Parallelization

18-613: Foundations of Computer Systems
8th Lecture, **April 9, 2019**

Instructor:

Franz Franchetti

Outline

- **Example: Solving a Linear System of Equations**
- Parallel machine models
- Algorithmic approaches
- Amdahl, strong and weak scaling
- CUDA
- MPI, OpenMP, OpenACC

Solving a Linear System of Equations

- **Problem specification**

Find x s.t. $Ax = b$ with

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

- **Textbook approach: Gauss Elimination**
  - Augmented matrix
  - Elementary row operations
  - Reach echelon form

$$\left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 1 & 1 & -1 & 1 \\ 3 & 11 & 5 & 35 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 0 & -2 & -2 & -8 \\ 0 & 2 & 2 & 8 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 0 & -2 & -2 & -8 \\ 0 & 0 & 0 & 0 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 0 & -2 & -3 \\ 0 & 1 & 1 & 4 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

Do you see any issues here?

Gauss-Seidel and Jacobi Iterations

- **Gauss Seidel: in-place updates**

```
for (t=0; t<T; t++) {
   for (i=1; i<N-1; i++) {
      A[i] = 0.33*(A[i-1] +
             A[i] + A[i+1]);
}
```

$$A = \begin{bmatrix} 0.33 & 0 & 0 & \dots & 0 \\ 0.33 & 0.33 & 0 & \dots & 0 \\ 0.33 & 0.33 & 0.33 & 0 & \dots \\ 0 & 0.33 & 0.33 & 0.33 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0.33 \end{bmatrix}$$

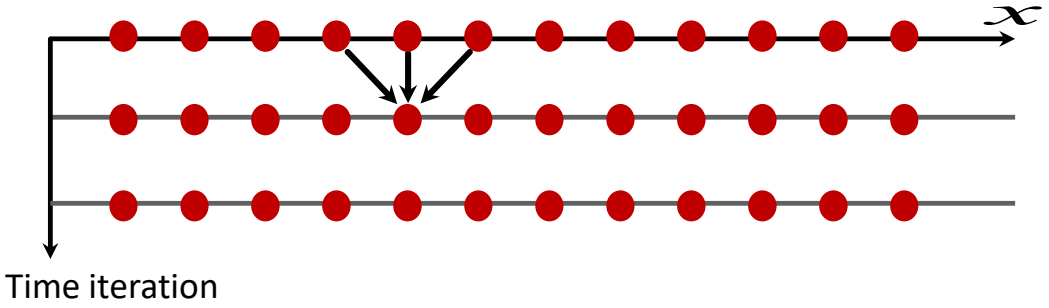
- **Jacobi Iteration**

```
for (t=0; t<T; t++) {
   for (i=1; i<N-1; i++) {
      B[i] = 0.33*(A[i-1] + A[i] + A[i+1]);
   for (i=1; i<N-1; i++)
      A[i] = B[i];
}
```

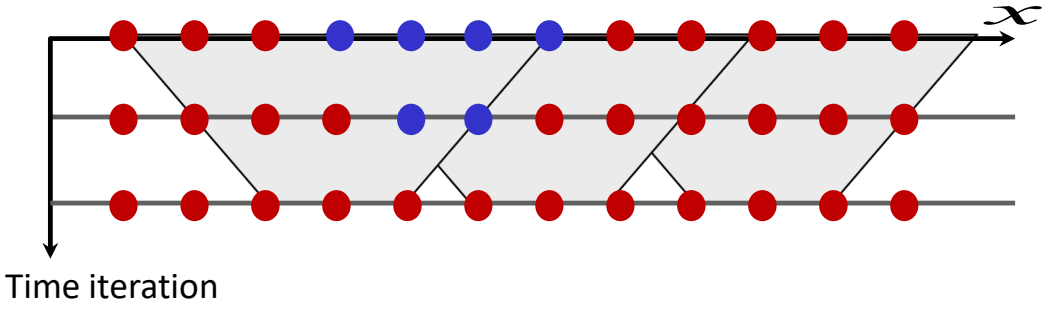
- **Special case: tri-diagonal matrix**

Blocking: Locality and Parallelism

- **Representation of iteration**

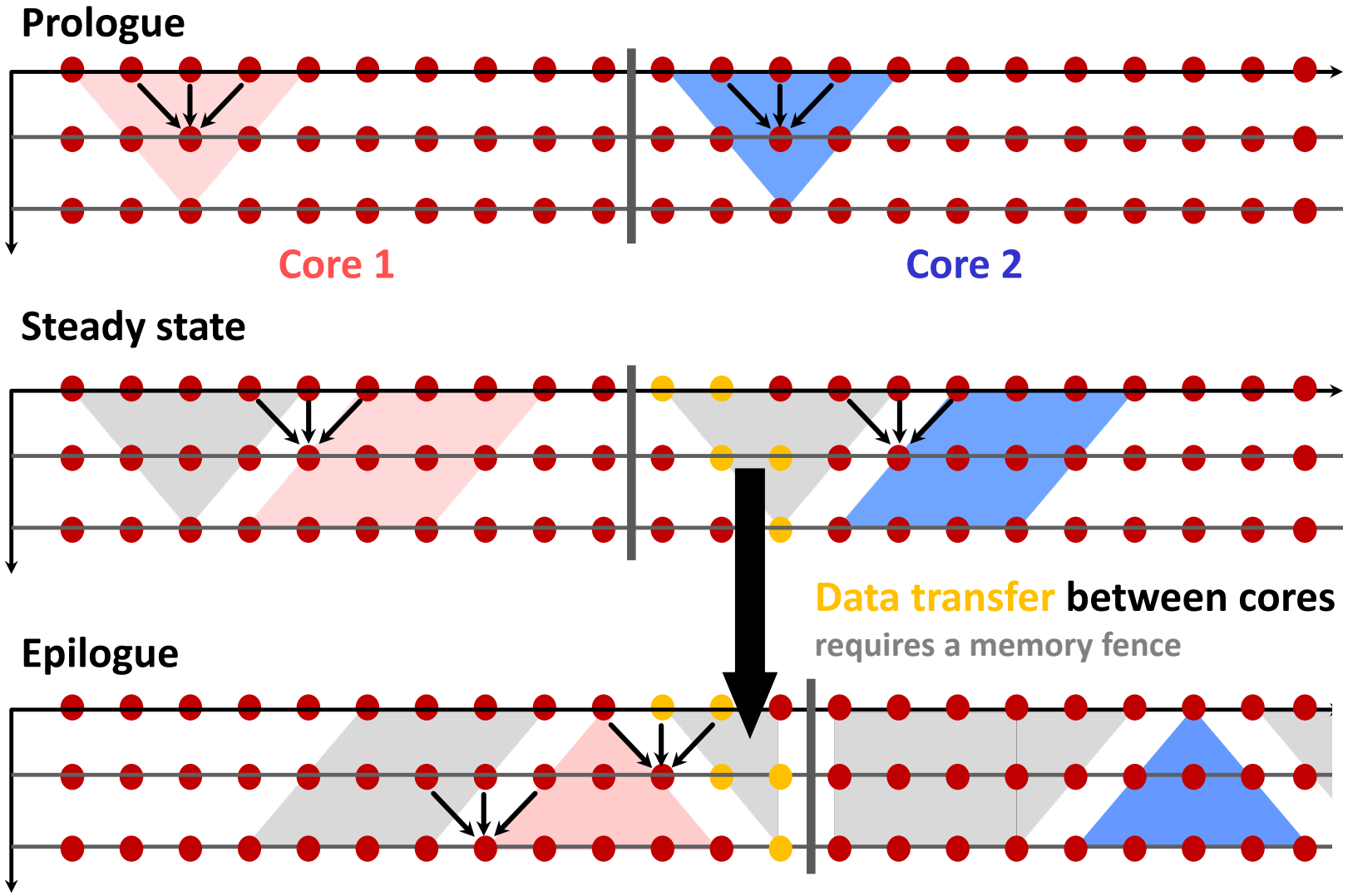


- **Trapezoidal blocking**



Overhead: recomputation, data reloading/communicating

Better Parallelization

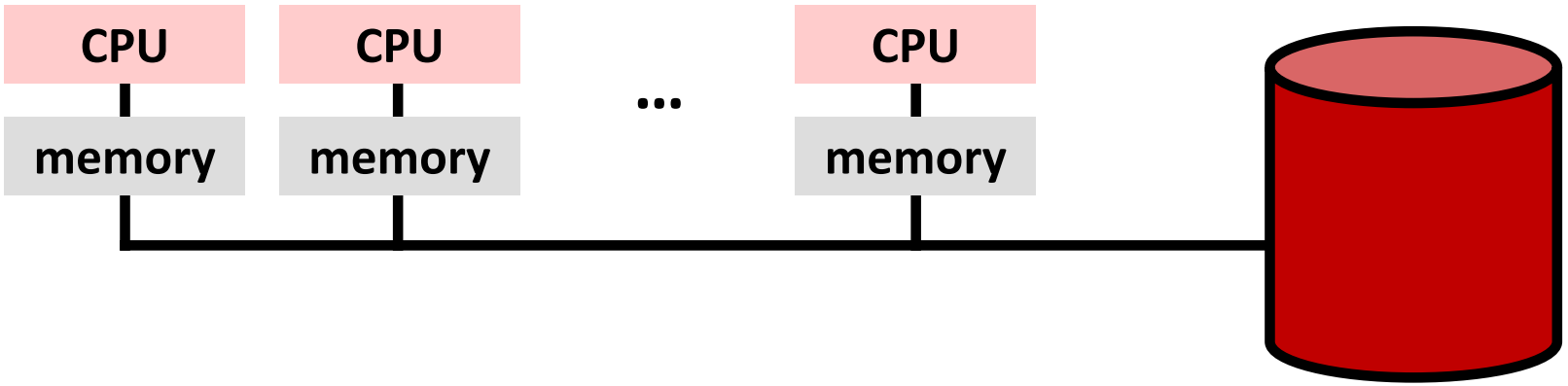


Outline

- Example: Solving a Linear System of Equations
- **Parallel machine models**
- Algorithmic approaches
- Amdahl, strong and weak scaling
- CUDA
- MPI, OpenMP, OpenACC

Distributed Memory: Clusters and MPP

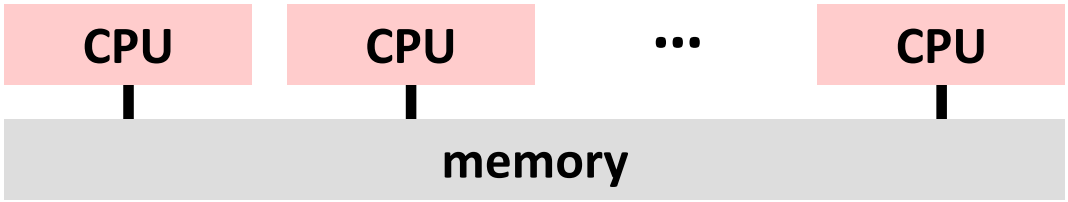
- **Topology: memory distributed, may have central storage**



- **Programming**
  - Programming model: Bulk synchronous parallel
  - Classical/cluster: message passing (MPI)
  - Modern/big data: MapReduce/Hadoop
  - Disks can be central or local (file system can hide that)

Shared Memory: SMP, NUMA, SIMT

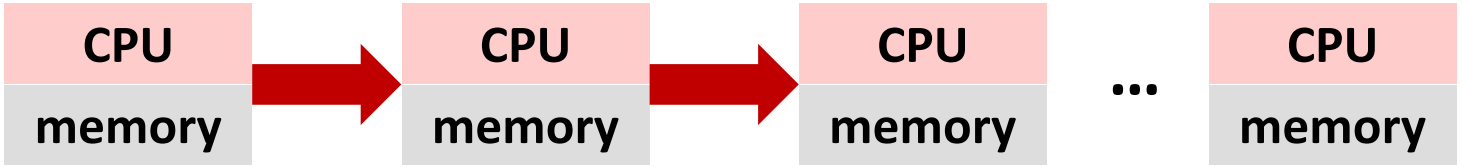
- **Topology: memory is globally addressable (may be physically partitioned)**



- **Programming**
  - Programming model: PRAM
  - OpenMP, pthreads
  - Cilk, TBB
  - CUDA, OpenCL

Pipelining: Systolic Arrays, Workflow

- **Topology: Data is pipelined from unit to unit**



- **Programming**
  - Programming model: data flow
  - TensorFlow
  - Simulink, Labview, StreamIt
  - Graphical tools

Outline

- Example: Solving a Linear System of Equations
- Parallel machine models
- **Algorithmic approaches**
- Amdahl, strong and weak scaling
- CUDA
- MPI, OpenMP, OpenACC

Data Parallelism vs. Task Parallelism

- **Data parallelism: same operation performed on all data**
  - Data is distributed across computing node
  - Parallelism is proportional to problem size
  - Often available in large scale scientific/engineering computations
  - Automatic parallelization well-studied/well-understood
- **Task parallelism: different operation performed across data**
  - More irregular problems
  - Limited parallelism
  - Large scale Parallelism often comes from solving many problems
  - Web servers, data bases
  - Often data parallelism now augmented by task parallelism support

Loop Parallelization

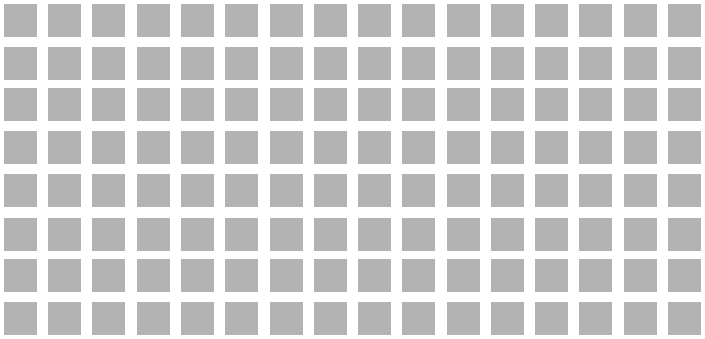
- **Idea: distribute iterations across processors**

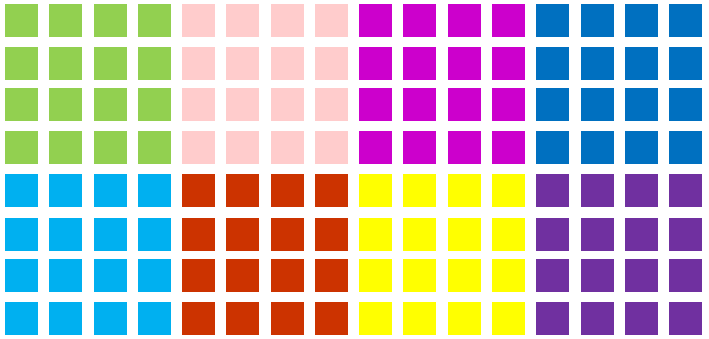
```
// sequential program
for (i=0; i<N; i++) {
   y[2*i]   = x[2*i] + x[2*i+1]
   y[2*i+1] = x[2*i] - x[2*i+1]
}

// run in parallel on processor i, N/2 processors
void iteration(double *x, double *y, int i) {
   y[2*i]   = x[2*i] + x[2*i+1]
   y[2*i+1] = x[2*i] - x[2*i+1]
}
```

- **Core approach for data parallelism across parallel architectures**
  - Shared memory: OpenMP
  - GPUs: CUDA, OpenCL, OpenACC
  - Distributed memory: MPI
  - Loop pipelining

Domain Decomposition

- **Break problem domain into pieces, distribute across processors**





- **Needed for scalable parallelization**
  - Originally: array-based data structures
  - Applies to general data sets
  - MUST for distributed memory, BUT needed everywhere for performance
  - Most systems require locality
  - Advanced: ghost cells, asynchronous updates
  - Distribution: cyclic, block-cyclic,…

Speculation and Transactions

- **How to parallelize sequential problems: try, allow to fail**

  Parse string (state machine):
  Find if string contains "AGCTACGTTAGC"

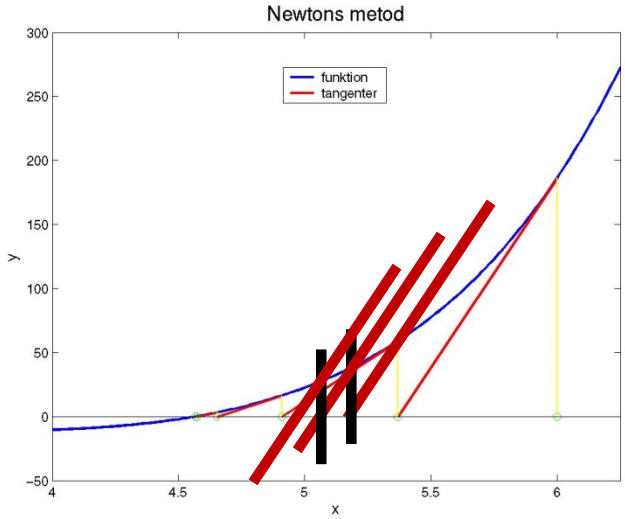
  

  In parallel:
  1) Find if string contains "AGCTAC"
  2) Find if string contains "GTTAGC"
  Then: see if locations are consecutive

- **Often can predict outcome with high probability of success**
  - In hardware: Branch predictions
  - Tree traversals: don't know which way to go—pick one (or all)
  - Must be able to roll back data structure if guess was wrong
  - Transactions: atomic operations that either succeed or fail
  - Important for parallelizing state machines, discrete simulations, etc.

Asynchronous Approaches

- **What if we can tolerate some stale (older) data?**



Newton method with fixed gradient

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

- **Algorithms are often stable w.r.t. old data**
  - Algorithms often converge (maybe slower)
  - PDEs, message passing algorithms
  - Machine learning algorithms: batching of vectors
  - Often trade-off cost of iteration vs. cost of communication/update
  - Some algorithms absolutely cannot tolerate stale data

Outline

- Example: Solving a Linear System of Equations
- Parallel machine models
- Algorithmic approaches
- **Amdahl, strong and weak scaling**
- CUDA
- MPI, OpenMP, OpenACC

Characterizing Parallel Program Performance

- ***p* processor cores, T_k is the running time using *k* cores**
- ***Def.* *Speedup:* $S_p = T_1 / T_p$**
  - S_p is *relative speedup* if T_1 is running time of parallel version of the code running on 1 core
  - S_p is *absolute speedup* if T_1 is running time of sequential version of code running on 1 core
  - Absolute speedup is a much truer measure of the benefits of parallelism
- ***Def.* *Efficiency:* $E_p = S_p / p = T_1 / (pT_p)$**
  - Reported as a percentage in the range (0, 100]
  - Measures the overhead due to parallelization
- **Is super-linear speed-up ($S_p > p$, $E_p > 100\%$) possible?**
  - Yes: Due to hyperthreading and cache effects

Amdahl's Law

- Gene Amdahl (Nov. 16, 1922 – Nov. 10, 2015)

- **Captures the difficulty of using parallelism to speed things up.**
- **Overall problem**
  - T Total sequential time required
  - p Fraction of total that can be sped up ($0 \leq p \leq 1$)
  - k Speedup factor
- **Resulting Performance**
  - $T_k = pT/k + (1-p)T$
    - Portion which can be sped up runs k times faster
    - Portion which cannot be sped up stays the same
  - Least possible running time:
    - $k = \infty$
    - $T_\infty = (1-p)T$

Amdahl's Law Example

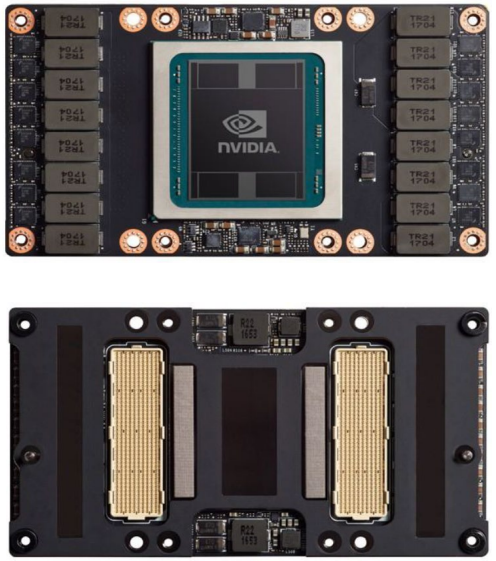
- **Overall problem**
  - T = 10 Total time required
  - p = 0.9 Fraction of total which can be sped up
  - k = 9 Speedup factor

- **Resulting Performance**
  - $T_9 = 0.9 * 10/9 + 0.1 * 10 = 1.0 + 1.0 = 2.0$
  - Least possible running time:
    - $T_\infty = 0.1 * 10.0 = 1.0$

- **Limit on *strong scaling*: fixed problem size, increasing cores**
- **Not on *weak scaling*: problem size scales with increasing cores**

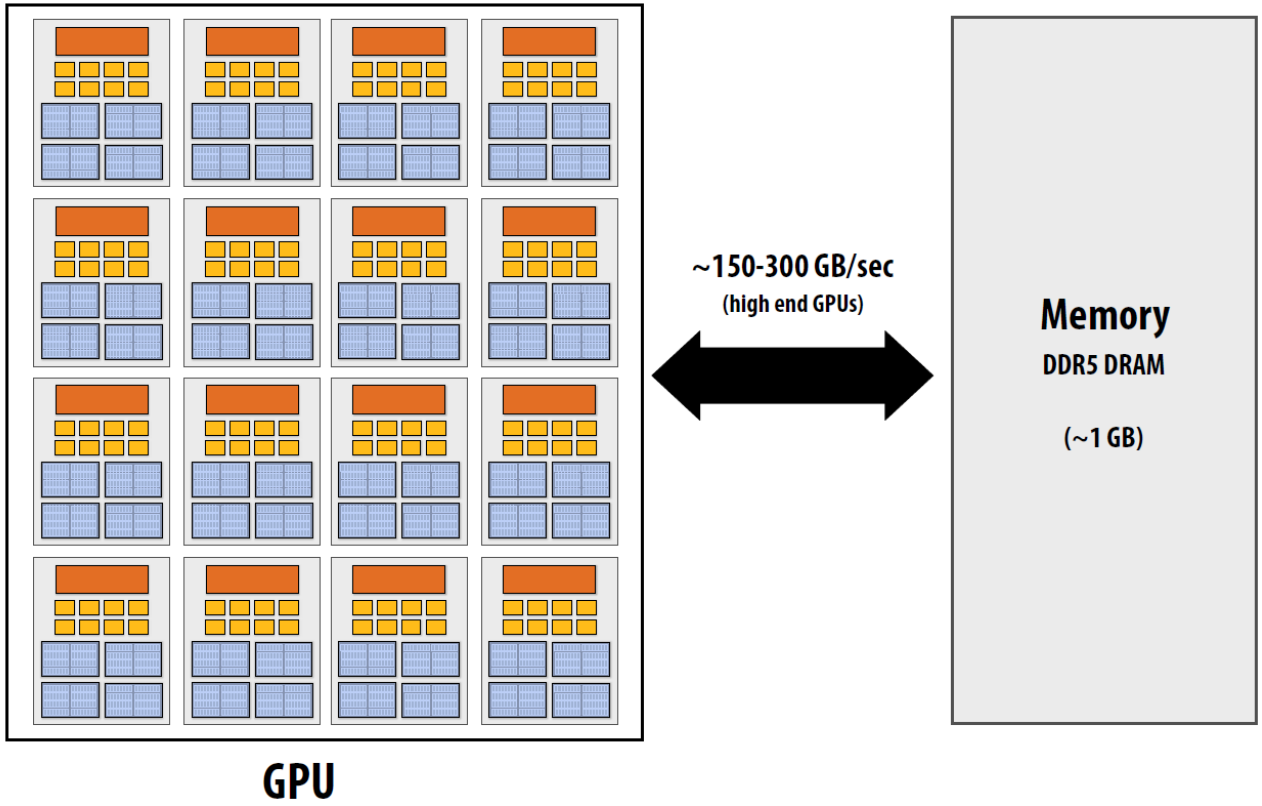
Outline

- Example: Solving a Linear System of Equations
- Parallel machine models
- Algorithmic approaches
- Amdahl, strong and weak scaling
- **CUDA**
- MPI, OpenMP, OpenACC



Based on "15-418/15-618: Parallel Computer Architecture and Programming" by Randy Bryant and Nathan Beckmann

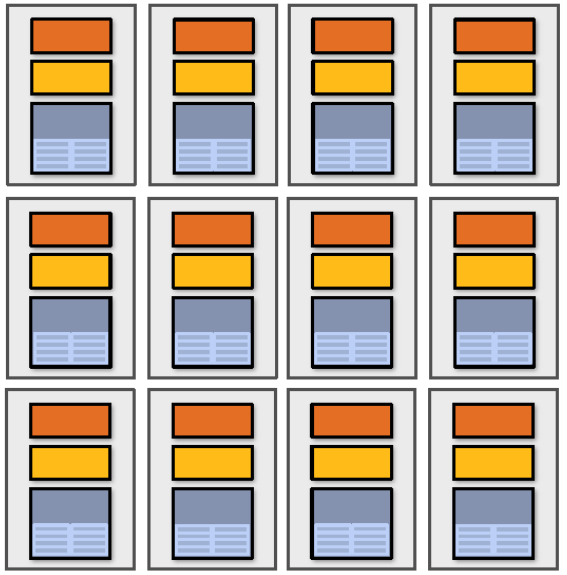
GPU Architecture



- Multi-core chip
- SIMD execution within a single core (many execution units performing the same instruction)
- Multi-threaded execution on a single core (multiple threads executed concurrently by a core)

NVIDIA Tesla architecture (2007)

- (GeForce 8xxx series GPUs)
  First alternative, non-graphics-specific ("compute mode") interface to GPU hardware
- Lets say a user wants to run a non-graphics program on the GPU's programmable cores…
  - Application can allocate buffers in GPU memory and copy data to/from buffers
  - Application (via graphics driver) provides GPU a single kernel program binary
  - Application tells GPU to run the kernel in an SPMD fashion ("run N instances")
  - Go! (launch(myKernel, N))



CUDA Programming Language

- **Introduced in 2007 with NVIDIA Tesla architecture**
- **"C-like" language to express programs that run on GPUs using the compute-mode hardware interface**
- **Relatively low-level: CUDA's abstractions closely match the capabilities/performance characteristics of modern GPUs (design goal: maintain low abstraction distance)**
- **Note: OpenCL is an open standards version of CUDA**
  - CUDA only runs on NVIDIA GPUs
  - OpenCL runs on CPUs and GPUs from many vendors
  - Almost everything we say about CUDA also holds for OpenCL

Basic CUDA Syntax

- **"Host" code:** serial execution
  Running as part of normal C/C++ application on CPU
- **Bulk launch of many CUDA threads**
  "launch a grid of CUDA thread blocks"
  Call returns when all threads have terminated
- **SPMD execution of device kernel function:**
- "CUDA device" code: kernel function (**`__global__`** denotes a CUDA kernel function) runs on GPU
- Each thread computes its overall grid thread id from its position in its block (**`threadIdx`**) and its block's position in the grid (**`blockIdx`**)

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

```
// kernel definition
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  C[j][i] = A[j][i] + B[j][i];
}
```

Clear Separation of Host and Device Code

- Separation of execution into host and device code is performed statically by the programmer

"Host" code : serial execution on CPU

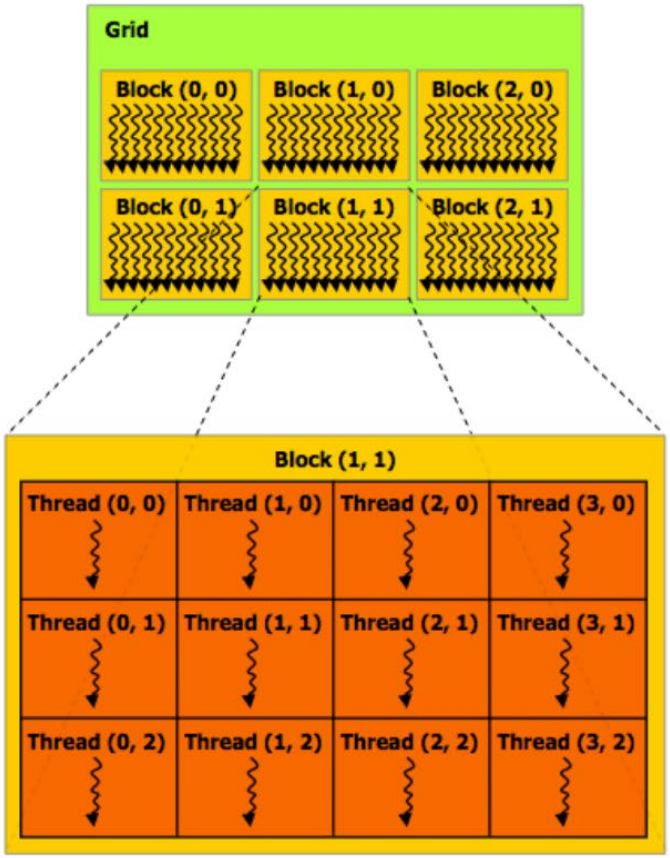
```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

"Device" code (SPMD execution on GPU)

```
__device__ float doubleValue(float x)
{
   return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(float A[Ny][Nx],
                                 float B[Ny][Nx],
                                 float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + doubleValue(B[j][i]);
}
```

Number of SPMD Threads is Explicit in Program

- Number of kernel invocations is not determined by size of data collection (a kernel launch is not map(kernel, collection) as was the case with graphics shader programming)



Regular application thread running on CPU (the "host")

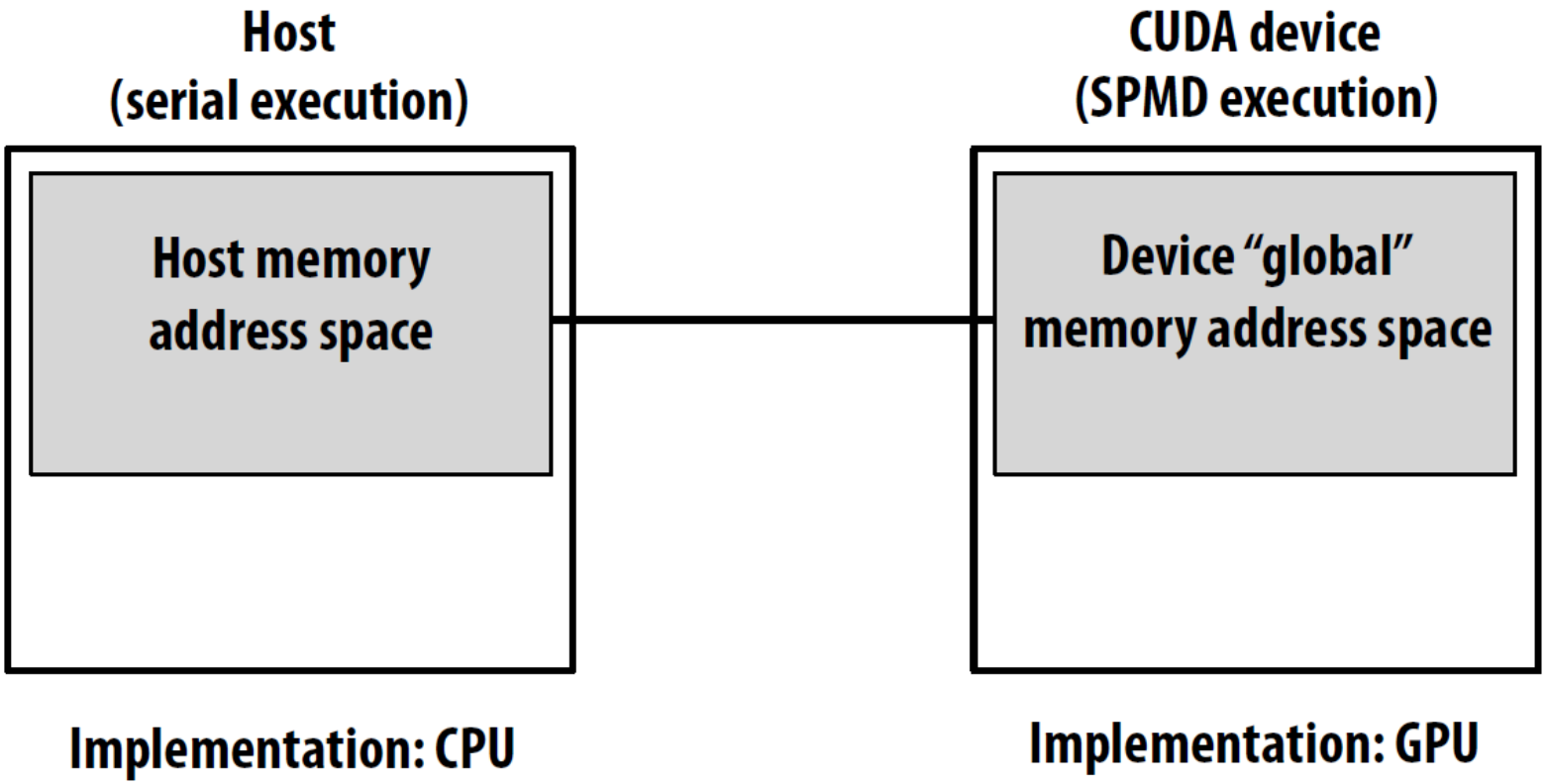
```
const int Nx = 11;  // not a multiple of threadsPerBlock.x
const int Ny = 5;   // not a multiple of threadsPerBlock.y

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks((Nx+threadsPerBlock.x-1)/threadsPerBlock.x,
               (Ny+threadsPerBlock.y-1)/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

CUDA kernel definition

```
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   // guard against out of bounds array access
   if (i < Nx && j < Ny)
      C[j][i] = A[j][i] + B[j][i];
}
```

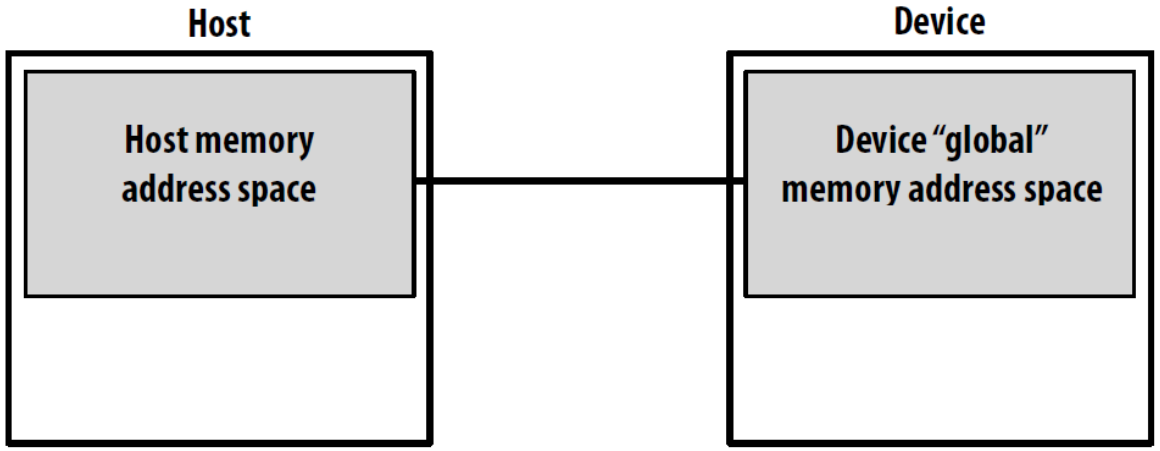
CUDA Memory Model

- Distinct host and device address spaces



memcpy Primitive

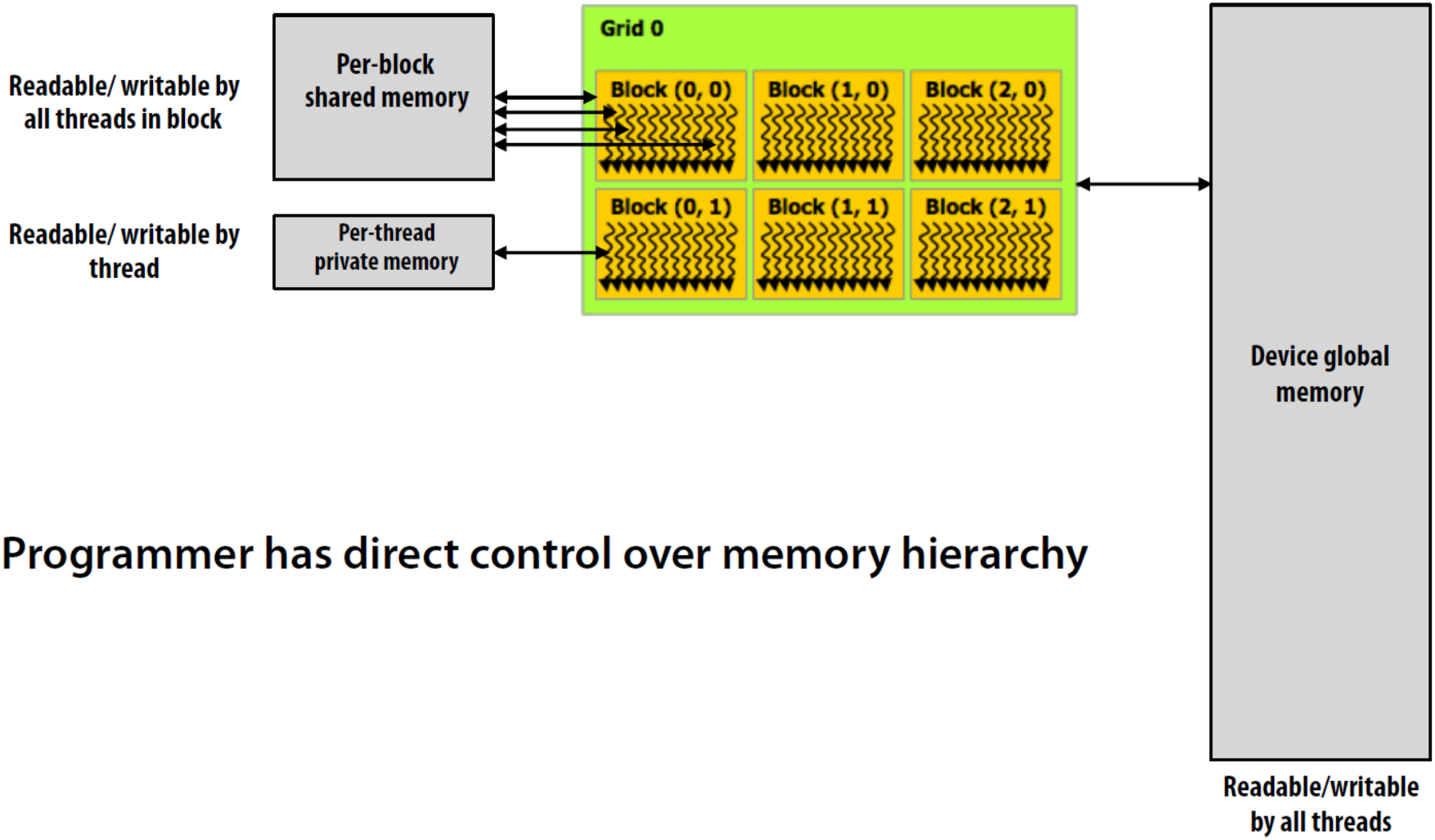
- Move data between address spaces




```
float* A = new float[N];       // allocate buffer in host mem

// populate host address space pointer A
for (int i=0 i<N; i++)
   A[i] = (float)i;

int bytes = sizeof(float) * N
float* deviceA;                 // allocate buffer in
cudaMalloc(&deviceA, bytes);    // device address space

// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);

// note: deviceA[i] is an invalid operation here (cannot
// manipulate contents of deviceA directly from host.
// Only from device code.)
```

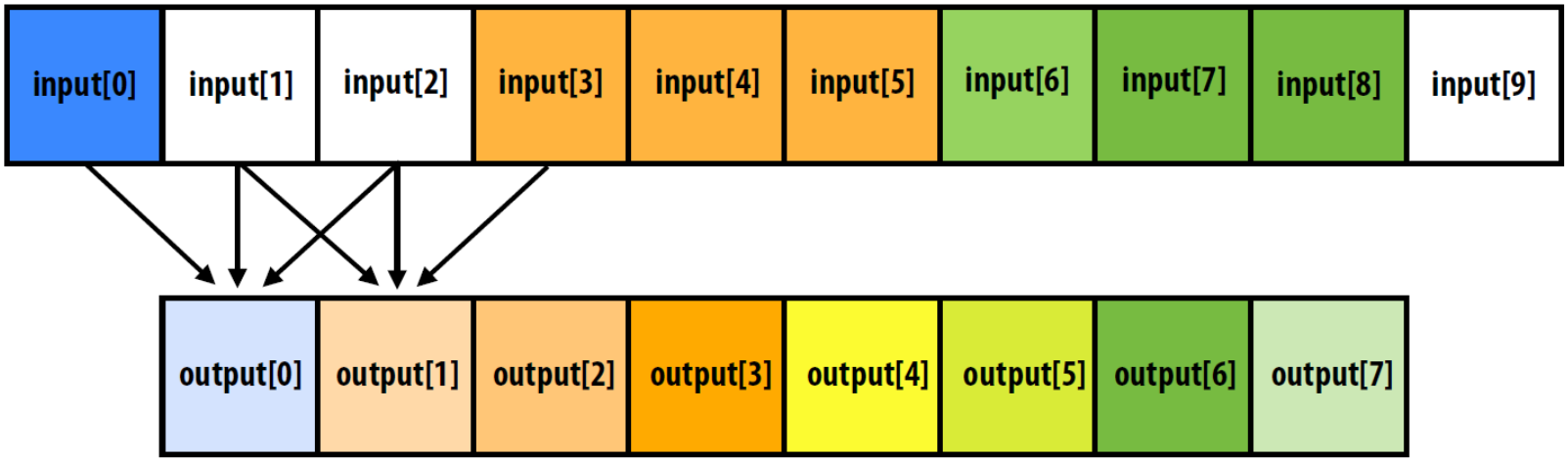
CUDA device Memory Model

- Three distinct types of memory visible to kernels



Programmer has direct control over memory hierarchy

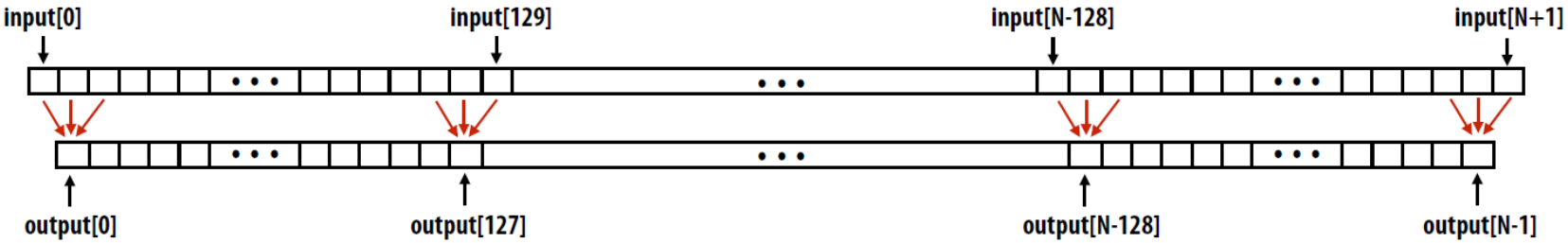
CUDA Example: 1D Convolution




```
output[i] = (input[i] + input[i+1] + input[i+2]) / 3.f;
```

1D Convolution in CUDA

One thread per output element



CUDA Kernel

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

   int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

   float result = 0.0f;  // thread-local variable
   for (int i=0; i<3; i++)
     result += input[index + i];

   output[index] = result / 3.f;
}
```

each thread computes result for one element

write result to global memory

Host code

```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );  // allocate array in device memory
cudaMalloc(&devOutput, sizeof(float) * N);      // allocate array in device memory

// Initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

CUDA Synchronization Constructs

- **`__syncthreads()`**
  Barrier: wait for all threads in the block to arrive at this point

- **Atomic operations**
  e.g., `float atomicAdd(float* addr, float amount)`
  Atomic operations on both global memory and shared memory variables

- **Host/device synchronization**
  Implicit barrier across all threads at return of kernel

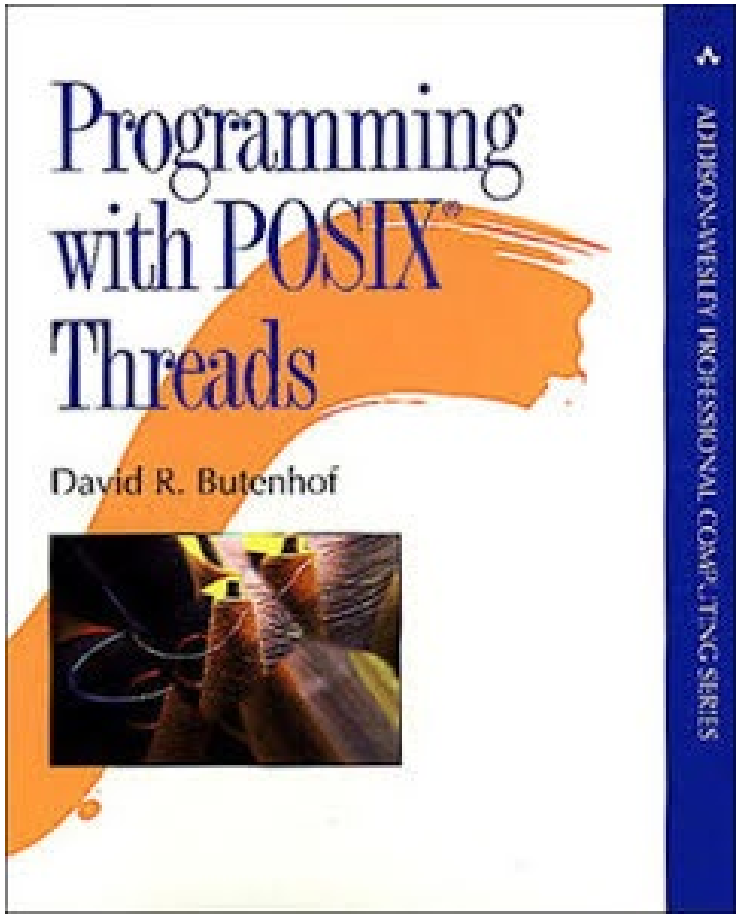
CUDA Abstractions

- **Execution: thread hierarchy**
  - Bulk launch of many threads
  - Two-level hierarchy: threads are grouped into thread blocks
- **Distributed address space**
  - Built-in memcpy primitives to copy between host and device address spaces
  - Three different types of device address spaces
  - Per thread, per block (“shared”), or per program (“global”)
- **Barrier synchronization primitive for threads in thread block**
- **Atomic primitives for additional synchronization**
  shared and global variables

Outline

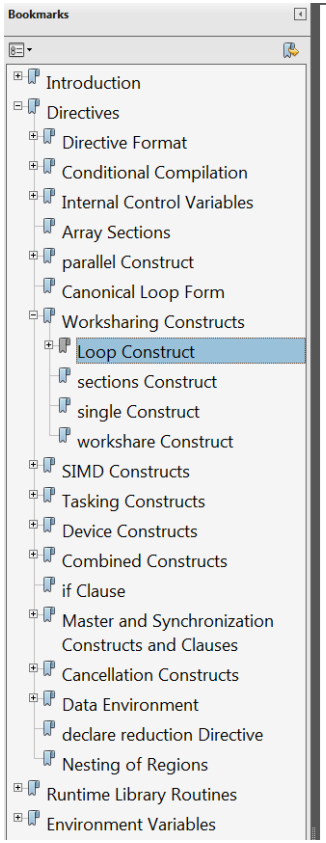
- Example: Solving a Linear System of Equations
- Parallel machine models
- Algorithmic approaches
- Amdahl, strong and weak scaling
- CUDA
- **MPI, OpenMP, OpenACC**

PThreads




```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
  pthread_t thread1, thread2;
  char *message1 = "Thread 1";
  char *message2 = "Thread 2";
  int iret1, iret2;

  iret1 = pthread_create(&thread1, NULL, print_message_function,
    (void*) message1);
  iret2 = pthread_create( &thread2, NULL, print_message_function,
    (void*) message2);

  pthread_join( thread1, NULL);
  pthread_join( thread2, NULL);

  printf("Thread 1 returns: %d\n",iret1);
  printf("Thread 2 returns: %d\n",iret2);
  exit(0);
}

void *print_message_function( void *ptr )
{
  char *message;
  message = (char *) ptr;
  printf("%s \n", message);
}
```

OpenMP



Syntax

C / C++

The syntax of the loop construct is as follows:

```
#pragma omp for [clause[ [,] clause] ... ] new-line
    for-loops
```

where clause is one of the following:

private(*list*)

firstprivate(*list*)

lastprivate(*list*)

linear(*list*[: *linear-step*])

reduction(*reduction-identifier* : *list*)

schedule([*modifier* [, *modifier*]:]*kind*[, *chunk_size*])

collapse(*n*)

ordered[(*n*)]

nowait

```
void conv_openmp(int n, float *a, float *b) {
  int i;
#pragma omp parallel for
  for (i=1; i<n-1; i++) /* i is private by default */
    b[i] = (a[i-1] + a[i] + a[i+1]) / 3.0;
}
```

More OpenMP

```
#include <stdio.h>
#include <omp.h>

int main() {
  int x;
  x = 2;
#pragma omp parallel num_threads(2) shared(x)
  {
     if (omp_get_thread_num() == 0) {
      x = 5;
    } else {
      /* Print 1: the following read of x has a race */
      printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x );
    }
#pragma omp barrier
    if (omp_get_thread_num() == 0) {
      /* Print 2 */
      printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
    } else {
      /* Print 3 */
      printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
    }
  }
  return 0;
}
```

OpenCL

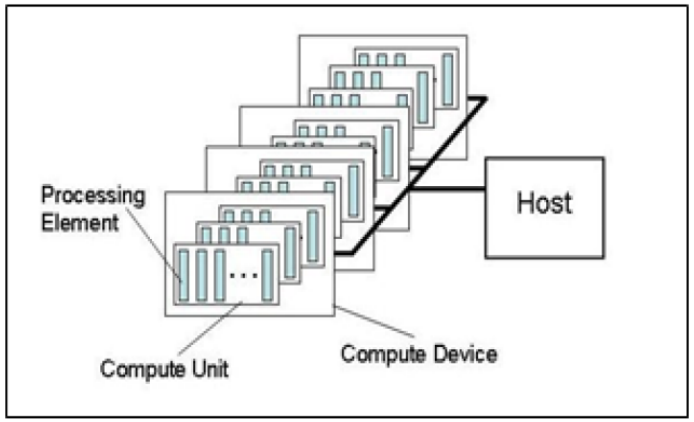


Figure 3.1: *Platform model . . . one host plus one or more compute devices each with one or more compute units composed of one or more processing elements.*

Programmers provide programs in the form of SPIR-V source binaries, OpenCL C or OpenCL C++ source strings or implementation-defined binary objects. The OpenCL platform provides a compiler to translate program input of either form into executable program objects. The device code compiler may be *online* or *offline*. An *online compiler* is available during host program execution using standard APIs. An *offline compiler* is invoked outside of host program control, using platform-specific methods. The OpenCL runtime allows developers to get a previously compiled device program executable and be able to load and execute a previously compiled device program executable.

OpenCL defines two kinds of platform profiles: a *full profile* and a reduced-functionality *embedded profile*. A full profile platform must provide an online compiler for all its devices. An embedded platform may provide an online compiler, but is not required to do so.

A device may expose special purpose functionality as a *built-in function*. The platform provides APIs for enumerating and invoking the built-in functions offered by a device, but otherwise does not define their construction or semantics. A *custom device* supports only built-in functions, and cannot be programmed via a kernel language.

All device types support the OpenCL execution model, the OpenCL memory model, and the APIs used in OpenCL to

OpenACC

General Syntax

C/C++

#pragma acc *directive [clause [[,] clause]...] new-line*

FORTRAN

!$acc *directive [clause [[,] clause]...]*

An OpenACC construct is an OpenACC directive and, if applicable, the immediately following statement, loop or structured block.Compute Construct

A compute construct is a **parallel**, **kernels**, or **serial** construct.

Parallel Construct

A **parallel** construct launches a number of gangs executing in parallel, where each gang may support multiple workers, each with vector or SIMD operations.

C/C++

#pragma acc parallel *[clause [[,] clause]…] new-line*
{ structured block }

FORTRAN

!$acc parallel *[clause [[,] clause]…]*
structured block
!$acc end parallel

https://www.openacc.org

Kernels Construct

A **kernels** construct surrounds loops to be executed on the device, typically as a sequence of kernel operations.

C/C++

#pragma acc kernels *[clause [[,] clause]…] new-line*
{ structured block }

FORTRAN

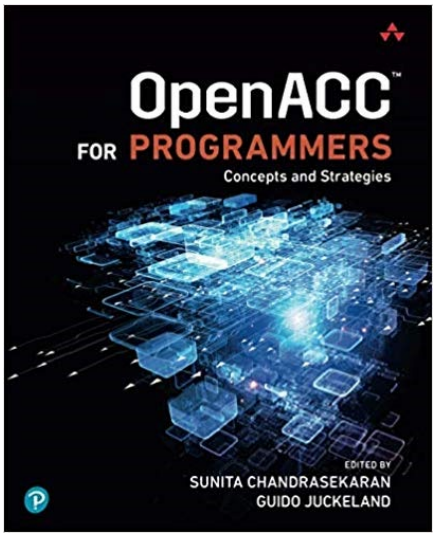
!$acc kernels *[clause [[,] clause]…]*
structured block
!$acc end kernels

CLAUSES

if(*condition* **)**
default(none)
default(present)
device_type or **dtype(** *[* | device-type-list]* **)**
async *[(expression)]*
wait *[(expression-list)]*
num_gangs(*expression* **)**
num_workers(*expression* **)**
vector_length(*expression* **)**

See Compute Construct Clauses.

copy(*list* **)**
copyin(*list* **)**
copyout(*list* **)**
create(*list* **)**
no_create(*list* **)**
present(*list* **)**
deviceptr(*list* **)**
attach(*list* **)**



OpenACC Example

```
#pragma acc data copy(A) create(Anew)
while ( error > tol  &&  iter  <  iter_max )  {
  error = 0.0;
#pragma acc kernels {
#pragma acc loop independent collapse(2)
  for (  int  j = 1; j < n-1;  j++ )  {
    for (  int  i = 1; i < m-1; i++ )  {
      Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                              A [j-1] [i] + A [j+1] [i]);
      error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
    }
  }
}
```

Thread Building Blocks

intel Developer Zone

Join Today > Log in

Development > Tools > Resources >

Search our content library...

Search document...

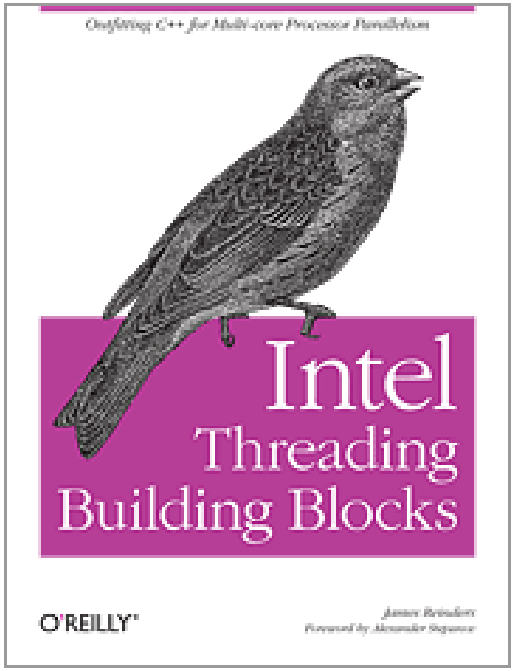
Intel® Threading Building Blocks Documentation

- Home
- Legal Information
- Getting Help and Support
- Introducing the Intel® Threading Building Blocks (Intel® TBB)
- Intel TBB Benefits
- Notational Conventions
- Intel® Threading Building Blocks (Intel®TBB) Developer Guide
- **Intel® Threading Building Blocks Developer Reference**
  - General Conventions

Intel® Threading Building Blocks Developer Reference

Parent topic: Intel® Threading Building Blocks

- General Conventions
- Environment
- Algorithms
- Containers Overview
- Flow Graph
- Thread Local Storage
- Memory Allocation
- Synchronization
- Timing
- Task Groups
- Task Scheduler
- Exceptions
- Threads
- Appendices

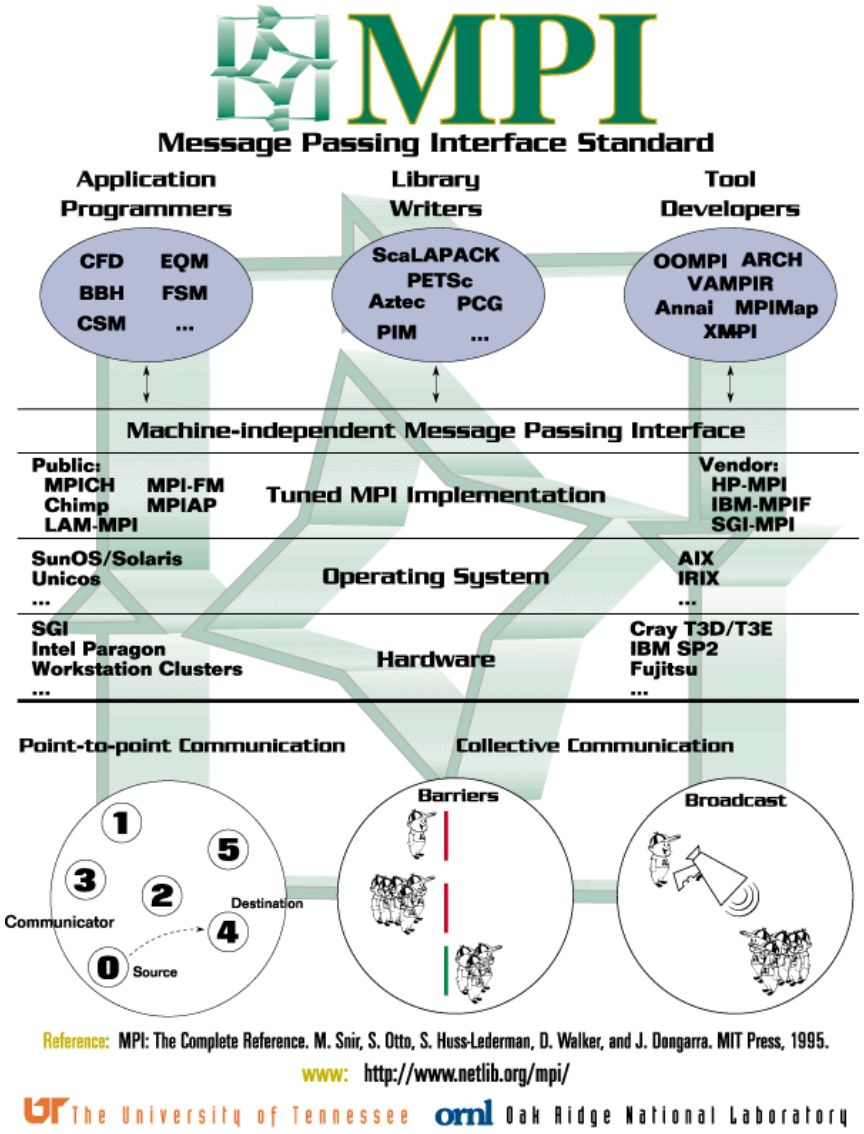


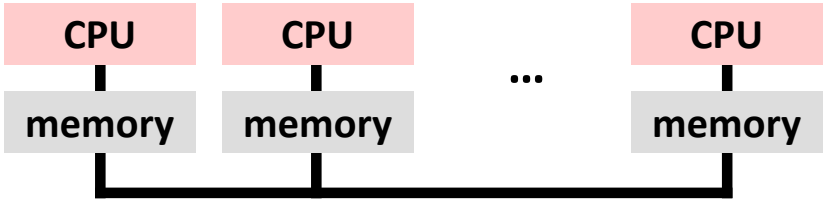
Prev
Task Isolation

Next
General Conventions

For more complete information about compiler optimizations, see our Optimization Notice.

MPI





PSC HPC

11k cores

200 GPUs

21.35 Pflop/s

MPI

```
#include <stdio.h>
#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;
    for (i=my_id*my_steps; i<(my_id+1)*my_steps ; ++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);

    if (my_id==0) {
        printf("pi = %f\n", pi);
    }
}
```

Summary

- Example: Solving a Linear System of Equations
- Parallel machine models
- Algorithmic approaches
- Amdahl, strong and weak scaling
- CUDA
- MPI, OpenMP, OpenACC

18-847G

Special Topics in Computer Systems:

Computational Problem Solving for Engineers

Franz Franchetti
Instructor

TBD
Teaching Assistants

This is Section G. Other sections (F, RW, SH) are different courses.