Parallel Programming Paradigms

A Long History

- IVTRAN (Parallel Fortran) language for the ILLIAC IV (1966-1970)
- Several other Fortran language based programming languages followed (Fortran D, KAP, Vienna Fortran, Paraphrase, Polaris etc. etc.)
- Experimental new approaches: Linda, Irvine Dataflow (Id), Decoupled Access Execute
- Vector Languages: Cray Fortran, FX/Fortran

Most Commonly Used

- MPI: Message Passing Interface
  - ARPA, NSF, Esprit
- Pthreads: POSIX Threads Linux Standard
  - **P**ortable **O**perating-**S**ystem **I**nterface (IEEE, the Open Group)
- OpenMP: Open Multi-Processing
  - AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, and more.
- CUDA: Compute Unified Device Architecture
  - Nvidia

MPI

- Communication between processes in a distributed program is typically implemented using MPI: Message Passing Interface.
- MPI is a generic API that can be implemented in different ways:
  - Using specific interconnect hardware, such as InfiniBand.
  - Using TCP/IP over plain Ethernet.
  - Or even used (emulated) on Shared Memory for inter process communication on the same node.

Some MPI basic functions

- `#include <mpi.h>`
- Initialize library:
```
MPI_Init(&argc, &argv);
```
- Determine number of processes that take part:
```
int n_procs;
MPI_Comm_size(MPI_COMM_WORLD,
  &n_procs);
```
  (MPI_COMM_WORLD is the initially defined universe intracommunicator for all processes)
- Determine ID of this process:
```
int id;
MPI_Comm_rank(MPI_COMM_WORLD, &id);
```

Sending Messages

```
MPI_Send(buffer,count,datatype,dest,tag,comm);
```

- buffer: pointer to data buffer.
- count: number of items to send.
- datatype: data type of the items (see next slide).
  - All items must be of the same type.
- dest: rank number of destination.
- tag: message tag (integer), may be 0.
  - You can use this to distinguish between different messages.
- comm: communicator, for instance MPI_COMM_WORLD.

- Note: this is a blocking send!

MPI data types

- You must specify a data type when performing MPI transmissions.
- For instance for built-in C types:
  - "int" translates to MPI_INT
  - "unsigned int" to MPI_UNSIGNED
  - "double" to MPI_DOUBLE, and so on.
- You can define your own MPI data types, for example if you want to send/receive custom structures.

Other calls

- `MPI_Recv()`
- `MPI_Isend()`, `MPI_Irecv()`
  - Non-blocking send/receive
- `MPI_Scatter()`, `MPI_Gather()`
- `MPI_Bcast()`
- `MPI_Reduce()`

Shutting down

- `MPI_Finalize()`

Pthreads

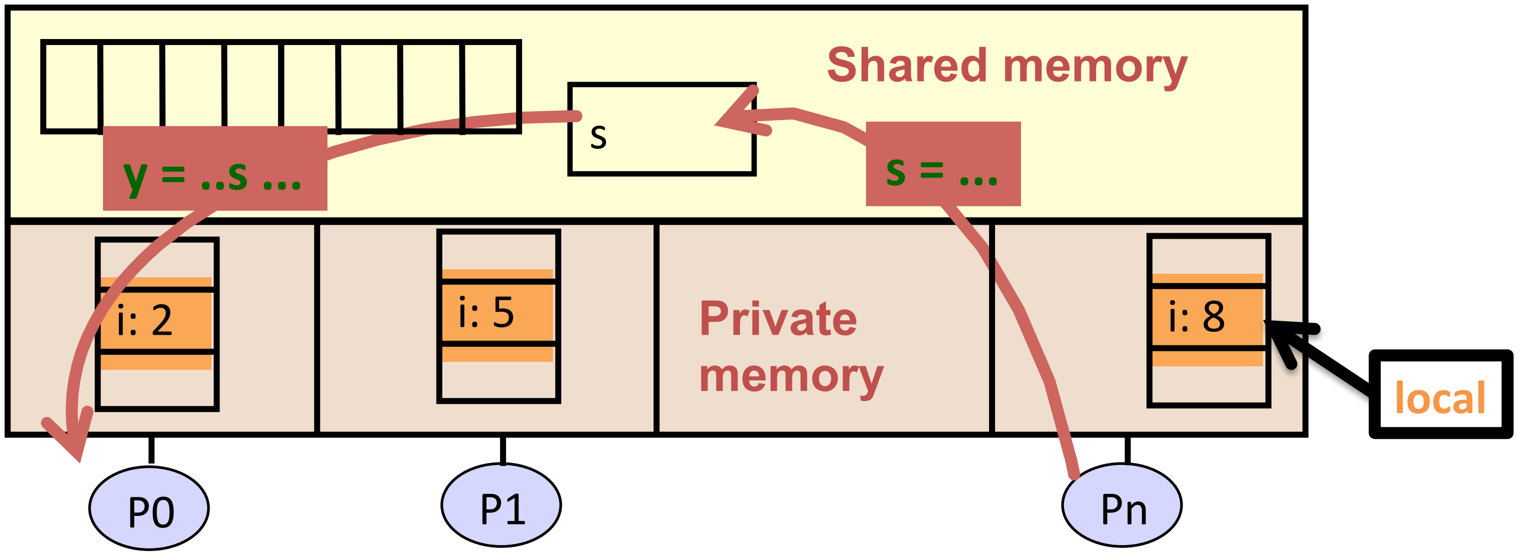
Pthreads defines a set of C programming language types, functions and constants. It is implemented with a pthread.h header and a thread library.

There are around 100 Pthreads procedures, all prefixed "pthread_" and they can be categorized into four groups:

- Thread management - creating, joining threads etc.
- Mutexes
- Condition variables
- Synchronization between threads using read/write locks and barriers

The POSIX semaphore API works with POSIX threads but is not part of threads standard, having been defined in the *POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993)* standard. Consequently the semaphore procedures are prefixed by "sem_" instead of "pthread_".

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables



Pthreads Supports

- Creating parallelism
- Synchronizing

No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

“Forking” Threads

Signature:

```
int pthread_create(pthread_t *thread_id,
                   const pthread_attr_t *thread_attribute,
                   void * (*thread_fun)(void *),
                   void *funarg);
```

Example call:

```
errcode = pthread_create(&thread_id, &thread_attribute,
                         thread_fun, &fun_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attribute: minimum stack size
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

Example

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello,
  NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(&threads[tn], NULL);
  }
  return 0;
}
```

Some More Functions

- **`pthread_yield();`**
  - Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.
- **`pthread_exit(void *value);`**
  - Exit thread and pass value to joining thread (if exists)
- **`pthread_join(pthread_t *thread, void **result);`**
  - Wait for specified thread to finish. Place exit value into *result.

Others:

- **`pthread_t me; me = pthread_self();`**
  - Allows a pthread to obtain its own identifier pthread_t thread;
- **`pthread_detach(thread);`**
  - Informs the library that the threads exit status will not be needed by subsequent pthread_join calls resulting in better threads performance. For more information consult the library or the man pages, e.g., man -k pthread..

Shared Data and Threads

- Variables declared outside of main are shared
- Object allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems

- Often done by creating a large "thread data" struct
  - Passed into all threads as argument
  - Simple example:

```
char *message = "Hello World!\n";

pthread_create(&thread1,
               NULL,
               print_fun,
               (void*) message);
```

Basic Types of Synchronization: Barrier

- Especially common when running multiple copies of the same function in parallel
  - SPMD "Single Program Multiple Data"
- simple use of barriers -- all threads hit the same one

```
work_on_my_subgrid();
barrier;
read_neighboring_values();
barrier;
```

- more complicated -- barriers on branches (or loops)

```
if (tid % 2 == 0) {
  work1();
  barrier
} else { barrier }
```

- barriers are not provided in all thread libraries

Creating and Initializing a Barrier

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```
pthread_barrier_t b;
pthread_barrier_init(&b,NULL,3);
```

- The second argument specifies an attribute object for finer control; using NULL yields the default attributes.

- To wait at a barrier, a process executes:

```
pthread_barrier_wait(&b);
```

Basic Types of Synchronization: Mutexes

- Threads are working mostly independently
- There is a need to access common data structure

```
lock *l = alloc_and_init();    /* shared */
acquire(l);
 access data
release(l);
```

- Locks only affect processors using them:
  - If a thread accesses the data without doing the acquire/ release, locks by others will not help
- Semaphores generalize locks to allow the use of the same locks across different processes

Mutexes in POSIX Threads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex =
PTHREAD_MUTEX_INITIALIZER;
   // or pthread_mutex_init(&amutex, NULL);
```

- To use it:

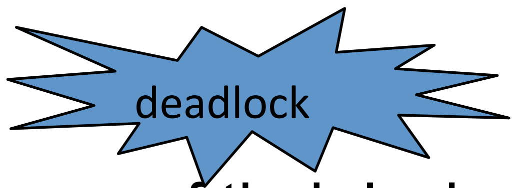
```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Multiple mutexes may be held, but can lead to problems:

```
      thread1            thread2
      lock(a)            lock(b)
      lock(b)            lock(a)
```



- Deadlock results if both threads acquire one of their locks, so that neither can acquire the second

Summary of Programming with Threads

- POSIX Threads are based on OS features
  - Can be used from multiple languages (need appropriate header)
  - Familiar language for most of program
  - Ability to shared data is convenient

- OpenMP is commonly used today as an alternative

Introduction to OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - "Standard" API for defining multi-threaded shared-memory programs
  - openmp.org – Talks, examples, forums, etc.

- High-level API
  - Preprocessor (compiler) directives (~ 80%)
  - Library Calls (~ 19%)
  - Environment Variables (~ 1%)

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Exact behavior depends on OpenMP *implementation*!
  - Requires compiler support (C or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions*, rather than T concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

Programming Model – Concurrent Loops

- OpenMP easily parallelizes loops
  - Requires: No data dependencies (reads/write or write/write pairs) between iterations!

- Preprocessor calculates loop bounds for each thread directly from *serial* source

```
#pragma omp parallel for

for( i=0; i < 25; i++ )
{
  printf("Foo");
}
```

Programming Model – Loop Scheduling

- `Schedule` Clause determines how loop iterations are divided among the thread team
  - **`static([chunk])`** divides iterations statically between threads
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default **`[chunk]`** is **`ceil( # iterations / # threads )`**
  - **`dynamic([chunk])`** allocates **`[chunk]`** iterations per thread, allocating an additional **`[chunk]`** iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default **`[chunk]`** is 1
  - **`guided([chunk])`** allocates dynamically, but **`[chunk]`** is exponentially reduced with each allocation

Data Sharing

PThreads:

- Global-scoped variables are shared
- Stack-allocated variables are private

OpenMP:

- shared variables are shared
- private variables are private

OpenMP Synchronization

- OpenMP Critical Sections
  - Named or unnamed
  - No *explicit* locks / mutexes
- Barrier directives
- Single-thread regions *within* parallel regions
  - **`master, single`** directives

CUDA NVIDIA

Programming Approaches

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Apps	Maximum Flexibility

Development Environment

Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

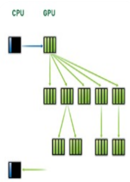
Open Compiler Tool Chain

Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

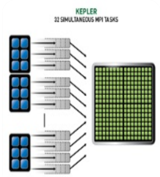
Hardware Capabilities

SMX

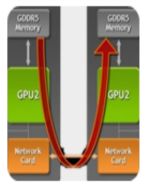
Dynamic Parallelism



HyperQ



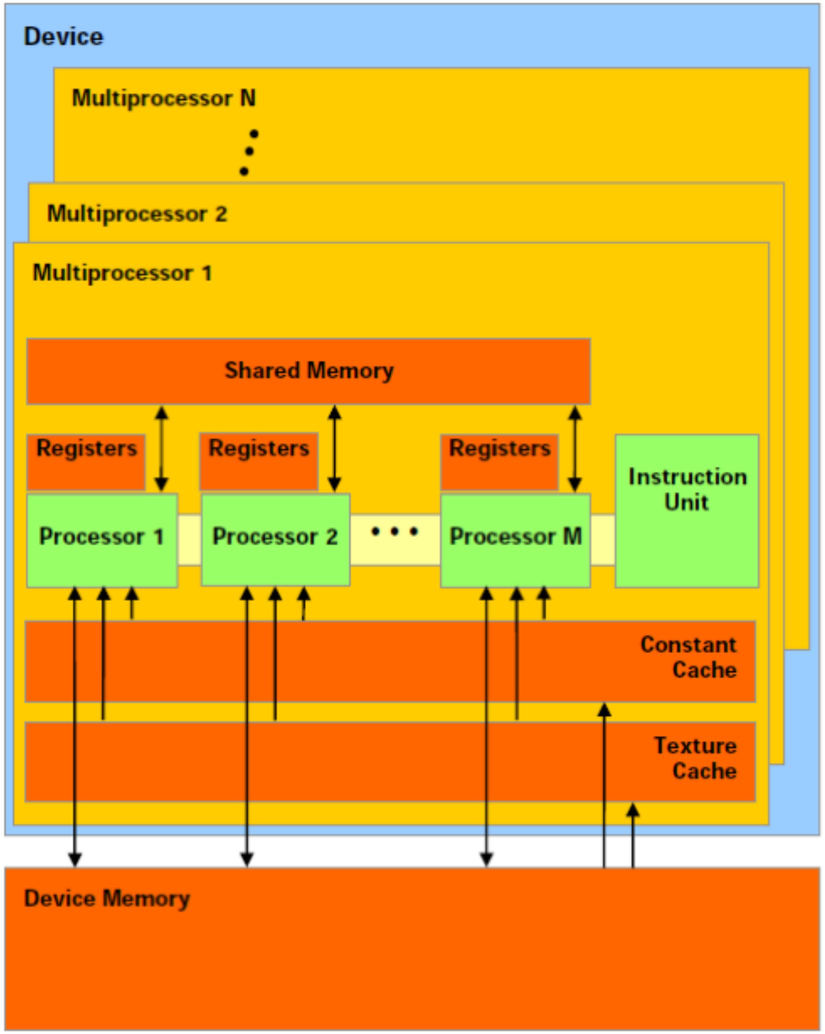
GPUDirect



NVIDIA GPU Platform

- A scalable array of multithreaded Streaming Multiprocessors (SMs), each SM consists of
  - 8 Scalar Processor (SP) cores
  - 2 special function units for transcendentals
  - A multithreaded instruction unit
  - On-chip shared memory
- GDDR3 SDRAM *
- PCIe interface

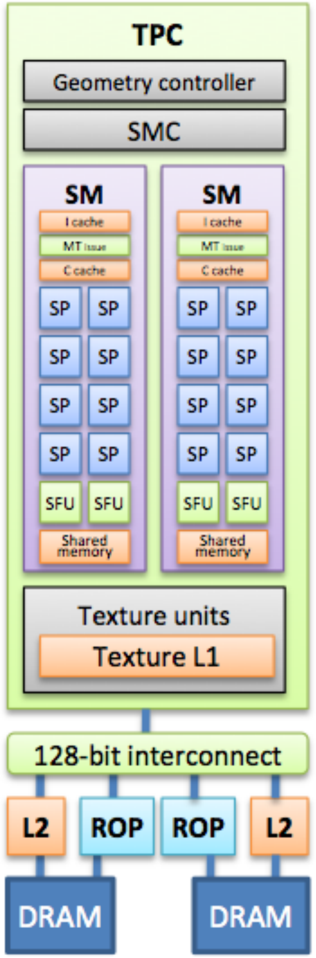
  Peripheral Component Interconnect Express



*** Graphics Double Data Rate Synchronous Dynamic Random Access Memory (DDR3 vs DDR2: larger prefetch buffer, ie 8 bits instead of 2 bits)**

Sample Platforms

NVIDIA GeForce9400M G GPU

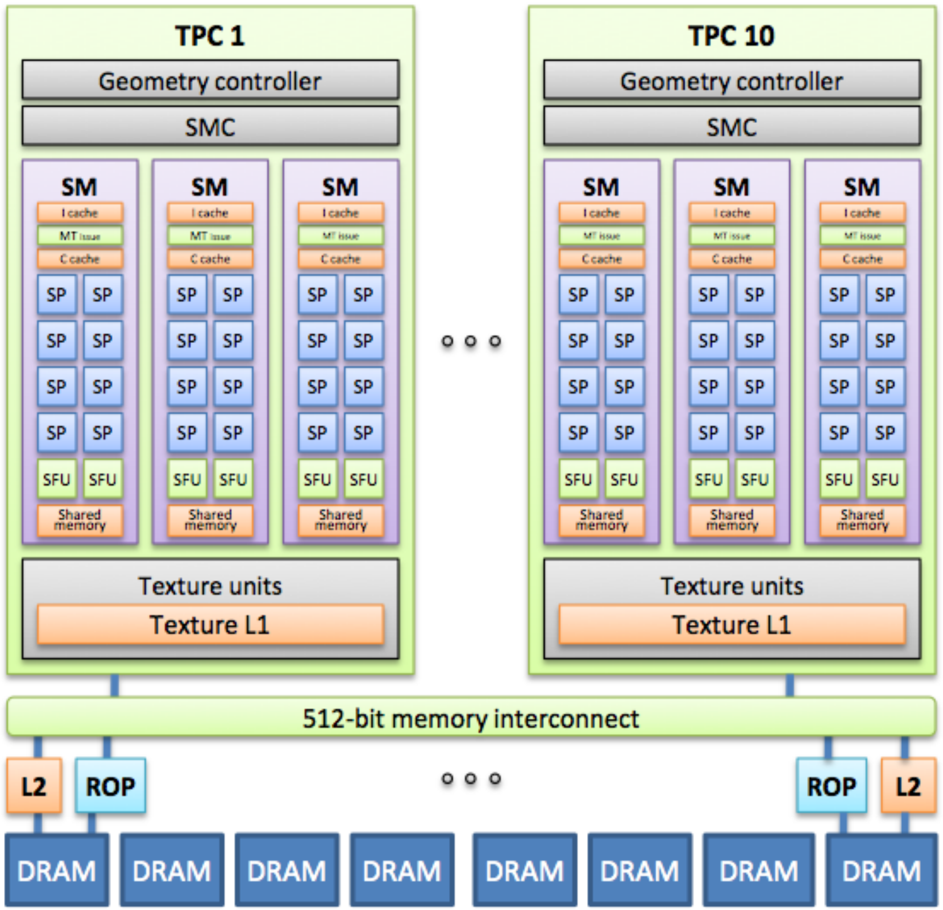


Render Output Unit (ROP)

- 16 streaming processors arranged as 2 streaming multiprocessors
- At 0.8 GHz this provides
  - 54 GFLOPS in single-precision (SP)
- 128-bit interface to off-chip GDDR3 memory
  - 21 GB/s bandwidth

Sample Platforms

NVIDIA Tesla C1060 GPU



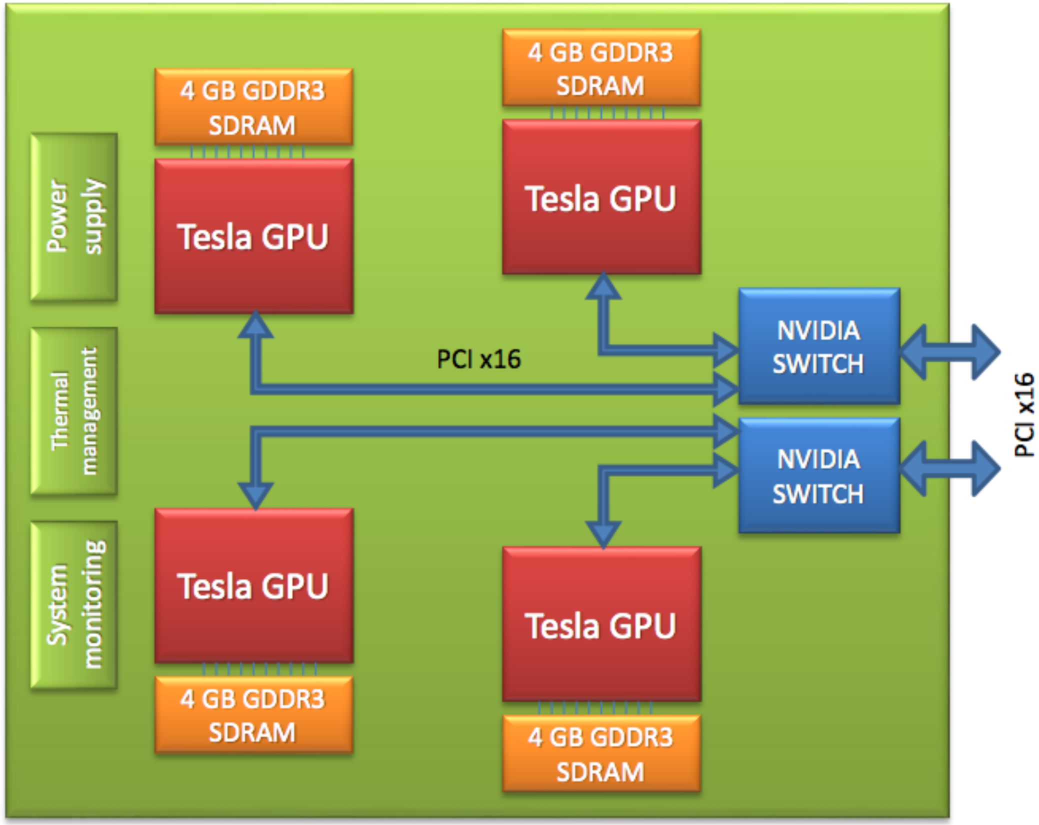
- 240 streaming processors arranged as 30 streaming multiprocessors
- At 1.3 GHz this provides
  - 1 TFLOPS SP
  - 86.4 GFLOPS DP
- 512-bit interface to off-chip GDDR3 memory
  - 102 GB/s bandwidth

Distributed over 10 Texture Processor Clusters

Sample Platforms

NVIDIA Tesla S1070 Computing Server

- 4 T10 GPUs



How to program GPU's

Let's take Vector Addition written in C for a CPU:

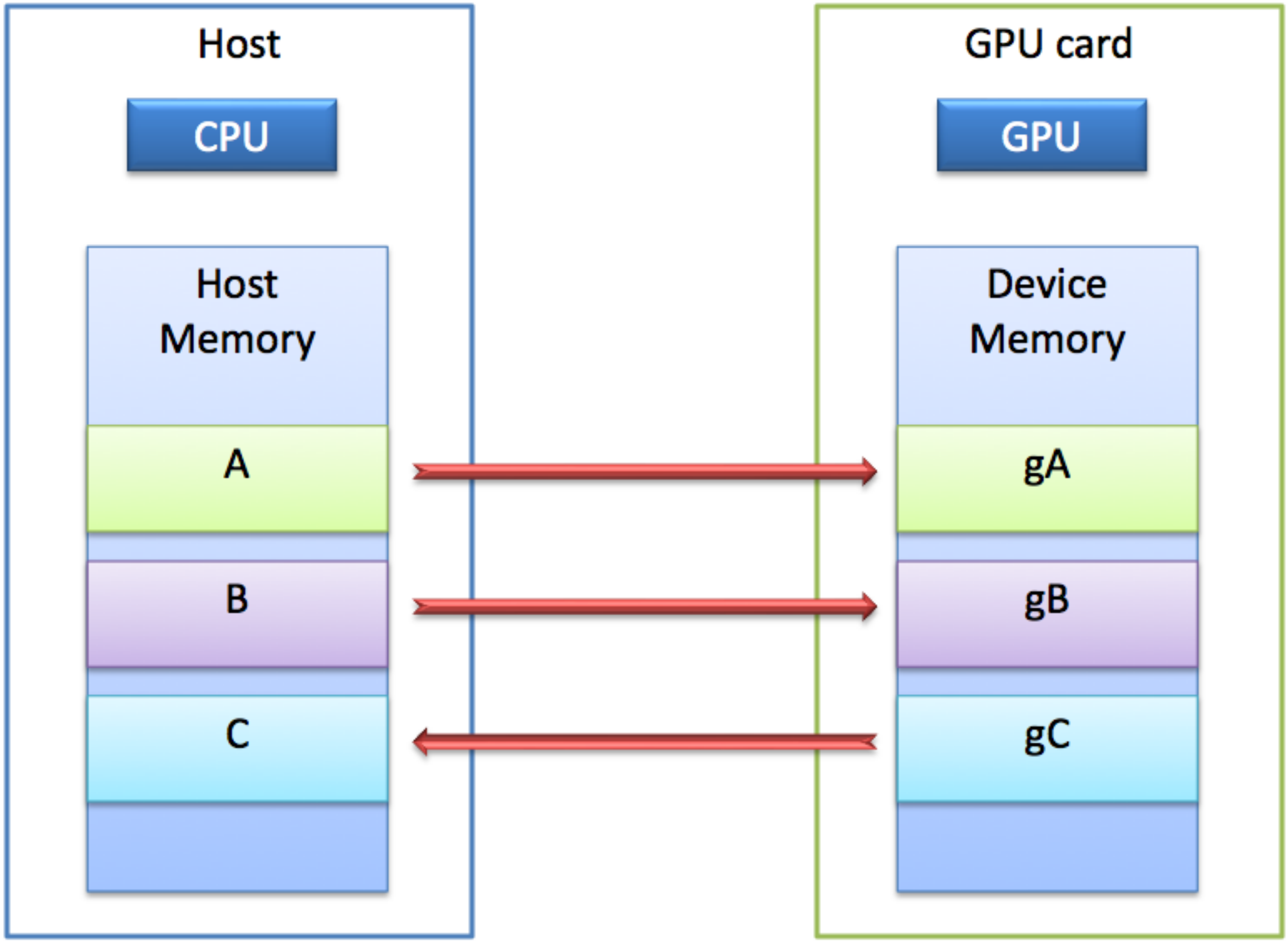
```
void vecAdd(int N, float* A, float* B, float* C) {
   for (int i = 0; i < N; i++)  C[i] = A[i] + B[i];
}
```

← Computational kernel

```
int  main(int argc, char **argv)
{
   int N = 16384;  // default vector size

   float *A = (float*)malloc(N * sizeof(float));
   float *B = (float*)malloc(N * sizeof(float));
   float *C = (float*)malloc(N * sizeof(float));

   vecAdd(N, A, B, C);  // call compute kernel

   free(A); free(B); free(C);
}
```

- Memory allocation: the three malloc lines
- Kernel invocation: vecAdd(N, A, B, C);
- Memory de-allocation: free(A); free(B); free(C);

How to get the GPU involved



Memory Spaces

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
- Host (CPU) manages device (GPU) memory
  - cudaMalloc(void** pointer, size_t nbytes)
  - cudaFree(void* pointer)
  - cudaMemcpy(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind direction);
    - returns after the copy is complete
    - blocks CPU thread until all bytes have been copied
    - does not start copying until previous CUDA calls complete
  - enum cudaMemcpyKind
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice

Example

```
int  main(int argc, char **argv)
{
   int  N = 16384;  // default vector size

   float *A = (float*)malloc(N * sizeof(float));
   float *B = (float*)malloc(N * sizeof(float));
   float *C = (float*)malloc(N * sizeof(float));

   float *devPtrA, *devPtrB, *devPtrC;

   cudaMalloc((void**)&devPtrA, N * sizeof(float));
   cudaMalloc((void**)&devPtrB, N * sizeof(float));
   cudaMalloc((void**)&devPtrC, N * sizeof(float));

   cudaMemcpy(devPtrA, A, N * sizeof(float),  cudaMemcpyHostToDevice);
   cudaMemcpy(devPtrB, B, N * sizeof(float),  cudaMemcpyHostToDevice);
```

Memory allocation on the GPU card

Copy data from the CPU (host) memory to the GPU (device) memory

Example continued

```
vecAdd<<<N/512, 512>>>(devPtrA, devPtrB, devPtrC);

cudaMemcpy(C, devPtrC, N * sizeof(float),  cudaMemcpyDeviceToHost);

cudaFree(devPtrA);
cudaFree(devPtrB);
cudaFree(devPtrC);

  free(A);
  free(B);
  free(C);
}
```

Kernel invocation

Copy results from device memory to the host memory

Device memory de-allocation

Example continued: VecAdd

- **CPU version**

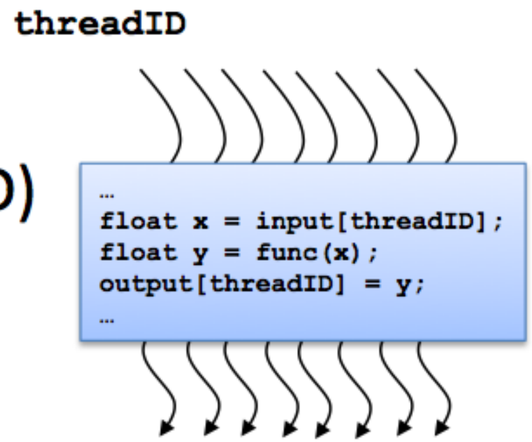
```
void vecAdd(int N, float* A, float* B, float* C)
{
   for (int i = 0; i < N; i++)
      C[i] = A[i] + B[i];
}
```

- **GPU version**

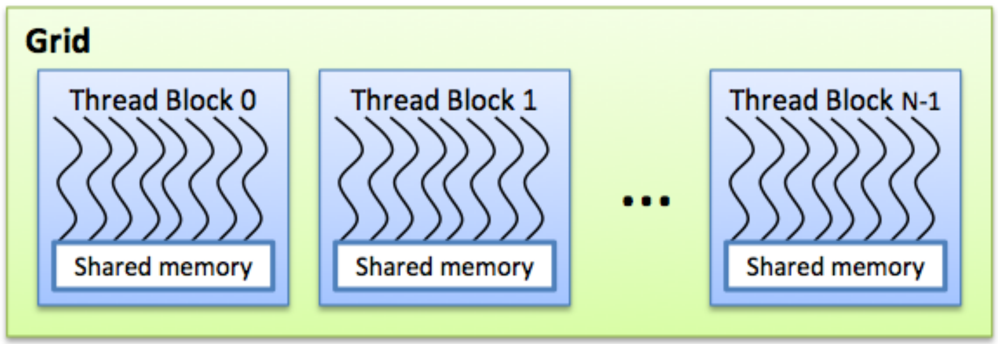
```
__global__  void vecAdd(float* A, float* B, float* C)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   C[i] = A[i] + B[i];
}
```

Example continued: Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



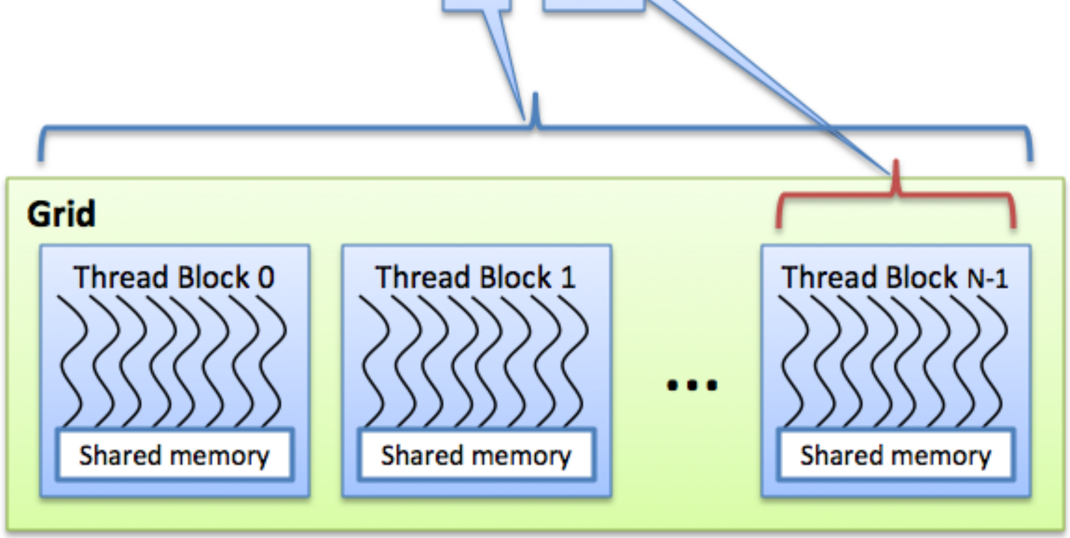
- Threads are arranged as a grid of thread blocks
  - Threads within a block have access to a segment of shared memory



Example continued: Kernel Invocation

grid & thread block dimensionality

```
vecAdd<<<32, 512>>>(devPtrA, devPtrB, devPtrC);
```




```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

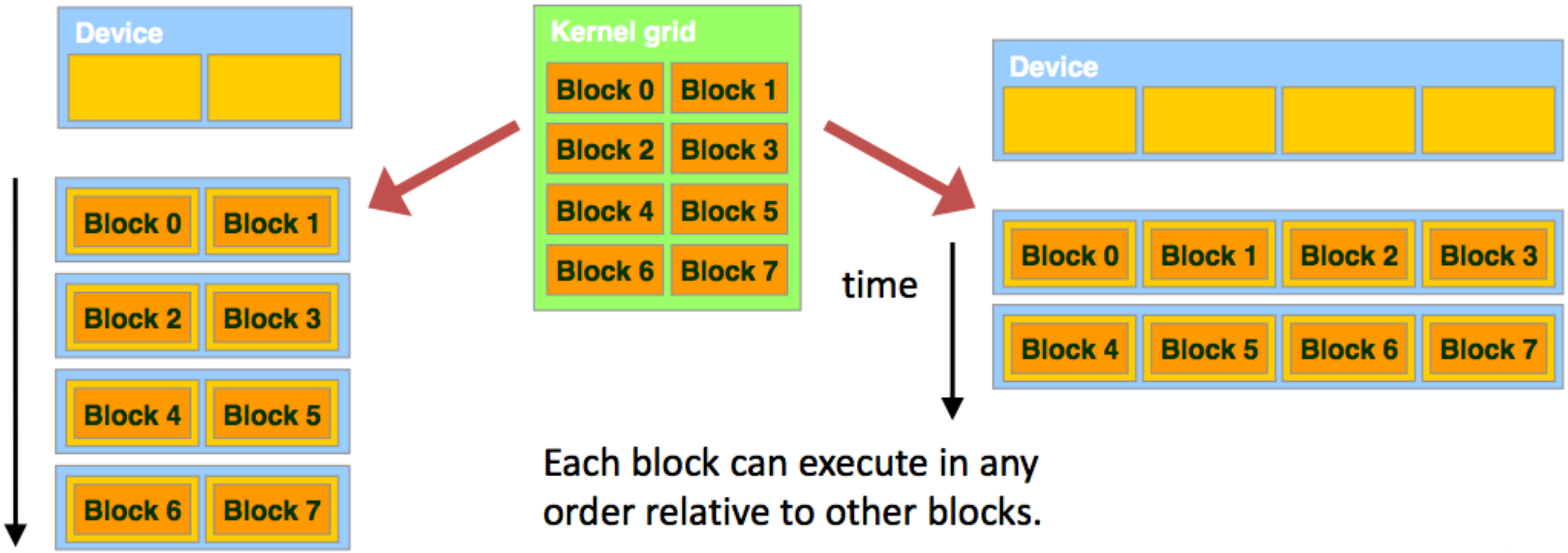
block ID within a grid

number of threads per block

thread ID within a thread block

Mapping Threads to the Hardware

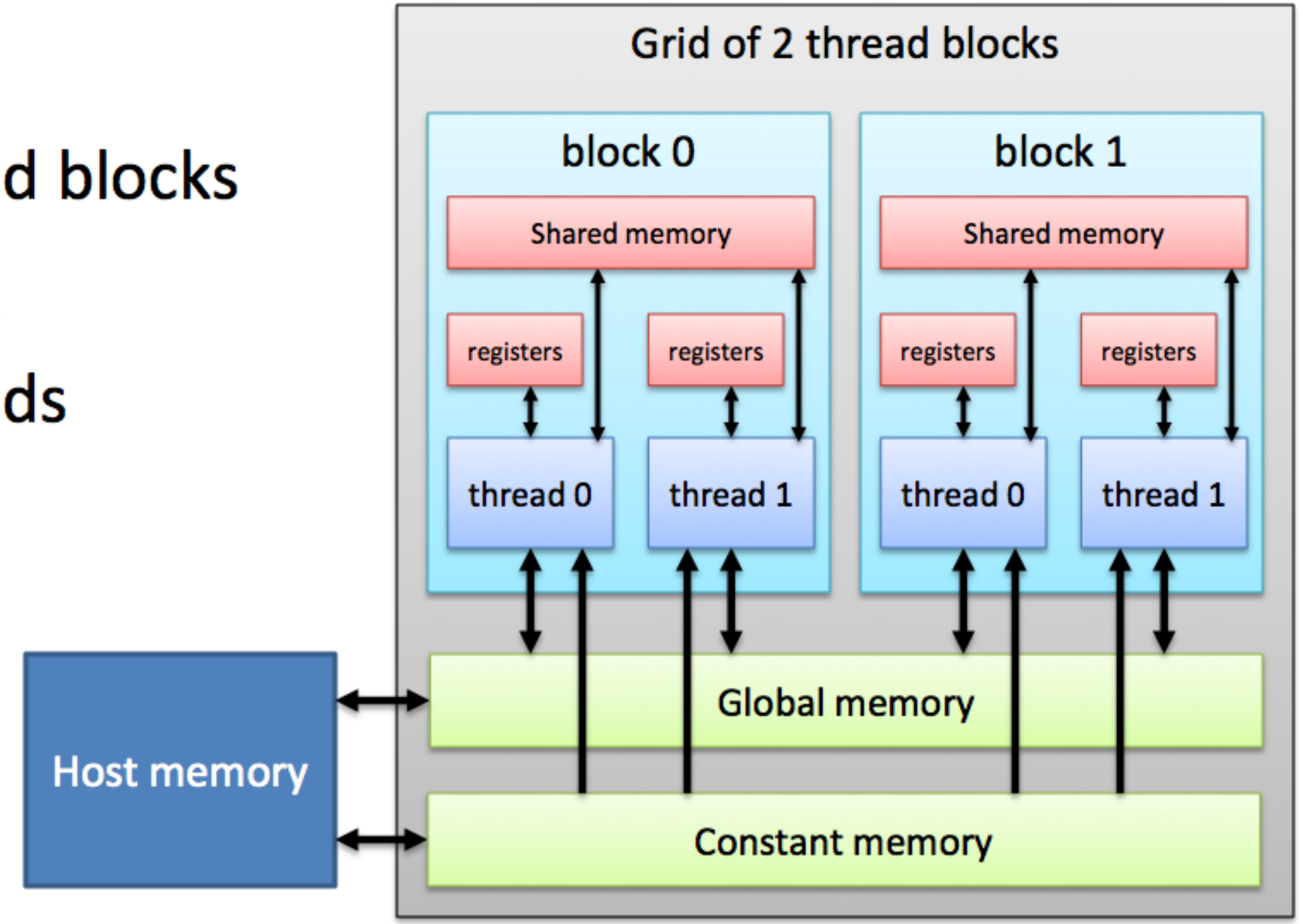
- Blocks of threads are transparently assigned to SMs
  - A block of threads executes on one SM & does not migrate
  - Several blocks can reside concurrently on one SM
- Blocks must be independent
  - Any possible interleaving of blocks should be valid
  - Blocks may coordinate but not synchronize
  - Thread blocks can run in any order



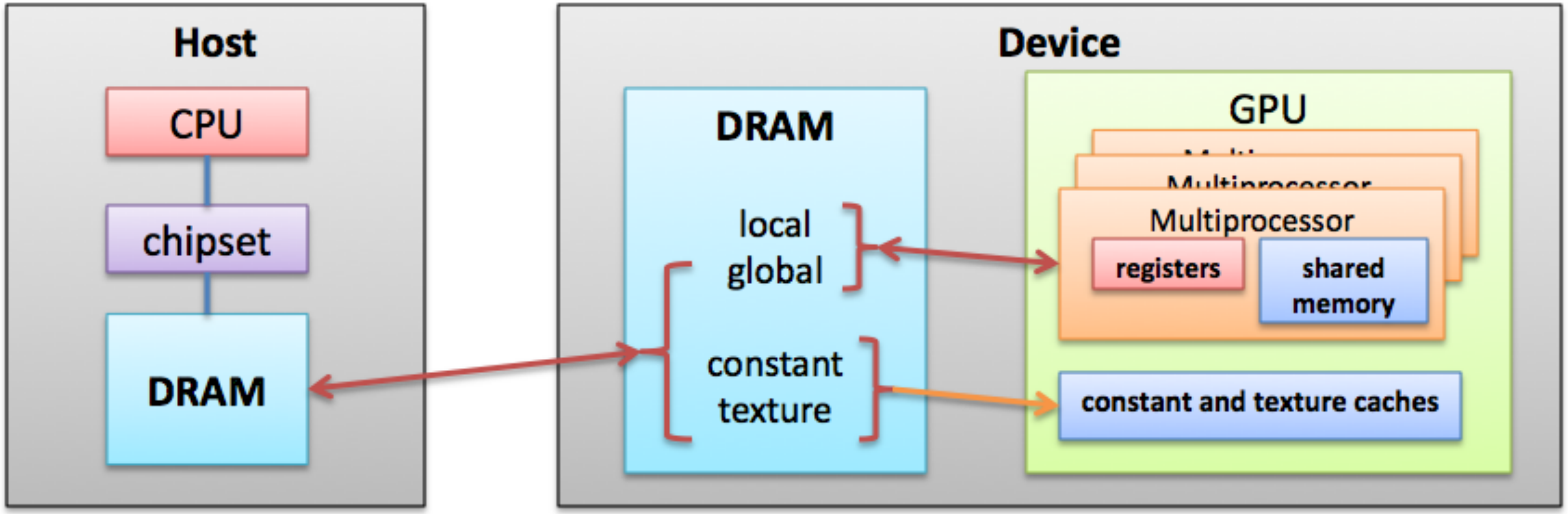
Each block can execute in any order relative to other blocks.

Mapping Threads to the Hardware

- 1D grid
  - 2 thread blocks
- 1D block
  - 2 threads



GPU Memory Hierarchy (Summary)



Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

Other Parallel Programming Paradigms

- Parallel Functional Programming
- MapReduce: HADOOP
- Coordination Languages: Linda
- Platform Specific: OCCAM (Transputer)