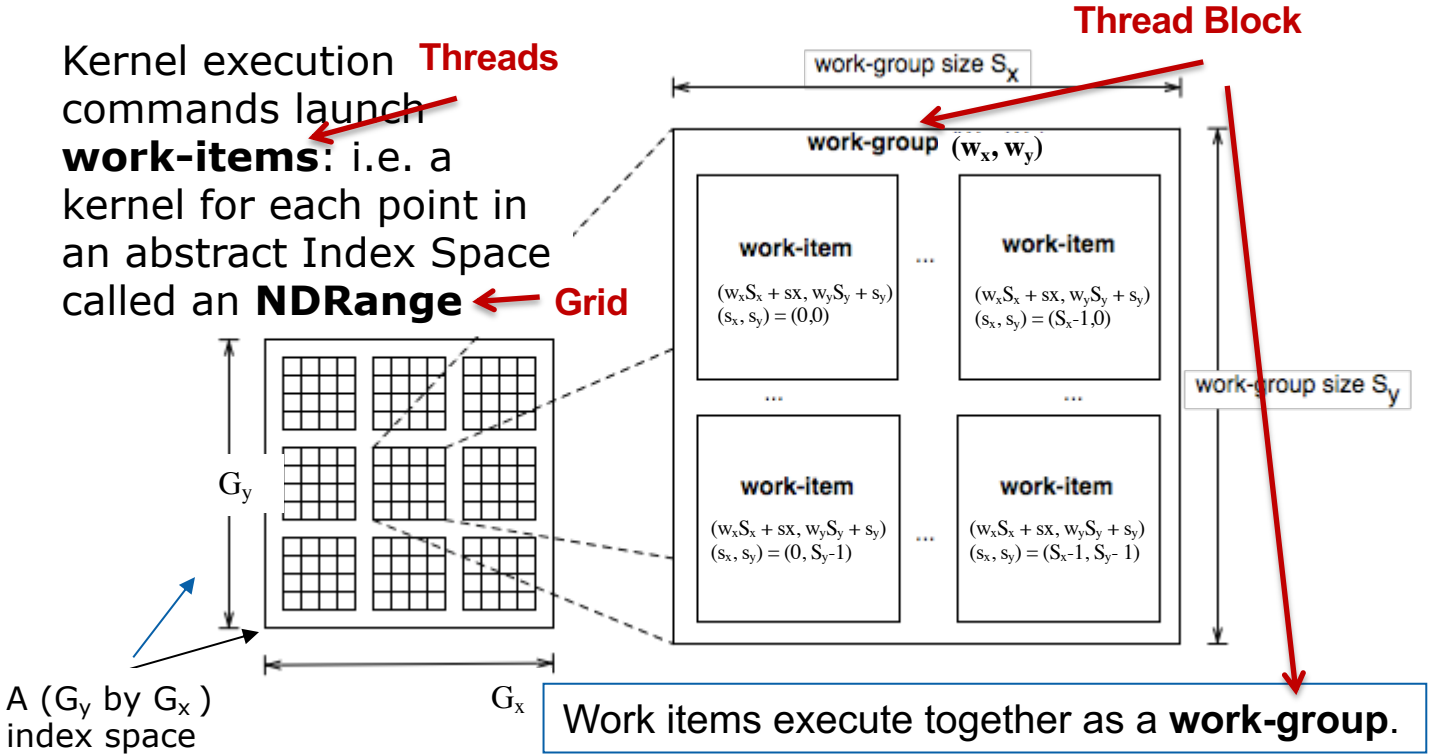
CUDA Stream

Thread Block

Threads

Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**

Grid



Work items execute together as a **work-group**.

Vector Addition - Kernel

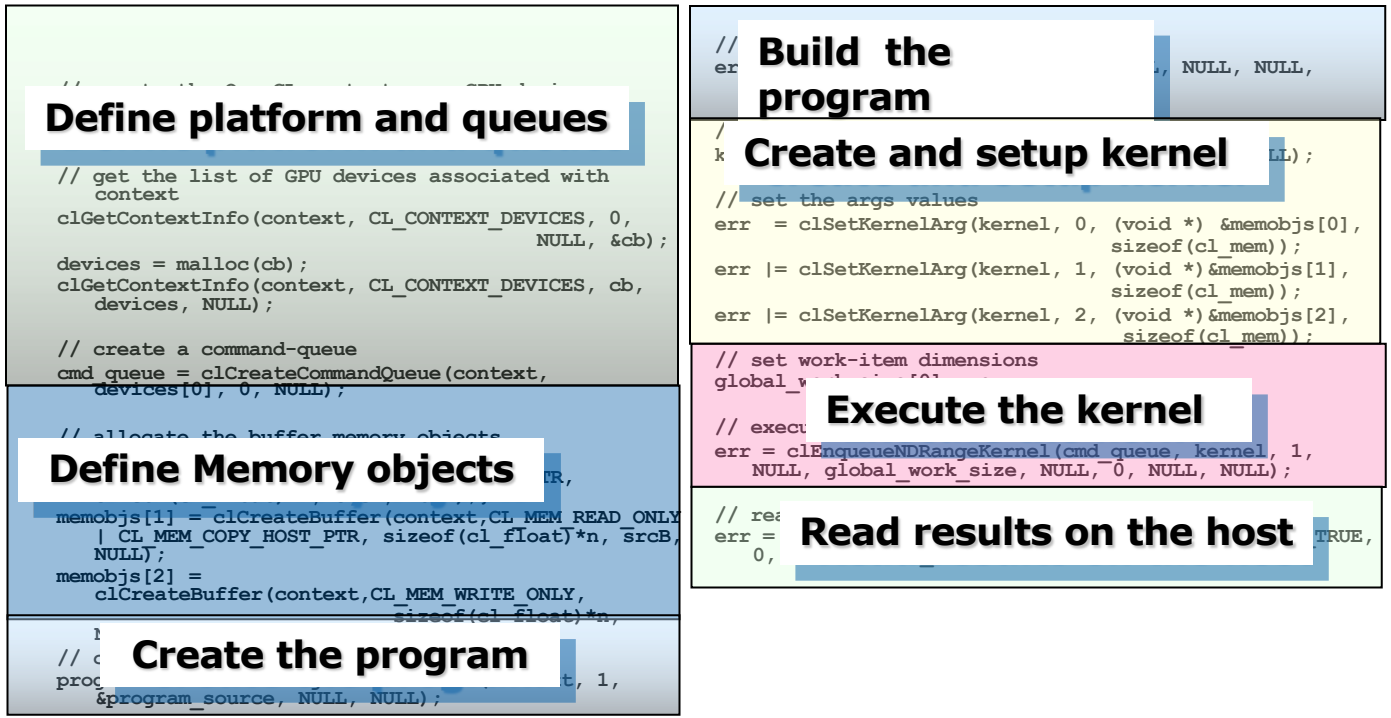

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
 {
     int gid = get_global_id(0);
     c[gid] = a[gid] + b[gid];
}
```

Vector Addition: Host Program


```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
   CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with
   context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,
   devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   sizeof(cl_float)*n, srcA,
                                        NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY
   | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
                                        NULL);
memobjs[2] =
   clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                          sizeof(cl_float)*n,
   NULL,
                                        NULL);
// create the program
program = clCreateProgramWithSource(context, 1,
   &program_source, NULL, NULL);
```

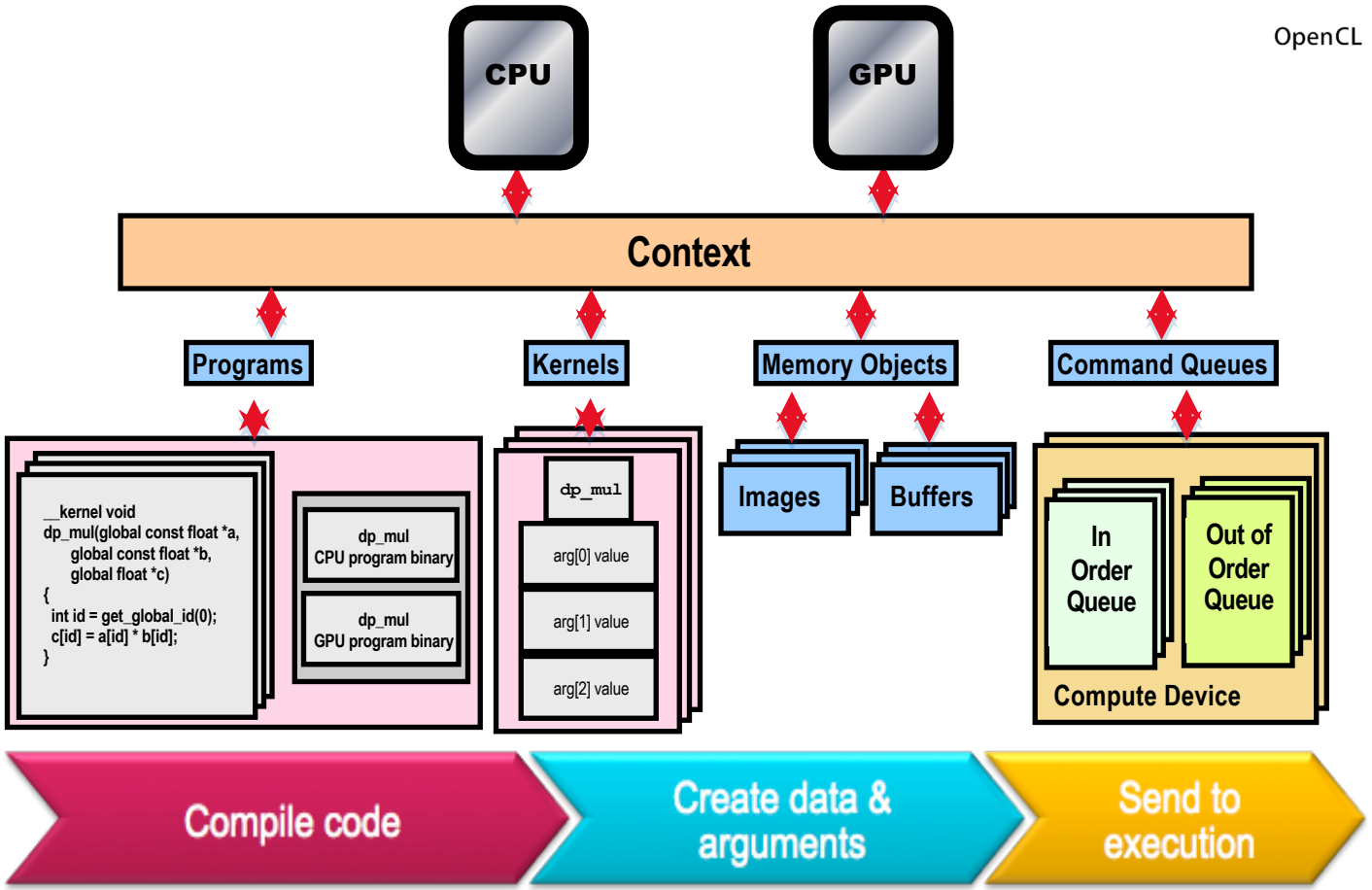
```
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
                                        NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                   sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                   sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                    sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
   CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

Vector Addition: Host Program



It's complicated, but most of this is "boilerplate" and not as bad as it looks.

OpenCL summary

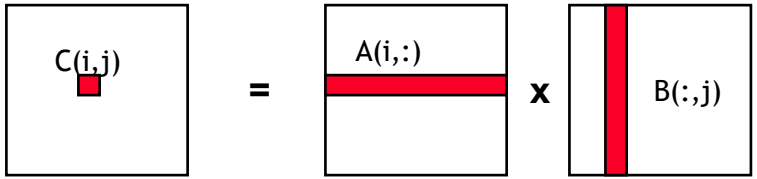


Whining about performance Portability

- Do we have performance portability today?
  - NO: Even in the “serial world” programs routinely deliver single digit efficiencies.
  - If the goal is a large fraction of peak performance, you will need to specialize code for the platform.
- However there is a pretty darn good performance portable language. It’s called OpenCL

Matrix multiplication example:
Naïve solution, one dot product per element of C

- Multiplication of two dense matrices.



Dot product of a row of A and a column of B for each element of C

- To make this fast, you need to break the problem down into chunks that do lots of work for sub problems that fit in fast memory (OpenCL local memory).

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
          C[i*N+j] += A[i*N+k] * B[k*N+j];
        }
      }
    }
}
```

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
          C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Let's get rid of all those ugly brackets

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (i = ib*NB; i < (ib+1)*NB; i++)
     for (jb = 0; jb < NB; jb++)
       for (j = jb*NB; j < (jb+1)*NB; j++)
         for (kb = 0; kb < NB; kb++)
           for (k = kb*NB; k < (kb+1)*NB; k++)
             C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Break each loop into chunks with a size chosen to match the size of your fast memory

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Rearrange loop nest to move loops over blocks "out" and leave loops over a single block together

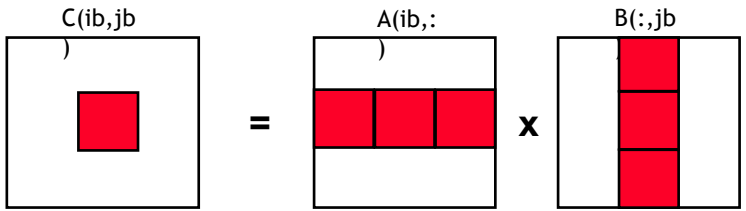
Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

This is just a local matrix multiplication of a single block

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // C_ib,jb = A_ib,kb * B_kb,jb
}
```

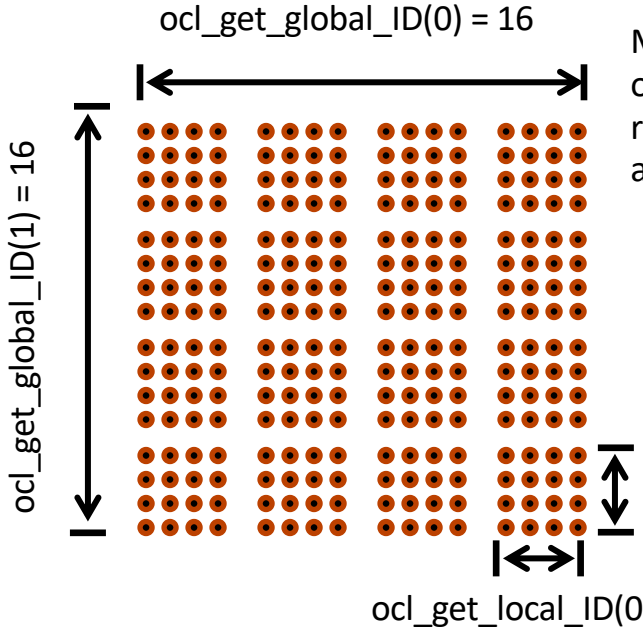


Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Mapping into A, B, and C from each work item

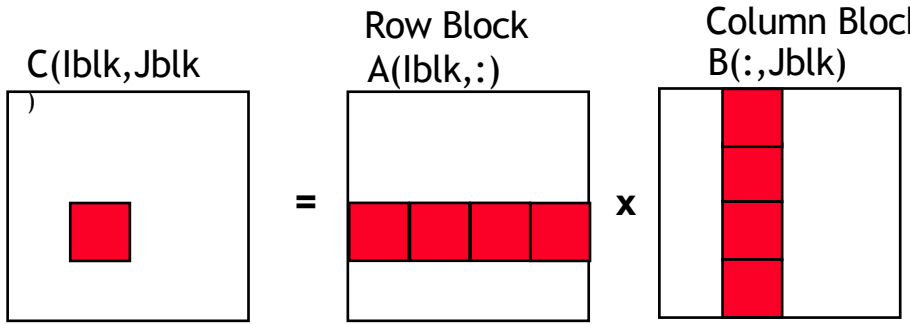
Understanding index offsets in the blocked matrix multiplication program.

16 x 16 NDRange with workgroups of size 4x4



Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

Mapping into A, B, and C from each work item

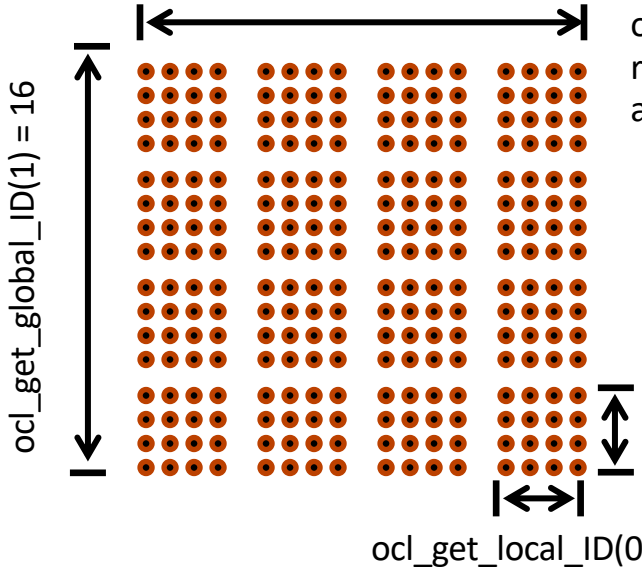


Understanding index offsets in the blocked matrix multiplication program.

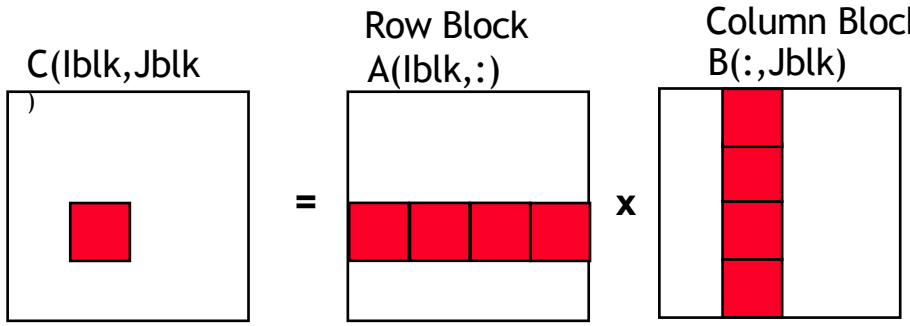
16 x 16 NDRange with workgroups of size 4x4

ocl_get_global_ID(0) = 16

Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).



Mapping into A, B, and C from each work item



Understanding index offsets in the blocked matrix multiplication program.

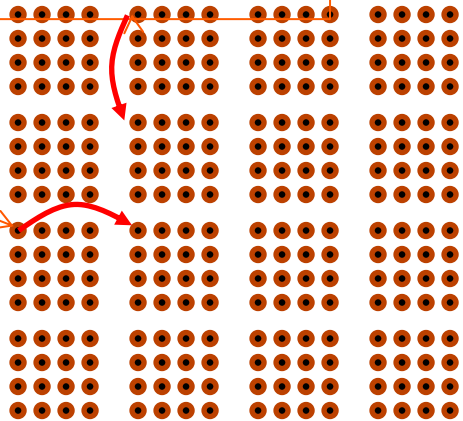
16 x 16 NDRange with workgroups of size 4x4

Consider indices for computation of the block C(Iblk=2, Jblk=1)

Bbase = Jblk*blksz = 1*4

Abase = Iblk*N*blksz
= 1 * 16 * 4

Subsequent A blocks by shifting index by Ainc = blksz = 4

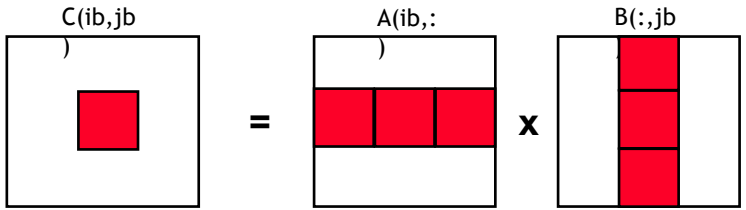


Subsequent B blocks by shifting index by Binc = blksz * N
= 4 * 16 = 64

Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

Portable performance: dense matrix multiplication

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // C_ib,jb = A_ib,kb * B_kb,jb
}
```



Transform the basic serial matrix multiply into multiplication over blocks

Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
              const unsigned int N,
              __global float* A,
              __global float* B,
              __global float* C,
              __local  float* Awrk,
              __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
      //Each work-item loads a single element of the two
      //blocks which are shared with the entire work-
group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      Abase += Ainc;    Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Blocked matrix multiply: kernel

It's getting the indices right that makes this hard

```
#define blksz 16
__kernel void mmul(
              const unsigned int N,
              __global float* A,
              __global float* B,
              __global float* C,
              __local  float* Awrk,
              __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
      //Each work-item loads a single element of the two
      //blocks which are shared with the entire work-
group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      Abase += Ainc;    Bbase += Binc;
  }
  C[j*N+i] = Ctmp;
}
```

Load A and B blocks, wait for all work-items to finish

Wait for everyone to finish before going to next iteration of Kblk loop.

Matrix multiplication … Portable Performance (in MFLOPS)

- Single Precision matrix multiplication (order 1000 matrices)

Case	CPU	Xeon Phi	Core i7, HD Graphics	NVIDIA Tesla
Sequential C (compiled /O3)	224.4		1221.5	
C(i,j) per work-item, all global	841.5	13591		3721
C row per work-item, all global	869.1	4418		4196
C row per work-item, A row private	1038.4	24403		8584
C row per work-item, A private, B local	3984.2	5041		8182
Block oriented approach using local (blksz=16)	12271.3	74051 (126322*)	38348 (53687*)	119305
Block oriented approach using local (blksz=32)	16268.8			

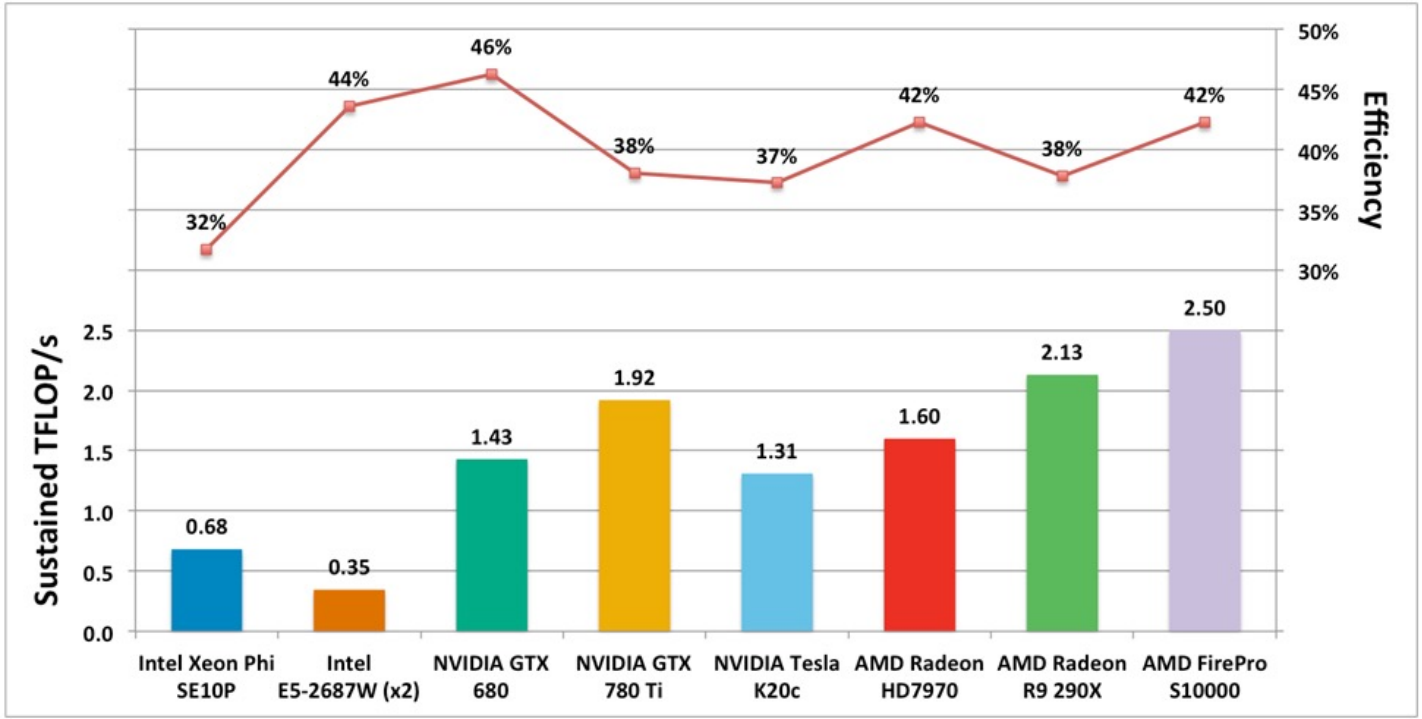
Xeon Phi SE10P, CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB = 4 MB
* The comp was run twice and only the second time is reported (hides cost of memory movement.

Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, OpenCL SDK 2013, MKL 11.0 update 3.
Intel Core i7-4850HQ @ 2.3 GHz which has an Intel HD Graphics 5200 w/ high speed memory. ICC 2013 sp1 update 2.
Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

BUDE: Bristol University Docking Engine

One program running well on a wide range of platforms

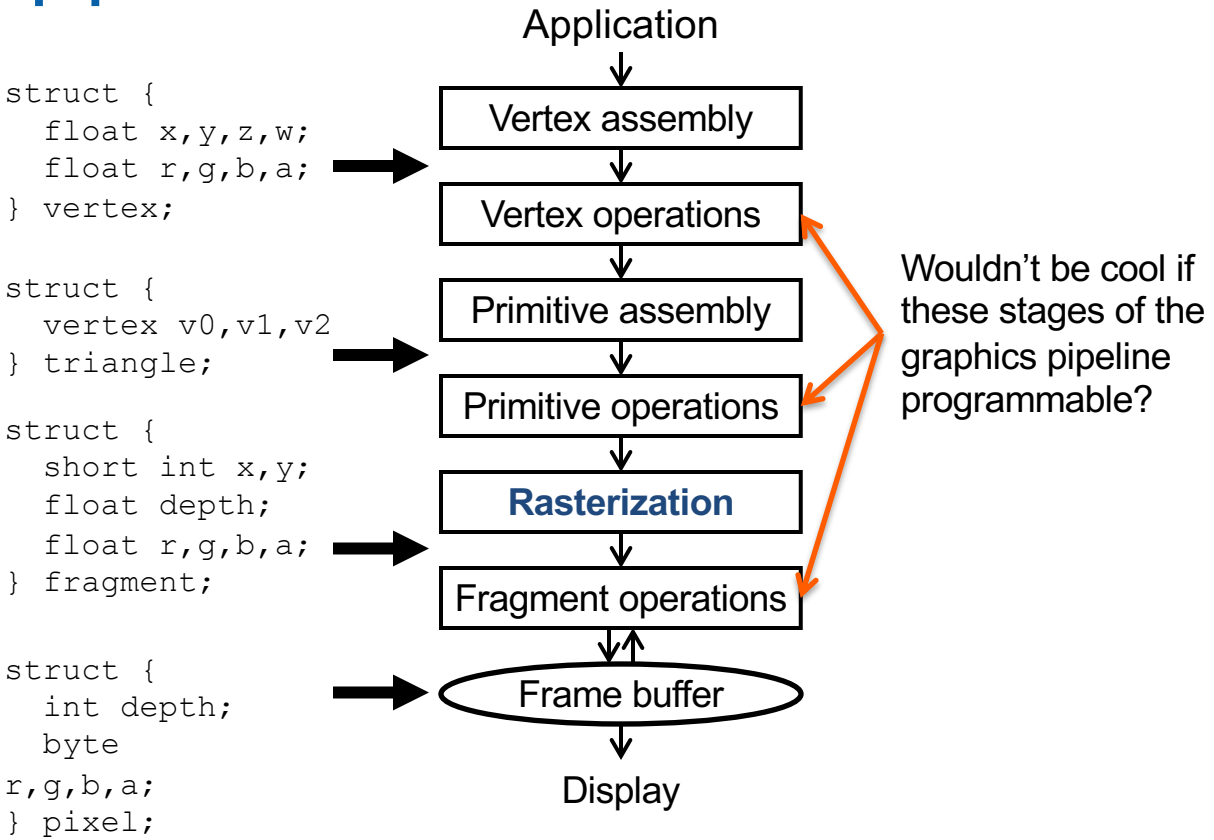


Backup Content

(or what I'd cover if we had a couple more hours)

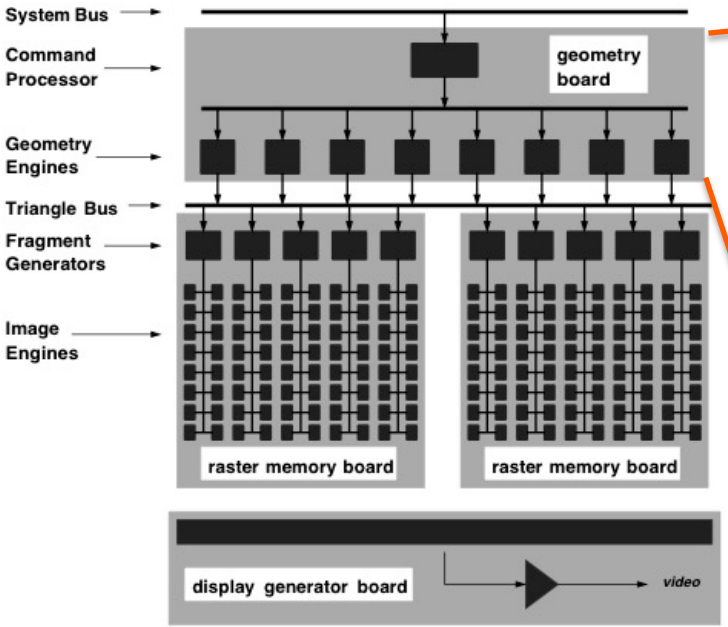
- Developing a complex kernel in OpenCL; the dream of performance portability
- The history of General Purpose GPU programming (GPGPU)
- Debunking the 100X GPU vs. CPU Myth

Let's take a deeper look at the GPU: The vertex pipeline



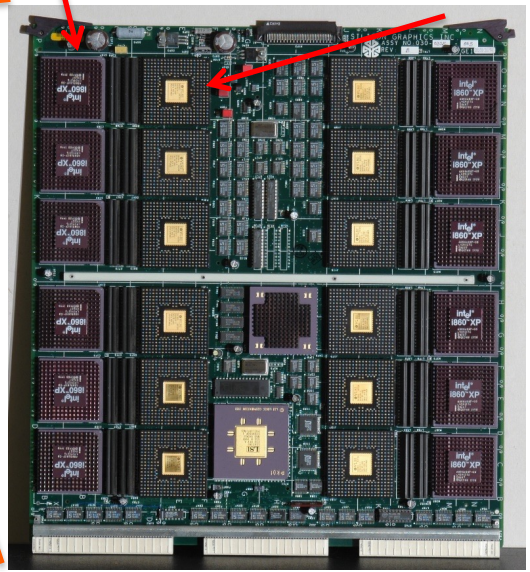
Thanks to Kurt Akeley

High-end GPUs have historically been programmable



Intel i860 RISC CPU

Custom ASIC for processor interconnect



Silicon Graphics RealityEngine GPU
1993

- I860 billed as a “Cray-on-a-chip”
  0.80 micron technology
  2.5M transistors

Programming GPUs

First paper on GPGPU programming I could find dates to 1995 … though the term GPGPU didn't appear in the literature until ~2000.

Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware

Brian Cabral, Nancy Cam, and Jim Foran
Silicon Graphics Computer Systems*

Abstract

Volume rendering and reconstruction centers around solving two related integral equations: a volume rendering integral (a generalized Radon transform) and a filtered back projection integral (the inverse Radon transform). Both of these equations are of the same mathematical form and can be dimensionally decomposed and approximated using Riemann sums over a series of resampled images. When viewed as a form of texture mapping and frame buffer accumulation, enormous hardware enabled performance acceleration is possible.

1 Introduction

Volume Visualization encompasses not only the viewing but also the construction of the volumetric data set from the more basic projection data obtained from sensor sources. Most volumes used in rendering are derived from such sensor data. A primary example being Computer Aided Tomographic (CAT) x-ray data. This data is usually a series of two dimensional projections of a three dimensional volume. The process of converting this projection data back into a volume is called *tomographic reconstruction*.[1] Once a volume is tomographically reconstructed it can be visualized using volume rendering techniques.[5, 7, 13, 15, 16, 17]

These two operations have traditionally been decoupled, being handled by two separate algorithms. It is, however, highly beneficial to view these two operations as having the same mathematical and algorithmic form. Traditional volume rendering techniques can be reformulated into equivalent algorithms using hardware texture mapping and summing buffer. Similarly, the Filtered Back Projection CT algorithm can be reformulated into an algorithm which also uses texture mapping in combination with an accumulation or summing buffer.

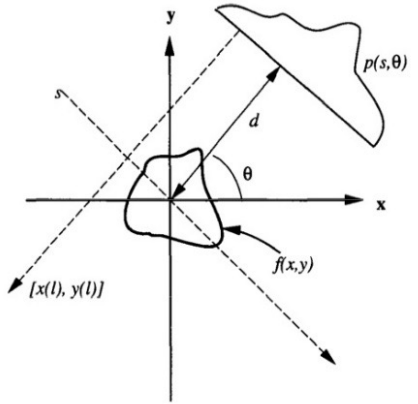


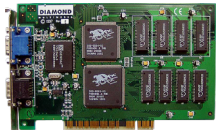
Figure 1: The Radon transform represents a generalized line integral projection of a 2-D (or 3-D) function $f(x, y, z)$ onto a line or plane.

der and reconstruct volumes at rates of 100 to 1000 times faster than CPU based techniques.

2 Background: The Radon and Inverse Radon Transform

We begin by developing the mathematical basis of volume rendering and reconstruction. The most fundamental of which is the Radon

The evolutions of the GPU



1st generation: Voodoo 3dfx (1996)

2nd Generation:
GeForce 256/Radeon 7500 (1998)

3rd Generation: GeForce3/Radeon 8500 (2001). The first GPU to allow a limited programmability in the vertex pipeline.

4th Generation: Radeon 9700/GeForce FX (2002): The first generation of "fully-programmable" graphics cards.

5th Generation: GeForce 8800/HD2900 (2006) and the birth of CUDA

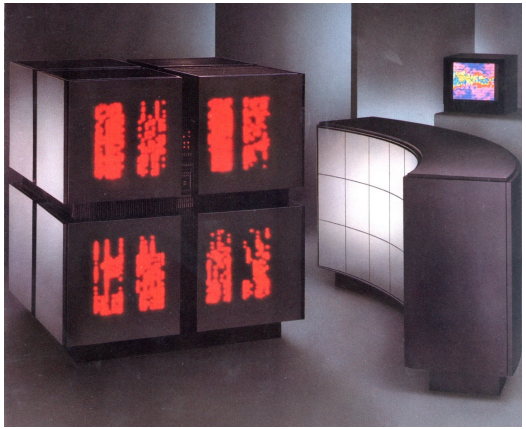
GPGPU arrives: 2006

- GeForce 8800/HD2900:
  - Ground-up GPU redesign
  - Support for Direct3D 10
  - Geometry Shaders
  - Stream out / transform-feedback
  - Unified shader processors
- ***Support for General GPU programming***

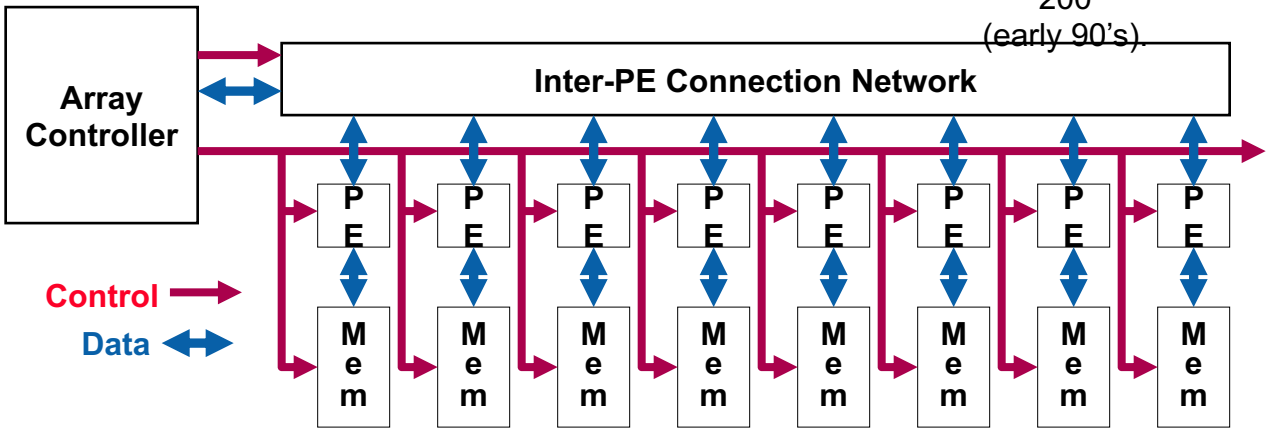
- Fortunately for NVIDIA, the academic community had been working on GPGPU programming for almost a decade.
- Ian Buck at Stanford was wrapping up his dissertation “Stream computing on Graphics Hardware” and the language “Brook”.
- He moved over to NVIDIA and led the effort to create CUDA.
- CUDA was extremely influential … Late in 2008 Apple, AMD, Intel, NVIDIA, Imagination Technologies and several other companies released a vendor-neutral, portable standard for stream computing called OpenCL.

Understanding GPGPU programming: SIMD Architecture

- Single Instruction Multiple Data (SIMD)
- Central controller broadcasts instructions to multiple processing elements (PEs)
  - Only requires one controller for whole array
  - Only requires storage for one copy of program
  - All computations fully synchronized

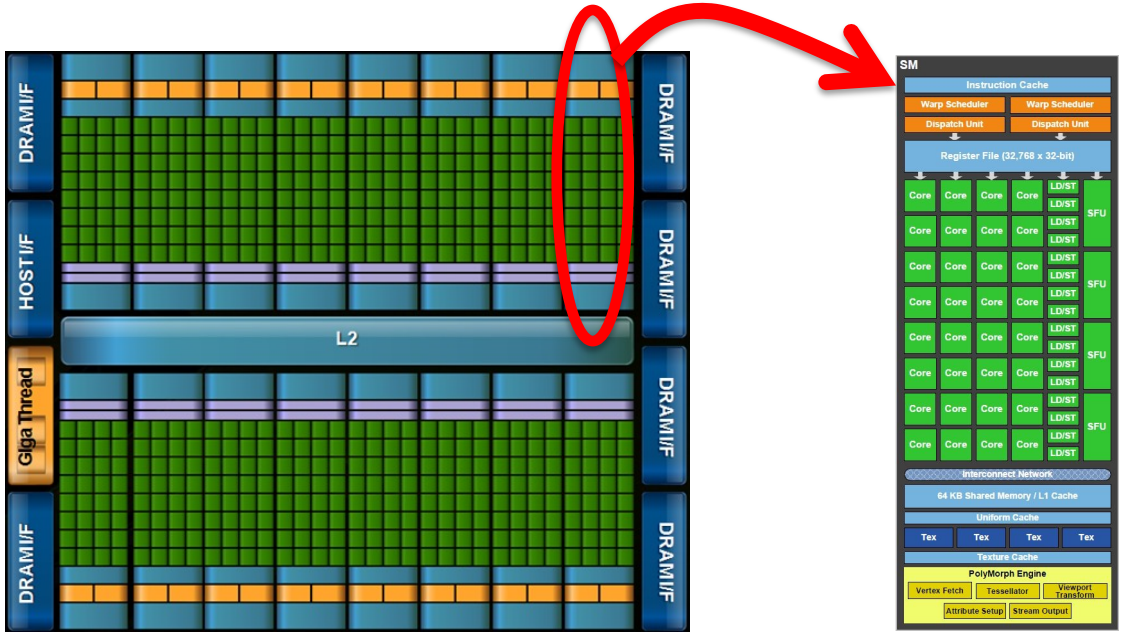


Thinking Machines Corp CM-200
(early 90's).

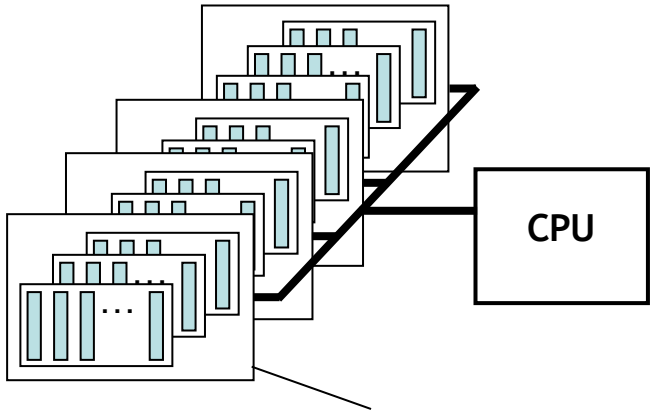


Nvidia GPU Architecture

- Nvidia GPUs are a collection of “Streaming Multiprocessors”
  - Each SM is analogous to a core of a Multi-Core CPU
- Each SM is a collection of SIMD execution pipelines that share control logic, register file, and L1 Cache



GPU Platform Model



- The GPUs are driven by a CPU which …
  - Manages the code to execute on the GPUs
  - Maintains a queue of kernels to execute
  - Manages memory on the GPU and movement between the CPU and the GPU

Backup Content

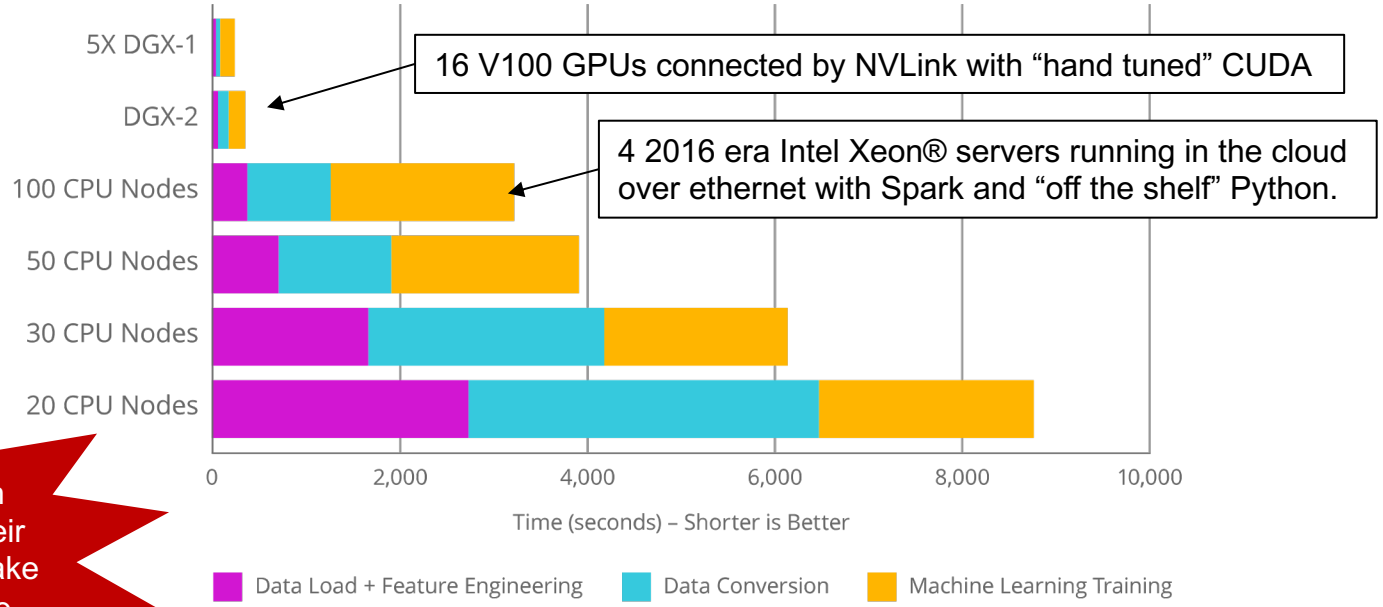
(or what I'd cover if we had a couple more hours)

- Developing a complex kernel in OpenCL; the dream of performance portability
- The history of General Purpose GPU programming (GPGPU)
- Debunking the 100X GPU vs. CPU Myth

NVIDIA Performance claims

An Nvidia slide from CLSAC'18 talk

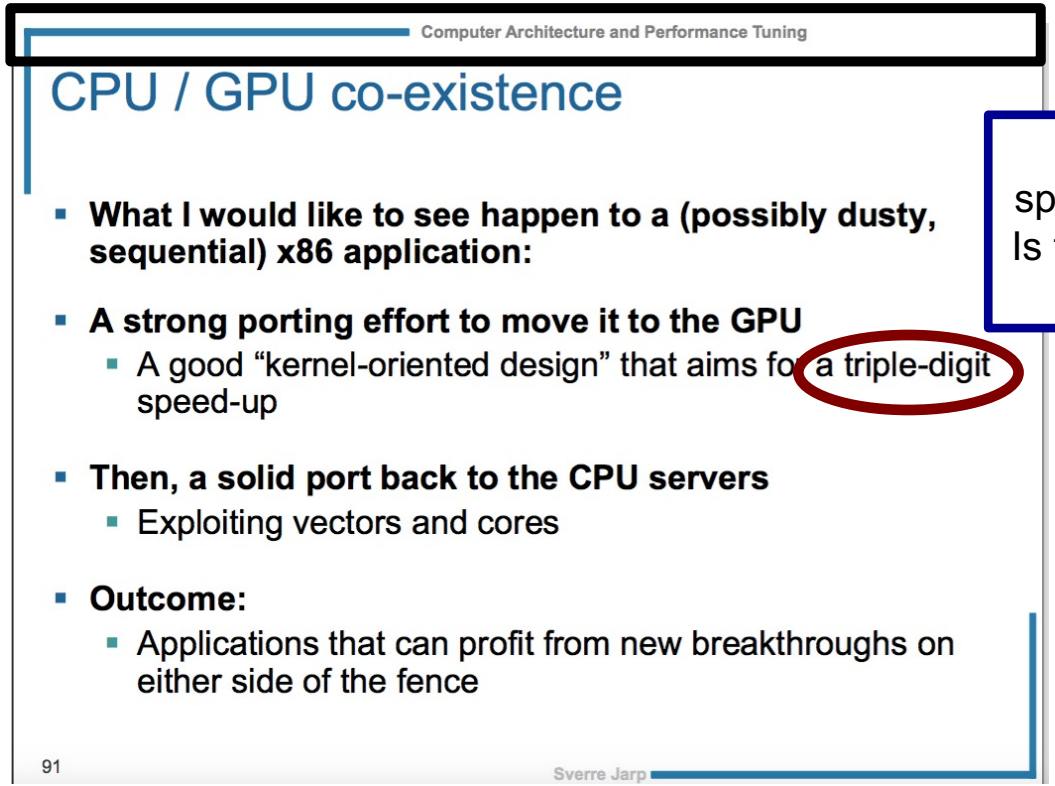
End-to-End Faster Speeds on RAPIDS



You gotta watch these guys … their marketing folks "take liberties" with the facts

"CPU Node" = 1 AWS Broadwell Intel® Xeon® E5 vCPU (thread)

100X speedups from GPUS: a common myth



Triple digit speedups? Really? Is this a reasonable goal?

Source: a great ESC'15 lecture by a smart person who made a mistake!!!

A high-level view of performance

•Well optimized applications are either compute or bandwidth bounded

- **For bandwidth bound applications:**

  *Performance = Arch efficiency * Peak Bandwidth Capability*

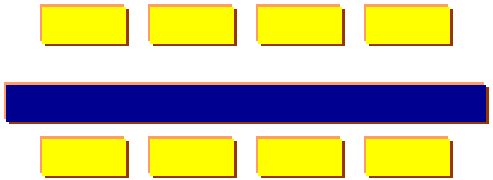
- **For compute bound applications:**

  *Performance = Arch efficiency * Peak Compute Capability*

Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

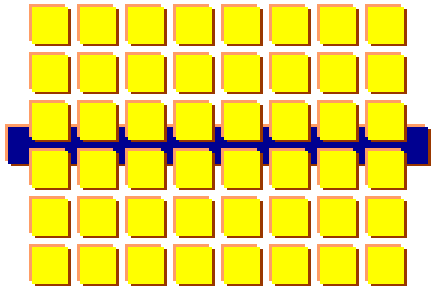
Reasonable Speedup Expectations

- Chip A



$Perf_A = Eff_A * Peak_A(Comp \text{ or } BW)$

- Chip B



$Perf_B = Eff_B * Peak_B(Comp \text{ or } BW)$

$$Speedup\frac{B}{A} = \frac{Perf_B}{Perf_A} = \frac{Eff_B}{Eff_A} * \frac{Peak_A(Comp\_or\_BW)}{Peak_B(Comp\_or\_BW)}$$

Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

Speedup expectations for well optimized code: CPU vs. GPU

Core i7 960

- Four OoO Superscalar Cores, 3.2GHz
- Peak SP Flop: 102GF/s
- Peak BW: 30 GB/s

GTX 280

- 30 SMs (w/ 8 In-order SP each), 1.3GHz
- Peak SP Flop: 933GF/s*
- Peak BW: 141 GB/s

Assuming both Core i7 and GTX280 have the same efficiency:

	Max Speedup: GTX 280 over Core i7 960
Compute Bound Apps: (SP)	933/102 = 9.1x
Bandwidth Bound Apps:	141/30 = 4.7x

* 933GF/s assumes mul-add and the use of SFU every cycle on GPU

Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

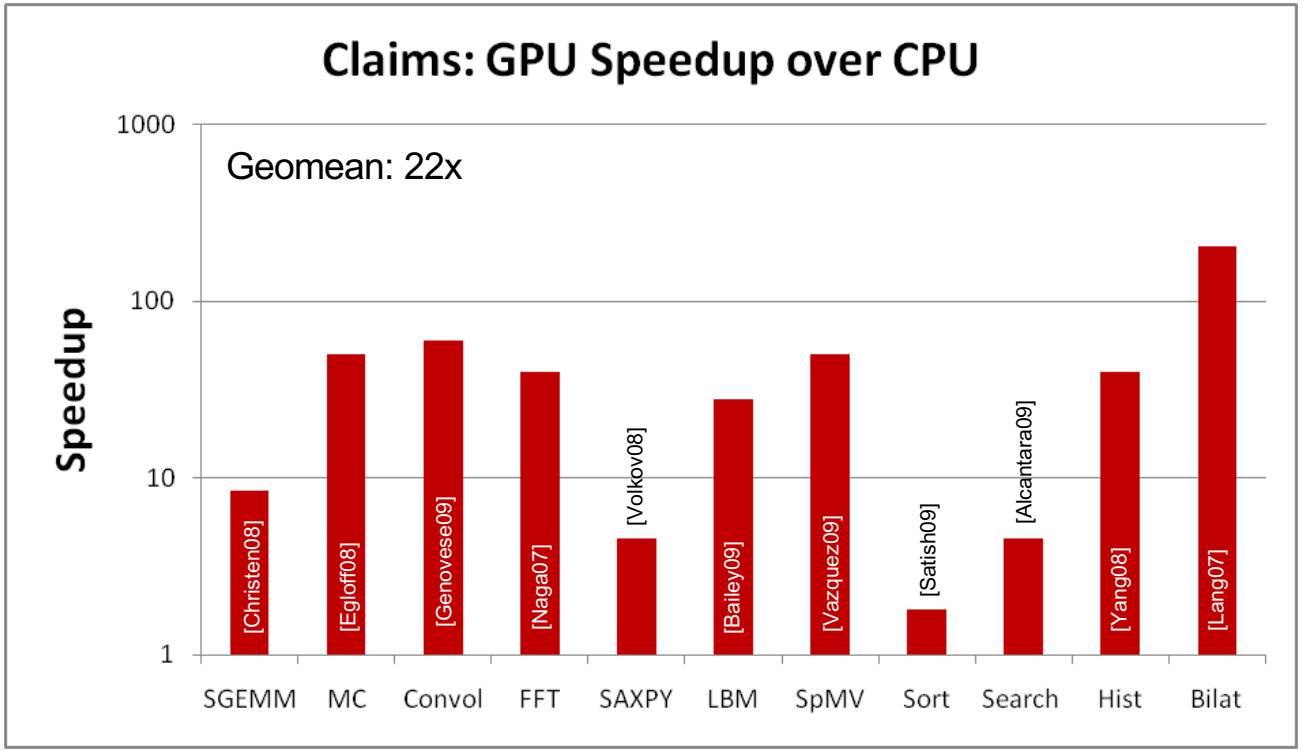
A fair comparison of CPUs and GPUs: Methodology

- Start with previously best published code / algorithm
- Validate claims by others
- Optimize BOTH CPU and GPU versions
- Collect and analysis performance data

Note: Only computation time on the CPU and GPU is measured. PCIe transfer time and host application time are not measured for GPU. Including such overhead will lower GPU performance

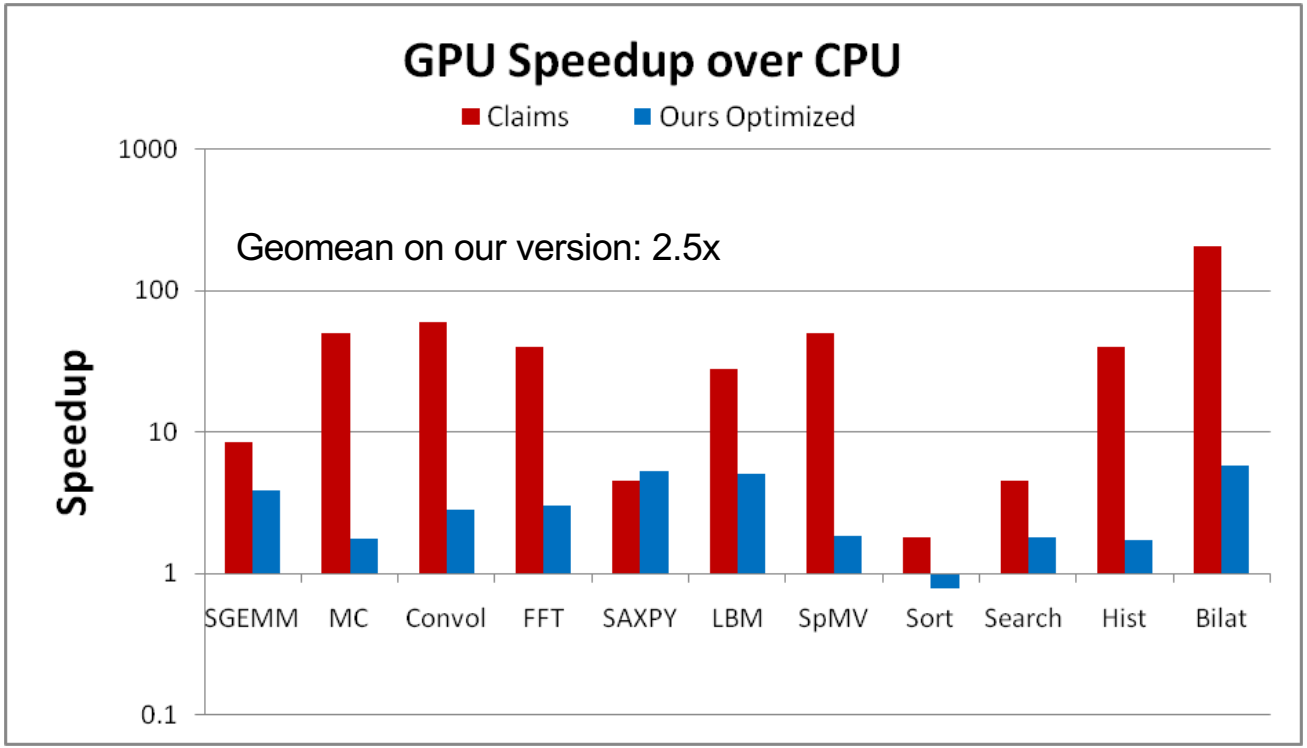
Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

What was claimed



Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

What we measured

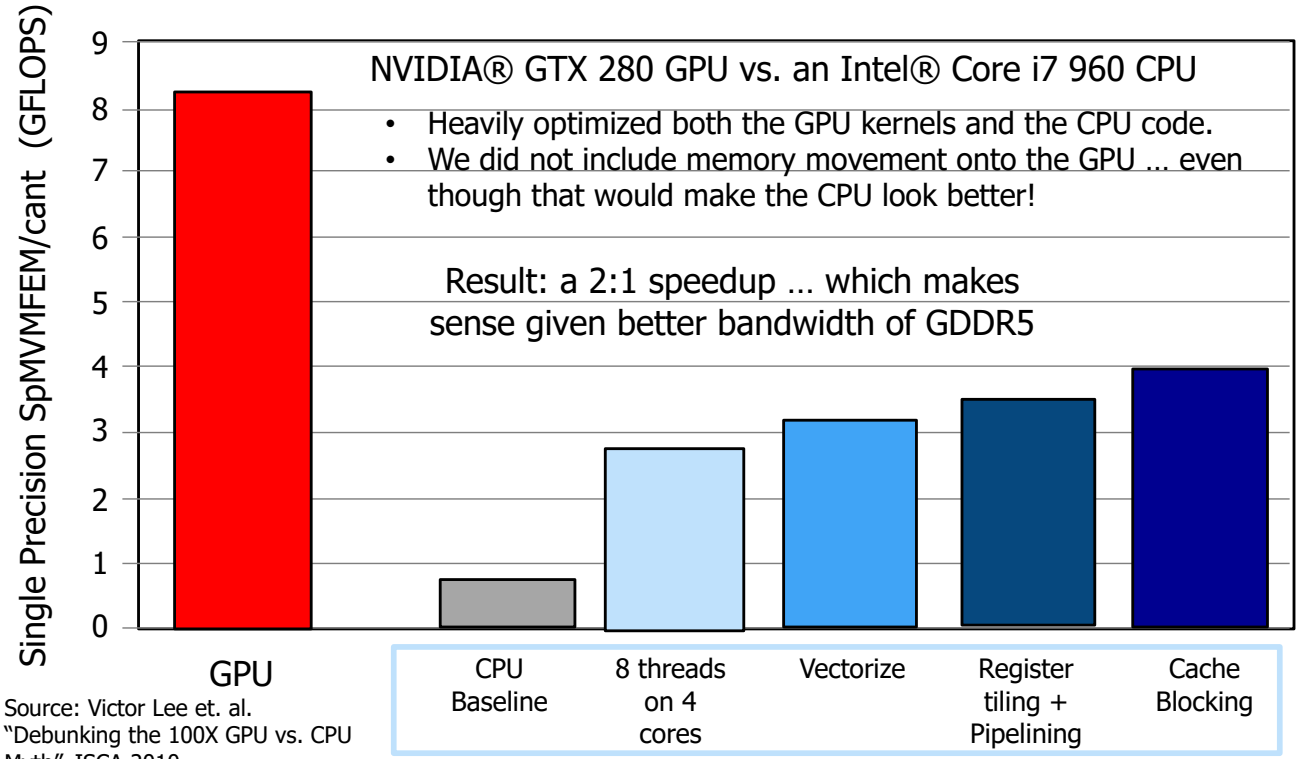


Apps.	SGEMM	MC	Conv	FFT	SAXPY	LBM	Solv	SpMV	GJK	Sort	RC	Search	Hist	Bilat
Core i7-960	94	0.8	1250	71.4	16.8	85	103	4.9	67	250	5	50	1517	83
GTX280	364	1.4	3500	213	88.8	426	52	9.1	1020	198	8.1	90	2583	475

Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

Sparse matrix vector product: GPU vs. CPU

- [Vazquez09]: reported a 51X speedup for an NVIDIA® GTX295 vs. a Core 2 Duo E8400 CPU … but they used an old CPU with unoptimized code



Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

Common Mistakes when comparing a CPU and a GPU

- Compare the latest GPU against an old CPU
- Highly optimized GPU code vs. unoptimized CPU code
  - I've seen numerous papers compare optimized CUDA vs. Matlab or python
- Parallel GPU code vs. serial, unvectorized CPU code.
- Ignore the GPU penalty of moving data across the PCI bus from the CPU to the GPU

GPUs are great and depending on the algorithm can show two to four fold speedups. But not 100+ … that's just irresponsible and should not be tolerated!!