CS3006 Parallel & Distributed Systems

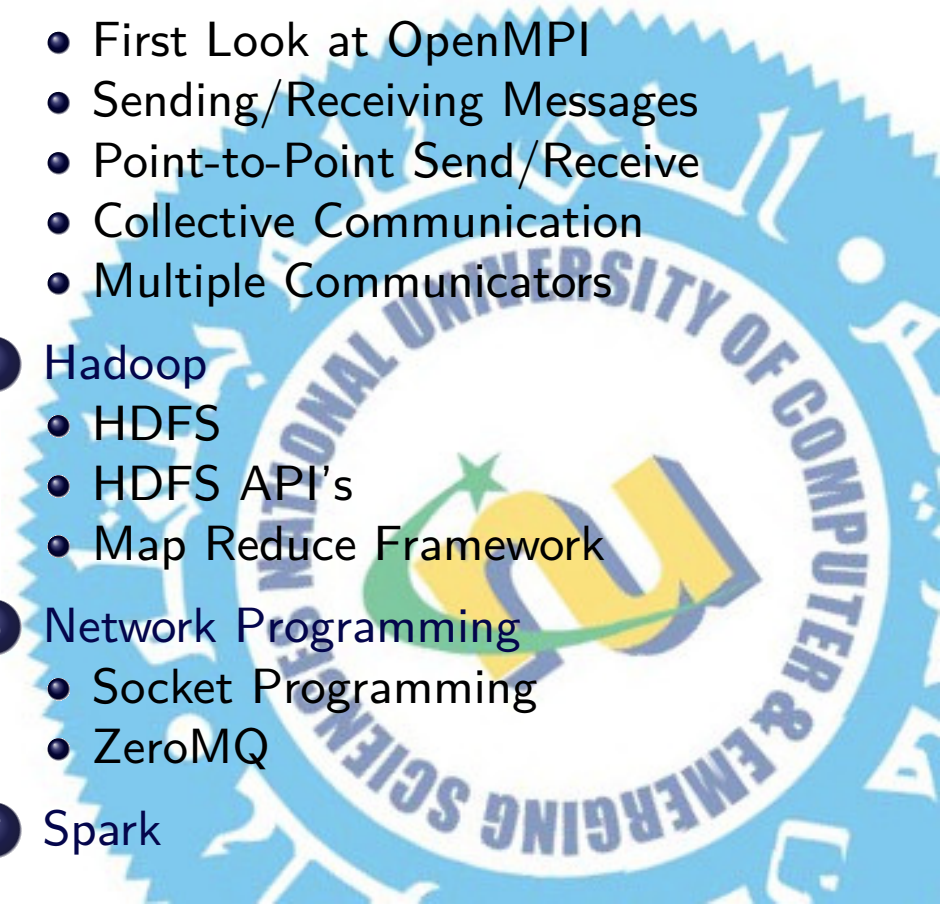
Dr. Omar Usman Khan
omar.khan@nu.edu.pk

Department of Computer Science
National University of Computer & Emerging Sciences, Peshawar

December 21, 2021

Syllabus

1. Introduction
   - Brief Overview
   - Hardware Models
2. Shared Memory Programming Models: pThreads, OpenMP
   - Overview
   - Posix Threads
   - First Look at OpenMP
   - Profiling
   - Work-Sharing Constructs
   - Synchronization
3. Vector Programming: OpenCL/CUDA
   - Overview
   - GPGPU's: OpenCL & CUDA
   - OpenCL Specification
   - First Look at OpenCL
   - CUDA Programming
   - Optimization
4. Distributed Memory Programming Model: MPI
   - Overview
   - Communicators
   - First Look at OpenMPI
   - Sending/Receiving Messages
   - Point-to-Point Send/Receive
   - Collective Communication
   - Multiple Communicators
5. Hadoop
   - HDFS
   - HDFS API's
   - Map Reduce Framework
6. Network Programming
   - Socket Programming
   - ZeroMQ
7. Spark



About the Course

Learning Outcomes

CLO1	Have a good understanding of concurrent and distributed systems
CLO2	Be able to write programs for Concurrent systems
CLO3	Be able to write programs for Distributed systems
CLO4	Be able to write programs for specialized Performance Hardwares

Marks Breakdown

- Sessional I: 15%
- Sessional II: 15%
- Final Examination: 50%
- Assignments: 20% (including Takehome Lab Activities)

About the Course (cont.)

Assignment Instructions

- Will accept assignments after last date, provided said assignment for entire class is not marked.
- Parts of assignment may appear in examinations. If not, then similarity checks may be applied on submitted assignments.

- HPC Setup with support for various parallel and distributed computing frameworks will be made available to all students for duration of semester (and beyond upon request)

Generating Public Key for Assignments

- On Windows, Use https://winscp.net/eng/docs/ui_puttygen
- On Linux, use **ssh-keygen** command
- Share generated key with me by email.

Where are We

- Clock rates 40 MHz (MIPS R3000 1988) $\rightarrow$ 4.0 GHz (Intel Core-i7-4790K 2015), 4.4 GHz (Intel Xeon X5698 2015)
- Higher processor speed $\propto$ More **heat dissipated**
- Transistor has reached size of 32 nm (Generation 3 Core-i7). **Size limit**: How much more smaller can it get?
- What is a single clock cycle worth?
  - **xchg**: Exchange values in two registers
  - **push**: Push operation using registers
  - **pop**: Pop operation using registers
  - **add**,**sub**: 3 additions/subtractions involving registers
  - **cmp**: 3 comparisons involving registers
  - **mul**: half a fp multiplication involving registers
- Can we get a 8 GHz clock frequency?
- What will happen if we reach a 8 GHz clock frequency?
- Support for executing multiple instructions per clock cycle, fast cache technologies, superscalar architectures ...
- Dramatic increase in Floating Point Operations per Second (FLOPs)

Where are We (cont.)

Figure 1: AMD breaks Guinness World Record for fastest processor frequency at 8.429 gigahertz (2011)

Where are We (cont.)

A question to think about

- Akram has a 12 GHz single core processor. Akbar has a quad-core 3 GHz multiprocessor. Which one is faster?

Premise: Top 10 Supercomputers

Application Areas: Astrophysics, financial markets (portfolio management), scientific research (computational modelling, biology, chemistry, physics, mathematics, geometry, graphics, signal processing, etc.), . . . Will be covered again in detail

Rank	Site	System	Cores	TFLOPs	Power (kWh)
1	Guangzhou, China	Tianhe-2: Intel Xeon	3,120,000	54,902	17,808
2	Oak Ridge Lab, USA	Titan: Cray Opteron, NVIDIA K20	560,640	27,112	8,209
3	DoE, USA	Sequoaia:, IBM BlueGene/Q	1,572,864	20,132	7,890
4	RIKEN Ins., Japan	K-Computer: Fujitsu SPARC64	705,024	11,280	12,660
5	DoE, USA	Mira: IBM BlueGene/Q	786,432	10,066.3	3,945
6	Swiss S.Comp., Switzerland	Cray, Intel Xeon, NVIDIA K20	115,984	7,788	2,325
7	Univ. Texas, USA	Stampede: Intel Xeon	462,462	8,520	4510
8	Forschchung Juelich, Germany	JUQUEEN: IBM BlueGene/Q	458,752	5,872	2,301
9	DoE, USA	Vulcan: IBM BlueGene	393,216	5,033	1,972
10	US Govt	Cray, Intel Xeon, NVIDIA K40	72,800	6,131	1,499
-	Your Home	Intel Core-i7	4	.026	0.3
-	Your Home	NVIDIA K40	2,880	4.29	0.3

Table 1: Top 10 Supercomputers: February 2015, top500.org

Premise: Top 10 Supercomputers (cont.)

Figure 2: China's Tianhe-2 . . . The Heaven River, The Milky Way, developed and operated by 1,300 scientists and engineers. Plans to double the computing power were stopped by the US government when they rejected Intel's application of granting an export license for selling CPU's and motherboards to the chinese government. Total of 3.12 Million Cores, and 1,375 TB RAM, runs Kylin Linux, and Slurm job management

Premise: Top 10 Supercomputers (cont.)

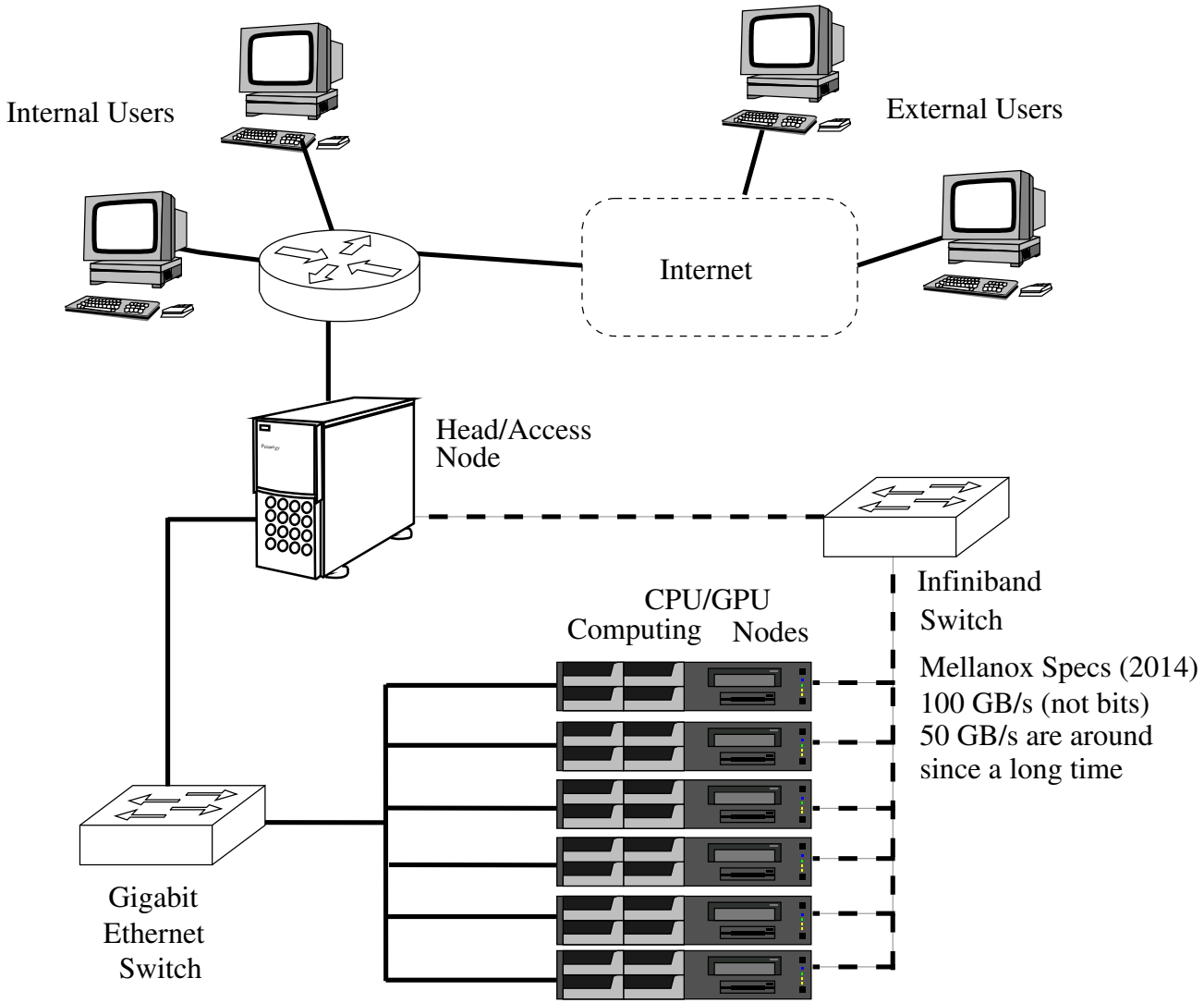


Figure 3: Setup of a typical HPC facility

Top 10 Supercomputers over Time

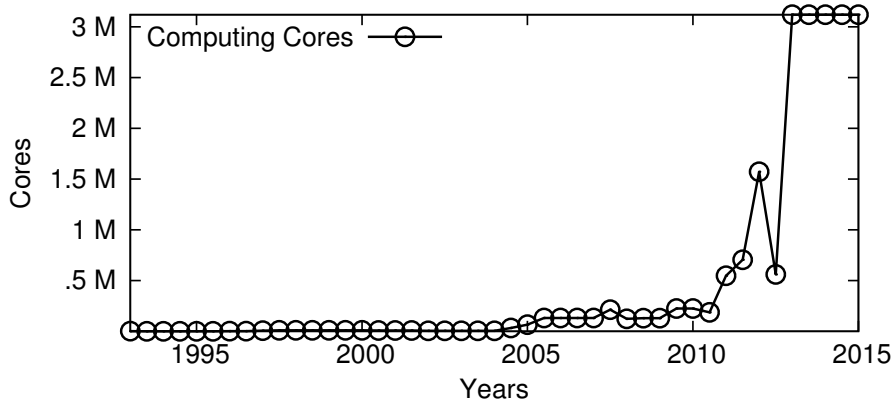
Year	Cores	FLOPs
2006	131,072	367,000,000,000,000 (367 TFLOP)
2007	212,992	596,000,000,000,000 (596 TFLOP)
2008	129,600	1,456,700,000,000,000 (1.4 PFLOP)
2009	224,162	2,331,000,000,000,000 (2.3 PFLOP)
2010	224,162	4,701,000,000,000,000 (4.7 PFLOP)
2011	705,024	11,280,400,000,000,000 (11.2 PFLOP)
2012	1,572,864	27,112,500,000,000,000 (27.1 PFLOP)
2013	3,120,000	54,902,400,000,000,000 (54.9 PFLOP)
2014	3,120,000	54,902,400,000,000,000 (54.9 PFLOP)
2015	3,120,000	54,902,400,000,000,000 (54.9 PFLOP)

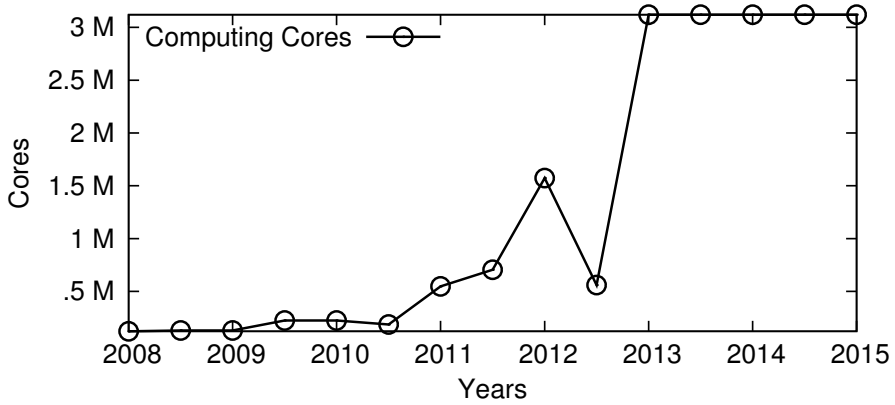
Table 2: Timeline Table of the Top Super-Computer, top500.org

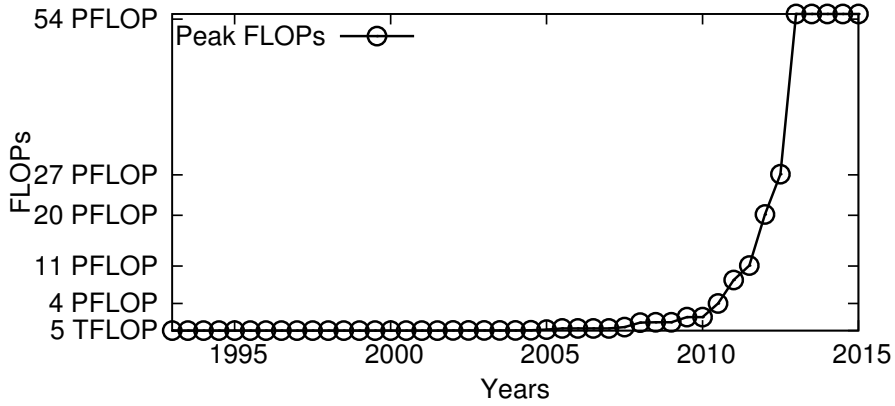
atto	10^{-18}		exa	10^{18}
femto	10^{-15}		peta	10^{15}
pico	10^{-12}		tera	10^{12}
nano	10^{-9}		giga	10^{9}
micro	10^{-6}		mega	10^{6}
milli	10^{-3}		kilo	10^{3}

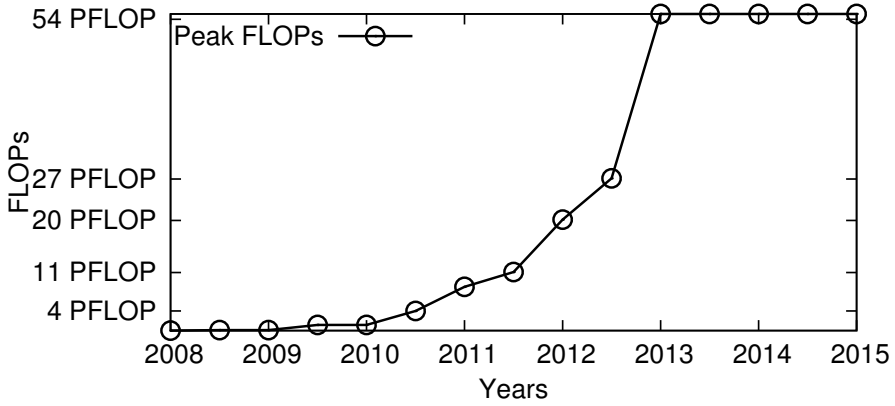
Table 3: Scales

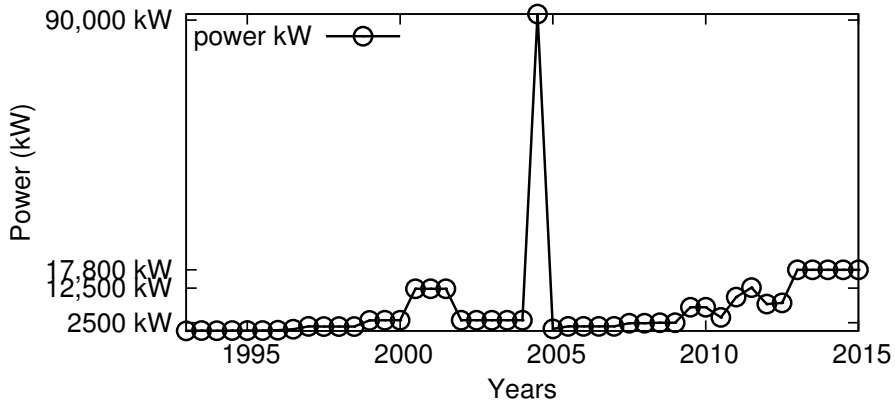
Top 10 Supercomputers over Time (cont.)

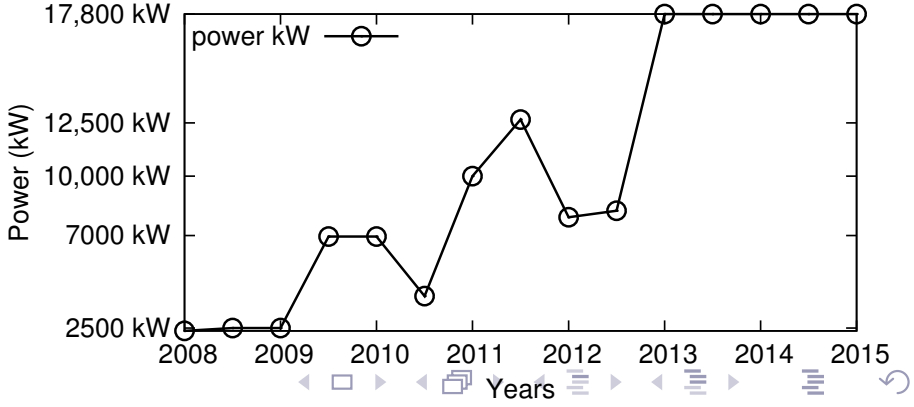












Why do we Want More than 3 Million Cores (or more) ??

- Because some computations are not possible using current hardware. E.g.,
  - Modelling of Global Climate over decades
  - Atomic scale simulations of large materials
  - Aerodynamics simulation of an entire aircraft
  - Modelling of fine resolution air/liquid turbulence

Example Calculations

$10\mu m^3$ volume discretized into cells of size $5nm^3$.

$$\frac{10 \times 10 \times 10\mu m}{5 \times 5 \times 5nm} = \frac{1000\mu m}{125 \times 10^{-9}\mu m} = 8 \times 10^9 \tag{1}$$

RAM required (M) $= 8 \times 10^9 \times sizeof\,(double) = 6.4 \times 10^{10}$ bytes = **59.6 GB**. Assuming each cell takes 1 ms and complexity of $O\,(n)$, Time required (t) = $8 \times 10^9 \times 10^{-3}/\,(60 \times 60 \times 24 \times 30)$ = **3.08 months**. **This is too much.** Changing cell size to $15nm^3$ gives us.

$$\frac{10 \times 10 \times 10\mu m}{15 \times 15 \times 15nm} = \frac{1000\mu m}{3375 \times 10^{-9}\mu m} = 2.96 \times 10^8 \tag{2}$$

RAM required (M) $= 2.96 \times 10^8 \times sizeof\,(double)$ = **2.2 GB**. Time required (t) = $2.96 \times 10^8 \times 10^{-3}/\,(60 \times 60 \times 24)$ = **3.42 days**. **This is acceptible**.

Moore's Law, 1965

A prediction that would define the pace of digital revolution. Many interpretations:

- Computing would increase in **power** exponentially
- Computing would decrease in relative **cost** exponentially
- **Transistory density** will double every year (revised to double every 18 months)

Cramming more components onto integrated circuits

With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip

By Gordon E. Moore

Director, Research and Development Laboratories, Fairchild division of Fairchild Camera and Instrument Corp.

The future of integrated electronics is the future of electronics itself. The advantages of integration will bring about a proliferation of electronics, pushing this science into many new areas.

Integrated circuits will lead to such wonders as home computers—or at least terminals connected to a central computer—automatic controls for automobiles, and personal portable communications equipment. The electronic wristwatch needs only a display to be feasible today.

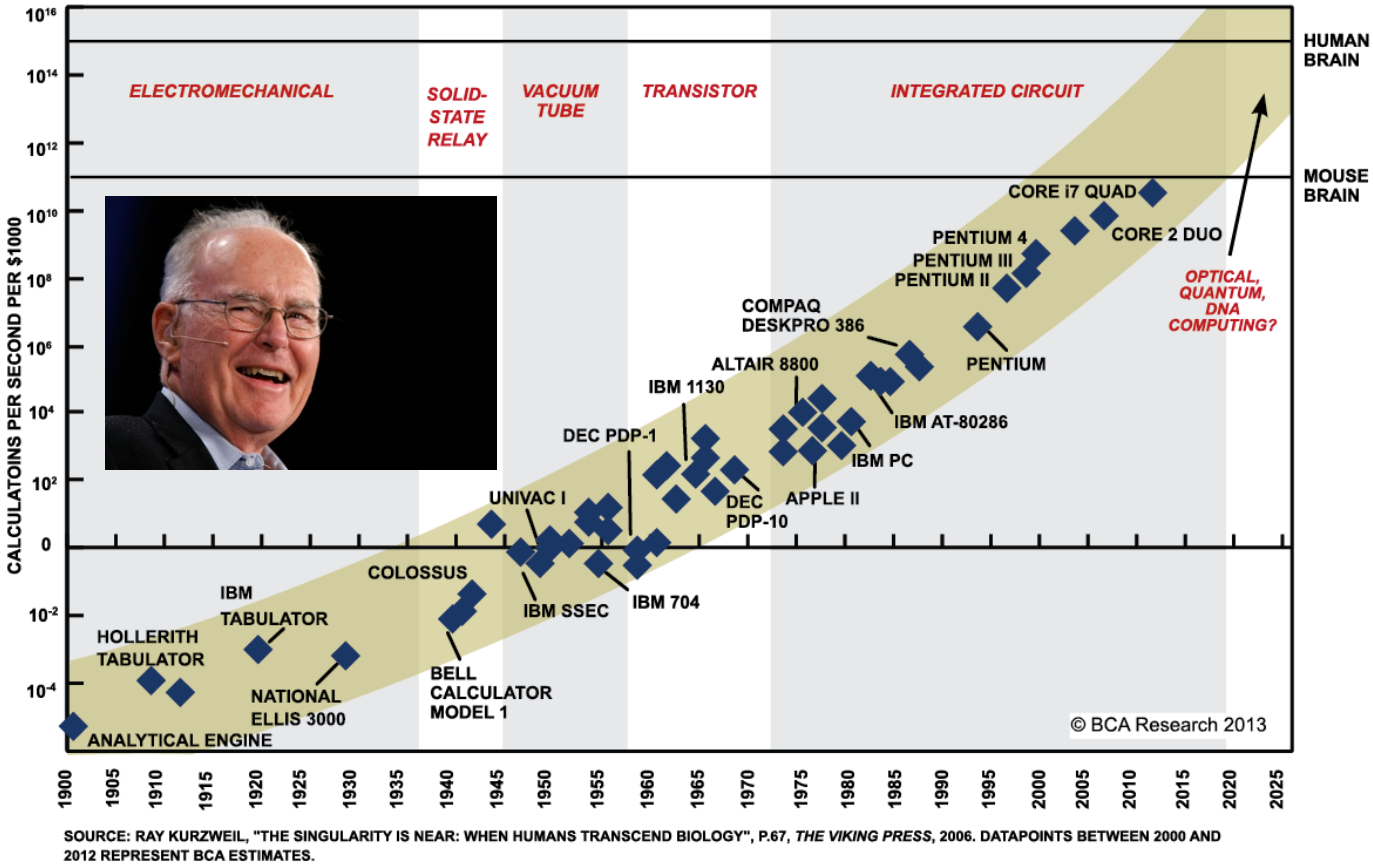
But the biggest potential lies in the production of large systems. In telephone communications, integrated circuits in digital filters will separate channels on multiplex equipment. Integrated circuits will also switch telephone circuits and perform data processing.

Computers will be more powerful, and will be organized in completely different ways. For example, memories built of integrated electronics may be distributed throughout the

The author

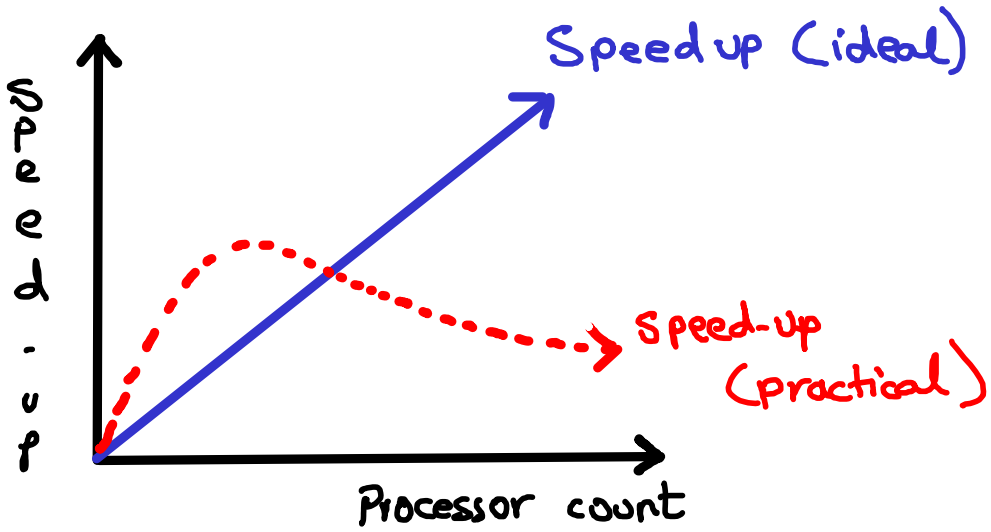
Dr. Gordon E. Moore is one of the new breed of electronic engineers, schooled in the physical sciences rather than in electronics. He earned a B.S. degree in chemistry from the University of California and a Ph.D. degree in physical chemistry from the California Institute of Technology. He was one of the founders of Fairchild Semiconductor and has been director of the research and development laboratories since 1959.

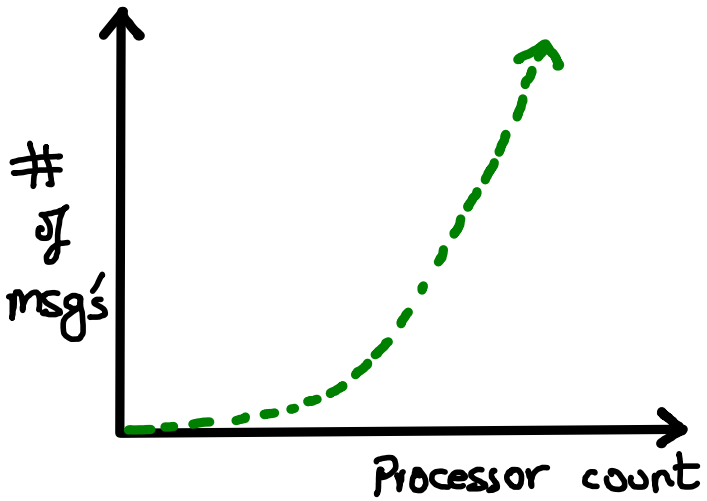
Electronics, Volume 38, Number 8, April 19, 1965



SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

Premise: Can software exploit this hardware for speed??





- Why ideal speedup is not possible: data transfer (through message exchanges), I/O bottlenecks, race conditions, dependencies, contention (critical section), load balancing, deadlocks, synchronization, node failures, **lazy programmers, programmers lacking skills in concurrent and distributed programming** ...

Theoretical Speedup

n Problem Size

p Processors Quantity

$\sigma(n)$ Sequential portion of computation

$\phi(n)$ Parallelizable portion of computation

$\kappa(n,p)$ Parallelization overhead

$T(n,1)$ Sequential Execution time $T(n,1) = \sigma(n) + \phi(n)$

$T(n,p)$ Parallel Execution time $T(n,p) = \sigma(n) + \kappa(n,p) + \frac{\phi(n)}{p}$

$s(n,p)$ Speedup $s(n,p) = \frac{T(n,1)}{T(n,p)}$

$\epsilon(n,p)$ Efficiency $\epsilon = \frac{T(n,1)}{pT(n,p)}$

Theoretical Speedup (cont.)

Amdahl's Law

Ignoring overhead, speedup is:

$$s(n,p) = \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p}} \tag{3}$$

Serial vs Parallel vs Distributed

Serial Computing Systems

- Single control mechanism, determines the next instruction to be executed.
- All data operations are fetched from memory one at a time.
- Speedup can be introduced using:
  - Instruction Caching (large caches with low latency)
  - Pipelining (super-scalar architectures)
  - Overclocking
  - Overlapping of I/O and computation using Direct Memory Access (DMA)
- **Note: Two serial programs can execute concurrently (multi-taskng)**.

- **Multi-Processing**: Multiple CPU cores executing more than 1 program at a given time
- **Multi-Tasking**: A single CPU core "appearing" to be executing more than 1 program at a given time
- **Multi-Threading**: The presence of more than 1 "threads" in a single program.

Serial vs Parallel vs Distributed (cont.)

Parallel Computing Systems

- Several processors that are located within a small distance of each other
- Their main purpose is to perform a computation task jointly
- Communication between processors, if required, is reliable, fast, and predictable.

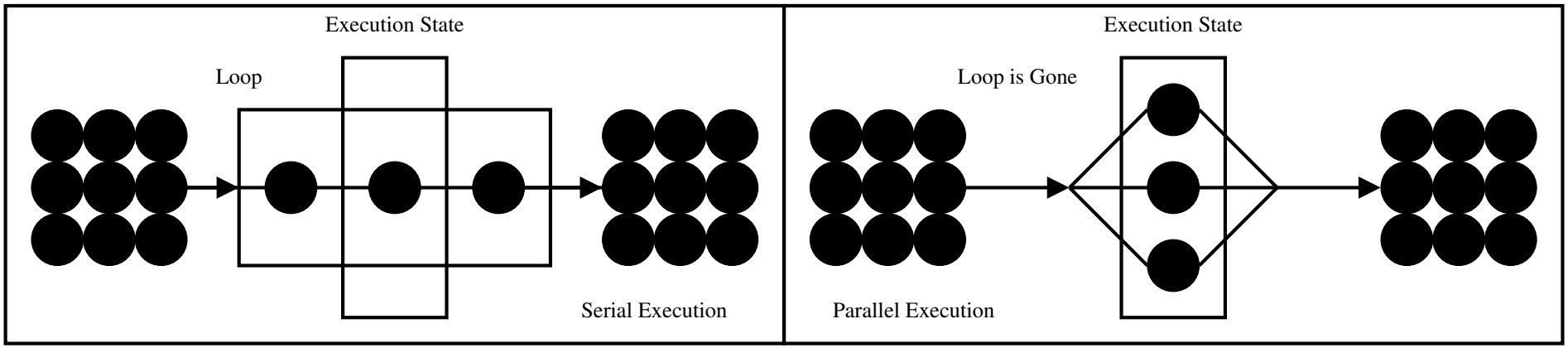


Figure 4: Difference between serial and parallel execution

Serial vs Parallel vs Distributed (cont.)

Distributed Computing Systems

- Processors may be far apart (are loosely coupled and usually independent of others)
- Challenges
  - Communication latency between processors is large and unpredictable
  - Communication links may be unreliable
  - Topology of system may undergo changes.

Serial vs Parallel vs Distributed (cont.)

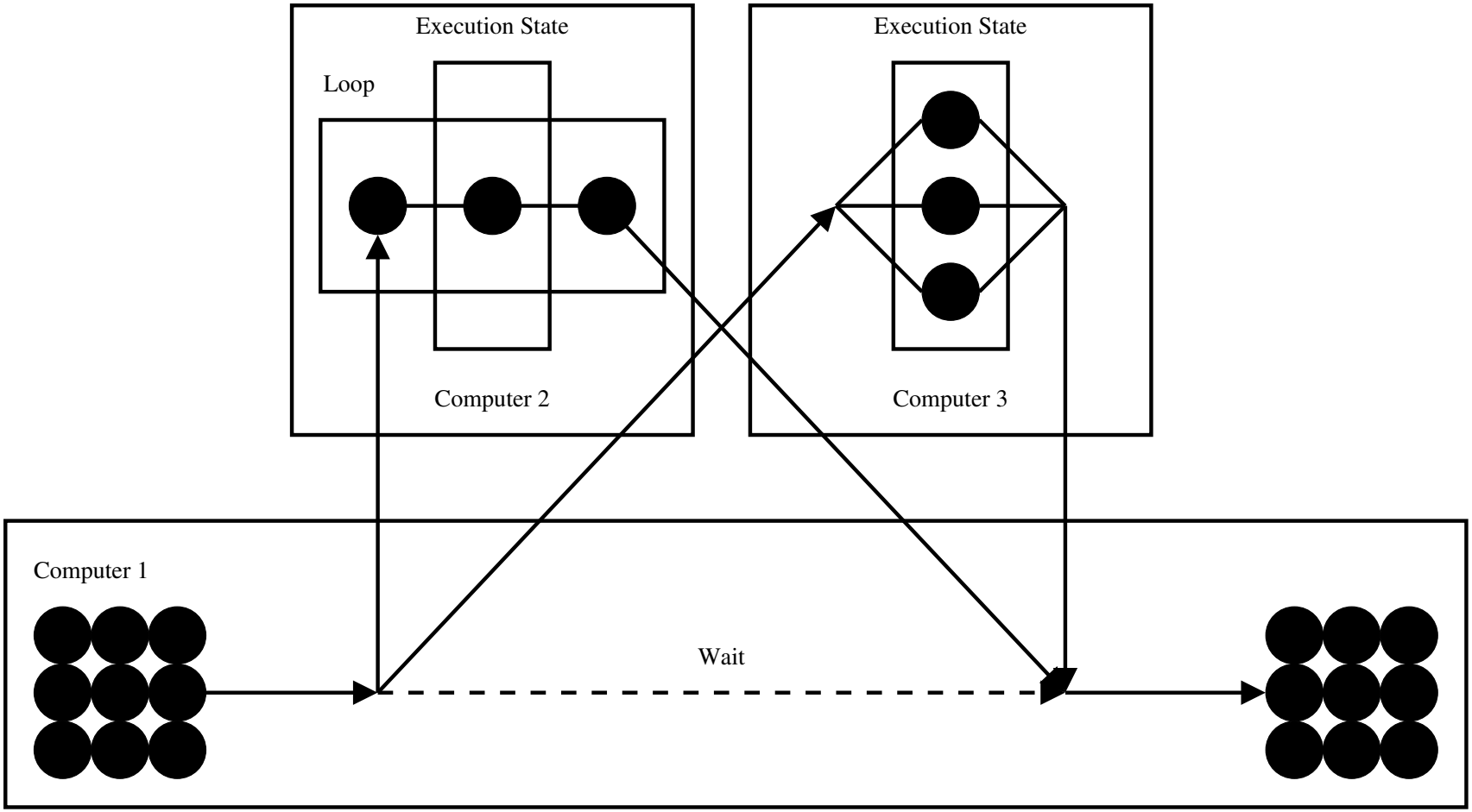


Figure 5: Model of execution on a distributed system (non-client/server). This shows a hybrid. But usually developers don't like these setups.

What is there in Parallel, that is not available in Serial?

Some Examples . . .

- **Task Allocation**: Breakdown of total workload into smaller tasks, such that each smaller task is assigned to a different processor
- **Interim Communication**: Since each processor is working on independent tasks, they may need to communicate outcome of computation with other processors.
  - Is communication automatic? or defined by programmer?
  - Depends on type of multi-threading tool that is being used
- **Synchronization**: A waiting interval by a processor for certain tasks
  - E.g., arrival of data (before computation on it can be performed, or before bring printed).
  - This would have positive effect on accuracy but negative effect on performance.
- . . .

Applications and Examples

General Purpose Programs and Purposes

- Word processor (spell check, auto save, . . .)
- Spreadsheet (automatic background recalculations after cell edits)
- Operating system (User threads, Kernel threads, . . .)
- Server (multi client handling, database servers serving millions of transactions per second)
- GUI's (Event listeners)
- Video games (rendering of animation, taking care of physics, artificial intelligence, network connections, user I/O, . . .))
- Web browser (tabs, multi-segment download accelerators, . . .)
- Search Engines (Information searching, retrieval)
- Encoding/Decoding of video (compression/decompression)
- . . .

Applications and Examples (cont.)

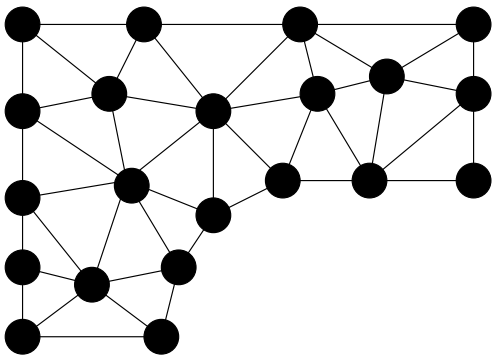
Scientific Purposes

- Mathematical problems (Ordinary differential equations, partial differential equations, linear algebra, numerical analysis)
  - Climate modelling and weather prediction
  - Fluid dynamics (video games, simulation of dam bursts, tsunamies, etc.)
  - Computational Biology, Finance, Physics, Chemistry
  - Machine learning, statistical methods
  - Image processing (medical images, spectral images, satellite images)
- Finding Aliens, Finding cure for Alzheimers, Cancers, Parkinson's, ...

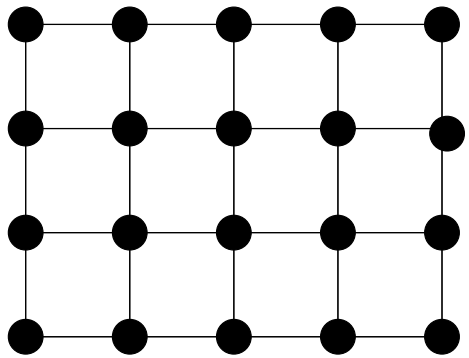
Figure 6: The Folding@Home project by Stanford University

Difference between both types of Applications

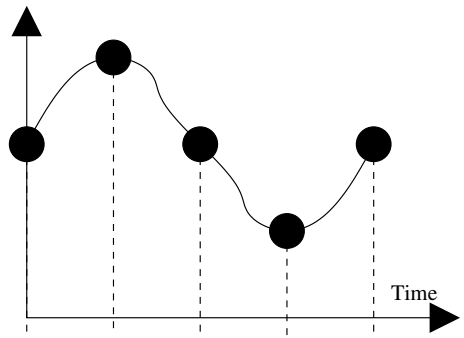
- General Purpose Applications: Threads are usually involved in doing **different things**. This is known as **task-parallelism**.
- Scientific Applications: Threads are usually involved in doing **similar things**, but at different locations. This is known as **data-parallelism**
- Different locations? Take a problem and perform both **spatial decomposition**, as well as **time decomposition**.
- Each decomposed location (involving predominantly local interactions) will be handled ideally by a single processor. If it involves non-local interactions, then processors would need to communicate.
- What level of decomposition would be required ?? A question answered by concerns of cost, speed, storage memory, functional requirements, etc.



Non–Uniform Spatial Decomposition

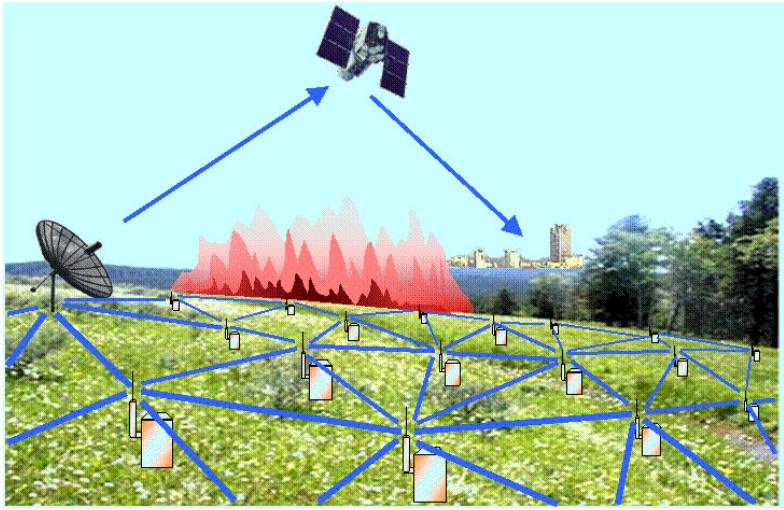


Uniform Spatial Decomposition



Uniform Time Decomposition

Applications



← Fig: Sensor Networks Deployment

- Approach-1: Client/Server Distributed Models
- Approach-2: Decentralized Distributed Models (P2P, WSN, MANET)

- Functions: Information Acquisition, Extraction, Aggregation and Control in geographically distributed systems
- E.g., sensor networks, where geographically distributed sensors **acquire** local information). This information is communicated to a control station through multiple hops. At each hop, information may be **extracted**, **aggregated** (to remove redundancies), and repackaged before forwarded. The control station ultimately recieves the packet and performs **control** operations (e.g., fire control, smart cooling, etc.)
- Concerns: unreliable communication, device failures, cost, speed, deployment, energy constraints

Examples of Processors

- Hundreds of architectures, dozens in main-stream, hundreds now obsolete
- Many ways of classifying these architectures. The most simple classification is those that are "parallel", and those that are "distributed"

Array Processors (Vector Processors)

- Array of coupled processors (with nearest neighbor inter-connects)
- Most array processors perform "combined" or "group" execution.
- Ideal for spatial decomposed problems, mage processing, scientific computing, etc. (not good for task-parallelism)
- A host processor may monitor progress on the processor array and serves as a "control processor"
- Distributed/shared memory array processor possible
- More about this in Hardware Classifications (Flynn)

Examples of Processors (cont.)

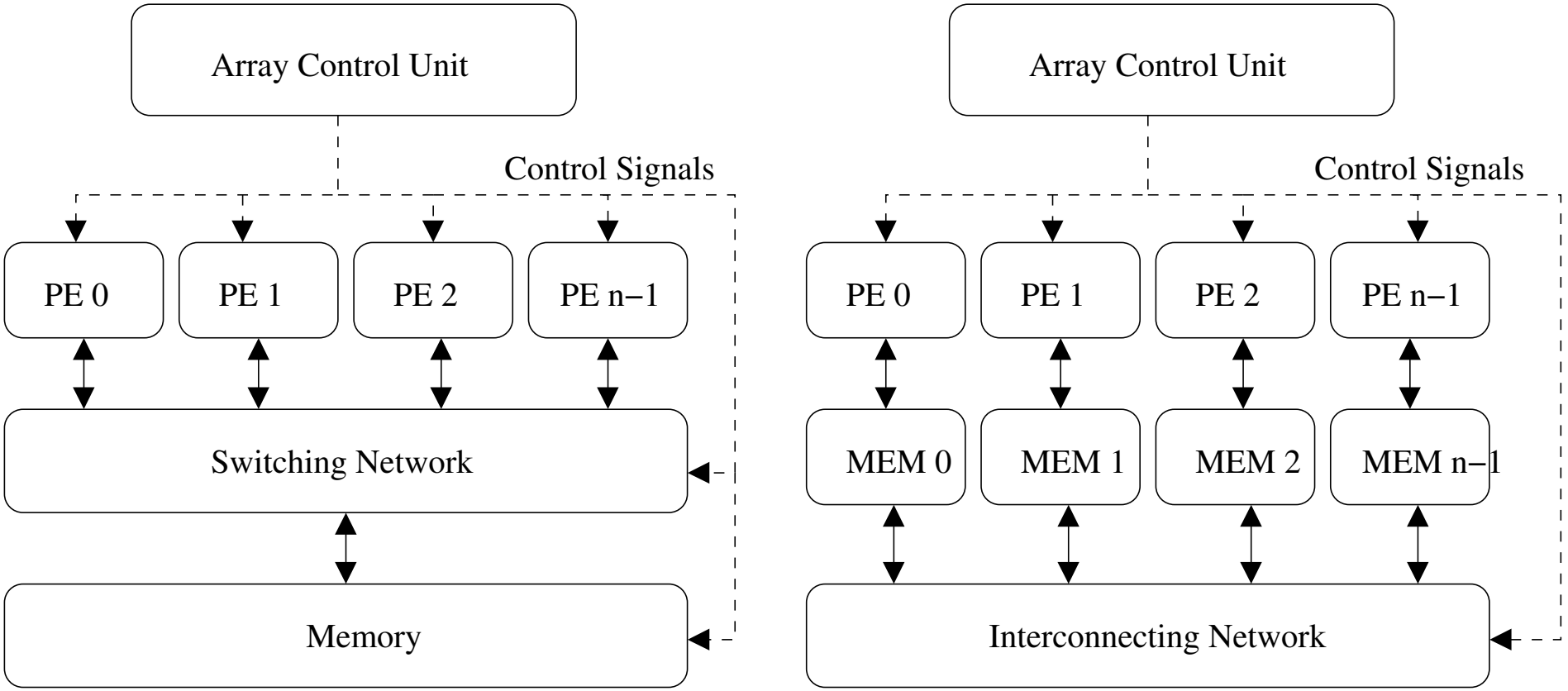


Figure 7: Array Processors (left) Shared Memory (right) distributed memory (IMG: Roland N Ibbett & Nigel P Topham, 1996) (Key: PE = Processing Element)

Examples of Processors (cont.)

Multiprocessors

- Loosely coupled processors, where each processor has more control on what to do and when to do.
- Well suited for task parallel problems (general purpose multi-threaded applications)
- Most common type is known as "symmetric multiprocessor system"

Systolic Arrays

- Makes use of "Very Large Scale Integration" (VLSI) Technology, where all computation elements are placed on a single chip.
- Processors are designed for a single purpose only and show fixed behaviour (code is embedded in system hardware)
- E.g., solving a linear system of equations, performing a fast Fourier transform
- Data "moves" across each processor (must be aligned before execution can commence)

Examples of Processors (cont.)

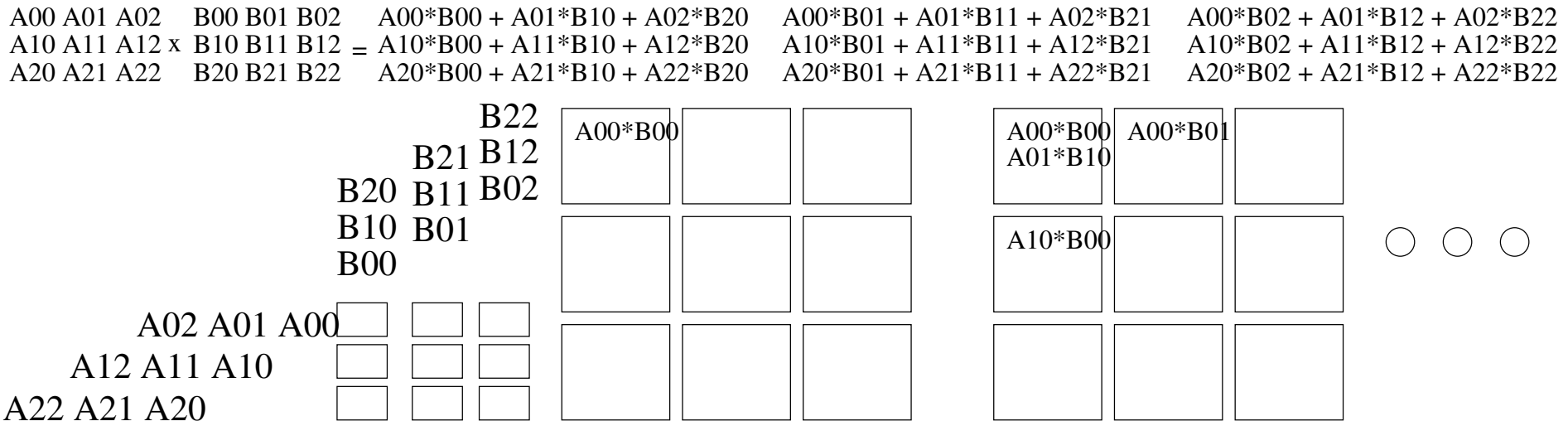


Figure 8: Movement of Data across systolic array for Matrix Multiplication

Parameters Describing Types of Parallel and Distributed Computers

Number of Processors

- Thousands of Processors: Massively Parallel Systems
- Limited No. of Processors: Coarse Grained Parallel Systems
- Clock frequency of massively parallel systems is usually "less" as compared to coarse-grained ones.

Presence/Absence of Global Control Mechanism

- Who is controlling parallel processing ???
- Global control mechanism in Massively Parallel Systems **starts** a program, then **loads** data onto parallel processors, and then **allows** each parallel processor to run on its own, before finally **collecting** data back from each processor
- In coarse grained parallel systems, no central control exists and all processor collaborate in an ad-hoc manner.

Parameters Describing Types of Parallel and Distributed Computers (cont.)

Synchronous vs Asynchronous Operations

- Is there a common global clock available?
- If available, all processors will "synchronize" their execution according to the global clock. (Synchronous Operation)
- Synchronous operations help remove race conditions, but on the other hand, may introduce overhead and delays.

Parameters Describing Types of Parallel and Distributed Computers (cont.)

Processor Interconnects: How Processors Exchange Information

Shared Memory

- Global shared memory accessible by all processors
- Method of Communication: 1 Processor writes to shared memory, the others read from it
- Access latency is very low
- Simultaneous access to shared memory may lead to race conditions (To rectify, use hardware switching Note: software solution like semaphores etc. cannot be used here)

Message Passing

- Processors communicate through messages using an interconnecting network
- Each processor has private local memory
- Ideal interconnecting network would have a link between each and every processor (but this is expensive, hence simpler interconnects such as shared bus can be used)

Parameters Describing Types of Parallel and Distributed Computers (cont.)

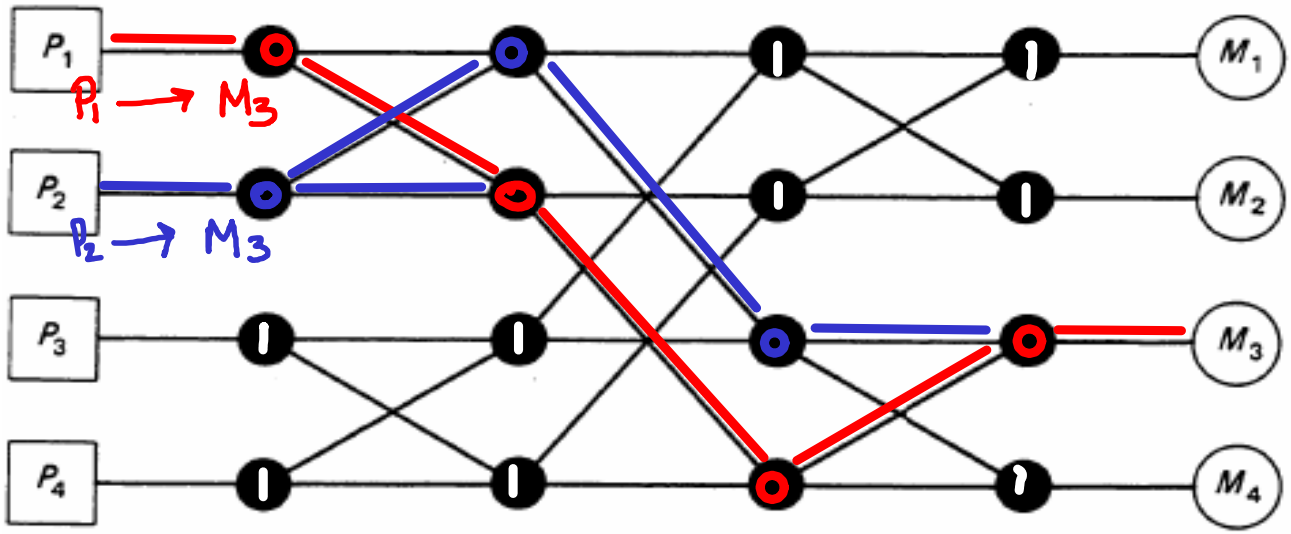


Figure 9: Switching System, with all switches set to 1. p_1 access m_3 and sets the switches to 0 along the access path. p_2 wants to access m_3 but will have to find an access path where the switch is 1. Number of switches depends on number of processors and memory locations, and type of switching network. The depth of switch network would ideally be small (usually at least $\log_2 p$)

Parameters Describing Types of Parallel and Distributed Computers (cont.)

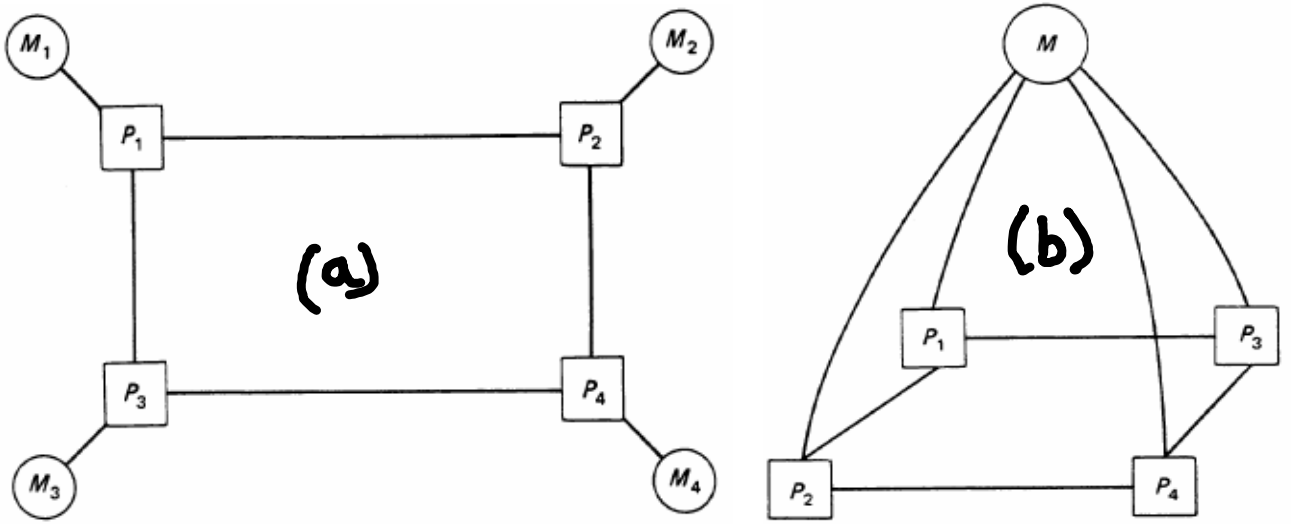


Figure 10: (a) A simple message passing system with local memories assigned to each processor. Processors are interconnected using a ring network. (b) A hybrid system where processors can communicate directly using the ring network, or through the single shared global memory. Global memory contention is ignored in the illustration.

Flynn-Johnson Taxonomy (Classification)

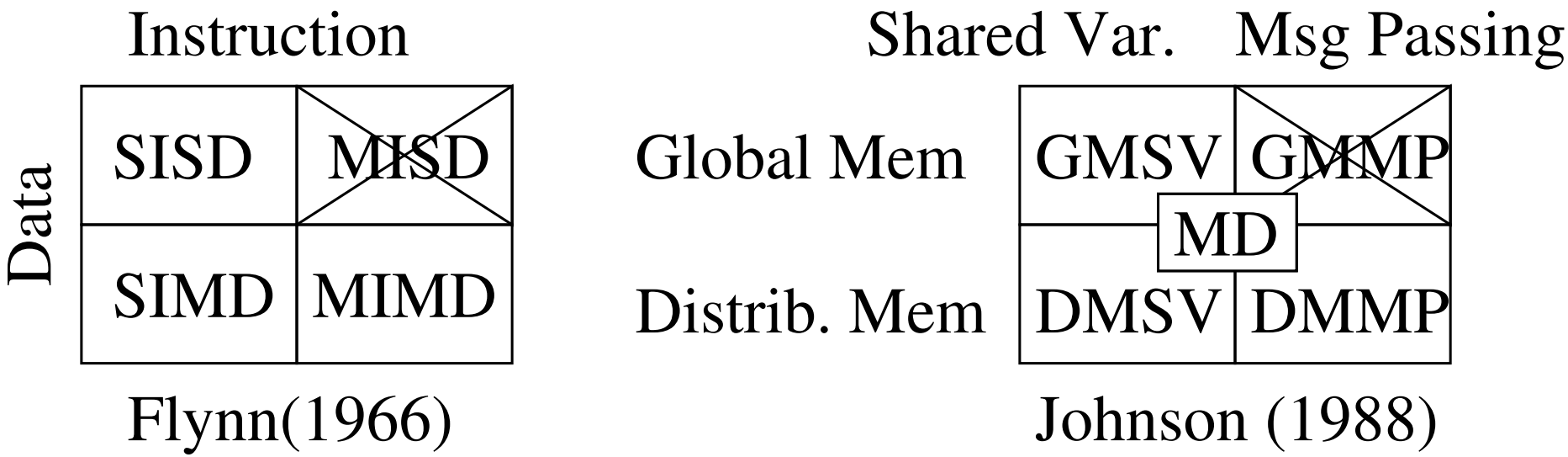


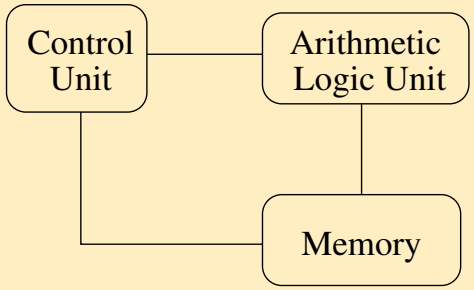
Figure 11: (l) Flynn's Classification (1966) based on instruction and data streams, and (r) Johnson's Classification (1988) based on memory and communication structures

- **Instruction Stream**: Sequence of Instructions
- **Data Stream**: Sequence of Data

Flynn-Johnson Taxonomy (Classification) (cont.)

SISD

- Uni-Processor architectures
- Logic of program is sequential (sequential flow of instructions)
- Concurrent execution using context-switching can be possible
- *It's still SISD if you follow sequential programming on multi-core machines*



Typical Fetch/Execute Cycle: (1) Instruction Fetch, (2) Ins. Decode, (3) Operand Address Calculation, (4) Operand Fetch, (5) Execute, (6) Store Result

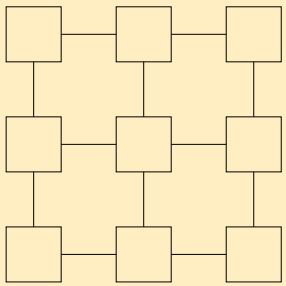
MISD

- Not commercially implemented
- E.g. multiple processors working on decrypting a single encrypted object (using different decrypting algorithms.

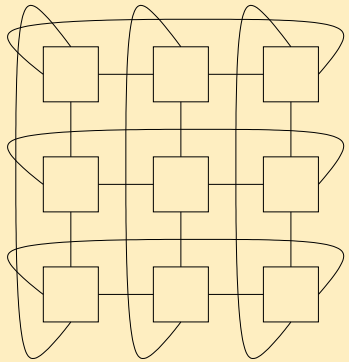
Flynn-Johnson Taxonomy (Classification) (cont.)

SIMD/MIMD

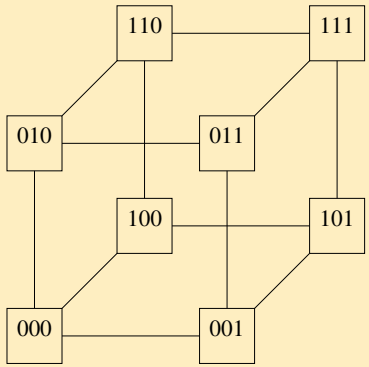
- Array/Streaming/Vector Processors interconnected using nearest-neighbour inter-connects.
- Processing elements contain only "Arithmetic Unit" instead of "Arithmetic and Logic Unit" (hence combined execution)
- Possible to disable some processing elements, if not needed
- Possible Issues
  - Data Alignment Issues
  - Conditional execution



Simple Nearest Neighbor Inter–connects



3 by 3 Nearest Neighbor Inter–connects



3–cube Inter–connects

Flynn-Johnson Taxonomy (Classification)

Johnson's Taxonomy

- **GMSV** Symmetric Multi-Processors (OpenMP, PThreads)
- **DMMP** Distributed systems (If simulated: Omnet++, NS2/3, etc.), NUMA based Cluster systems (OpenMPI, RPC), GPU Computing (OpenCL, CUDA), HPC based systems
- **DMSV** Cloud Model: Memory is distributed but access is through same address space (Hadoop, MapReduce)

Threads

- Smallest execution path
- Runs within scope/address space of a process
- No. of threads = No. of processors ??
- Fork-Join Model

```
main()
{
  // serial region
  fork();
  // parallel region
  join();
  // serial region
}
```

Overview

- Open Multi Processing
- Portable (Windows, Linux), shared-memory threading API for C/C++/Fortran (around 60 methods)
- Combines serial and parallel code in single source file
- Current specification: OpenMP 4.0
  - Compiler directives (Native to C/C++/Fortran)
  - Runtime library routines (e.g. increase/decrease threads as required)
  - Environment variables

Compiler Directives

```
#pragma omp construct clauses
```

- **#pragma omp** Sentinel
- **construct** The construct, or directive Name
- **clauses** e.g., shared, schedule, etc.

Overview (cont.)

Runtime Routines

- 61 routines
- Perform tasks such as Control number of threads, Query thread information, lock management, wall clock time monitoring, etc.
- Example:

```
omp_set_num_threads(8);
```

Environment Variables

- Alternative to runtime routines

```
export OMP_NUM_THREADS=8 && source /etc/profile
```

Posix Threads **pthreads**

- Before there was OpenMP, common approach to support parallel programming was(is) **pthreads**
- **P**ortable **O**perating **S**ystem **I**nterface for UNI**X**
- Originally for UNIX and Linux, but meant for all operating systems that are POSIX standard compliant (Windows did not fall down this way)

```
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
   printf("\n%d: Hello World!\n", threadid);
   pthread_exit(NULL);
}

int main() {
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t < NUM_THREADS;t++) {
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc) printf("ERROR; return code from pthread_create() is %d\n", rc);
   }
   pthread_exit(NULL);
}
```

Posix Threads **pthreads** (cont.)

- Compiled as

  `gcc filename.c -lpthread`

- **Joining a Thread**: Making one thread wait for another (e.g., calling thread waiting for called thread)

```
void *foo() {
   printf("Hello Thread\n");
}

int main() {
   pthread_t tid;
   pthread_create(&tid, NULL, foo, NULL);
   pthread_join(tid, NULL);
   printf("Hello Process\n");
   exit(0);
}
```

- Many programmers find posix to be hard, cumbersome
  - Function pointers
  - Crypted functions calls such as:

    `pthread_create()`, `pthread_exit()`, `pthread_join()`

- Low chances that a compiler may optimize automatically for the above code
- Code is dependent on Posix compatible platforms (operating systems) only.
- Not designed for data-parallelism (scientific computing)

The Code (No. of Threads)

- Master thread spawns a team of threads as needed. (all thread creation work done transparently, developer saved from details)

```
#include <omp.h>

int main(int args, char *argv) {
  #pragma omp parallel num_threads(4)
  {
    tid = omp_get_thread_num();
    printf("Hello from %d\n", tid);
  }
}
```

- Compilation (Intel)

```
icc file.c -openmp
```

- Compilation (GNU)

```
gcc file.c -fopenmp
```

- Common Runtime Routines:

```
omp_set_num_threads(int);
/* Who am I */
omp_get_thread_num();
omp_get_max_threads();
/* Are we in a parallel region?*/
omp_in_parallel();
/* How many processing cores*/
omp_get_num_procs();
/* Lock a thread */
omp_set_lock();
```

- Shared vs Private (What will happen below?)

```
int tid;
#pragma omp parallel private(tid)
{
   tid = omp_get_thread_num()
}
#pragma omp parallel shared(tid)
{
   tid = omp_get_thread_num()
}
```

The Code (Conditional Parallelism)

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

int main(int args, char *argv)
{
  int tid, doIt = 0;

  #pragma omp parallel if(doIt == 1) private(tid)
  { /* Parallel Region Construct Starts here */
    tid = omp_get_thread_num();
    printf("Hello from %d\n", tid);

    if (tid == 0)
    {
      printf("Hello %d threads\n", omp_get_num_threads());
    }
  } /* Parallel Region Construct Ends here */
}
```

The Code (Reduction)

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

int main(int args, char *argv)
{
 int tid, N = 1000, sum = 0;
 int array[N];

 for (i = 0; i < N; i++) {
  array[i] = i+1;
 }

 #pragma omp parallel num_threads(N) private(tid) reduction(+:sum)
 {
  tid = omp_get_thread_num();
  sum += array[tid];
 }

 printf("Sum: %d\n", sum);
}
```

Profiling

Profiling

An analysis that may measure usage of instructions in terms of their frequency or duration of execution. The goal of profiling is to aid program **optimization**.

Clocks (Wall, User, System)

```
time ./a.out
# output of my program, if any.
real    0m0.003s # Wall clock time
user    0m0.001s # Userspace (accumulated processors)
sys     0m0.001s # Kernelspace
```

General Tips

- Run code for sufficiently long period of times
- Allow relaxation period in code for better approximation

Profiling (cont.)

Event based Methods for Profiling

- HW-based performance counters: Special purpose registers that store counts of HW-related activities (or execution of special instructions)
- Operating system hooks (e.g., function/system call interception)
- Userspace libraries (e.g., sys/times.h)

```
struct timespec start, finish; /* Defined in time.h */
/* struct timespec contains tv_sec, and tv_nsec */
/* struct timeval  contains tv_sec, and tv_usec */
clock_gettime(CLOCK_MONOTONIC, &start); /* vs CLOCK_REALTIME (NTP dependent) */
  /* Code Here */
clock_gettime(CLOCK_MONOTONIC, &finish);
double elapsed = (finish.tv_sec - start.tv_sec);
    elapsed += (finish.tv_nsec - start.tv_nsec) / 1e9;
```

Profiling (cont.)

Statistical Tools for Profiling (only percentages, no absolute time

- System-Level: (profiling through kernel using OProfile)
- User-Level: (Using tools like **gprof**, kcachegrind, valgrind, google performance tools)

Flat Profile of GPROF

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
39.47	12.01	12.01	1	12.01	21.38	func1
30.79	21.38	9.37	1	9.37	9.37	func2
30.79	30.75	9.37	1	9.37	9.37	new_func1
0.13	30.79	0.04				main

Profiling (cont.)

Call Graph of GPROF

```
index % time    self  children    called     name
[1]    100.0    0.04   30.75                 main [1]
               12.01    9.37       1/1           func1 [2]
                9.37    0.00       1/1           func2 [3]
-----------------------------------------------
               12.01    9.37       1/1           main [1]
[2]     69.4   12.01    9.37       1         func1 [2]
                9.37    0.00       1/1           new_func1 [4]
-----------------------------------------------
                9.37    0.00       1/1           main [1]
[3]     30.4    9.37    0.00       1         func2 [3]
-----------------------------------------------
                9.37    0.00       1/1           func1 [2]
[4]     30.4    9.37    0.00       1         new_func1 [4]
```

Profiling (cont.)

```
#include <omp.h>

double start_time = omp_get_wtime();
#pragma omp parallel [...]
   // code
double time = omp_get_wtime() - start_time;
```

Work Sharing

- **Work Sharing**: General term describing distribution of work across threads
- Can be performed using three constructs:
  - **for** construct (for data parallelism)
  - **sections** construct (for task parallelism)
  - **tasks** construct (for irregular problems, e.g. unbounded loops, recursive codes)

Data Parallelism

- Assuming that there is data independence across loop iterations, try:

```
#pragma omp parallel [clauses]
{
   #pragma omp for [for clauses]
   for (loop control) {
      // statements
   }
}
```

- OpenMP (or it's compilers) cannot (always) automatically identify data dependencies
- Threads "share" the iterations of the for loop
- Equivalent Code:

```
#pragma omp parallel for [for clauses]
for (loop control) {
   // statements
}
```

Data Parallelism (cont.)

```
#pragma omp parallel num_threads(2) private(tid)
{
   tid = omp_get_thread_num();
   #pragma omp for
   for (i = 0; i < 20; i++) {
      printf("%3d by %3d\n", i, tid);
   }
}
```

Data Parallelism (cont.)

Schedule Clauses (How loop iterations are mapped to threads)

Static Scheduling

- Low-overhead
- Load imbalance
- threads assigned "chunks" of iterations

Dynamic Scheduling

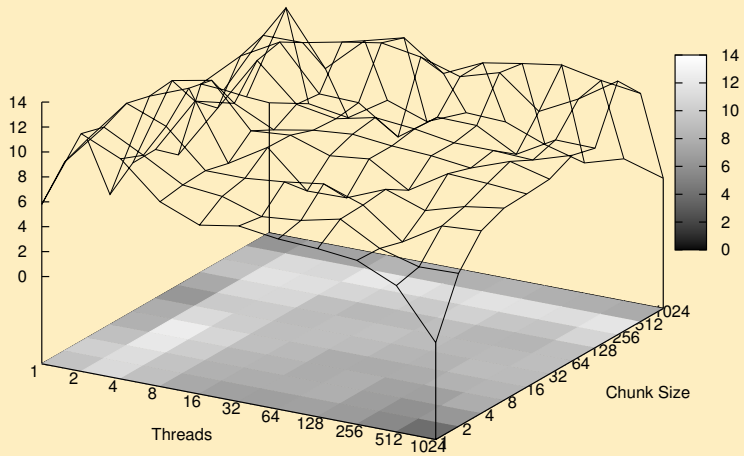
- High-overhead
- Reduces load imbalance
- Threads grab "chunks" of iterations

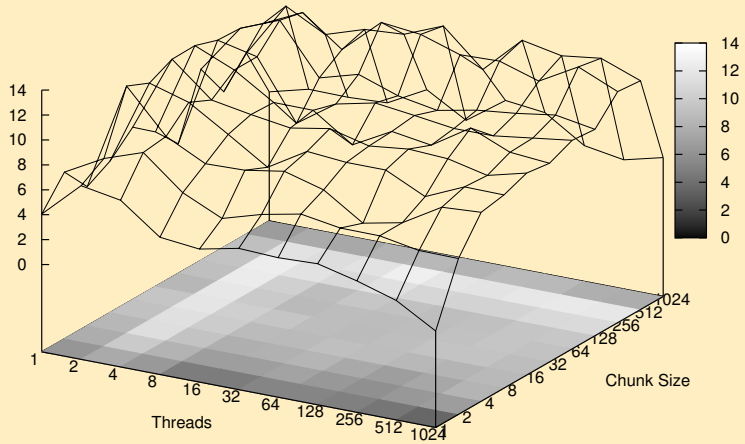
- `#pragma omp for schedule(static,chunksize)`
- `#pragma omp for schedule(dynamic,chunksize)`

Data Parallelism (cont.)

Effect of Chunk Size and Thread Quantities

- Computed 1024×1024 matrix multiplication using static and dynamic scheduling
- (Left) Static Scheduling (Right) Dynamic Scheduling





Data Parallelism (cont.)

Is Schedule Class Really Necessary?

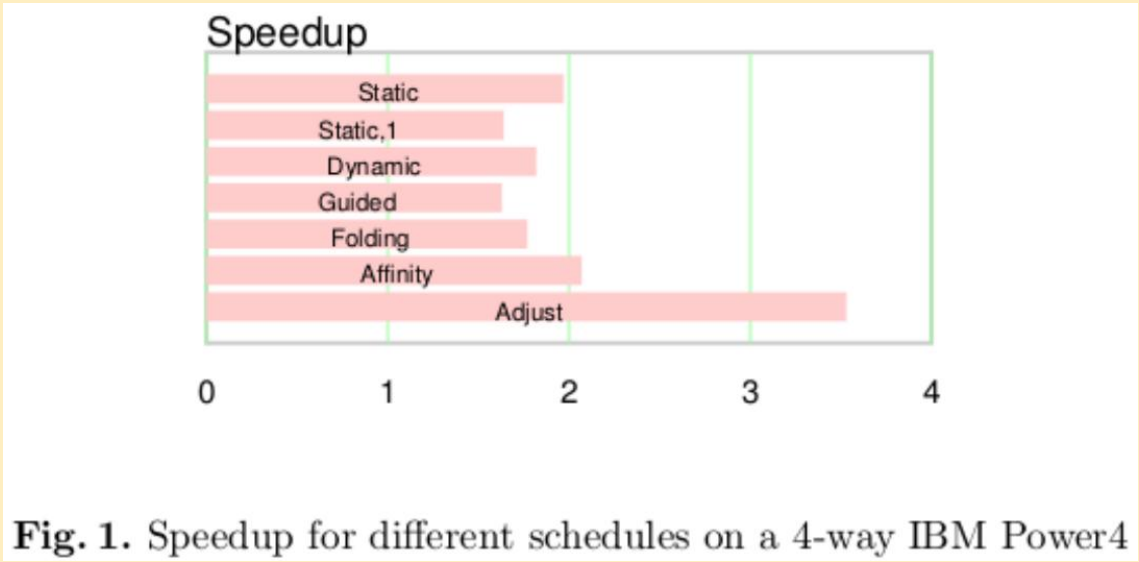


Fig. 1. Speedup for different schedules on a 4-way IBM Power4

- Ayguade et. al., Is the schedule clause really necessary in OpenMP?, Technical Report, Springer Verlag, 2003

Task Parallelism

- Considering following scenario:

```
p = pcibus();
n = networkCard(p);
w = wifiCard(p);
s = ssh(n,w);
h = http(n,w);
f = ftp(n,w);
```

- n, w can be executed in parallel
- s, h, and f can be executed in parallel
- Task Parallelism using **sections** in OpenMP:

```
#pragma omp parallel [clauses]
{
   #pragma omp sections
   {
      #pragma omp section
      // Code of first task

      #pragma omp section
      // Code of second task
   }
}
```

Task Parallelism (cont.)

- **Sections** must be inside a parallel region. **Sections** itself provides enclosure for an individual **omp section**

```
p = pcibus();
#pragma omp parallel sections num_threads(2)
{
   #pragma omp section
   n = networkCard(p);
   #pragma omp section
   w = wifiCard(p);
}
#pragma omp parallel sections num_threads(3)
{
   #pragma omp section
   s = ssh(n,w);
   #pragma omp section
   h = http(n,w);
   #pragma omp section
   f = ftp(n,w);
}
```

- If no. of threads is $<$ no. of tasks, threads first attempt the beginning tasks before jumping to next ones.
- If no. of threads is $>$ no. of tasks, each task is performed by one thread, while the remaining remain idle

OpenMP Tasks for Task Parallelism

- OpenMP Tasks suitable for irregular problems (problems without loops, unbounded loops, recursive algorithms, etc.)
- Example: Summation across loop:

```
int summation(int *array, int N)
{
   int i, sum = 0;
   for (i = 0; i < N; i++)
      sum += array[i];
   return sum;
}

int main()
{
   int N = 1000;
   int array[N];

   for (i = 0; i < N; i++)
      array[i] = i+1;

   int sum = summation(array, N);

   printf("Sum: %d\n", sum);
}
```

- Example: Summation without loop (recursive):

OpenMP Tasks for Task Parallelism (cont.)

```
int summation(int *array, int N)
{
   if (N == 0)       return 0;
   else if (N == 1)   return *array;

   int half = N / 2;
   return summation(array, half) + summation(array+half, N - half);
}
```

- Replacing recursive call with OpenMP Tasks:

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task shared(x)
   x = summation(array, half);
   #pragma omp task shared(y)
   y = summation(array+half, N - half);

   #pragma omp taskwait

   x += y;
}
return x;
```

- Tryout: Fibonacci Series $f(1) = 1, f(2) = 1, f(n) = f(n-1) + f(n-2)$
- Tryout: Traversing Linked List (unbounded loop)

OpenMP Tasks for Task Parallelism (cont.)

```
struct LL {
   int x;
   struct LL *next;
};

int main()
{  int i, random = 10;    // rand() % 10;

   struct LL *head, *tail, *current;
   head = malloc(sizeof(struct LL));
   head->x = 0;
   head->next = NULL;

   tail = head;
   for (i = 1; i < random; i++) {            // populating the linked list
      current     = malloc(sizeof(struct LL)); // allocate space for new node
      current->x    = i;                   // set its value to i
      current->next = NULL;                  // next is obviously null
      tail->next    = current;               // set tail of last node
      tail        = current;               // set new tail
   }
   current = head;
   while(current) {                        // this is the actual work
      printf("%d\n", current->x);             // task which we modify
      current = current->next;               // on the next slide
   }
}
```

So modifying the work into tasks from previous slide, we have:

OpenMP Tasks for Task Parallelism (cont.)

```
#pragma omp parallel
#pragma omp single
{
   current = head;
   printf("single is: %d\n", omp_get_thread_num());
   while(current) {
      #pragma omp task
      printf("%d by %d\n", current->x, omp_get_thread_num());
      current = current->next;
   }
}
```

- **Properties of Tasks**
  - Tasks are independent units of work (but if dependencies exist, then their execution may be deferred)
  - One thread assigned to perform work of a given task
  - Tasks can be nested (e.g. task within task)
  - All threads cooperate in execution of each other.
  - Tasks may be deferred, or executed immediately (decided by runtime automatically)
  - Task threads can suspend the execution of a task and start/resume another
- Why is **single** construct being used?
  - One thread enters single construct and creates all tasks (you don't want all threads to enter construct and create duplicate tasks)

OpenMP Tasks for Task Parallelism (cont.)

- This single thread will create a pipeline (task queue) on which other tasks will be placed.
- By default, all threads will start execution of tasks **only when** single construct (thread) finishes its job. This can be overridden by passing **nowait** clause to the single construct.

- How are both segments and tasks executed?
  - All **segment** based threads will be "alive" together. (In other words, all threads will implicitly wait at a termination barrier). (Or, in other words, no thread will exit unless the other ones are ready to exit.)
  - In tasks, many possibilities can exist. **Default is:** Worker threads will wait for thread running single construct to create tasks. As soon as the tasks are created, they, including the thread running single construct may pick any task and execute it.

Synchronization

- **Synchronization**: A technique that (a) imposes order of execution, and/or (b) provides protected access to shared data
  - **High Level Synchronization**: barrier, taskwait, critical, atomic
  - **Low Level Synchronization**: locks (simple, nested)

Synchronization (cont.)

High Level Synchronization

- **Barriers** (All work being performed by any thread is guaranteed to be completed at barrier exit). Usage:

  ```
  #pragma omp barrier
  ```

- For OpenMP Tasks, the barrier is the **taskwait** clause. Usage:

  ```
  #pragma omp taskwait
  ```

- **Critical Section** provides for mutual exclusion: only one thread can enter critical section at a time

  ```
  int myValue = 0;
  #pragma omp parallel num_threads(500) shared(myValue)
  {
     #pragma omp critical
     myValue++;
  }
  printf("Value: %d\n", myValue);
  ```

- **atomic** also provides mutual exclusion by making sure read/write to memory location is protected and singular. (Different from critical clause, which can be any sequence/type of code)

  ```
  #pragma omp atomic
  ```

Synchronization (cont.)

Low Level Synchronization

- OpenMP **Simple Locks**: A lock is available if it is **unset**

```
omp_lock_t myLock;          // the lock object
omp_init_lock(&myLock);     // initialize to unset
#pragma omp parallel
{
   omp_set_lock(&myLock);   // set lock (wait if already locked)
   // do some work
   omp_unset_lock(&myLock); // unset lock
}
omp_destroy_lock(&myLock);  // destroy/deallocate lock
```

  - Can also test if a lock is set or not (without blocking) by calling:

```
omp_test_lock(&myLock);  // returns true if set, false otherwise
```

- OpenMP **Nested Locks**:
  - A set is an increment, an unset is a decrement.
  - Can be locked a number of times. Doesn't unlock until you have unset it as many times it was locked.

```
omp_nest_lock_t   myLock;
omp_init_nest_lock(&myLock);
omp_set_nest_lock(&myLock);
omp_unset_nest_lock(&myLock);
omp_destroy_nest_lock(&myLock);
```

Vector Programming Overview

- Execution Models for GPU Architecture: MIMD, SIMD, SIMT

Figure 12: Fundamental Design Philosophy of CPU vs GPU

- **Control Logic**: Allow Parallel and/or Out-of-Order execution of threads. (Centralized on CPU, Decentralized on GPU)
- **ALU**: Perform arithmetic and bitwise operations (One ALU for each core on CPU, One ALU for each core on GPU, or One Arithmetic Unit (FPU) for each core)
- **Cache**: On-chip Memory to reduce instruction and data access latencies (Very small capacity on GPU)

Vector Programming Overview (cont.)

- **DRAM CPU**: Off-chip Memory to store different processes

Why are people switching to GPU's?

- Performance Reasons (Offload numerically intensive parts to GPU)
- Processor availability in Market (Program for the dominant processor. 10 years ago, parallel parallel computing limited to governments and large corporations/universities. This has all changed now with GPUs, thanks to video games.)
- Massive scalability in limited space (Embedded applications requiring parallelism could not include large cluster-based machines. With GPUs, they can)
- IEEE Floating Point Compliancy (Early GPU's were not entirely IEEE compliant. Hence programmers refrained to use them. This is now almost history, unless you buy an old GPU)
- Graphics Programming no longer required to operate on Graphics Cards. We have **GPGPU** compliant API's.

Example: NVIDIA Kepler K40

- GP-GPU, Scientific Computing
- Slave Processors
- GPU Giants (NVIDIA + AMD)
- CUDA: NVIDIA based GPU's
- OpenCL: Open coding standard for cross-device execution (Mobile Phones, GPU, CPU, Altera FPGA's), established by Khronos Group (2008)

-	Kepler K40	Intel i7-4900MQ
Cores	2,880	4 (8 Threads)
RAM	12 GB	-
Cache	48 Kb	8 Mb
Clock	876 MHz	2.8 GHz
Bwidth	288 GB/s	25.6 GB/s
FLOP (d)	1.40 TFLOP	13.97 GFLOP
FLOP (f)	4.29 TFLOP	25.76 GFLOP

NVIDIA k40, 2880 cores, 12 GB RAM

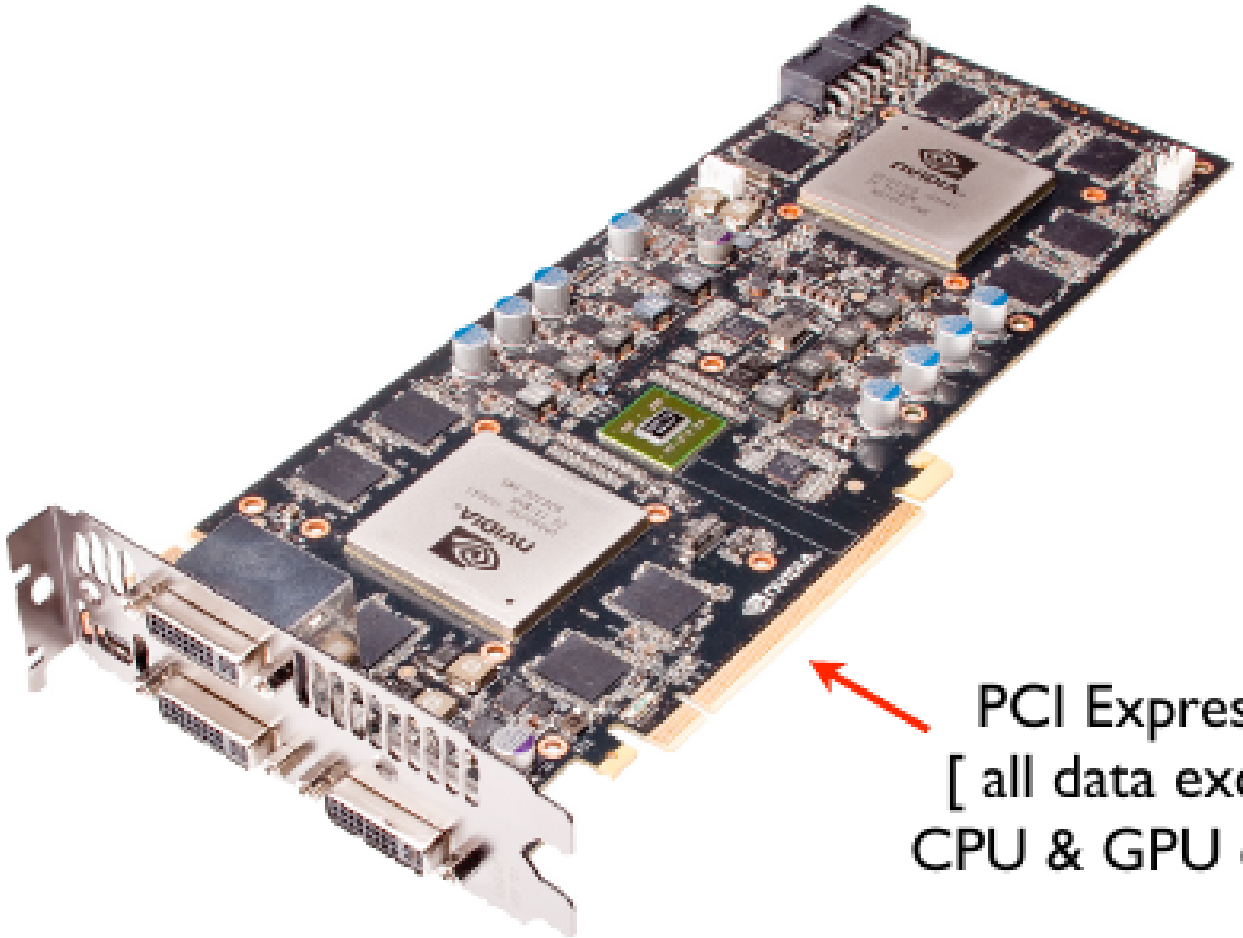
Example: NVIDIA GTX 590

GTX 590 GPU

GTX 590 GPU

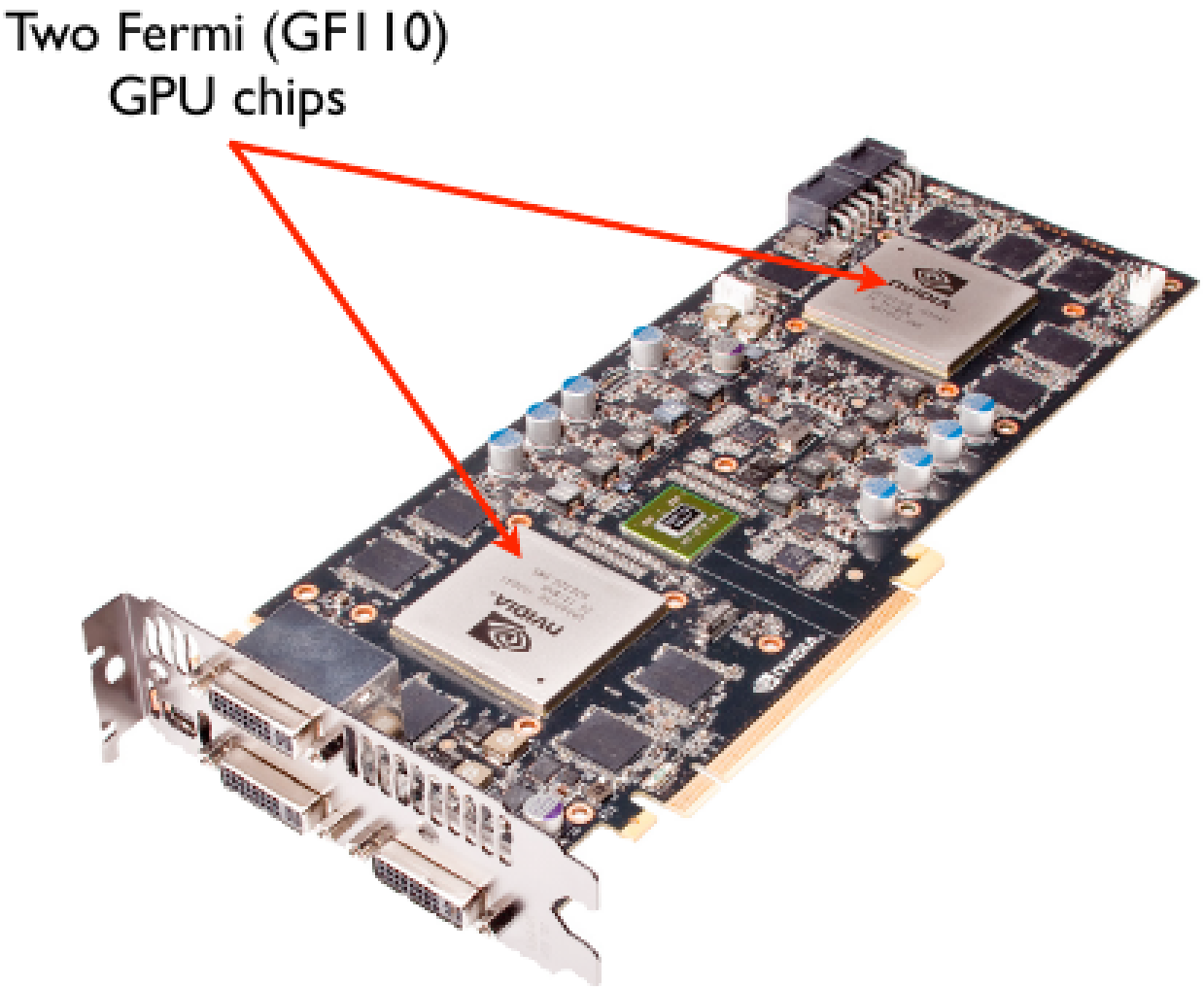
- Cores: 1024, Processor Clock: 1215 MHz, Memory: 3 GB

Example: NVIDIA GTX 590 (cont.)

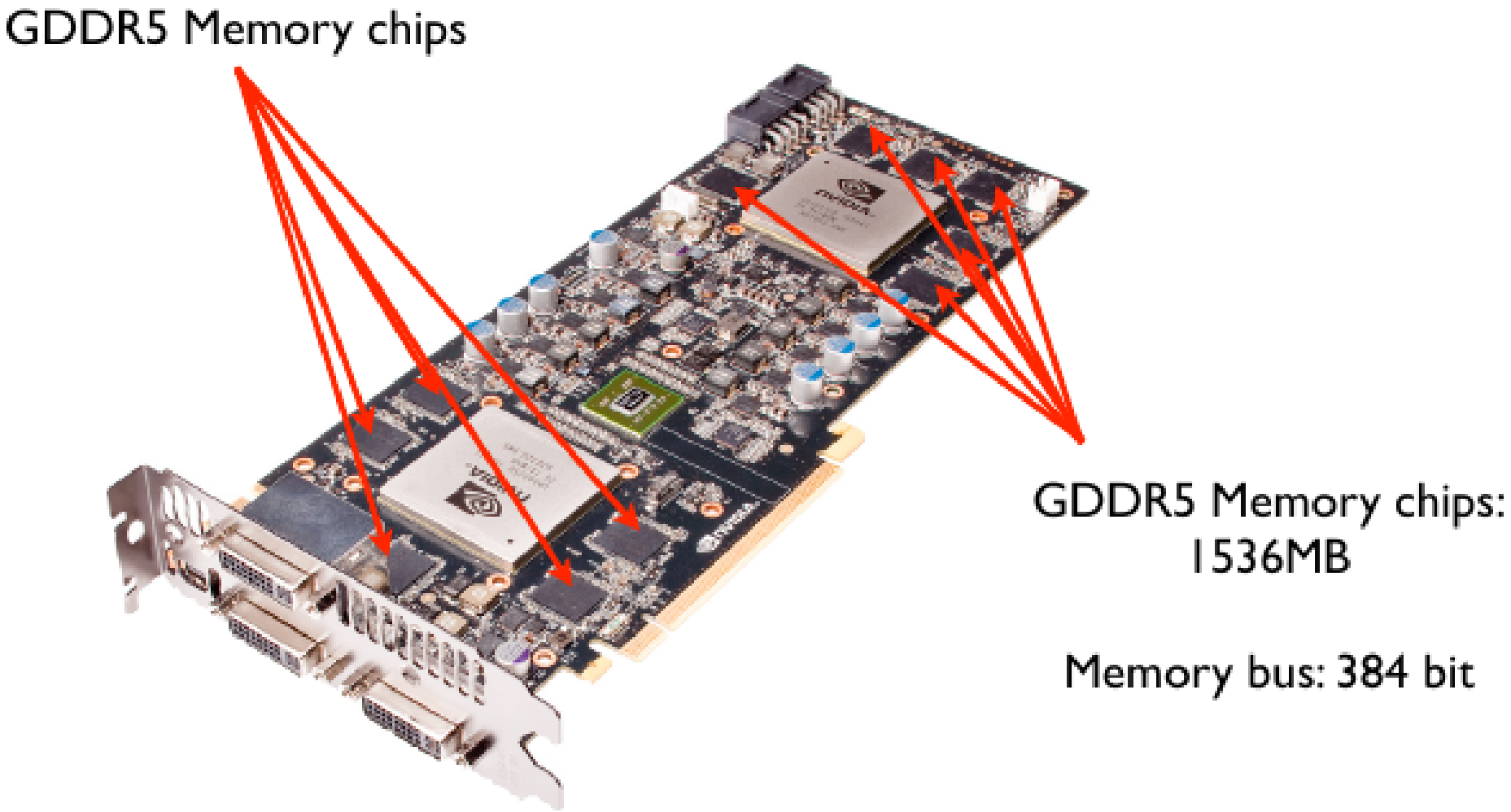


PCI Express 2.0 interface
[all data exchange between
CPU & GPU crosses this bus]

Example: NVIDIA GTX 590 (cont.)



Example: NVIDIA GTX 590 (cont.)



Example: NVIDIA GTX 590 (cont.)

Fermi: GPU <> GDDR5
memory bandwidth ~200 GB/s

PCI Express 2.0 bus
bandwidth ~6 GB/s

Xeon 5520 <> DDR3
memory transfer rate < 25 GB/s

Ethernet bandwidth:
~0.2 GB/s

USB3 bandwidth:
< 0.5 GB/s

SATA3 HD bandwidth:
< 0.8 GB/s

Figure 13: The Bottleneck

Trying it Out

Access Details

```
ssh 121.52.146.108 -l cdc-username -XY
```

where, **username** is your FAST-roll number
and **password** is same as username (one time only)
Example session with password = p116003 would be:

```
ssh 121.52.146.108 -l cdc-p116003 -XY
```

Introduction to GPGPU's

Open Computing Language (**OpenCL**)

- An **open** standard from **Khronos**; the makers of OpenGL (v1.0 Release December 2008)
- Cross Platform/Vendor/Architecture (CPU, GPU, DSP, FPGA, ...)
- GPU Giants (NVIDIA + AMD)
- Other Major Players (Apple, Intel, Qualcomm, Samsung, Xilinx, Altera)
- OpenCL is a {Standard, Language (based on C99), API/Library, Runtime SIMT based Compilation and Execution Environment}
- Two-way inter-operatable with OpenGL

Compute Unified Device Architecture (**CUDA**)

- **Proprietary** platform/API, released by **NVIDIA** (v1.0 released in January 2007)
- Handles only one platform/vendor, i.e., NVIDIA manufactured Graphic Cards
- CUDA is a {Language, API/Library, Non-runtime compilation environment, and Runtime execution environment}
- Inter-operatability with OpenGL is one-way (OpenGL can view CUDA buffers, but CUDA cannot view OpenGL buffers)

Introduction to GPGPU's (cont.)

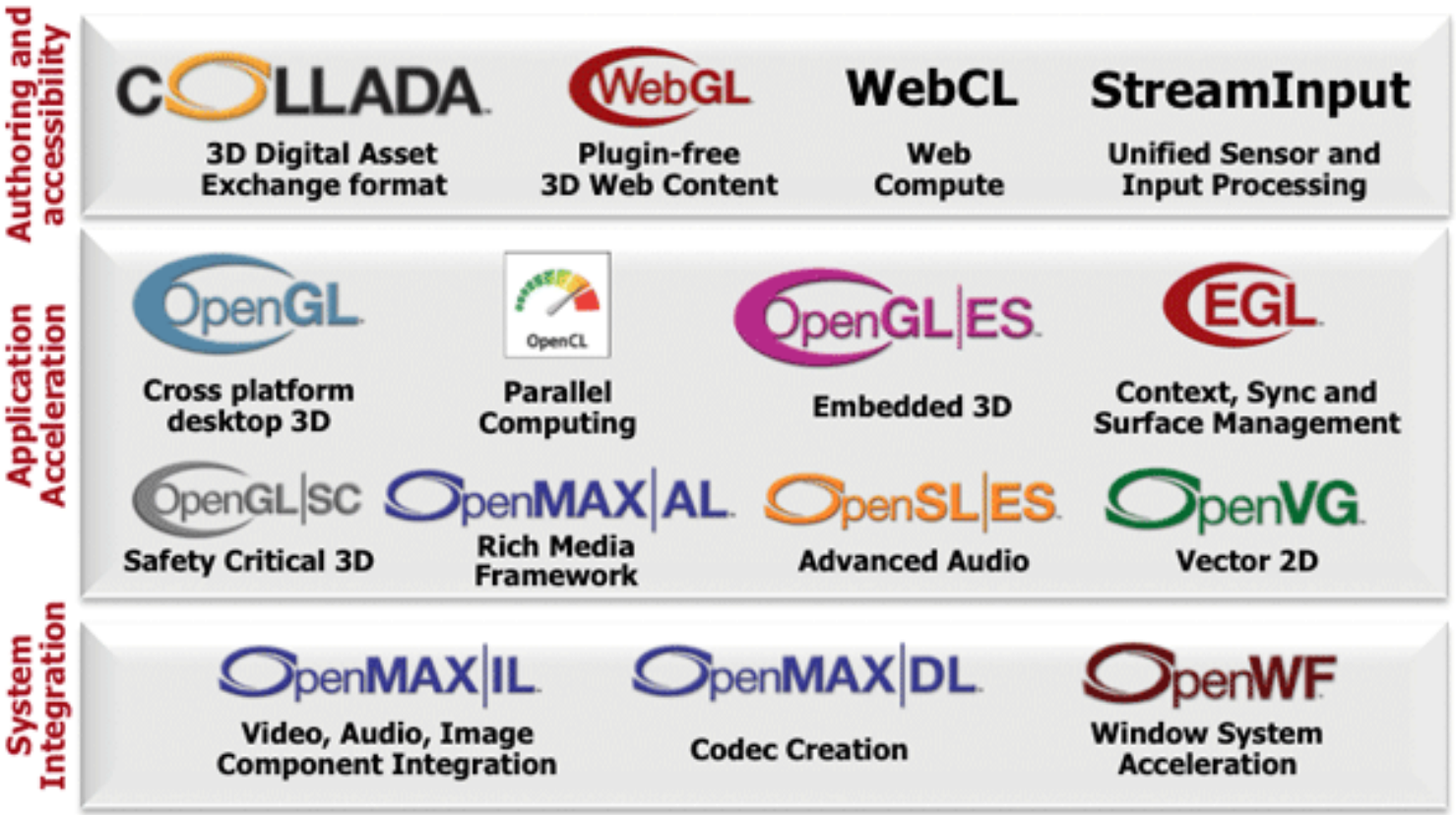
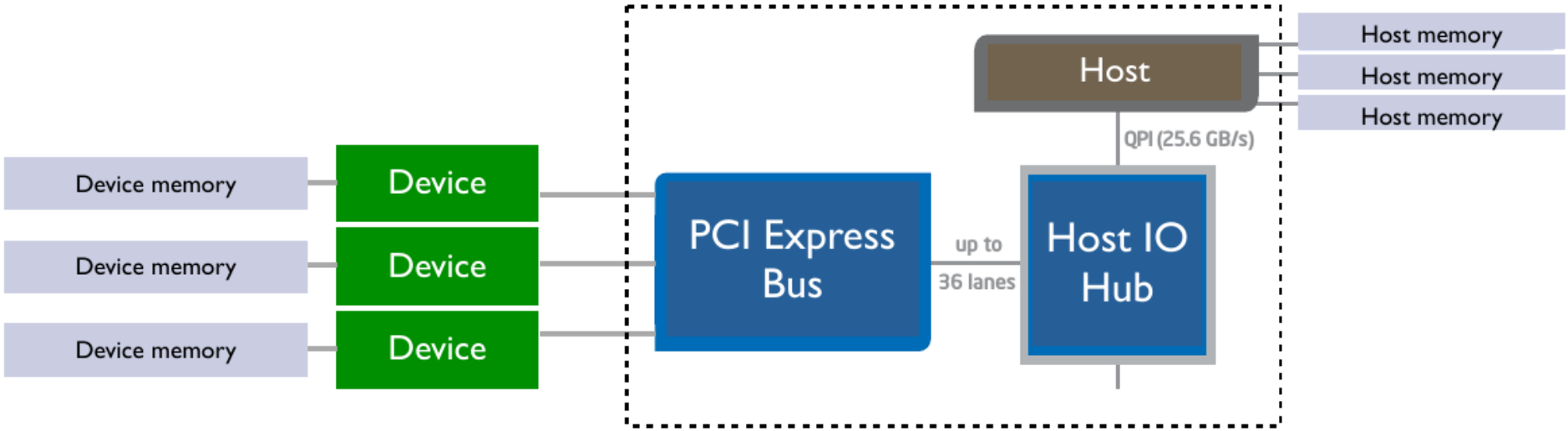


Figure 14: Khronos Group Open Specifications

[Img] http://www.khronos.org/about

Data Migration



Data Migration (cont.)

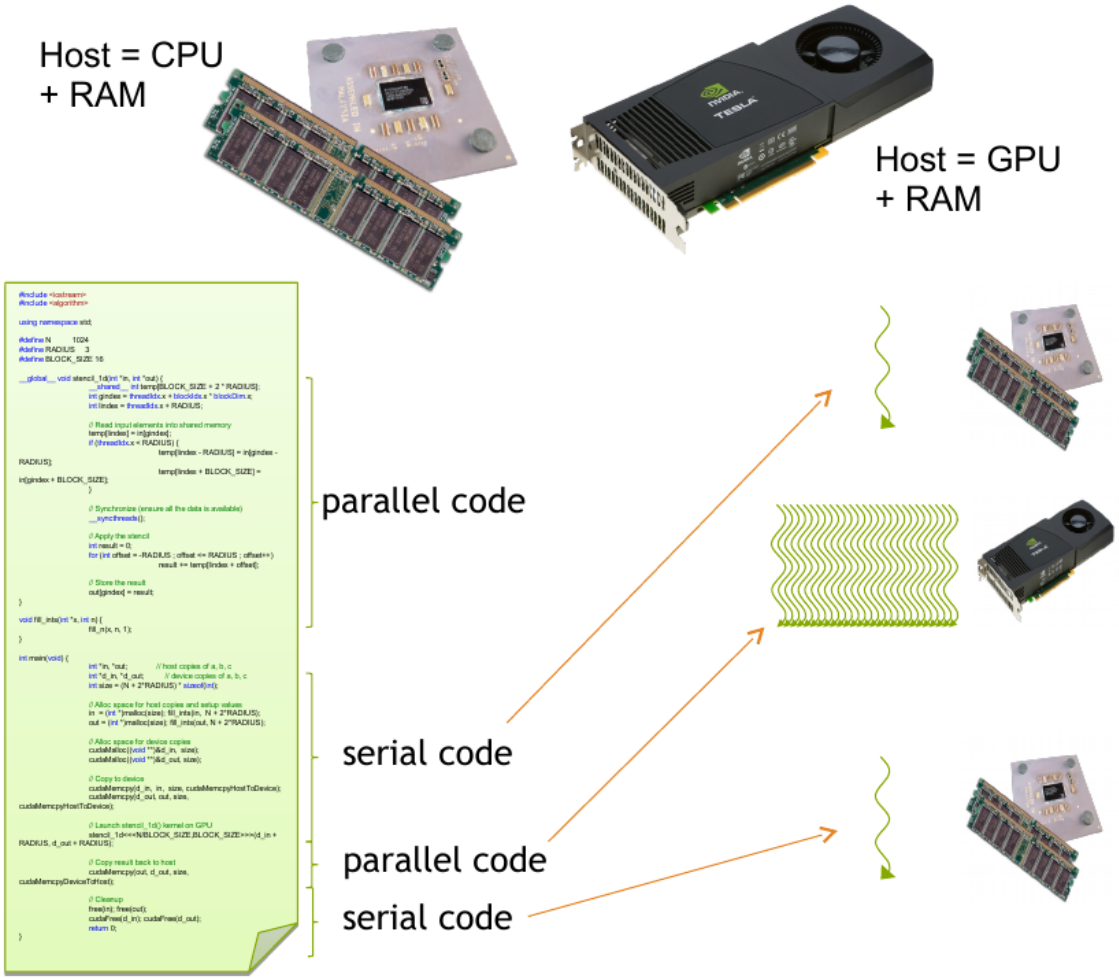
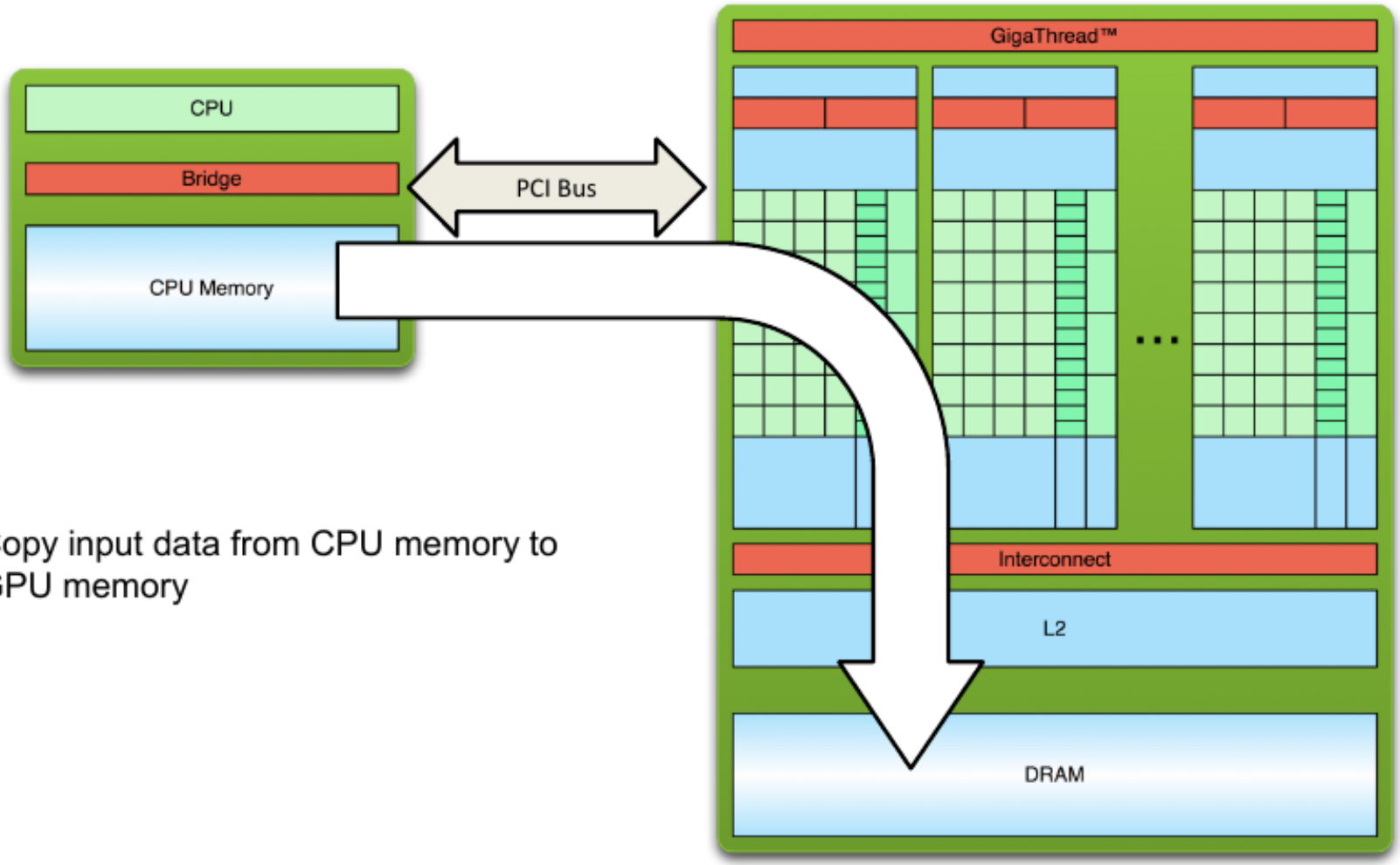


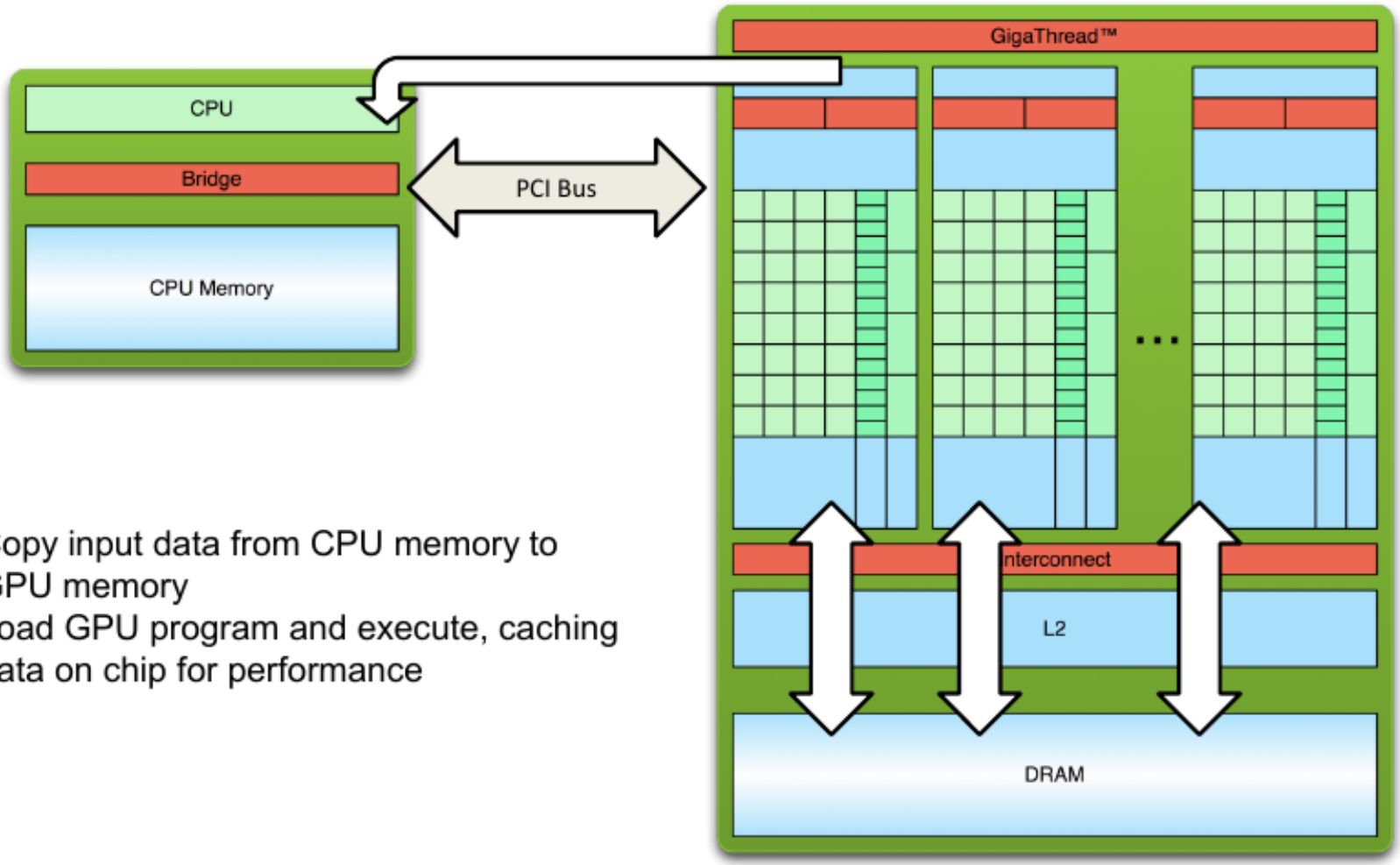
Figure 15: Porting portions of your code to GPU

Data Migration (cont.)



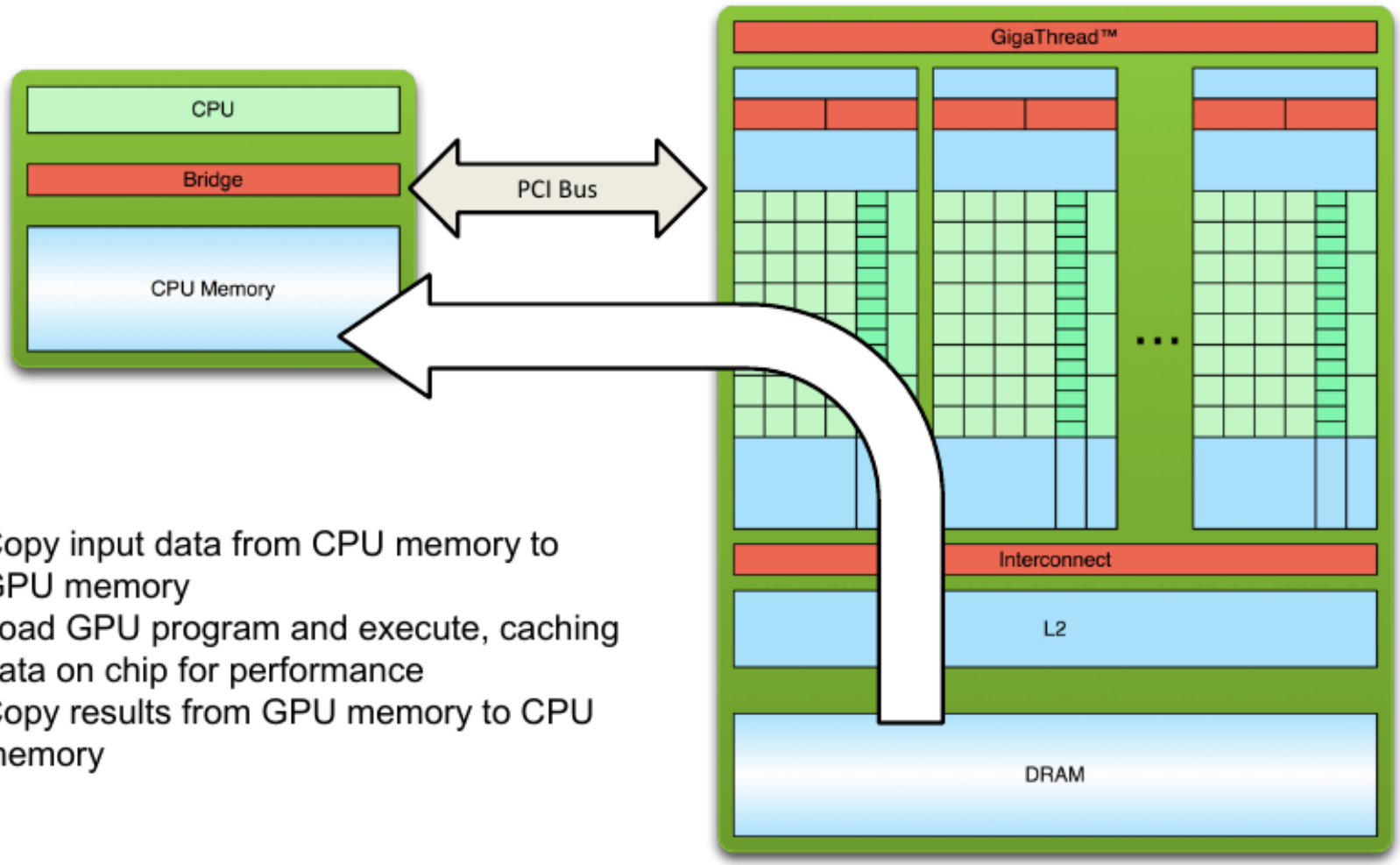
1. Copy input data from CPU memory to GPU memory

Data Migration (cont.)



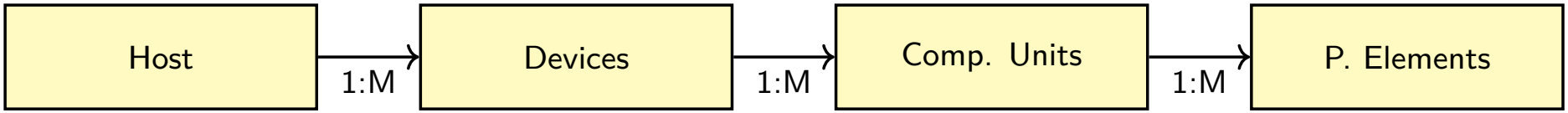
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

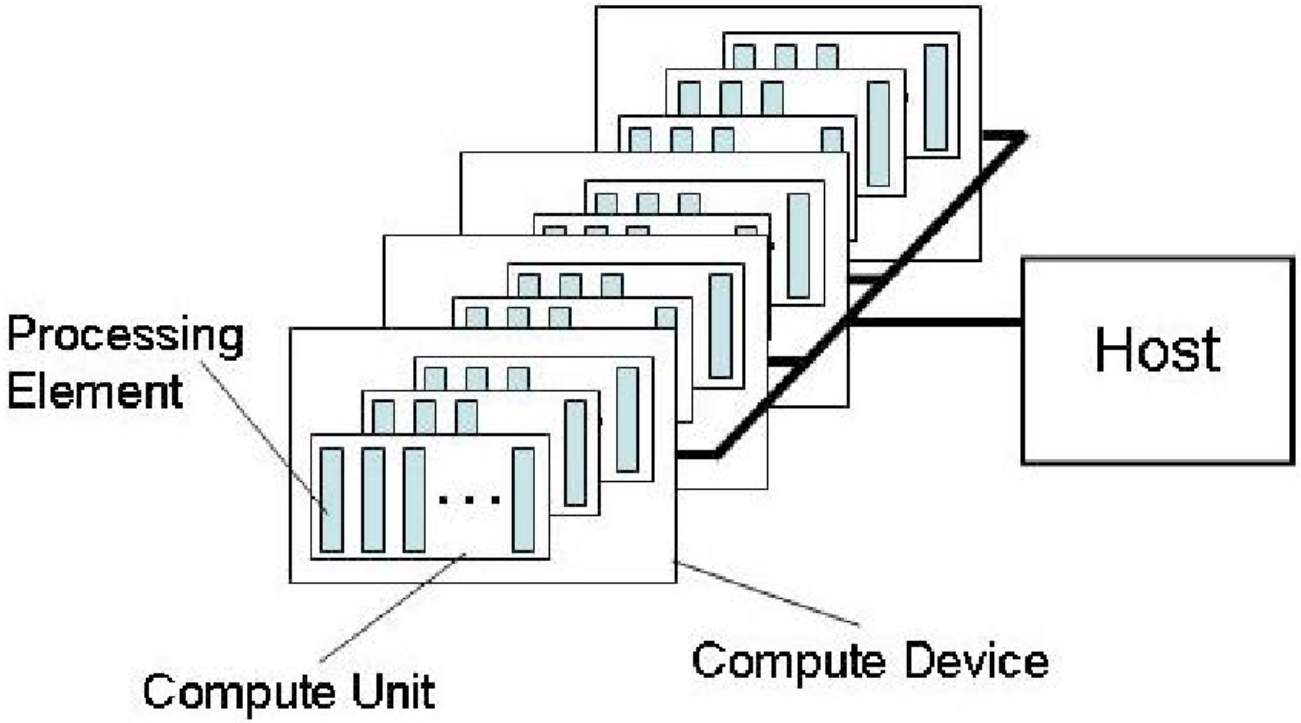
Data Migration (cont.)



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

OpenCL Specification: Platform Model





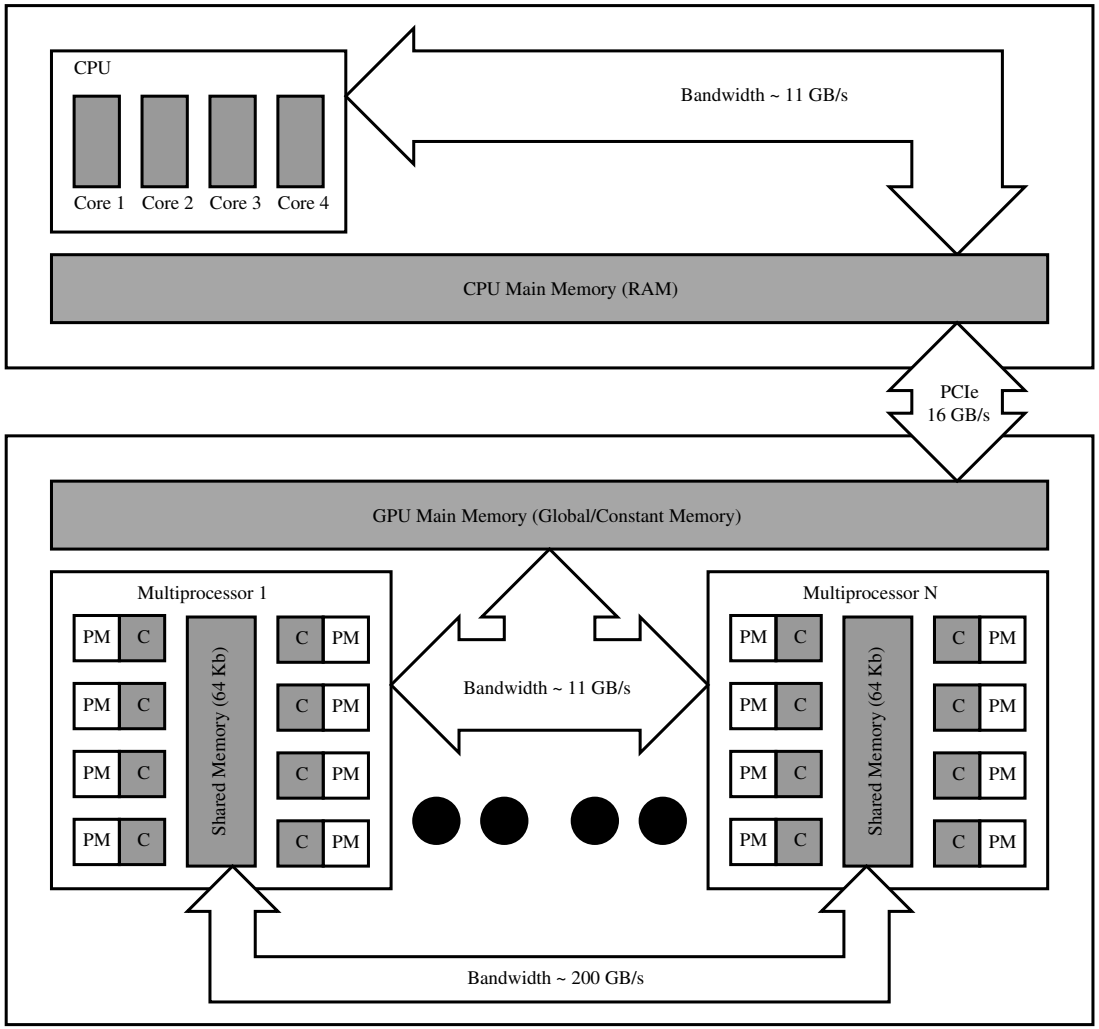
- NVIDIA-SDK: oclDeviceQuery

OpenCL Specification: Platform Model (cont.)

NVIDIA/CUDA	AMD/OpenCL
CUDA Core	Processing Element
Streaming Multiprocessor	Compute Unit
GPU Device	Compute Device

Table 4: Terminology Difference between CUDA and OpenCL

Memory Model



Global Memory

- Permits R/W access to all work-items in all work-groups
- May be cached (hardware dependent)

Constant Memory

- Belongs to same address space as global memory
- Host initializes and places contents into it
- Contents remain constant (R/O) to work-items

Memory Model (cont.)

Local (Shared) Memory

- Can be used to allocate variables that are shared by all work-items in a particular work-group.
- Have much faster access speed than global memory (./oclBandwidthTest)

Private Memory (Registers)

- Can be used to allocate variables for a given work-item
- Visibility scope belongs to work-item only

Memory Address Space Specifiers

Declaration (OpenCL)	Declaration (CUDA)	Memory	Scope	Lifetime
__private__	none	register	thread	kernel
__local__	__shared__	shared	block	kernel
__global__	__device__	global	grid	application
__constant__	__constant__	constant	grid	application

Memory Model (cont.)

```
CL_DEVICE_MAX_MEM_ALLOC_SIZE:        2879 MByte
CL_DEVICE_GLOBAL_MEM_SIZE:           11519 MByte
CL_DEVICE_LOCAL_MEM_SIZE:            48 KByte
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 64 KByte
CL_DEVICE_REGISTERS_PER_BLOCK_NV:    65536
```

OpenCL Specification: Execution Model

Host Cuda/OpenCL applications have a single host program (that runs on the CPU. It is responsible for preparing the execution environment for the GPU.

Kernel A host program launches a number of instances of a kernel.

Command Queue All kernel management related tasks (performed by the CPU) such as memory transfer from device RAM to CPU RAM, from CPU RAM to device RAM, launching kernels, etc. are placed as commands on the command queue.

Work Item Each instance of a kernel is known as a work-item.

Work Group Arrangements of work-items into logical groups by developer.

Warp Arrangements of work-items into physical groups by thread scheduler.

Index Space An N-Dimensional range of values (N can only be 1, 2, or 3) that is used for identifying a thread uniquely. Range of values represented by integer arrays of length N.

OpenCL Specification: Execution Model (cont.)

NVIDIA/CUDA	AMD/OpenCL
Kernel	Kernel
Stream	Command Queue
Thread	Work Item
Block	Work Group
Warp	Warp
Grid	NDrange

Table 5: Execution Model Terminology Differences between CUDA and OpenCL

OpenCL Specification: Execution Model (cont.)

Index Space

Global Size Size of NDRange integer array along each dimension (G_x, G_y, G_z)

Global ID Identify of a thread in global context along each dimension (g_x, g_y, g_z). Thus, total range of threads would be $[0, 1, \ldots, G_x - 1], [0, 1, \ldots, G_y - 1], [0, 1, \ldots, G_z - 1]$

Group Size Total number of work-groups along each dimension (W_x, W_y, W_z)

Group ID Identify of each work-group along each dimension (w_x, w_y, w_z)

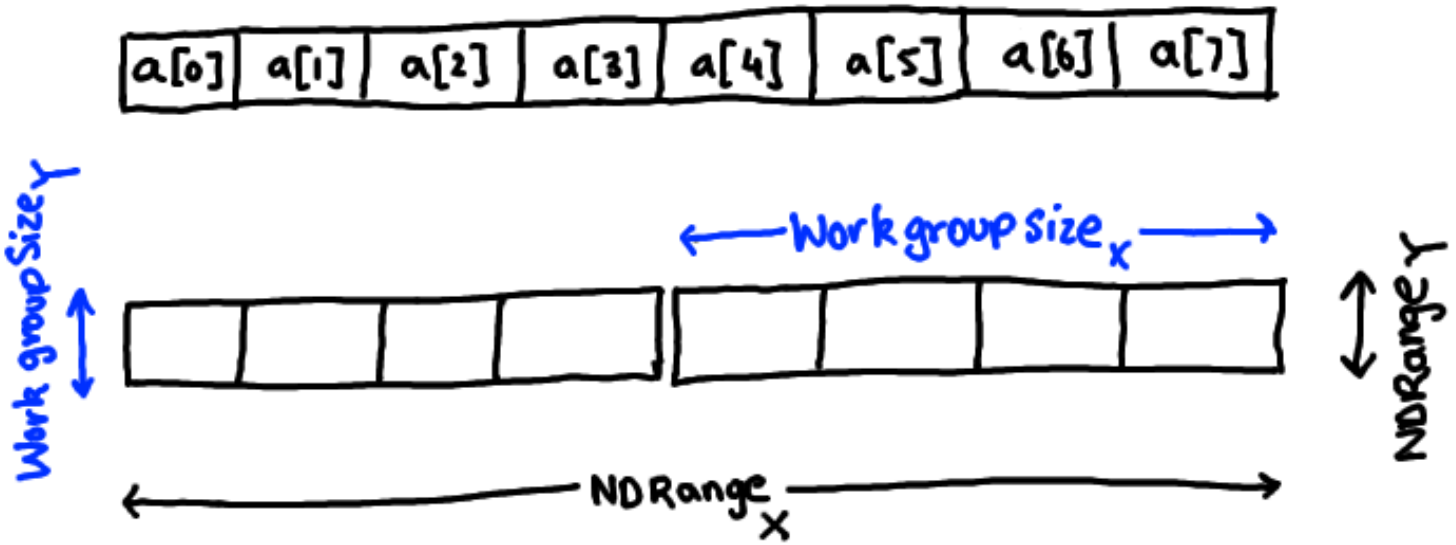
Local Size Size of work-group along each dimension (L_x, L_y, L_z). For GPU programming, the Local Size must **evenly** divide the Global Size.

```
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:        3
CL_DEVICE_MAX_WORK_ITEM_SIZES:      1024 / 1024 / 64 (512/512/32)
CL_DEVICE_MAX_WORK_GROUP_SIZE:      1024 (512)
```

Local ID Identify of a thread in local context (of work-group) along each dimension. Thus, total range of threads would be $[0, 1, \ldots, L_x - 1], [0, 1, \ldots, L_y - 1], [0, 1, \ldots, L_z - 1]$ for each work-group.

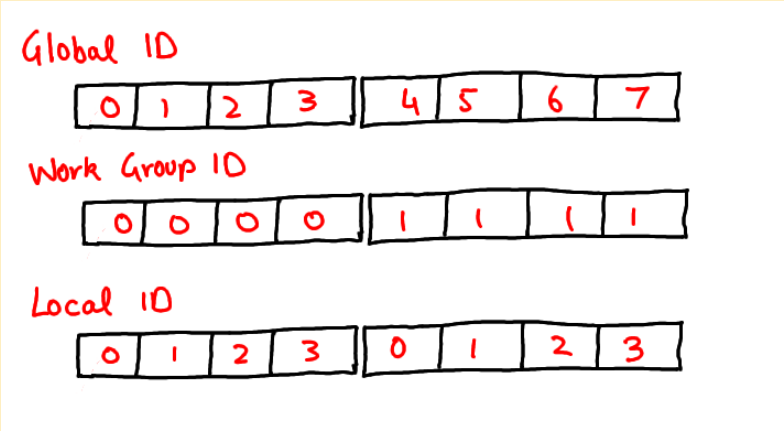
OpenCL Specification: Execution Model (cont.)

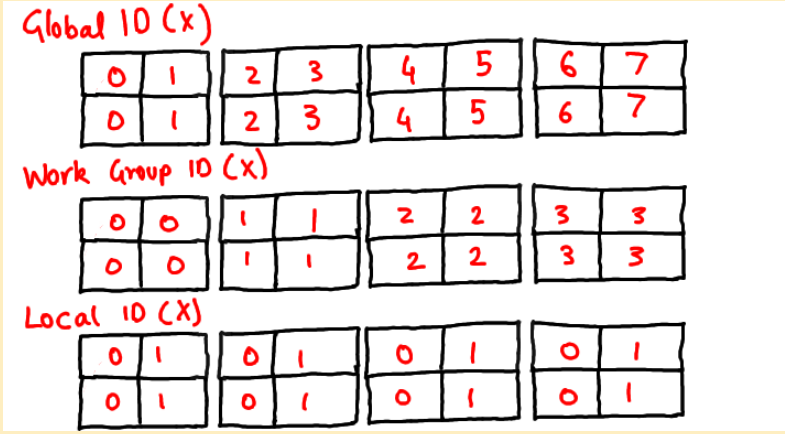
```
int get_work_dim();       // Return number of dimensions
int get_num_groups(int);  // Return number of work-groups W_{x,y,z}
int get_global_size(int); // Return NDRange Size G_{x,y,z}
int get_local_size(int);  // L_{x,y,z} = G_{x,y,z}/W_{x,y,z}
int get_global_id(int);   // g_{x,y,z} = w_{x,y,z} × L_{x,y,z} + l_{x,y,z}
                          // g_{x,y,z} = w_{x,y,z} × L_{x,y,z} + l_{x,y,z} + o_{x,y,z}
int get_group_id(int);    // w_{x,y,z} = g_{x,y,z}/L_{x,y,z}
int get_local_id(int);    // l_{x,y,z} = g_{x,y,z}%L_{x,y,z}
```

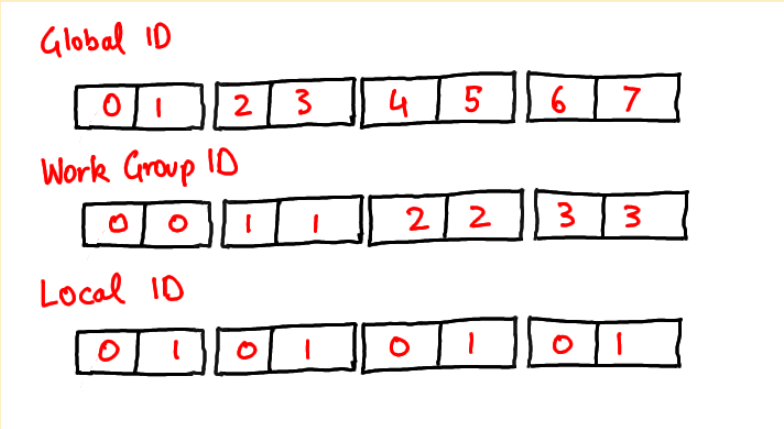


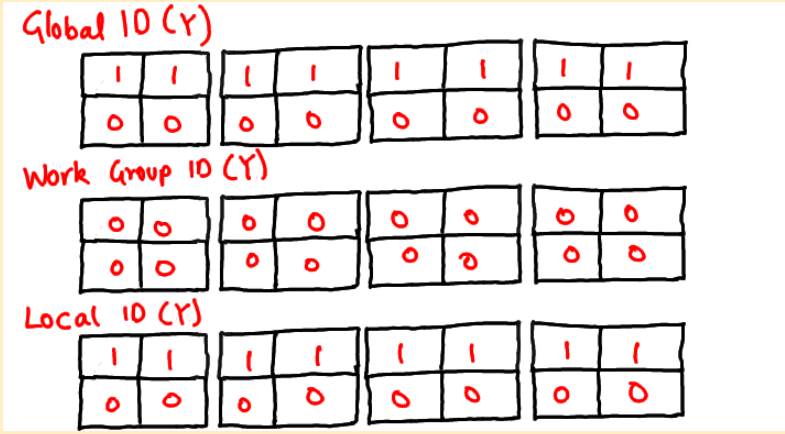
OpenCL Specification: Execution Model (cont.)

Examples of Index Space









OpenCL Specification: Execution Model (cont.)

The OpenCL Context

The execution environment created by host for executing the kernels. Comprises of:

Devices Collection of OpenCL devices to be used by the host

Kernels OpenCL functions that would run on OpenCL devices

Program Objects The kernel source code and/or executables. These will be built at runtime **Within the Host Program**. Source code can be specified through three means:

- Regular string definition within code
- Read line-by-line from file
- **generated** dynamically inside the host program through string manipulation

Memory Objects Sets of objects in memory that are visible to OpenCL devices and contain values that can be be operated upon by instances of kernels.

OpenCL Specification: Execution Model (cont.)

Command Queue

- Interaction between host and OpenCL device occurs through commands posted on the command queue. Types of commands supported:
  - Kernel execution commands
  - Memory transfer commands (from device RAM to CPU RAM, and CPU RAM to device RAM)
  - Synchronization commands
- Multiple command queues can be created for doing things simultaneously.
- Typical command sequence (copy from CPU RAM to Device RAM, Execute Kernel, copy from Device RAM to CPU RAM)
- **In-Order Execution**: Commands launched in order in which they appear in command queue. New command only starts when old one is completed.
- **Out-of-Order Execution**: Commands placed in order, but new commands can start executing even if previous one is not yet completed.

OpenCL Specification: Memory Model

Memory Objects

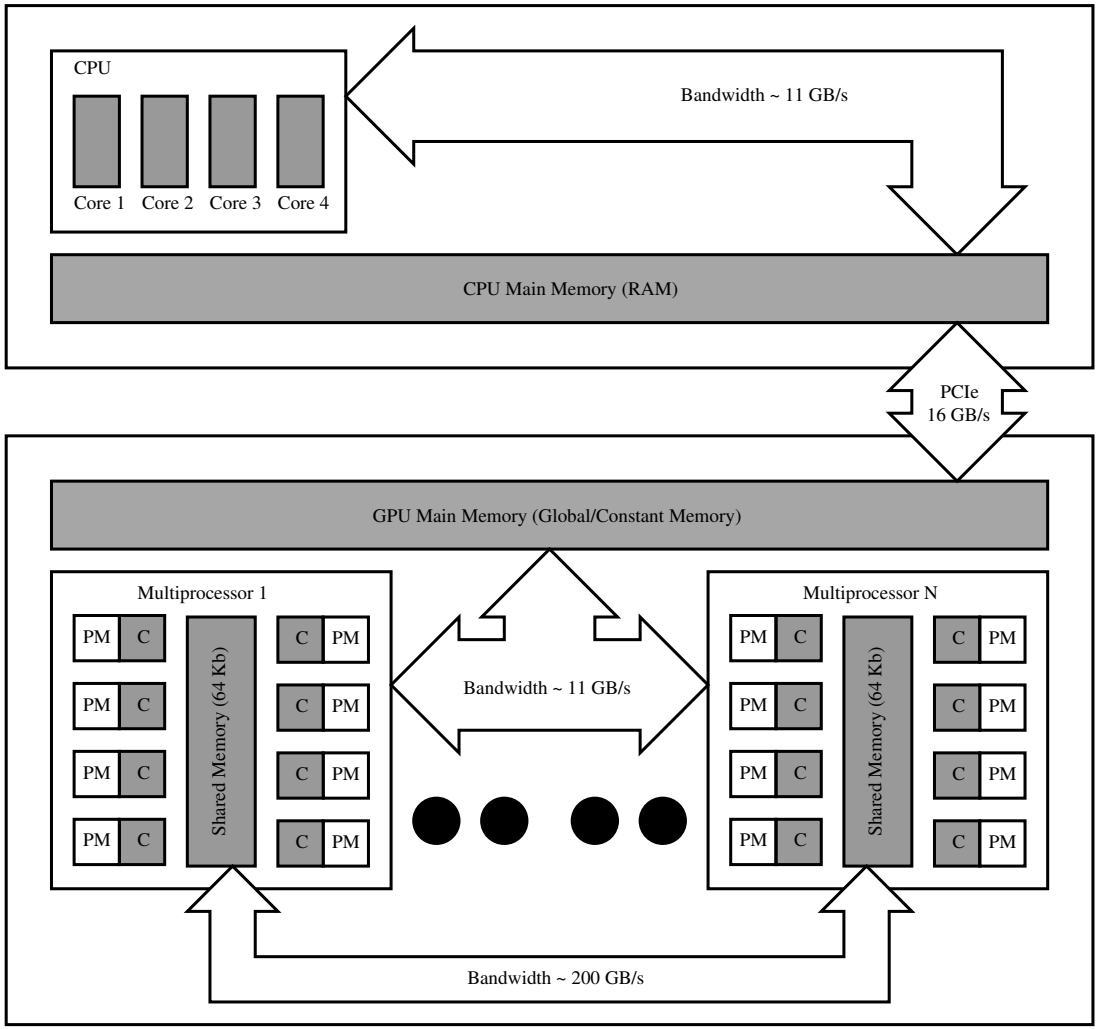
Buffer Objects Contiguous block of memory (basically an array).

- Can be present in either host RAM, or device RAM
- Can be accessed using pointers
- Can be read-only, or write-only, or read-write
- May be cached (hardware dependent)

Image Objects Used for storing images

- Can be present in GPU Texture Memory
- Can be accessed as values
- Can be read-only, or write-only
- Can be cached (irrespective of hardware)

OpenCL Specification: Memory Model



Global Memory

- Permits R/W access to all work-items in all work-groups
- May be cached (hardware dependent)

Constant Memory

- Belongs to same address space as global memory
- Host initializes and places contents into it
- Contents remain constant (R/O) to work-items

OpenCL Specification: Memory Model (cont.)

Local (Shared) Memory

- Can be used to allocate variables that are shared by all work-items in a particular work-group.
- Have much faster access speed than global memory (./oclBandwidthTest)

Private Memory

- Can be used to allocate variables for a given work-item
- Visibility scope belongs to work-item only

```
CL_DEVICE_MAX_MEM_ALLOC_SIZE:        2879 MByte
CL_DEVICE_GLOBAL_MEM_SIZE:           11519 MByte
CL_DEVICE_LOCAL_MEM_SIZE:            48 KByte
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 64 KByte
CL_DEVICE_REGISTERS_PER_BLOCK_NV:    65536
```

OpenCL Specification: Memory Model (cont.)

Memory Transfers

- Bandwidth: Varies for each level of memory hierarchy (./oclBandwidthTest: Choice is obvious.) Below: Output for NVIDIA K40:

```
Host to Device Bandwidth, 1 Device(s), Paged memory, direct access
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33554432                 6944.4

Device to Host Bandwidth, 1 Device(s), Paged memory, direct access
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33554432                 5067.5

Device to Device Bandwidth, 1 Device(s)
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33554432                 177162.9
```

- Optimization Techniques: Multiple Command Queues, Pinned Memory

Squaring an Array

C Code

```
void squareCPU(float *a, int size) {
   for (int x = 0; x < size; x++)
      a[x] *= a[x];
}
```

OpenCL Kernel Code

```
__kernel void squareCPU(__global float *a) {
   int x = get_global_id(0);
   a[x] *= a[x];
}
```

Squaring an Array (cont.)

CUDA Kernel Code

```
__global void hello(float *a) {
   int x = threadIdx.x;
   a[x] *= a[x];
}
```

OpenCL C (Restricted & Customized C99)

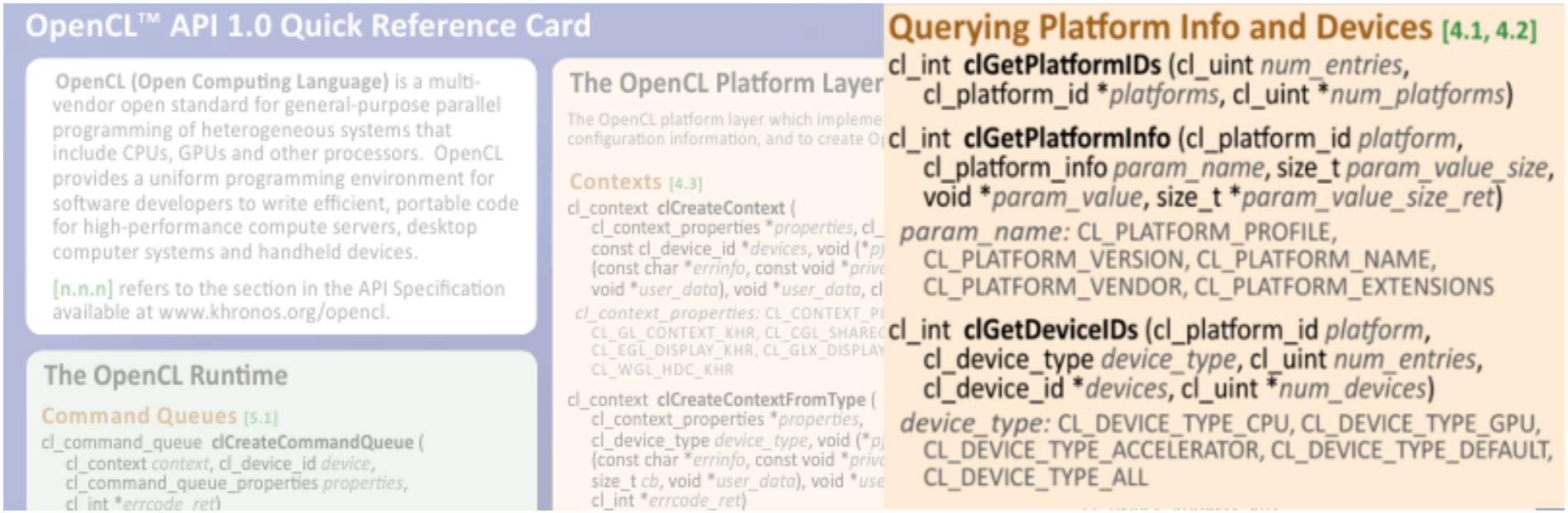
- Address Space pointers to Memory __global, __local, __constant, __private
- Entry points to functions marked by __kernel
- No function pointers
- No variable length arrays
- No recursion
- Support for Vector Types

Squaring an Array: Host Code

Figure 16: Host - Device Access Process (Step 0)

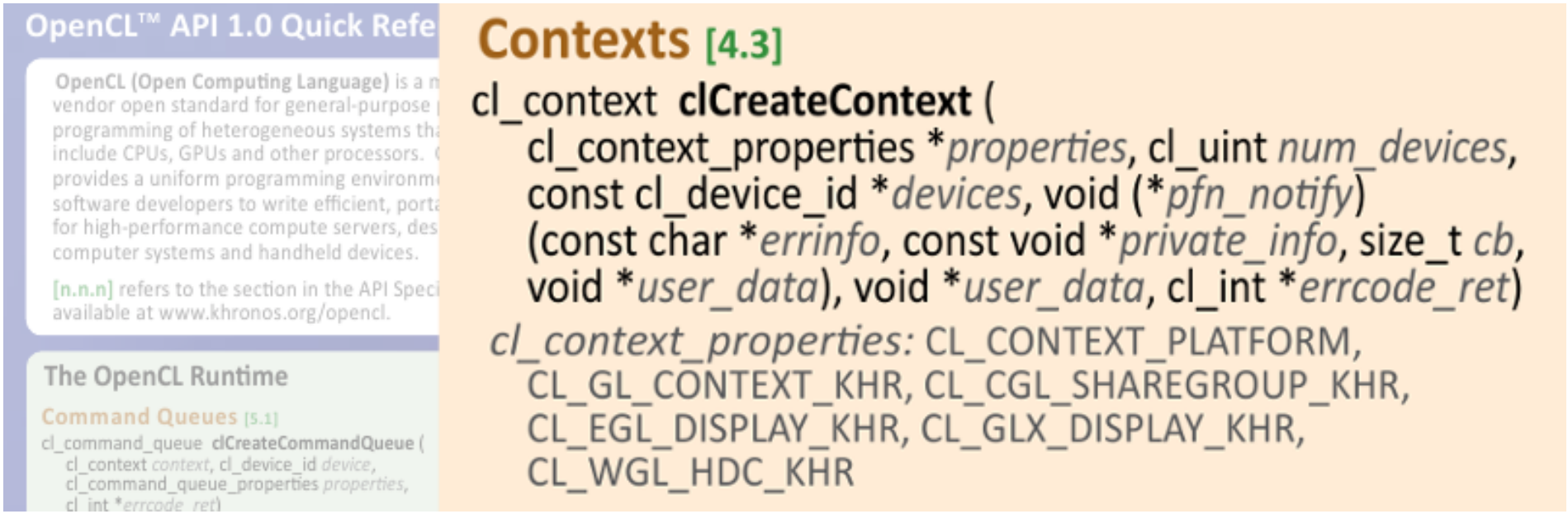
```
39    // Platform Calls
40    cl_platform_id  platform;
41    cl_uint ret_num_platforms;
42    clGetPlatformIDs(1, &platform, &ret_num_platforms);
43
44    // Device Calls
45    cl_device_id device;
46    cl_uint ret_num_devices;
47    clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1, &device, &ret_num_devices);
48
```

Squaring an Array: Host Code (cont.)



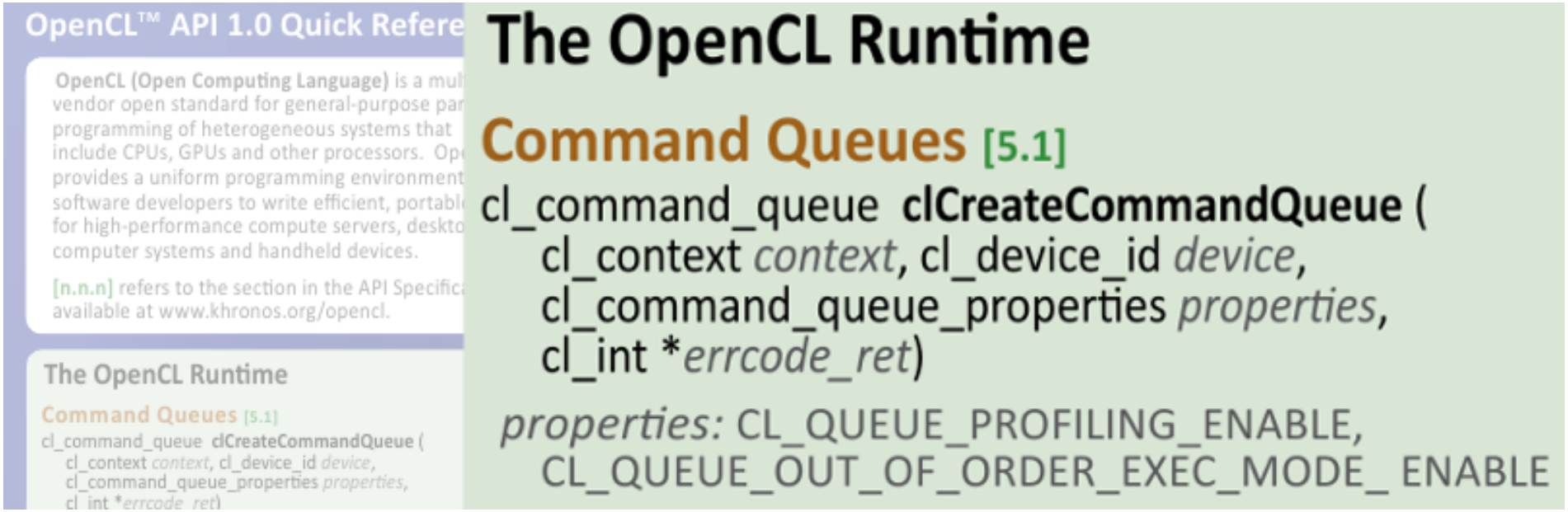

```
44      // Device Calls
45  2   cl_device_id device;
46      cl_uint ret_num_devices;
47      clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1, &device, &ret_num_devices);
48
49  3   // Context Creation
50      cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
51
```

Squaring an Array: Host Code (cont.)




```
48
49  // Context Creation
50  cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
51
52  // Command Queue Creation
53  cl_command_queue command_queue = clCreateCommandQueue(context, device, 0, NULL);
54
```

Squaring an Array: Host Code (cont.)




```
52  // Command Queue Creation
53  cl_command_queue command_queue = clCreateCommandQueue(context, device, 0, NULL);
54
55  // Fetch Program
56  const char* source_code =   "__kernel void square(__global float *a) { \
57                                  int x = get_global_id(0);\
58                                  a[x] *= a[x];\
59                              }";
60  cl_program program = clCreateProgramWithSource(context, 1, &source_code, NULL, NULL);
```

Squaring an Array: Host Code (cont.)

OpenCL™ API 1.0 Quick Reference C

OpenCL (Open Computing Language) is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices.

[n.n.n] refers to the section in the API Specification available at www.khronos.org/opencl.

The OpenCL Runtime

Command Queues [5.1]

```
cl_command_queue clCreateCommandQueue (
    cl_context context, cl_device_id device,
    cl_command_queue_properties properties,
    cl_int *errcode_ret)
```

Create Program Objects [5.4.1]

```
cl_program clCreateProgramWithSource (
    cl_context context, cl_uint count, const char **strings,
    const size_t *lengths, cl_int *errcode_ret)

cl_program clCreateProgramWithBinary (
    cl_context context, cl_uint num_devices,
    const cl_device_id *device_list, const size_t *lengths,
    const unsigned char **binaries, cl_int *binary_status,
    cl_int *errcode_ret)
```

Figure 17: Host - Device Access Process (Step 4+5)

Squaring an Array: Host Code (cont.)

```
61
62      // Build Program
63      clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
64
65      // Create Kernel
66      cl_kernel kernel = clCreateKernel(program, "square", NULL);
67
```

OpenCL™ API 1.0 Quick Reference Card

OpenCL (Open Computing Language) is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices.

Build Program Executable [5.4.2]

cl_int **clBuildProgram** (cl_program *program*,
cl_uint *num_devices*, const cl_device_id **device_list*,
const char **options*, void *(*pfn_notify)*
*(*cl_program, void **user_data)*, void **user_data*)

OpenCL™ API 1.0 Quick Referenc

OpenCL (Open Computing Language) is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. OpenCL provides a uniform programming environment for

Create Kernel Queries [5.5.1]

cl_kernel **clCreateKernel** (cl_program *program*,
const char **kernel_name*, cl_int **errcode_ret*)

Squaring an Array: Host Code (cont.)

Source Code Errors ??

```
char buffer[2048];
clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
                      sizeof(buffer), buffer, NULL);
printf("Build Log: %s\n", buffer);
```

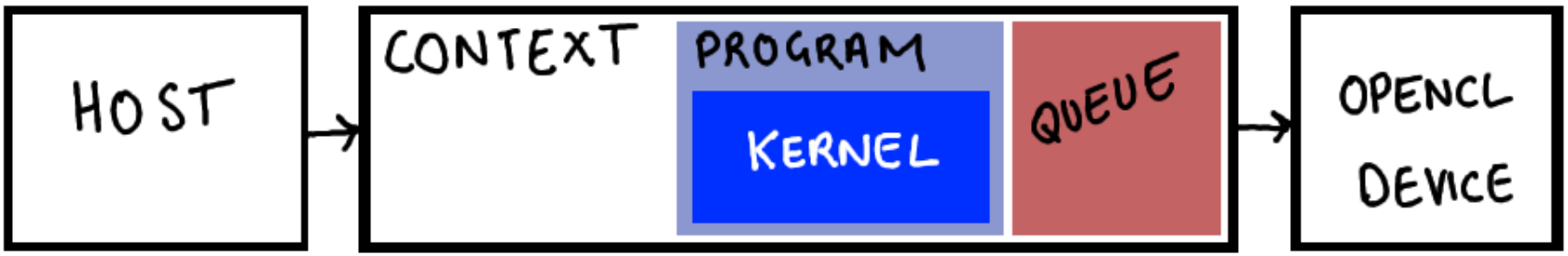
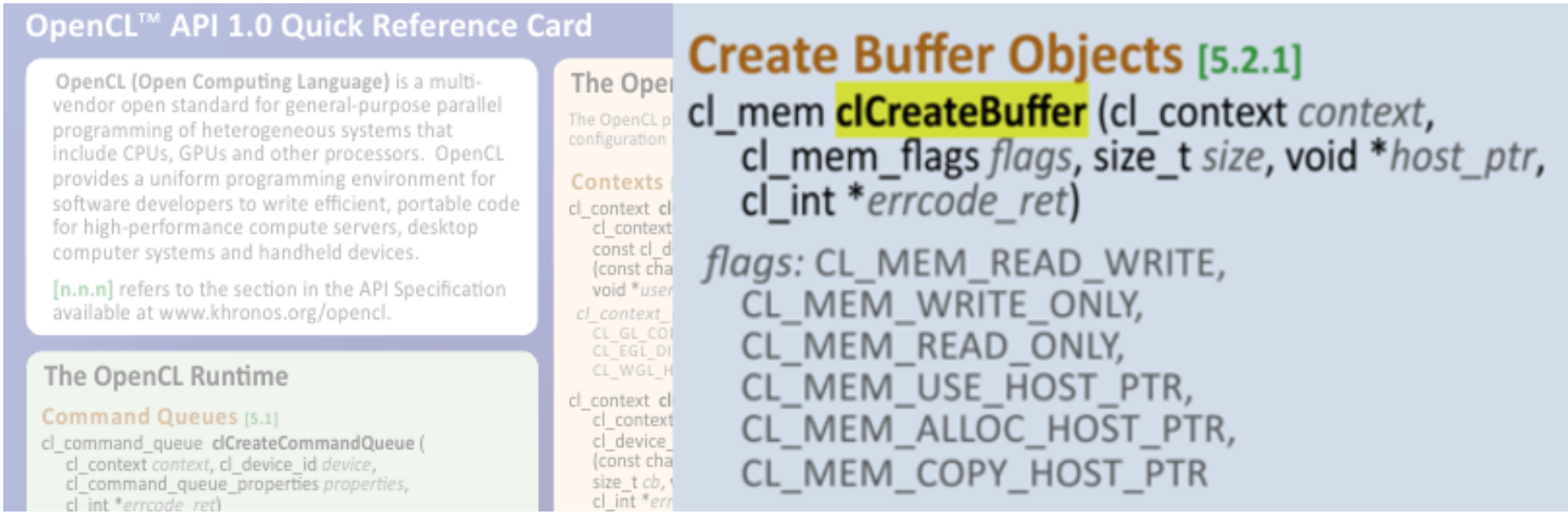


Figure 18: Host - Device Access Process (Step 7)

```
67
68  // Create Memories
69  cl_mem data = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*size, NULL, NULL);
70
```

Squaring an Array: Host Code (cont.)



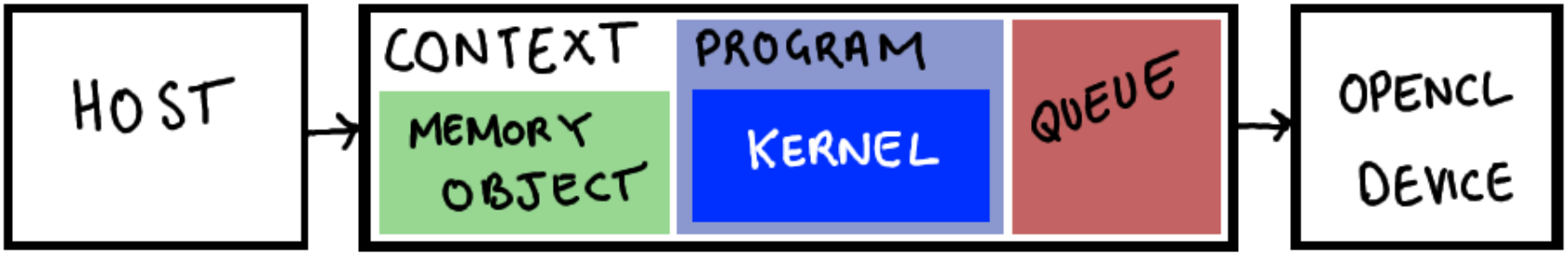


Figure 19: Host - Device Access Process (Step 7)

Squaring an Array: Host Code (cont.)

```
71  // Transfer Data
72  clEnqueueWriteBuffer(command_queue, data, CL_TRUE, 0, sizeof(float)*size, a, 0, NULL, NULL);
73
```

```
82
83  // Get Data Back
84  clEnqueueReadBuffer(command_queue, data, CL_TRUE, 0, sizeof(float)*size, a, 0, NULL, NULL);
```

OpenCL™ API 1.0 Quick Reference Card

OpenCL (Open Computing Language) is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices.

[n.n.n] refers to the section in the API Specification available at www.khronos.org/opencl.

The OpenCL Runtime

Command Queues [5.1]

cl_command_queue clCreateCommandQueue (
cl_context *context*, cl_device_id *device*,
cl_command_queue_properties *properties*,
cl_int *errcode_ret*)

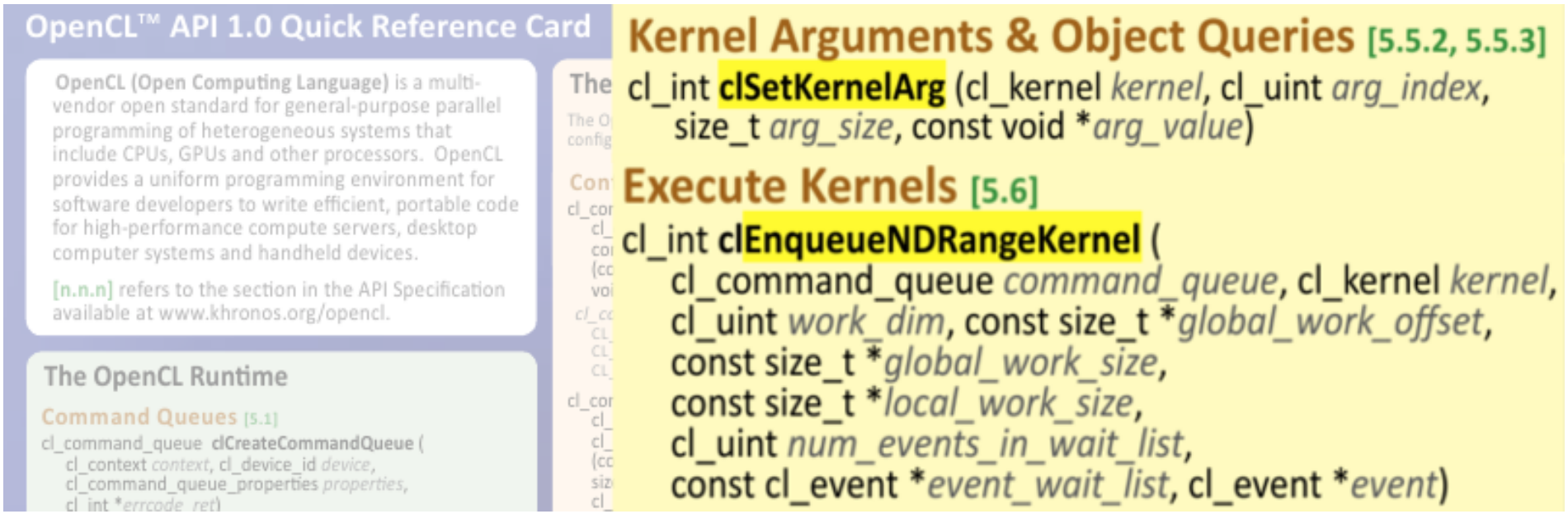
Read, Write, Copy Buffer Objects [5.2.2 - 5.2.3]

cl_int **clEnqueueReadBuffer** (
cl_command_queue *command_queue*, cl_mem *buffer*,
cl_bool *blocking_read*, size_t *offset*, size_t *cb*,
void *ptr, cl_uint *num_events_in_wait_list*,
const cl_event *event_wait_list, cl_event *event)

cl_int **clEnqueueWriteBuffer** (
cl_command_queue *command_queue*, cl_mem *buffer*,
cl_bool *blocking_write*, size_t *offset*, size_t *cb*,
const void *ptr, cl_uint *num_events_in_wait_list*,
const cl_event *event_wait_list, cl_event *event)

Squaring an Array: Host Code (cont.)

```
74    // Set Kernel Arguments
75    clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&data);
76
77    // Run Kernel
78    size_t globalSize[1], localSize[1];
79    globalSize[0]   = size;
80    localSize[0]    = 16;
81    clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, globalSize, localSize, 0, NULL, NULL);
```



Building up towards Hello World

```
int main()
{
  printf("hello world!\n");
  return 0;
}
```

Compilation

- nvcc hello_world.cu
- ./a.out

Building up towards Hello World: Inserting GPU Code

```
__global__ void myKernel(void)   // indicates function runs on
{                                // device and is called from
}                                // host code

int main()
{
  myKernel<<<1,1>>>();           // like function call, but
                                 // with more information
                                 // 1st digit number of blocks
                                 // 2nd digit threads per block
  printf("hello world!\n");
  return 0;
}
```

- nvcc will separate source code into host and device components
  - Device functions processed by NVIDIA compiler
  - Host functions processed by standard host compiler

```
/* a, b, c are pointers to memory location on the device */
__global__ void add(int *a, int *b, int *c)
{  *c = *a + *b;
}
int main()
{  int a=2, b=7, c, *da, *db, *dc;

   cudaMalloc((void **)&da, sizeof(int)); // allocate
   cudaMalloc((void **)&db, sizeof(int)); // on device
   cudaMalloc((void **)&dc, sizeof(int));

   // Copying in blocking-mode
   cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
   add<<<1,1>>>(da, db, dc);
   cudaMemcpy(&c, dc, sizeof(int), cudaMemcpyDeviceToHost);

   printf("a + b = %d\n", c);
   cudaFree(da); cudaFree(db); cudaFree(dc);
   return 0;
}
```

Running in Parallel

- so far; pointing parameters to GPU, and running single thread on GPU.
- Time to look at how to run things in parallel.

Code Change

```
// add<<<1, 1>>>(da, db, dc);
  add<<<N, 1>>>(da, db, dc);
```

- Instead of running add() once, execute it N times in parallel

Changes in Kernel Code

```
__global__ void add(int *a, int *b, int *c)
{
   c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Changes in Host Code

```
int main()
{  int *a, *b, *c, *da, *db, *dc, N=5, i;

   a = (int*)malloc(sizeof(int)*N); // allocate host mem
   b = (int*)malloc(sizeof(int)*N); // and assign random
   c = (int*)malloc(sizeof(int)*N); // memory

   cudaMalloc((void **)&da, sizeof(int)*N);
   cudaMalloc((void **)&db, sizeof(int)*N);
   cudaMalloc((void **)&dc, sizeof(int)*N);

   cudaMemcpy(da, &a, sizeof(int)*N, cudaMemcpyHostToDevice);
   cudaMemcpy(da, &a, sizeof(int)*N, cudaMemcpyHostToDevice);
   add<<<N,1>>>(da, db, dc);
   cudaMemcpy(&c, dc, sizeof(int)*N, cudaMemcpyDeviceToHost);

   for (i = 0; i < N; i++)
      printf("a[%d] + b[%d] = %d\n", i, i, c[i]);
}
```

That was Blocks in Parallel. What about Threads in Parallel

Function Call Change

```
// add<<<1, 1>>>(da, db, dc); // single thread GPU
// add<<<N, 1>>>(da, db, dc); // N blocks on GPU
  add<<<1, N>>>(da, db, dc); // N threads on GPU
```

Changes in Kernel Code

```
__global__ void add(int *a, int *b, int *c)
{
   c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- Rest of Host code would be the same

Blocks or Threads. Whatever. Code is running in Parallel anyways

- Let's look at sample specs for NVIDIA Kepler K40
  - Cores = 2,880
  - Multiprocessors = 15
  - Cores / multi-processor = 192
- If we make parallel blocks, 1 block will be assigned to 1 multi-processor. This means all multi-processors will be busy, but within the multi-processor, 1 core has work to do, the remaining others are free.
- If we make parallel threads and 1 block, only 1 multi-processor will be busy, the remaining will be free.
- In either case, the GPU is **under-utilized**. For maximum utilization, use both (blocks + threads)
- **Note: Above is programmer's interpretation. The actual execution model loosely follows this interpretation but has other restrictions also.**

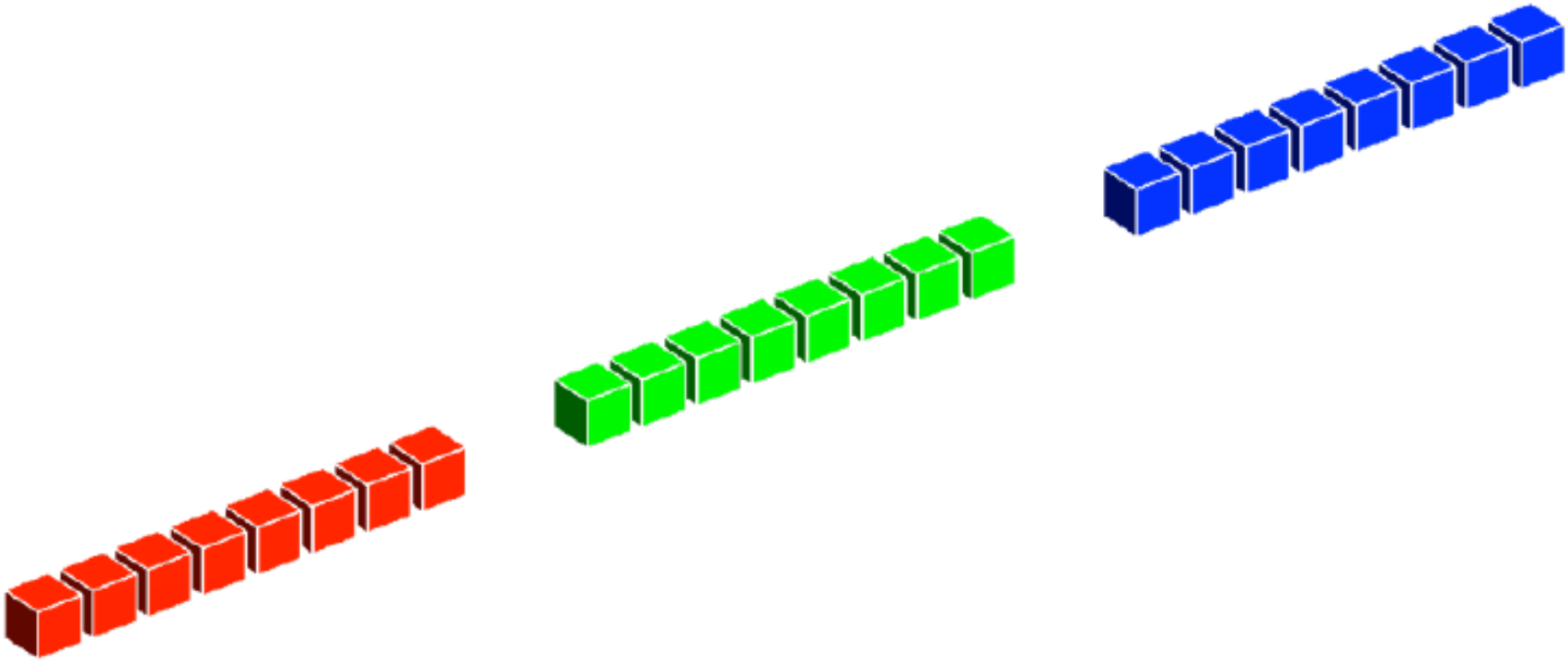
Case: Higher Dimensions

- So far, we have passed single values to kernel function call.
- For higher dimensions, use:
  - dimGrid(g_x, g_y, g_z) for dimensions of the grid
  - dimBlock(t_x, t_y, t_z) for dimensions of the thread block
- Total threads in thread-block cannot exceed 512 (or 1024 for some GPU's), i.e. condition $t_x \times t_y \times t_z \leq 512$ must be satisfied.
- Grid dimensions cannot exceed 32,768 in either dimension, i.e., condition $\max(g_x, g_y) \leq 32,768$
- dimGrid component must be evenly divisible by dimBlock component, i.e., condition $mod(g_n, t_n) = 0$ must be satisfied.

```
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:3
CL_DEVICE_MAX_WORK_ITEM_SIZES:      1024 / 1024 / 64 (512/512/32)
CL_DEVICE_MAX_WORK_GROUP_SIZE:      1024 (512)
```

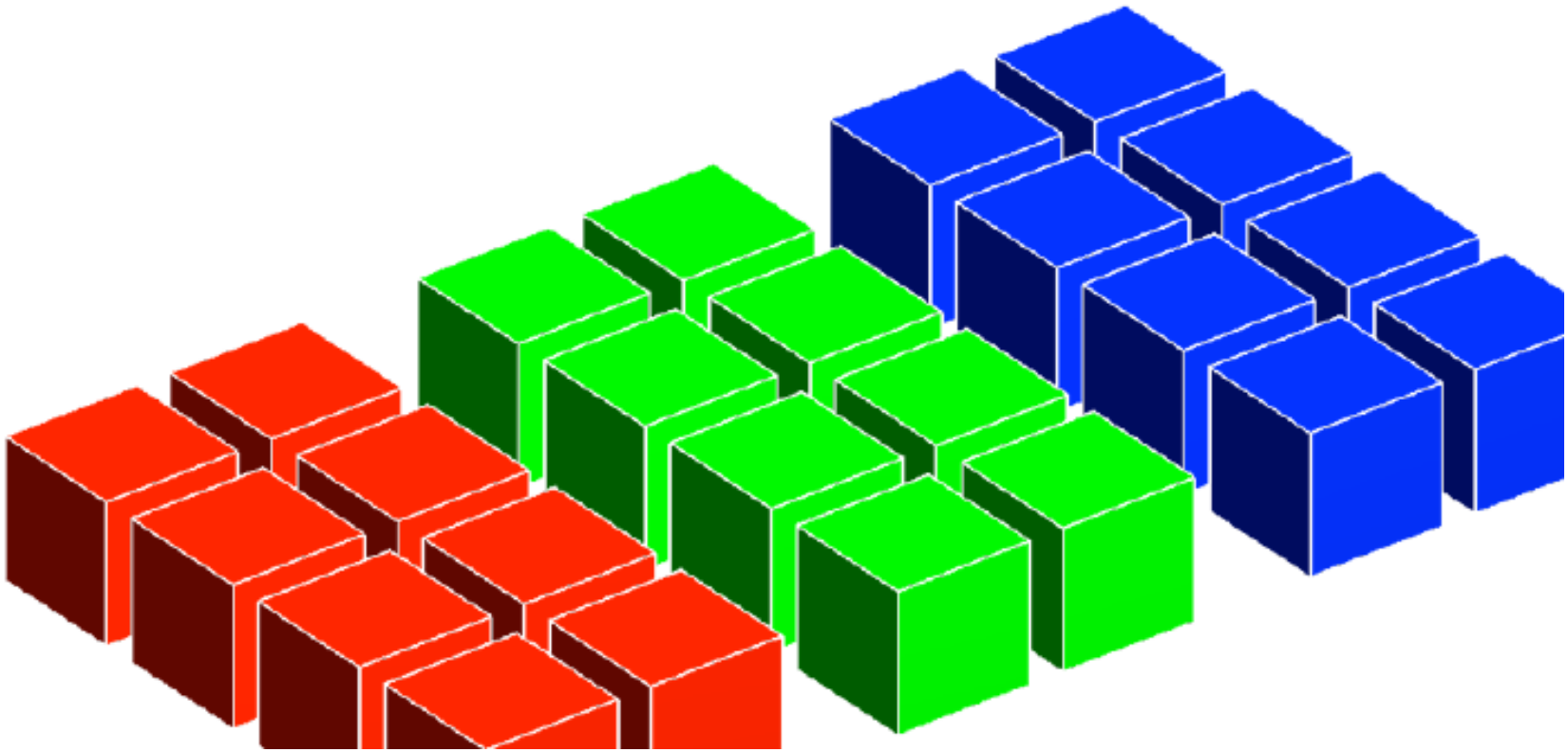
Case: Higher Dimensions (cont.)

```
dim3 dimGrid(3,1,1);
dim3 dimBlock(8,1,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```



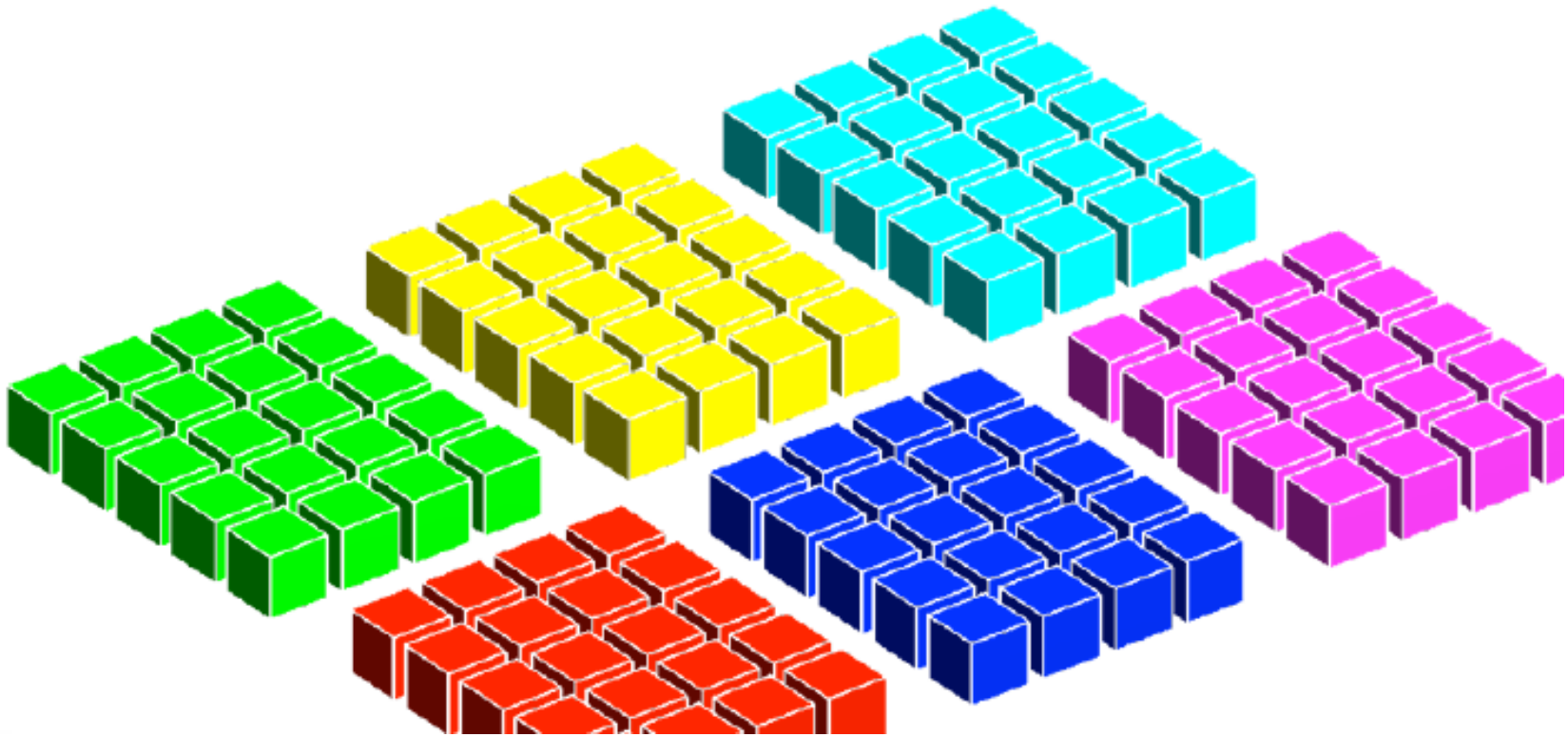
Case: Higher Dimensions (cont.)

```
dim3 dimGrid(3,1,1);
dim3 dimBlock(2,4,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```



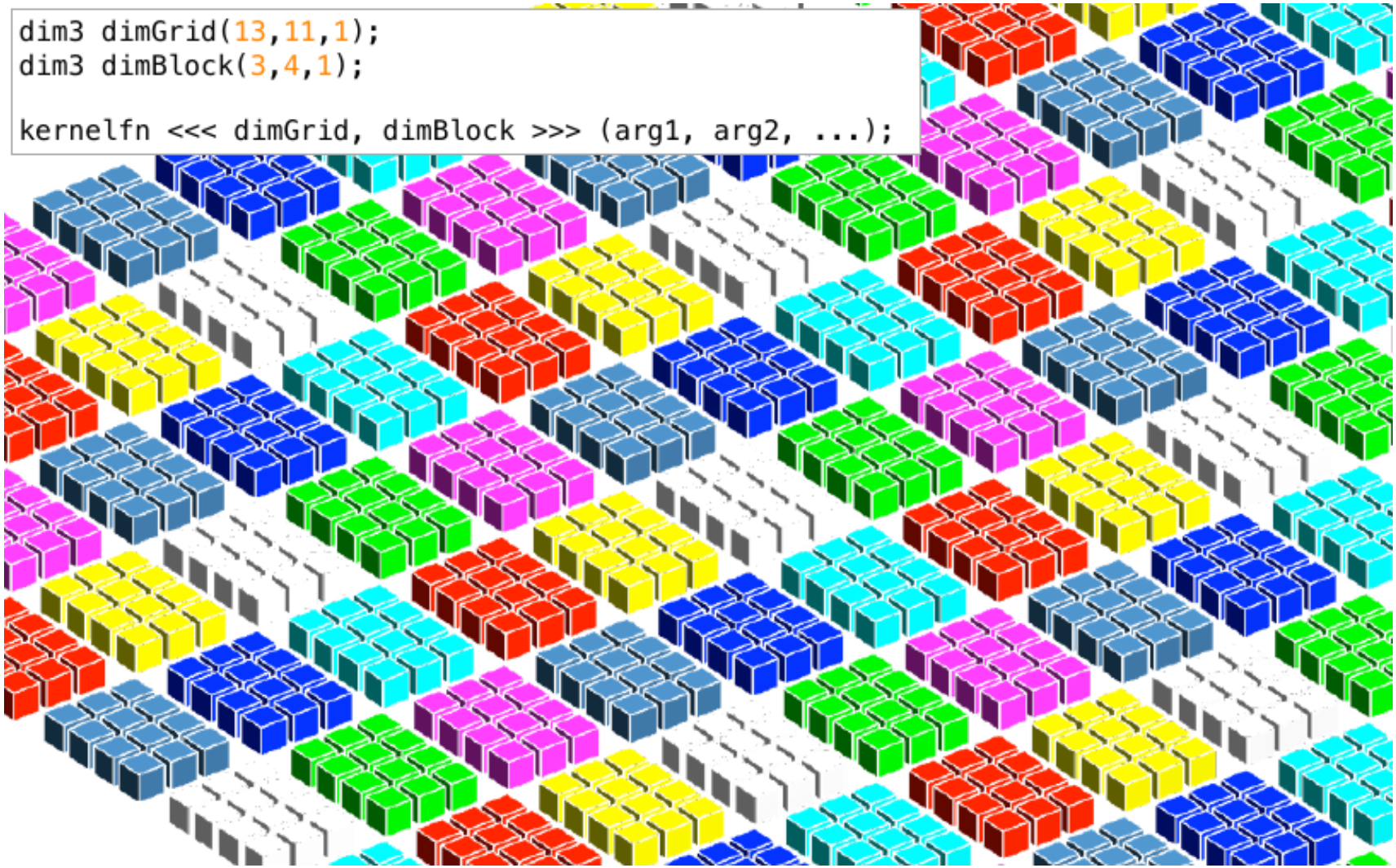
Case: Higher Dimensions (cont.)

```
dim3 dimGrid(3,2,1);
dim3 dimBlock(4,5,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```



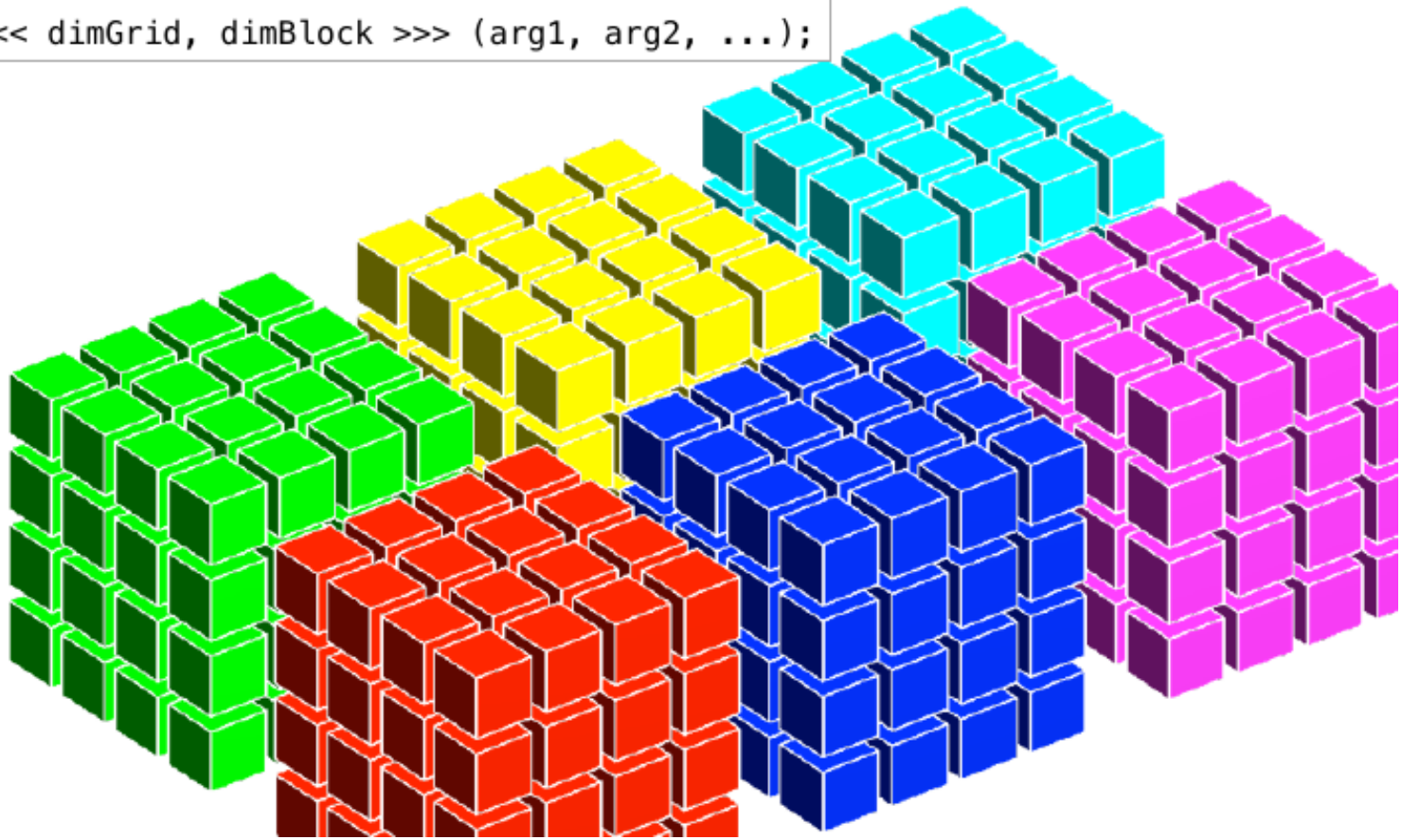
Case: Higher Dimensions (cont.)

```
dim3 dimGrid(13,11,1);
dim3 dimBlock(3,4,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```



Case: Higher Dimensions (cont.)

```
dim3 dimGrid(3,2,1);
dim3 dimBlock(4,4,4);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```



Combining both Concepts of Blocks and Threads

Formula to identify a point

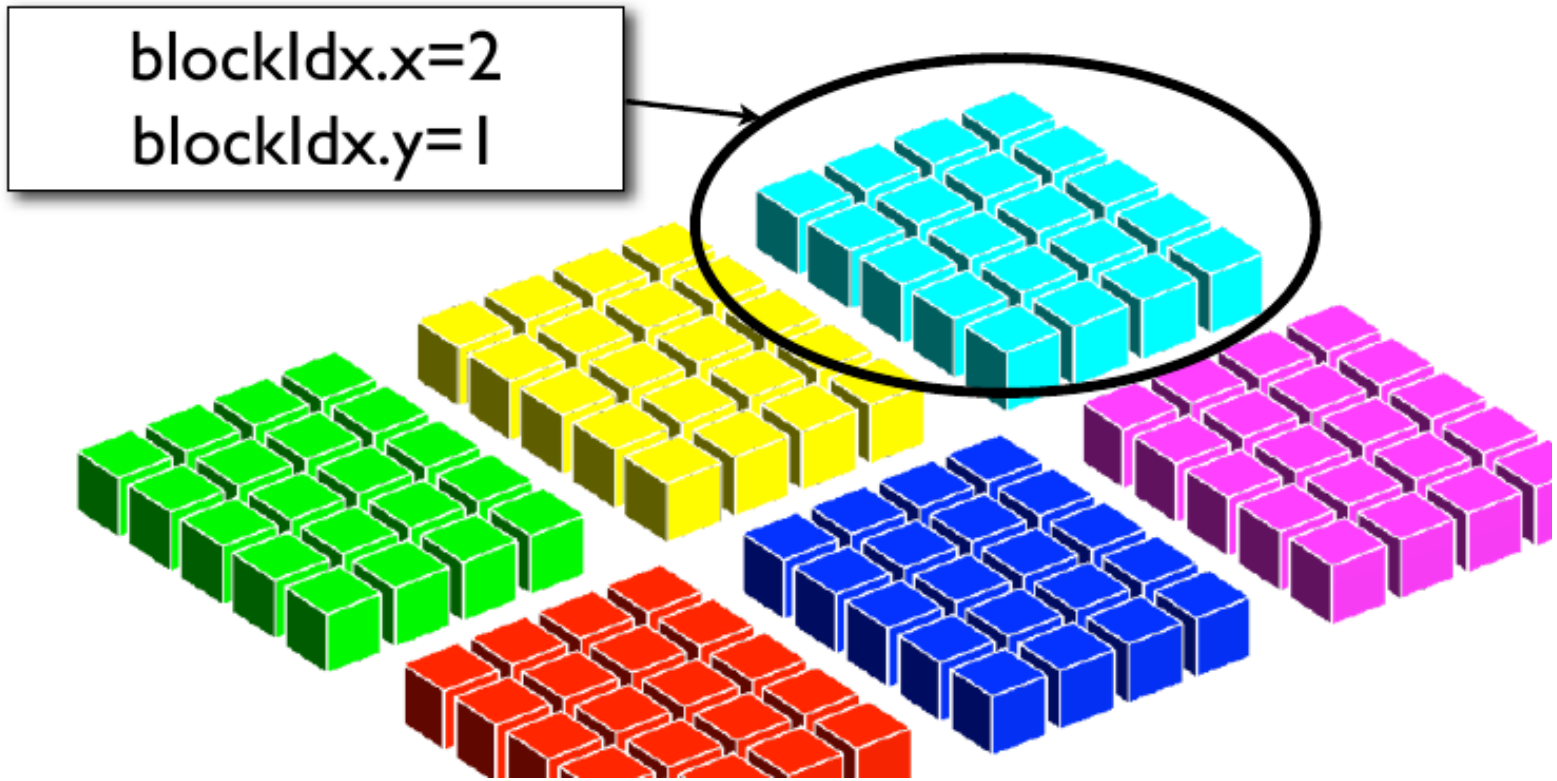
```
int index = blockIdx.x * blockDim.x + threadIdx.x;
```

- blockIdx.x = Index of thread block in grid
- blockDim.x = Number of threads in 1D thread-block
- threadIdx.x = Index of thread in 1D thread block

- Each thread execute kernel code is automatically provided the following variables
  - threadIdx.x, threadIdx.y, threadIdx.z
  - blockDim.x, blockDim.y, blockDim.z
  - blockIdx.x, blockIdx.y

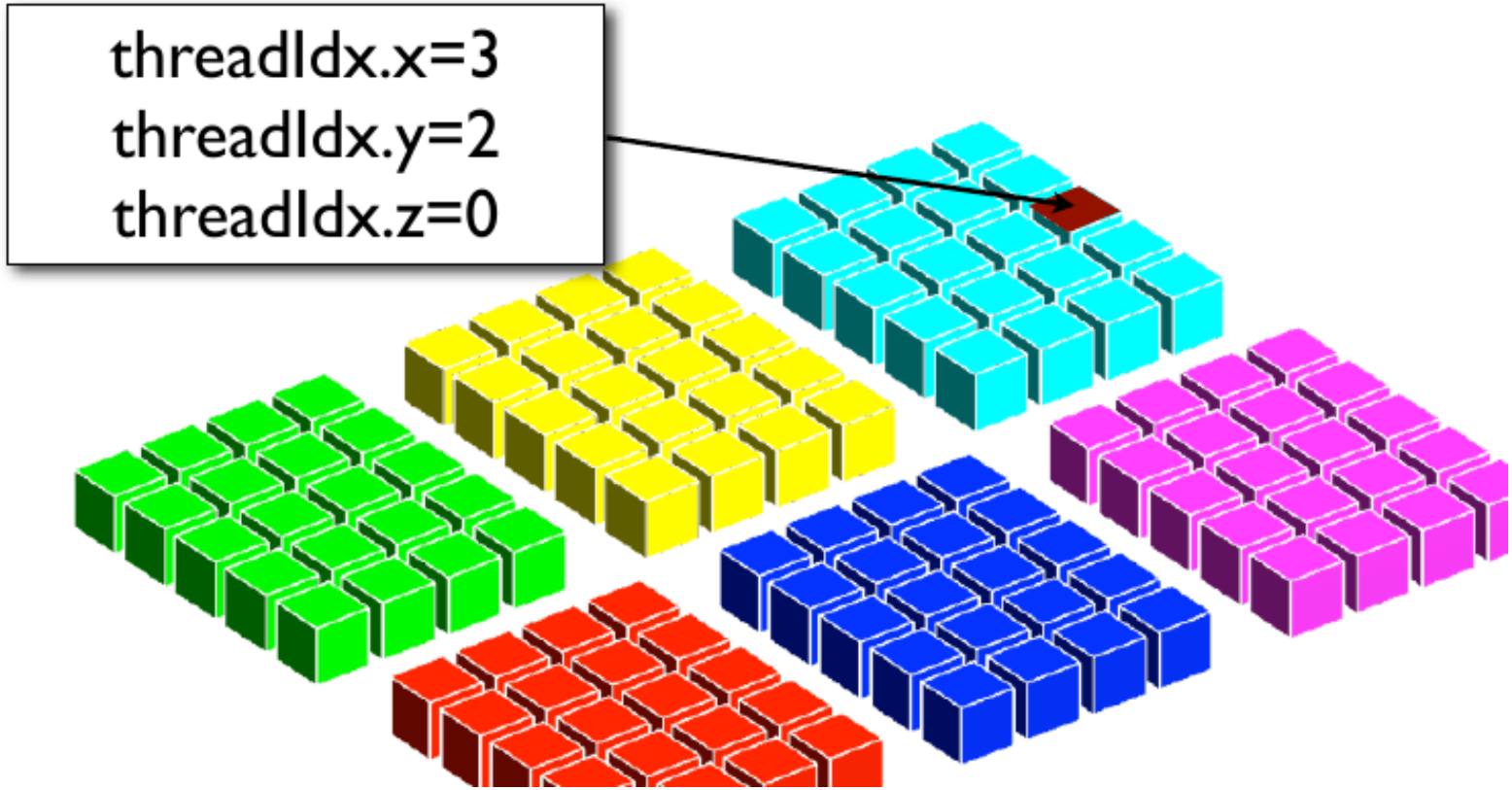
Combining both Concepts of Blocks and Threads (cont.)

```
dim3 dimGrid(3,2,1);
dim3 dimBlock(4,5,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```



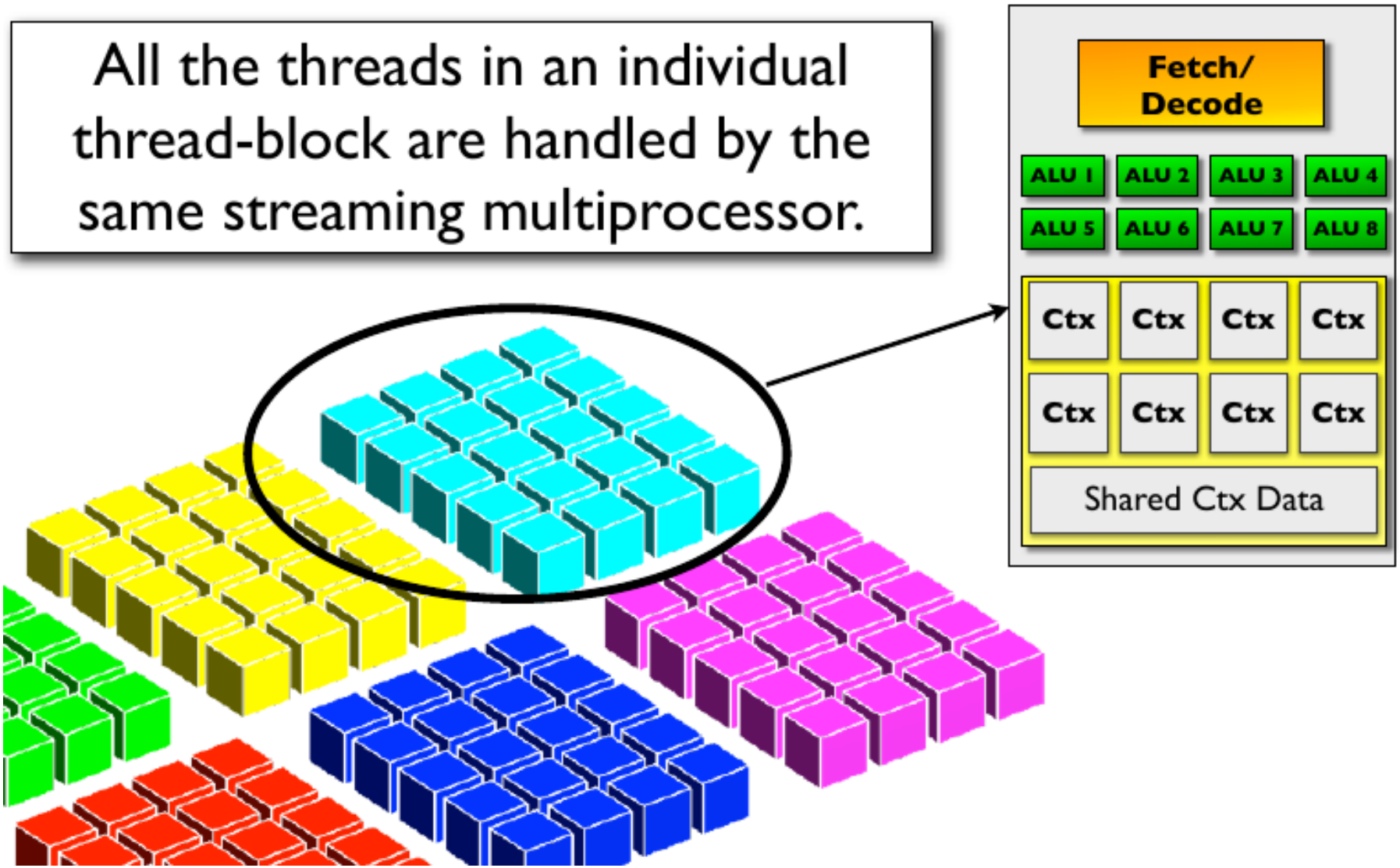
Combining both Concepts of Blocks and Threads (cont.)

```
dim3 dimGrid(3,2,1);
dim3 dimBlock(4,5,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```

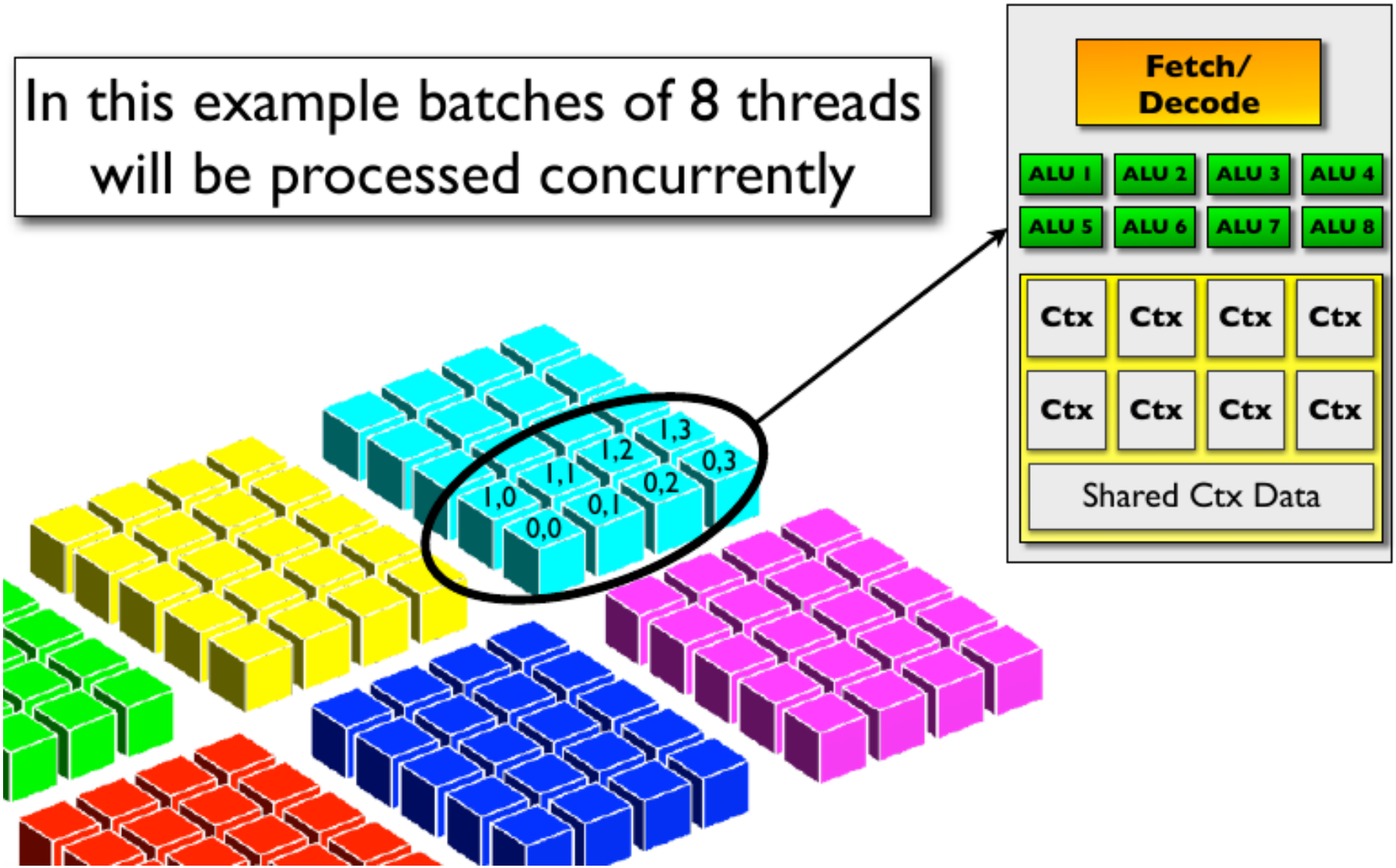


Higher Dimensions: Execution Model Revisited

All the threads in an individual thread-block are handled by the same streaming multiprocessor.



Higher Dimensions: Execution Model Revisited (cont.)



GPU Profiling using **nvprof**

- Provides textual reports on GPU and CPU activity

```
                nvprof ./a.out
ABCDEFGHIJKLMNOPQRSTUVWXYZ
==20580== NVPROF is profiling process 20580, command: ./a.out
25 25 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
==20580== Profiling application: ./a.out
==20580== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 42.82%  4.1280us         1  4.1280us  4.1280us  4.1280us  [CUDA memcpy HtoD]
 29.96%  2.8880us         1  2.8880us  2.8880us  2.8880us  square(char*)
 27.22%  2.6240us         1  2.6240us  2.6240us  2.6240us  [CUDA memcpy DtoH]

==20580== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.90%  604.38ms         1  604.38ms  604.38ms  604.38ms  cudaMalloc
  0.05%  318.97us        83  3.8430us     184ns  136.54us  cuDeviceGetAttribute
  0.02%  105.66us         1  105.66us  105.66us  105.66us  cudaFree
  0.01%  52.987us         1  52.987us  52.987us  52.987us  cuDeviceTotalMem
  0.01%  40.808us         2  20.404us  15.242us  25.566us  cudaMemcpy
  0.01%  36.906us         1  36.906us  36.906us  36.906us  cuDeviceGetName
  0.00%  16.190us         1  16.190us  16.190us  16.190us  cudaLaunch
  0.00%  2.7400us         1  2.7400us  2.7400us  2.7400us  cudaSetupArgument
  0.00%  1.6770us         2     838ns     357ns  1.3200us  cuDeviceGetCount
  0.00%  1.3360us         1  1.3360us  1.3360us  1.3360us  cudaConfigureCall
  0.00%     619ns         2     309ns     253ns     366ns  cuDeviceGet
```

Occupancy Calculator Usage

- Compile your program as:

  `nvcc myCode.cu --ptxas-options=-v`

  and you will see an output like:

```
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z6squarePc' for 'sm_20'
ptxas info    : Function properties for _Z6squarePc
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 6 registers, 40 bytes cmem[0]
```

ptxas PTX Assembly Code (The GPU sees this after compilation. Think of it as machine readable code)

gmem Global Memory

spilling Occurs when there is no more space in registers

cmem Constant Memory, 0 for kernel arguments, 2 for user defined constant arguments, 16 for compiler generated constants

compute capability Device dependent. See ./oclDeviceQuery provided to you.

Occupancy Calculator Usage (cont.)

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	3.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	6	
Shared Memory Per Block (bytes)	40	

(Don't edit anything below this line)

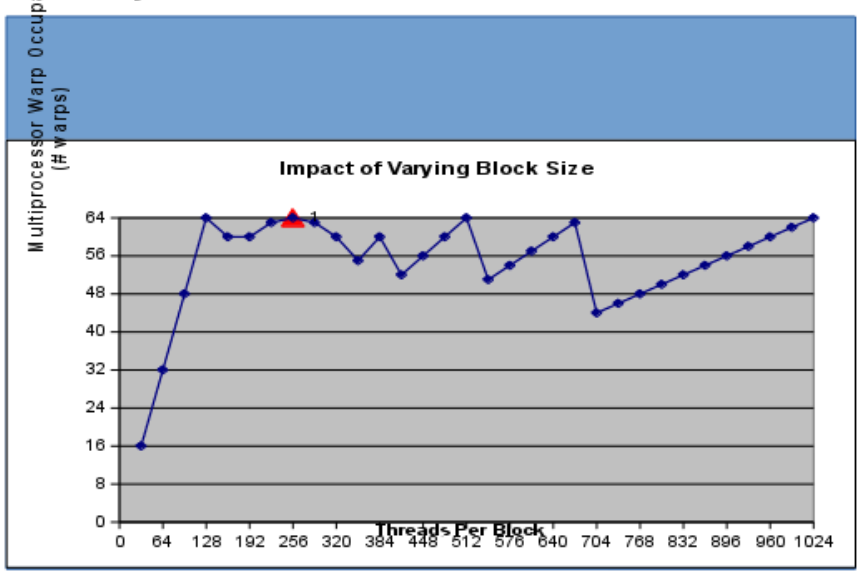
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	3.5
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16

lick Here for detailed instructions on how to use this occupancy calculator

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



Memory Coalescing

Code Snippet 1 (assume total threads = N)

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
a[x] = a[x] * a[x];
```

Code Snippet 2 (assume total threads = N)

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
if (x == N - 1)
   a[0] = a[x] * a[x];
else
   a[x] = a[x+1] * a[x+1];
```

- Looking at the hardware counters (NVIDIA Visual Profiler)
- Note: Will be more relaxed with higher Compute Capabilities on new graphic cards (Latest: 3.5)

Memory Coalescing (cont.)

Code Snippet 1

```
GLB MEM throughput(GB/s): 20.38
Peak for GT525M (GB/s): 28
Global M excess load(\%): 0.00
Global M excess store(\%): 0.00
```

Code Snippet 2

```
GLB MEM throughput(GB/s): 18.56
Peak for GT525M (GB/s): 28
Global M excess load(\%): 7.79
Global M excess store(\%): 0.00
```

Memory Coalescing (cont.)

Coalescing Rules

- Compute Capability 1.0 and 1.1
  - If 16 **aligned** threads require:
    - 4 bytes each, then 16 threads fetch 1 64 byte segment
    - 8 bytes each, then 16 threads fetch 1 128 byte segment
  - else if **not aligned** and require:
    - 4 bytes each, then 16 threads fetch 16 32 byte segment
    - 8 bytes each, then 16 threads fetch 16 32 byte segment
- Compute Capability 1.2 and 1.3
  - Prepare to Fetch 1 128 byte segment (for 16 threads)
  - If only lower/upper half of 128 byte is used, fetch 1 64 byte segment
  - If only lower/upper half of 64 byte is used, fetch 1 32 byte segment
- Compute Capability 2.x onwards
  - Fetch memory segment equal to size of cache

Pinned Memory

- Upto two times faster memory transfers.
- Uses concept of page-locked memory (prevents memory identified by pointer to be paged-out)
- Downside: Big matrices cannot be allocated using this approach.

```
cudaHostAlloc((void **)&hMemory, sizeof(hMemory),
                    cudaHostAllocDefault);
```

Example Code (Measured using **valgrind –tool=pagein**)

	Total	.data	.text	other
Pinned	137,447	4,594	129,502	4000
N. Pinned	248,143	114,597	129,502	4000

MPI Overview

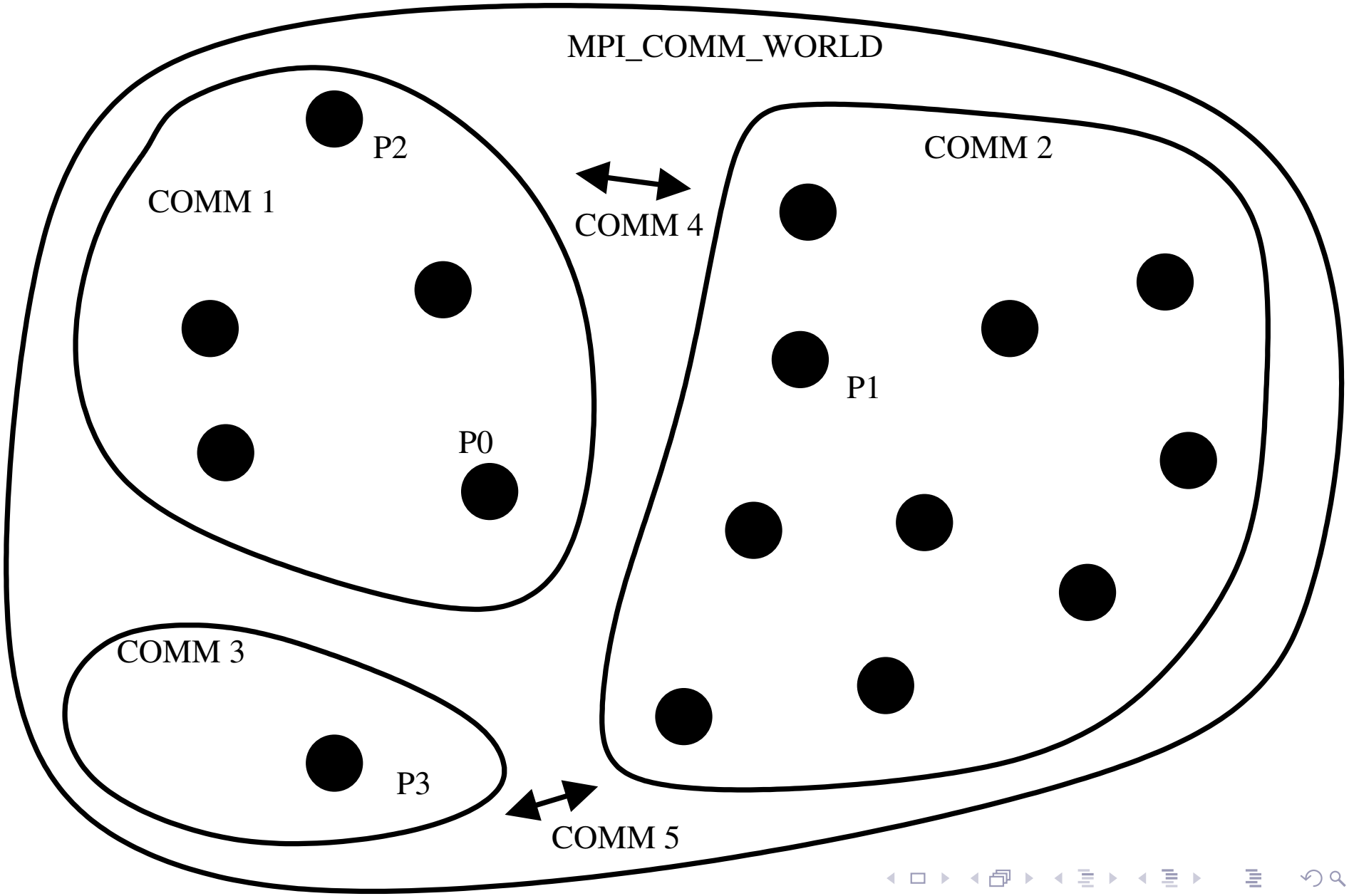
- **M**essage **P**assing **I**nterface
- MPI Can be used for Shared Memory, as well as Distributed Memory architectures (Hybrid, if requiredi)
- Supported by Fortran, C, C++ (but modules also available for python, & Java)
- Hides hardware details of underlying system (so portable)
- Many high performance libraries have MPI versions of API calls
- MPI version 3.0 specification has 400+ commands (function calls). Knowledge of only 11-12 of them can help you do the job in more than 90% of cases.

MPI **Communicators**

MPI_COMM_WORLD: Name of default MPI Communicator

- A communication universe (communication domain, communication group) for a group of processes
- Stored in variables of type **MPI_COMM**
- Communicators are used as arguments to all message transfer MPI routines
- Each process within communicator has a **rank**; a unique integer identifier ranging between $[0, \#processors - 1]$
- Multiple communicators can be established in a single MPI program
- **Intra-Communicator**: Used for communication within a single group
- **Inter-Communicator**: Used for communication between two disjoint groups

MPI **Communicators** (cont.)



First Look (hellompi.c)

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
   int size, my_rank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("Hello from %d out of %d\n", my_rank, size);

   MPI_Finalize();

   return 0;
}
```

```
mpicc hellompi.c # Compilation (mpiCC for C++, also gcc hellompi.c -lmpi)
mpirun -np 4 -hostfile filename a.out      # Execution
```

Configuring a Simple MPI based Distributed Computing Cluster

Requirements

- SSH Server

  ```
  apt-get install openssh-server
  ```

- OpenMPI Library

  ```
  apt-get install openmpi-bin openmpi-doc libopenmpi-dev
  ```

- NFS Network File System

  ```
  apt-get install nfs-server nfs-client
  ```

Configuring a Simple MPI based Distributed Computing Cluster (cont.)

Passwordless Login

- Generate a public-private key pair for your user name. Do not specify a pass-phrase when asked

  ```
  ssh-keygen
  ```

- Copy the public part of your key to the remote server

  ```
  ssh-copy-id <ip of remote computer>
  ```

- If the above command does not work, continue with these commands:
  - Copy the public key to the remote computer. For e.g., if the username is omar, the remote ip address is 1.2.3.4, then:

    ```
    scp /home/omar/.ssh/id_rsa.pub omar@1.2.3.4:/home/omar
    ```

  - Login to the remote computer using your username

    ```
    ssh 1.2.3.4 -l omar
    ```

  - Add the public key to authorized keys

    ```
    cat /home/omar/id_rsa.pub >> /home/omar/.ssh/authorized_keys
    ```

Configuring a Simple MPI based Distributed Computing Cluster (cont.)

Transfering Files

- There are many ways to transfer files. You can setup an **NFS** mountpoint, share files using dropbox, or send files using scp. The scp method is given below:
  `scp /location/of/a.out username@ipaddress:/home/username/a.out`
- **Note: All cluster nodes must be able to find the executable file at the same location as any other cluster node**

Important MPI Calls

```
MPI_Init(int*, char**);          // Initiate an MPI Computation
MPI_Finalize(void);             // Terminate an MPI Computation
MPI_Comm_size(MPI_COMM, int);     // How many processes
MPI_Comm_rank(MPI_COMM, int);     // Who am I?
MPI_Get_processor_name(char*, int); // What is the hostname?
MPI_Wtime(void);                // Elapsed time in seconds
MPI_Abort(MPI_COMM);             // Terminate all processes
```

Sending/Receiving

What may happen in code P0 (left) and P1 (right) below?

```
int a = 100;              int a;
send(&a, P1);              receive(&a, P0);
a = 0;                   printf("%d\n", a);
```

Approaches to Send/Receive

Blocking (Non-Buffered) Send/Receive

- Follow some form of "handshaking" protocol
  - Request to Send → Clear to Send → Send Data → Acknowledgement
  - Problem 1: Idling Overhead (both sender/receiver side)
  - Problem 2: Deadlock (sending at same time)

Blocking (Buffered) Send/Receive

- Copy send-data to designated buffer, and returns after "copy" operation is completed
- Problem 1: Buffer Size

```
for (i = 0; i < 1000; i++) {          for (i = 0; i < 1000; i++) {
   produce_data(&a);                     receive(&a, P0);
   send(&a, P1);                      consume_data(&a);
}                                     }
```

- Problem 2: Deadlock (sending at same time)

Approaches to Send/Receive (cont.)

Non-Blocking Send/Receive

- Return from Send/Receive operation before it is "safe" to return.
- Programmer responsibility to ensure that "sending data" is not altered immediately

- Blocking Operations: Safe and Easy Programming (at cost of overhead and risk of deadlocks)
- Non-Blocking Operations: Useful for Performance optimization, and breaking deadlocks (but brings in plenty of race-conditions if programmer not careful)

Point to Point Communication

Types of Point-to-Point Send/Receive Calls

- **Synchronous Transfer**: Send/Receive routines return only when the message transfer is completed. Not only does this transfer data, but it also synchronizes processes

```
MPI_Send()      // Blocking Send
MPI_Recv()      // Blocking Receive
```

- **Asynchronous Transfers**: Send/Receive do not wait for transfer data and proceeds with execution next line of instruction. (Precaution: Do not modify the send/receive buffers)

```
MPI_Isend()     // Non-Blocking Send
MPI_Irecv()     // Non-Blocking Receive
```

Point to Point Communication (cont.)

Sending

```
int MPI_Send(void *buffer,   int count,   MPI_DATATYPE datatype,
         int destination, int tag,    MPI_Comm comm);
```

- Send the data stored in **buffer**
- **Count** is the number of entries in the buffer
- What is the **datatype** of the buffer (MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_CHAR, etc.)
- **Destination** is the rank of process, to whom buffer is to be sent to, residing in communication universe **comm**
- The **tag** of the message (to distinguish between different types of messages)

Point to Point Communication (cont.)

Receiving

```
int MPI_Recv(void *buffer,   int count,  MPI_DATATYPE datatype,
         int source,    int tag,    MPI_Comm comm,
         MPI_Status *status);
```

- Store the received message in **buffer**
- **Count** is the number of entries to be received in the buffer. If number of entries is larger than the capacity of buffer, an overflow error MPI_ERR_TRUNCATE is returned.
- **Datatype** is the type of data that has been received
- **Source** is the rank of process, residing in communication domain **comm**, from whom buffer is received. Source can be hard-set, or a wild-card MPI_ANY_SOURCE.
- To retrieve message of certain type, set the **tag** argument. If there are many messages of same tag from same process, any one of them may be retrieved. If message of any tag is to be retrieved, use the wild-card MPI_ANY_TAG.
- Store status of received message in **status** (next slide). If not needed, use MPI_STATUS_IGNORE

Example 1: Blocking Send/Receive between Two Processes

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

if (myRank == 0) {                  // Sending Process
   int buffer = 3;
   MPI_Send(&buffer,          // Reference to the storage location
         1,              // 1 item is being sent
         MPI_INT,          // integer data type
         1,              // Destination rank
         123,             // tag of message
         MPI_COMM_WORLD);   // communicator
}
else {                      // receiving process
   int buffer;
   MPI_Recv(&buffer,          // reference to storage location
         1,              // receiving 1 item
         MPI_INT,          // of type integer
         0,              // from process of rank 0
         123,             // tag of message
         MPI_COMM_WORLD,    // communicator
         MPI_STATUS_IGNORE); // status of received message

   printf("%d\n", buffer);
}
```

Example 2: Sending an Array from one process to another

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

int x, array_size = 10, *buffer1, *buffer2;

if (myRank == 0) {                          // Sending Process
   buffer1 = malloc(sizeof(int)*array_size);   // allocate memory

   for (x = 0; x < array_size; x++)          // put something in it
         buffer1[x] = x+1;

   MPI_Send(buffer1, array_size, MPI_INT, 1, 123, MPI_COMM_WORLD);
}
else {                      // receiving process
   buffer2 = malloc(sizeof(int)*array_size);   // allocate memory
   MPI_Recv(buffer2, array_size, MPI_INT, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

   for (x = 0; x < array_size; x++)          // print it
         printf("%d\n", buffer2[x]2[x]);
}
```

Example 3: Doing distributed $x++$

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &mySize);

if (myRank == 0) { // Originating Process
   int x = 0;
   x++;

   MPI_Send(&x, 1, MPI_INT, myRank+1, 123, MPI_COMM_WORLD);
}
else if (myRank > 0 && myRank < n-1) {
   int x;
   MPI_Recv(&x, 1, MPI_INT, myRank-1, 123, MPI_COMM_WORLD,      MPI_STATUS_IGNORE);
   x++;
   MPI_Send(&x, 1, MPI_INT, myRank+1, 123, MPI_COMM_WORLD);
}
else if (myRank == n-1) {
   int x;
   MPI_Recv(&x, 1, MPI_INT, myRank-2, 123, MPI_COMM_WORLD,      MPI_STATUS_IGNORE);
   printf("Incremented to %d\n", x++);
}
```

MPI_Probe() and MPI_Get_count()

MPI_Status Structure

```
typedef struct _MPI_Status {
  int MPI_SOURCE;     // Rank of Sender
  int MPI_TAG;        // Tag of Message
  int MPI_ERROR;      // Any Error if occurred
} MPI_Status;
```

- **MPI_Probe()**: Check for incoming messages (without actually receiving them).

```
MPI_Probe(int source,  int tag,
          MPI_Comm comm, MPI_Status *status)
```

- **MPI_Get_count()**: Returns number of messages received

```
MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
              int *count)
```

Non-Blocking Send/Receive

- For sending, **MPI_Isend()** is used with the syntax:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
```

- For receiving, **MPI_Irecv()** is used with the syntax:

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int src, int tag, MPI_Comm comm,
              MPI_Request *request)
```

- To check whether a send/receive has completed or not, use **MPI_Test()** with syntax:

```
int MPI_Test(MPI_Request *req, int *flag, MPI_Status *status)
```

- To force a wait on a non-blocking send/receive, use **MPI_Wait()**:

```
int MPI_Wait(MPI_Request *req, MPI_Status *status);
```

- To wait for x requests, use **MPI_Waitall** as:

```
int MPI_Waitall(int x, MPI_Request *req, MPI_Status *stat);
```

Non-Blocking Send/Receive (cont.)

```
MPI_Request request;  // handle to the non-blocking operation
MPI_Status  status;   // contains information about the message
int flag;           // 1: sent/received, 0: not-sent/not-received
int buffer;         // the data buffer

if (myRank == 0) {                      // Sending Side
   buffer = 3;
   MPI_Isend(&buffer, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &request);
}
else {                                  // Receiving Side
   MPI_Irecv(&buffer, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &request);

   MPI_Test(&request, &flag, &status);     // check status of recv
   if (flag == 0) {
      MPI_Wait(&request, &status);       // force wait
   }

   printf("%d\n", buffer);
}
```

Collective Communications

- Point-to-Point: It is programmer's responsibility to ensure that all processes participate correctly in a given communication (Programmer's burden)
- MPI simplifies this using Collective Communication. Types are:
  - **Synchronization**:
    - Barriers: MPI_Barrier()
  - **Moving Data**:
    - Broadcasting: MPI_Bcast()
    - Scattering: MPI_Scatter()
    - Gathering: MPI_Gather()
  - **Collective Computation**:
    - Reduction: MPI_Reduce()
- Difference to point-to-point communications
  - No message tags
  - Most calls/versions support blocking communication only

Synchronization: Barrier

```
int MPI_Barrier(MPI_Comm comm)
```

```
int x, myRank;
MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

for (x = 0; x < 10; x++) {
   MPI_Barrier(MPI_COMM_WORLD);
   if (x == myRank) {
      printf("%d \n", myRank);
   }
}

MPI_Finalize();
```

Moving Data: Broadcast

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root,    MPI_Comm comm)
```

```
int arrSize = 10;
int *buffer = malloc(sizeof(int)*arrSize);

if (myRank == 0) {
   for (i = 0; i < arrSize; i++) {
      buffer[i] = i;
   }
}

MPI_Bcast(buffer, arrSize, MPI_INT, 0, MPI_COMM_WORLD);

printf("%d: ", myRank);
for (i = 0; i < arrSize; i++) {
       printf("%d ", buffer[i]);
}
printf("\n");
```

Moving Data: Scatter

```
int MPI_Scatter(
    void *sendbuf,        // full size buffer
    int sendcount,        // addresses to send
    MPI_Datatype sendtype, // data type to send
    void *recvbuf,        // chunk size buffer
    int recvcount,        // how many are to be received
    MPI_Datatype recvtype, // datatype of receive type
    int root,             // who is the source
    MPI_Comm comm);       // communication world
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &totalProcesses);

int fullSize = 100, reducedSize = fullSize/totalProcesses;

int *buffer  = malloc(sizeof(int)*fullSize);
int *buf     = malloc(sizeof(int)*reducedSize);

if (myRank == 0)    for (i = 0; i < fullSize; i++)    buffer[i] = i;

MPI_Scatter(buffer, reducedSize, MPI_INT, buf, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);

printf("%d: ", myRank);
for (i = 0; i < reducedSize; i++)       printf("%d ", buf[i]);
printf("\n");
```

Moving Data: Scatter & Gather

```
int MPI_Gather(
    void *sendbuf,       // chunk size buffer
    int  sendcount, MPI_Datatype sendtype,
    void *recvbuf,       // full size buffer
    int  recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &totalProcesses);

int fullSize = 100, reducedSize = fullSize/totalProcesses;
int *buffer1  = malloc(sizeof(int)*fullSize);
int *buffer2  = malloc(sizeof(int)*fullSize);
int *buf      = malloc(sizeof(int)*reducedSize);

if (myRank == 0)  for (i = 0; i < fullSize; i++)  buffer1[i] = i;

MPI_Scatter(buffer1, reducedSize, MPI_INT, buf, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);
for (i = 0; i < reducedSize; i++)   buf[i]*=2;
MPI_Gather(buf, reducedSize, MPI_INT, buffer2, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);

if (myRank == 0) {
   printf("%d: ", myRank);
   for (i = 0; i < fullSize; i++)
      printf("%d ", bufferR[i]);
}
```

Moving Data: All Gather

```
int MPI_Allgather(
    void *sendbuf,       // chunk size buffer
    int  sendcount, MPI_Datatype sendtype,
    void *recvbuf,       // full size buffer
    int  recvcount, MPI_Datatype recvtype, MPI_Comm comm) // Note: No root
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &totalProcesses);

int fullSize = 100, reducedSize = fullSize/totalProcesses;
int *buffer1  = malloc(sizeof(int)*fullSize);
int *buffer2  = malloc(sizeof(int)*fullSize);
int *buf      = malloc(sizeof(int)*reducedSize);

if (myRank == 0)  for (i = 0; i < fullSize; i++)  buffer1[i] = i;

MPI_Scatter(buffer1, reducedSize, MPI_INT, buf, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);
for (i = 0; i < reducedSize; i++)   buf[i]*=2;
MPI_Allgather(buf, reducedSize, MPI_INT, buffer2, reducedSize, MPI_INT, MPI_COMM_WORLD);

if (myRank == 0) {                  // Now can be changed to any rank !!
   printf("%d: ", myRank);
   for (i = 0; i < fullSize; i++)
      printf("%d ", bufferR[i]);
}
```

Collective Computation: Reduce

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

Where, *op* is any of:

MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_SUM	Summation	MPI_PROD	Product
MPI_LAND	Logical And	MPI_BAND	Bitwise And
MPI_LOR	Logical Or	MPI_BOR	Bitwise Or
MPI_LXOR	Logixal Xor	MPI_BXOR	Bitwise Xor

```
MPI_Init(&argc, &argv);

int myRank;
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

int buffer;
MPI_Reduce(&myRank, &buffer, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (myRank == 0) {
   printf("Final Sum: %d\n", buffer);
}

MPI_Finalize();
```

Collective Computation: All Reduce

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

Where, *op* is any of:

MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_SUM	Summation	MPI_PROD	Product
MPI_LAND	Logical And	MPI_BAND	Bitwise And
MPI_LOR	Logical Or	MPI_BOR	Bitwise Or
MPI_LXOR	Logixal Xor	MPI_BXOR	Bitwise Xor

```
MPI_Init(&argc, &argv);

int myRank;
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

int buffer;
MPI_Allreduce(&myRank, &buffer, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

if (myRank == 0) {              // now can be any rank
   printf("Final Sum: %d\n", buffer);
}

MPI_Finalize();
```

Collective Computation: Reduction (on buffer)

```
int myRank, mySize, i;
 MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
 MPI_Comm_size(MPI_COMM_WORLD, &mySize);

int *buffer = malloc(sizeof(int)*2);
buffer[0] = myRank;
buffer[1] = 1;

int *result = malloc(sizeof(int)*2);

MPI_Reduce(buffer, result, 2, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (myRank == 0) {
   printf("Final Sum: %d\n", result[0]);
   printf("Final Sum: %d\n", result[1]);
}
```

Multiple Groups and Communicators

- Useful if collective tasks are performed on a subset of processes
- **New communicators** created by splitting **old communicators**
- Grouping made on bases of **color** and **key**
- The old communicator will still be present

```
int MPI_Comm_split(
 MPI_Comm comm,     // The communicator to be split
 int color,         // to which color each process will belong?
 int key,           // rank order in new group
 MPI_Comm *newcomm) // the name of the new communicator
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

int myColor = myRank / 4;

MPI_Comm aRow;
MPI_Comm_split(MPI_COMM_WORLD, myColor, myRank, &aRow);

int myRankInRow;
MPI_Comm_rank(aRow, &myRankInRow);
```

Block Transfers

```
int MPI_Type_create_subarray(
    int          ndims,
    int          array_of_sizes[],
    int               array_of_subsizes[],
    int          array_of_starts[],
    int          order, // MPI_Order_C
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);

int MPI_Type_commit(MPI_Datatype *datatype)
```

Overview

Hadoop common | Hadoop File System | Map Reduce

Topics Covered

- Software Stack
- Distributed File System (HDFS)
- Physical Organization of Compute Nodes (Compute Nodes, Redundant Nodes, etc.)
- Map Reduce Framework
  - Key Value Pairs
  - Concept of Grouping Keys
  - Mappers & Reducers
- Sample Algorithms (See sample questions given)

Hadoop File System

Design Consideration

- Break files into blocks $B = \{b_1, b_2, \ldots, b_n\}$ of certain size s
- Distribute blocks across multiple (data) nodes $N = \{n_1, n_2, \ldots, n_m\}$ within a cluster
- Challenges:
  - For large m, node failures are quite probable. So introduce redundancy.
  - For large n, high throughput is required. So introduce concepts such as write once read many, and move computation closer to data.

Hadoop File System (cont.)

Performance Impact

- File Size, Block Length, Block Quantity, all affect performance
- Block quantity $\leftrightarrow$ Number of Threads
  - Queues
  - Thread Creation/Deletion Time
  - I/O
  - Messages exchanges between threads
- Strategies: Merge Files, Load Files in Sequence

Hadoop Common Architecture

Name Node (Master Server)

- Manages File System Namespace
- Regulates/Control access to files
  - Read/write requests from client
  - Create/Delete/Replicate blocks on data nodes

Data Nodes

- Manages Physical Storage of Blocks
- Serve Read/Write requests of Clients
- Serve Create/Delete/Replicate block requests of Name Node

Node Failures

Data Node Failure

- Server Crash / Disk Crash / Data Corruption / Network Failure
- Name node sends periodic heart-beats
- If true, mark dead, re-replicate block copy

Network Failure

- Denial of Service
- Physical Network failure

Namenode Failure

- Server Crash / Disk Crash / Data Corruption
- Send data/metadata to secondary nameserver (if configured)

HDFS Tuning Parameters

(Name,Value) properties in `/etc/hadoop/hdfs-site.xml`

- `dfs.replication` : `3` for Replication Factor
- `dfs.namenode.name.dir`: `/var/lib/hadoop/hdfs/name` for Name Node
- `dfs.datanode.data.dir`: `/var/lib/hadoop/hdfs/data` for Data Node
- `dfs.namenode.secondary.http-address` : `hdfs://localhost:50090` For Secondary Name node
- `dfs.permissions.superusergroup` : `hadoop` for User Permissions (must belong to this group)
- `dfs.block.size` : `134217728` for changing block size (across all clusters)
- `dfs.datanode.handler.count` : `10` for changing threads per data node.
- `dfs.namenode.fs-limits.max-blocks-per-file` : `100` for fixing maximum allowable blocks per file

- Dozens of other parameters specified in `hdfs-default.xml` (search online)

Specific Adjustments

- `hdfs dfs -D dfs.blocksize=134217728 -put test_128MB.csv /user`

HDFS Commands

- `hdfs dfs -ls /`
- `hdfs dfs -lsr /` ls with recursive display
- `hdfs dfs -du` Disk usage
- `hdfs dfs -dus` Disk usage summary
- `hdfs dfs -mv src dest`
- `hdfs dfs -rm xyz` Remove file or empty directory
- `hdfs dfs -rmr xyz` Recursive remove file or directory
- `hdfs dfs -put local remote`
- `hdfs dfs -get remote local`
- `hdfs dfs -cat file`
- `hdfs dfs -tail file`
- `hdfs dfs -head file`
- `hdfs dfs -chmod 777 file`
- `hdfs dfs -chown group file`
- `hdfs dfsadmin -report` Shows utilization of HDFS

Using C

libhdfs

- Part of the Hadoop distribution (Located pre-compiled in `$HADOOP_HDFS_HOME/lib/native/libhdfs.so`)
- Compatible with both Linux/Windows
- Java Native Interface (JNI) to Core Interface API in Java
- Thread Safe
- To compile (gcc)

```
gcc -I /opt/hadoop-3.2.1/include hdfsC.c
    -L /opt/hadoop-3.2.1/lib/native -lhdfs
    -L /opt/oracle-jdk-bin/jre/lib/amd64/server -ljvm|
```

- To run

```
CLASSPATH=$CLASSPATH:$(/opt/hadoop-3.2.1/bin/hadoop classpath --glob)
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/oracle-jdk-bin/jre/lib/amd64/server: \
                                 /opt/hadoop-3.2.1/lib/native
./a.out
```

Using C (cont.)

```c
#include "hdfs.h"
#include <string.h>
#include <stdio.h>

int main(int argc, char **argv) {
  hdfsFS       fs           = hdfsConnect("default", 0);

  const char  *writePath    = "/testfile.txt";
  hdfsFile     writeFile    = hdfsOpenFile(fs, writePath, O_WRONLY|O_CREAT, 0, 0, 0);

  char* buffer              = "Hello, World!";
  tSize num_written_bytes   = hdfsWrite(fs, writeFile, (void*)buffer, strlen(buffer)+1);

  hdfsFlush(fs, writeFile);
  hdfsCloseFile(fs, writeFile);
}
```

HTTP Rest API

- Support for HTTP Get, Put (-X PUT), Post (-X POST), Delete (-X DELETE)
- Configuration for `dfs.webhdfs.enabled` required in `hdfs-site.xml` (default is true)
- General Usage:

```
curl -i http://localhost:port/webhdfs/v1/[path|file]?
                                        [user.name=<user>&]
                                        op=[<options>]
```

- Default port: 50090 $\rightarrow$ 9870
- Responses in JSON

HTTP Rest API (cont.)

Sample Operations

- op=GETFILESTATUS Get Information about files
- op=MKDIRS for creating directory
- permission=755 for specifying Linux permissions
- op=CREATE for creating a (blank) file. For copying contents of an existing file, (use with -T <LOCAL_FILE>
- blocksize=<LONG> for Block Size
- replication=<SHORT> for replication factor
- op=APPEND for appending to a file
- op=OPEN for opening and reading a file
- op=RENAME&destination=<PATH> for renaming a file
- op=DELETE for deleting a file/directory, (Use with -X DELETE)
- . . . and others (See hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/WebHDFS.html

Map Reduce Framework

- Mapper Function
- Reducer Function
- Shuffle and Sort

Map Reduce Framework (cont.)

Example: Word Count

- <word, 1>
- Tokenize text and produce key-value pairs in a stream.
- Get new word from stream. If new word is same as previous word, increment a counter, else reset the counter.

```
import sys

for line in sys.stdin:
  line  = line.strip()
  words = line.split()
  for word in word:
    print ("%s\t1" % word)
```

```
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    count = int(count)
        if current_word == word:
        current_count += count
    else:
        print ('%s\t%s' % (current_word, current_count))
        current_count = count
        current_word = word
```

Map Reduce Framework (cont.)

To run Job

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-3.2.1.jar
            -file /home/omar/work/codes/hadoop/mapper.py
            -file /home/omar/work/codes/hadoop/reducer.py
            -mapper mapper.py
            -reducer reducer.py
            -input /4300-folder/*
            -output /4300-output
```

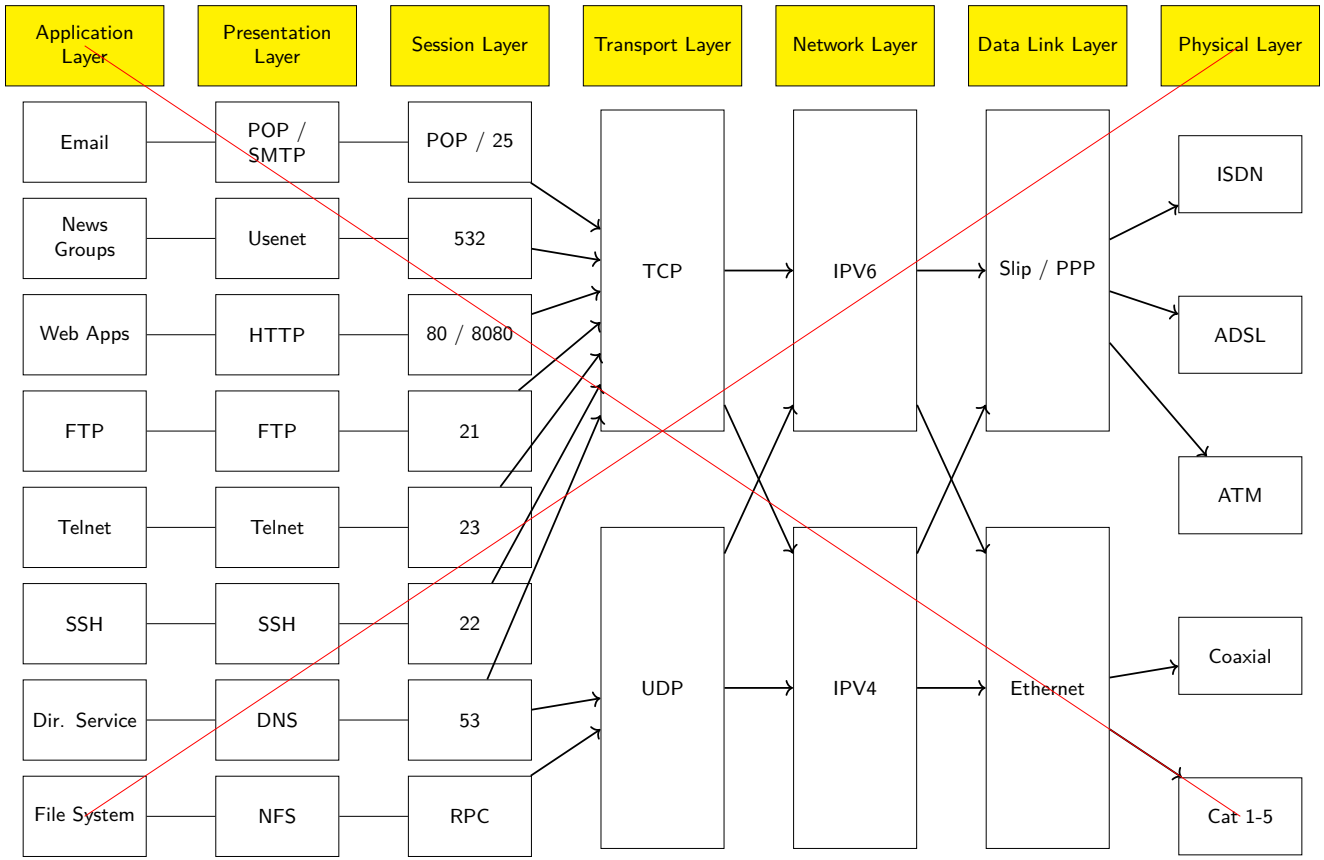
- Reducer Threads controlled using `-D mapred.reduce.tasks=16`

Yarn Configuration

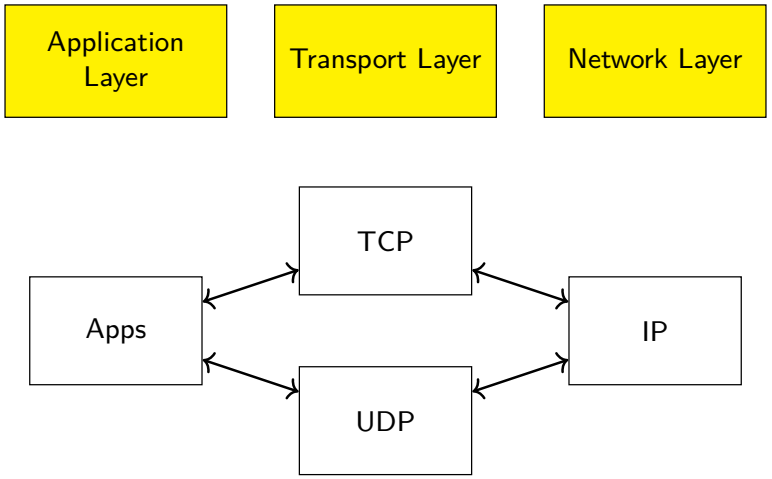
```
<property>
  <name>yarn.app.mapreduce.am.env</name>
  <value>HADOOP_MAPRED_HOME=${full path of your hadoop distribution directory}</value>
</property>
<property>
  <name>mapreduce.map.env</name>
  <value>HADOOP_MAPRED_HOME=${full path of your hadoop distribution directory}</value>
</property>
<property>
  <name>mapreduce.reduce.env</name>
  <value>HADOOP_MAPRED_HOME=${full path of your hadoop distribution directory}</value>
</property>
```

Client/Server Model

- Client: Request a Service
- Server: Provide a Service (Models: Iterative/Concurrent/Distributed)



Client/Server Model (cont.)



Overview

Info Required for Connection

- Protocol (TCP/UDP)
- Source + Destination IP Address
- Source + Destination Port Address

Transmission Control Protocol (TCP)

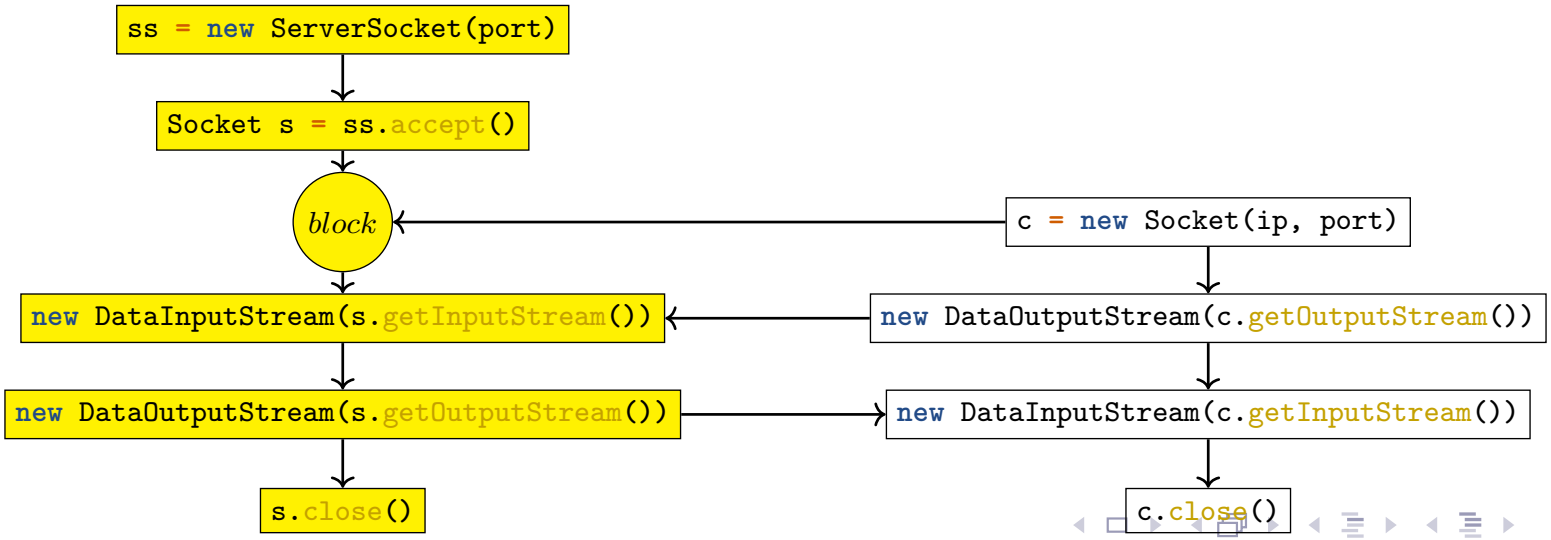
- Streams of Packets sent (each packet numbered for sequence)
- Acknowledgement sent for Each Packet (else retransmit)
- Packets also undergo error detection/correction techniques
- Slower, at the cost of integrity

User Datagram Protocol (UDP)

- Information content wise, datagram is same as packet (no numbering, packets can be received in any order)
- No acknowledgements sent
- No error detection/correction performed
- Faster, at the cost of integrity

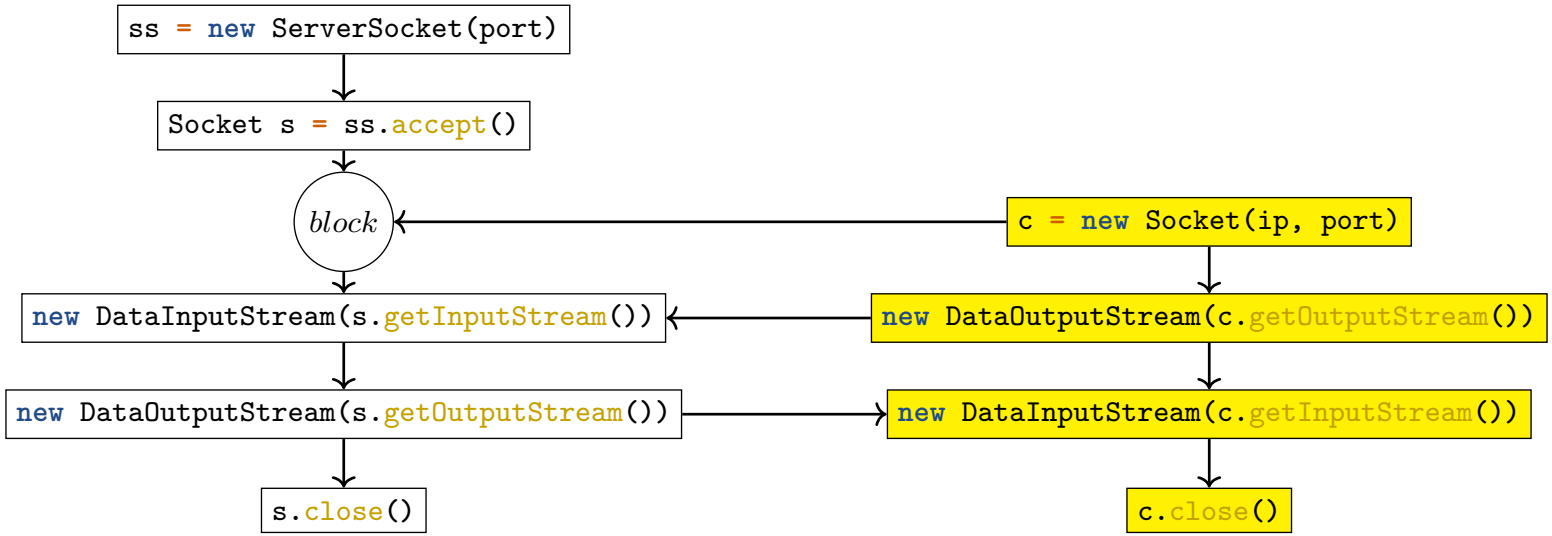
TCP Socket

- A method for achieving Inter-process communication on Linux systems (extended to networks)
- Person A phones Person B. Once connection is established. They can talk. Same Concept as Socket
- Just as File Read/Write through File Descriptor, similarly, Socket Send/Receive through File Descriptor as well
  - `java.net.Socket` and `java.net.ServerSocket` for Sockets
  - `java.io` for File Descriptors
- Most sockets use Half-Association
  - Protocol + Local IP Address + Local Port Number
  - Protocol + Remote IP Address + Remote Port Number



TCP Socket (cont.)

```
public class nu_18_socket_server {
  public static void main(String[] args) throws IOException {
    ServerSocket ss = new ServerSocket(50555);
    Socket s = ss.accept();
    System.out.println("connected to Client: " + s.getInetAddress() );

    DataInputStream in = new DataInputStream(s.getInputStream());
    System.out.println(in.readUTF());

    DataOutputStream out = new DataOutputStream(s.getOutputStream());
    out.writeUTF("Thank you from Server ");
    s.close();
  }
}
```



TCP Socket (cont.)

```
public class nu_18_socket_client {
  public static void main(String[] args) throws UnknownHostException, IOException {
    Socket c = new Socket("localhost", 50555);
        /* Data Output Stream / Input Stream from java.io */
    DataOutputStream out = new DataOutputStream(c.getOutputStream());
    out.writeUTF("Hello from " + c.getLocalSocketAddress());

    DataInputStream in = new DataInputStream(c.getInputStream());
    System.out.println("Server Says: " + in.readUTF());

    c.close();
  }
}
```

Important Methods from Socket Object

- Return Remote Port Number `c.getPort()`
- Return Local Port Number `c.getLocalPort()`
- Return Remote IP Address `c.getInetAddress()`
- Return Output Stream `c.getOutputStream()`
- Return Input Stream `c.getInputStream()`

InetAddress

- Get information about who is connecting to the server
- `InetAddress a = c.getInetAddress()`
- `InetAddress a = InetAddress.getByName("nu.edu.pk")`
- `InetAddress[] a = InetAddress.getAllByName("www.google.com")`
- `InetAddress a = InetAddress.getLocalHost()`
- Methods:
  - Get Client Host Name `a.getHostName()`
  - Get Client IP Address `a.getHostAddress()`
  - Is it Local Address? `a.isAnyLocalAddress()`
  - Is it Multicast Address? `a.isMulticastAddress()`
  - Ping Check (ms) `a.isReachable(2000)`

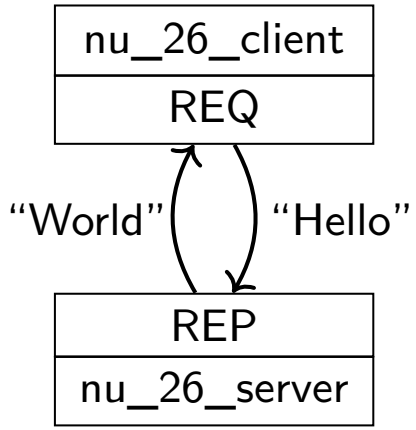
UDP

- `DatagramSocket s = new DatagramSocket(port)`
- `DatagramPacket d = new DatagramPacket(buffer, buffer.length)`
  - `byte[] buffer = new byte[1024]`
- `s.receive(d)`
- `DatagramPacket ack = new DatagramPacket(req.getData(), req.getLength(),`

ZeroMQ Overview

- ØMQ (www.zeromq.org)
- MQ Message Queue, Ø Symbolic (zero cost, zero latency, etc.)
- Intelligent Socket Library for Messaging
- FAST (8 Million msg/sec, 30 μ s Latency)
- Platform and Language Independent

Request Reply Message Pattern



- REQ-REP socket pair is in lockstep (Two `s.send()` will give -1 from the `s.recv()` call, and vice versa)
- **Context** helps manage socket (input: # of threads)
- All communication by bytes
- Connection String: inproc for thread to thread, ipc for process to process, tcp for box to box

```
import org.zeromq.ZMQ;

public class nu_26_zmqClient {
  public static void main(String[] args) {
    ZMQ.Context c = ZMQ.context(1);
    ZMQ.Socket  s = c.socket(ZMQ.REQ);
    s.connect ("tcp://localhost:5555");

    s.send ("Hello", 0);
    byte [] msg = s.recv(0);
    System.out.println (new String(msg));
    s.close();
    c.term();
  }
}
```

```
import org.zeromq.ZMQ;

public class nu_26_zmqServer {
  public static void main(String[] args) {
    ZMQ.Context c = ZMQ.context(1);
    ZMQ.Socket  s = c.socket(ZMQ.REP);
    s.bind ("tcp://*:5555");
    while (true) {
      byte [] msg = s.recv (0);
      s.send("World", 0);
    }
    s.close();
    c.term();
  }
}
```

Request Reply Message Pattern (cont.)

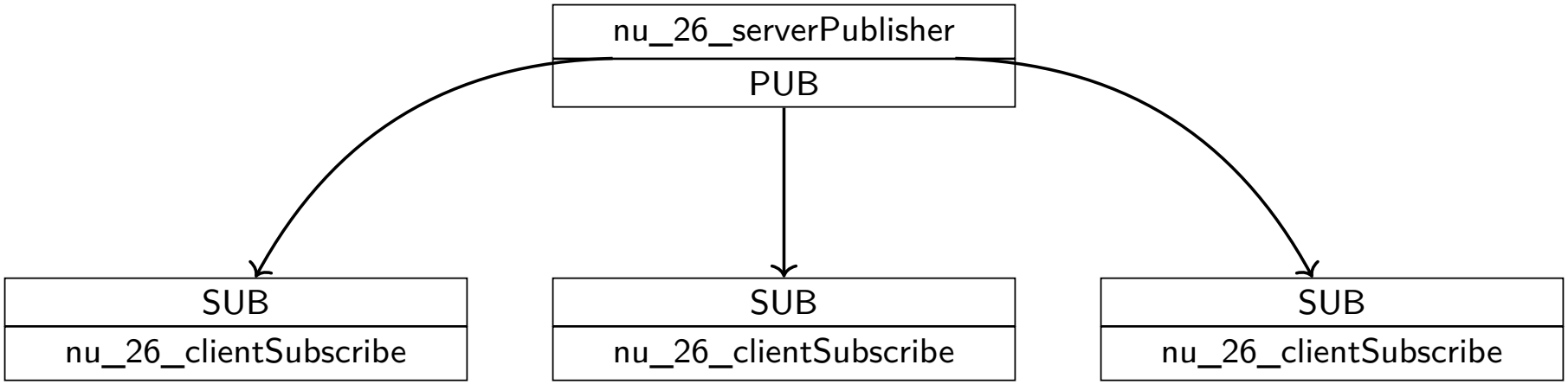
- Compilation & Running

```
javac -classpath /usr/local/share/java/zmq.jar nu_26_zmqClient.java
java -Djava.library.path=/usr/local/lib64 -classpath /usr/local/share/java/zmq.jar:. nu_26_zmqClient
```

- Server while loop condition can be set to

```
!Thread.currentThread().isInterrupted()
```

Publisher Subscriber Message Pattern



- Server Pushes Updates to a Number of Clients
- 1 Server Socket linked with Multiple Client Sockets (1:M communication)
- Publisher implements `bind()`, whereas subscribers implement `connect()` methods (it doesn't matter who is server and who is client)
- PUB-SUB socket pair is asynchronous

Publisher Subscriber Message Pattern (cont.)

```
ZMQ.Context c = ZMQ.context(1);
ZMQ.Socket s = c.socket(ZMQ.PUB);
s.bind ("tcp://*:5557");


int count = 0;
while ( /* Condition Check */ ) {
    String str = "Hello World " + count++;
    System.out.println(str);
    s.send(str.getBytes(), 0);
}
s.close();
c.term();
```

```
ZMQ.Context c = ZMQ.context(1);
ZMQ.Socket s = c.socket(ZMQ.SUB);
s.connect ("tcp://localhost:5557");

String filter = "World";
s.subscribe(filter.getBytes());


for (int i = 0; i < 100; i++) {
    byte [] buf = s.recv(0);
    System.out.println (new String(buf));
}
s.close();
c.term();
```

- Synchronization aspects:
  - Subscriber always misses a few messages that the publisher sends
  - Full synchronization if client starts before Server
  - If no subscriber available, then the publisher drops all messages (but they are generated)

Spark Overview

- Cluster computing platform designed for speed (mostly in-memory operations rather than file I/O)
- Support for Stream Processors (GPU's)
- API's available in Python, Java, Scala, and SQL

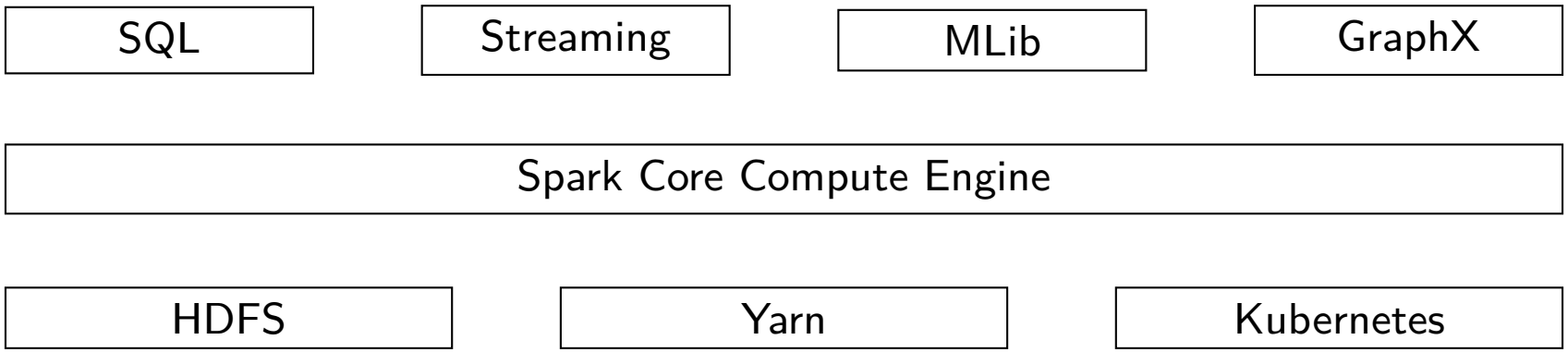
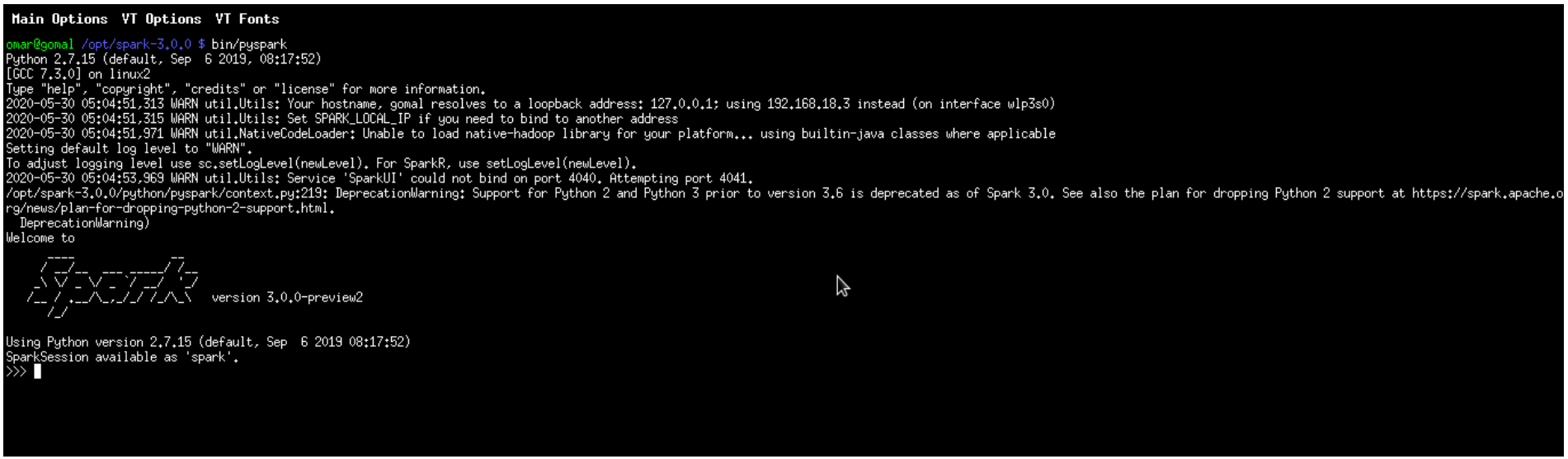


Figure 21: Spark System Architecture

- Web based Monitoring Interface on http://localhost:4040

Spark Shells

- `bin/pyspark` for Python
- `bin/spark-shell` for Scala
- `bin/sparkR` for R



- Can also be interfaced with Python Notebooks (e.g. Jupyter or iPython)

Hello World (Word Count)

```
lines = sc.textfile("README.md") # Must be on HDFS
                                 # lines is an RDD Object
lines.count()                    # Returns number of items
lines.first()                    # First line in RDD (or file)
```

Resilient Distributed Dataset RDD

- Computations performed on *distributed collections* that are automatically parallelized across a cluster.
- Immutable Objects (always new RDD returned)
- These Collections are the RDD's (more in coming slides)
- Created using `parallelize()` or `textfile()` methods of spark context.

Hello World (Word Count) (cont.)

Standalone Application

- Same API, but have to create Spark Contxt yourself
- Run using `bin/spark-submit`

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("My App")
sc   = SparkContext (conf = conf)

# User program from here onward

lines = sc.textFile("/4300-folder/4300-0.txt")
count = lines.count()

def hasBook(line):
    return "Book" in line

bookLines = lines.filter(hasBook)
bookcount = bookLines.count()
```

Core Spark Concepts

Driver Program (e.g., pyspark shell)

- Launches various parallel operations on a cluster
- Contains your applications main function
- Contains your applications distributed datasets
- Accesses Spark through a **Spark Context** object. This context is automatically created as object `sc`.

  ```
  <SparkContext master=local[*] appName=PySparkShell>
  ```
- RDD's created from Context

Executors

- Present on each Worker Node (computer)
- Managed by Driver

Core Spark Concepts (cont.)

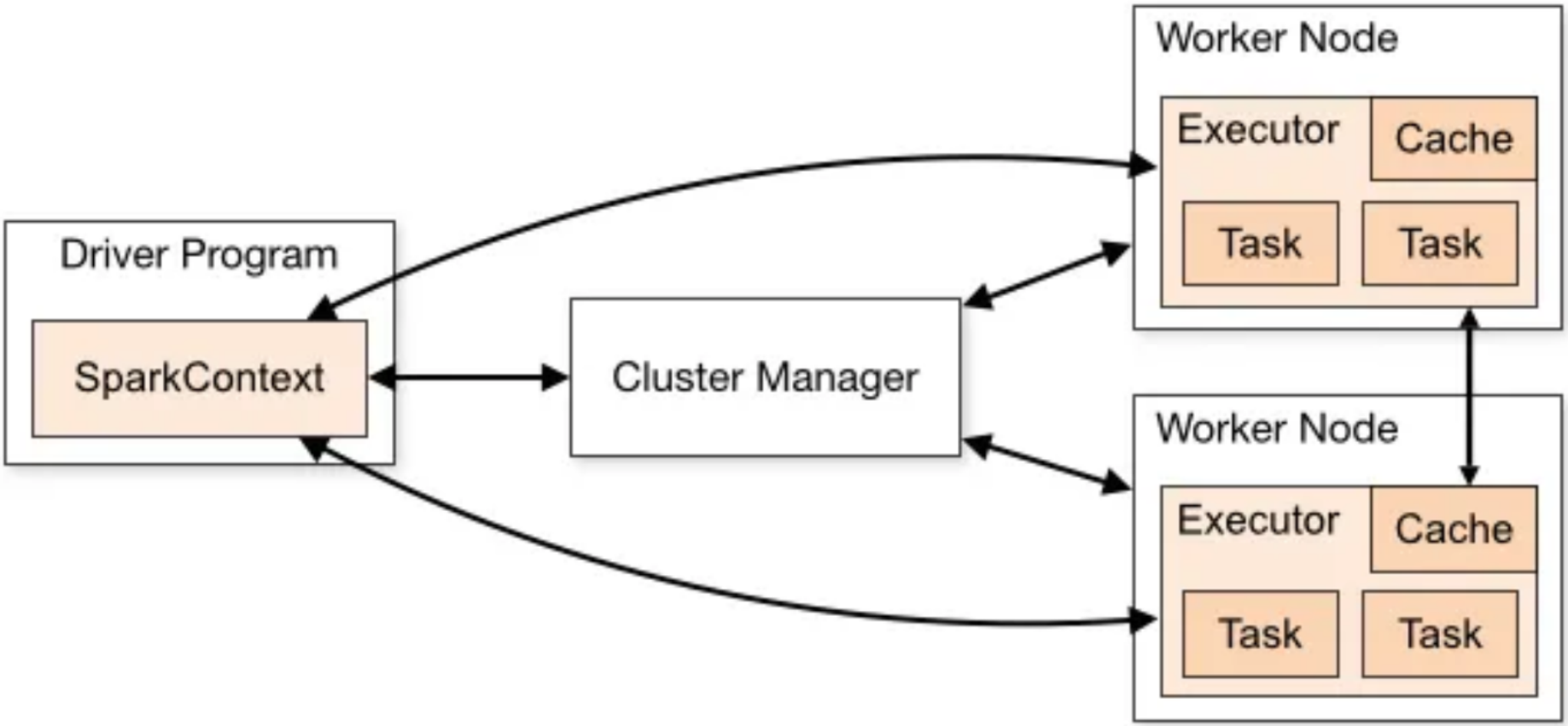


Figure 22: Components for Distributed Execution in Spark IMG: Apache Spark

Resilient Distributed Dataset (RDD)

Creation Approaches in Driver Program

- Loading an External Dataset from File `rddObj = sc.textFile("myFilename")`
- Collection of List objects `rddObj = sc.parallelize(["FAST", "I Like FAST"])`
  `rddObj = sc.parallelize([1, 2, 3, 4])`

Creation of DataFrame from RDD Object

```
dataCol = Seq("key", "value")
dataObj = Seq(("k1", v1), ("k2", v2))

rddObj  = sc.parallelize(dataObj)
dfRddObj = rddObj.toDF(dataCol)

dfRddObj.printSchema()
dfRddObj.show()
dfRddObj.select("key").show()
dfRddObj.filter(dfRddObj("value") > 100).show()
```

Resilient Distributed Dataset (RDD) (cont.)

Operations on RDD

- Actions (operations directly on RDD; Output displayed to Driver program, or to HDFS storage). For example: count(), first(), take(), collect()
- Transformations (new RDD from existing one). For example: Filtering(), Union(), Map(), FlatMap()

```
inputRDD  = sc.textFile("log.txt")

inputRDD.count()                                          # Action
inputRDD.first()                                          # Action

errorRDD   = inputRDD.filter(lambda x: "Error" in x)      # Transformation
warningRDD = inputRDD.filter(lambda x: "Warning" in x)    # Transformation

badLinesRDD = errorRDD.union(warningRDD)                  # Transformation

print("Bad Lines: " + badLinesRDD.count())                # Action
for line in badLinesRDD.take(10):                         # Action
```

Resilient Distributed Dataset (RDD) (cont.)

```
        print(line)                                          # or file write

for line in badlinesRDD.collect():                           # Danger Action
        print(line)
inputRDD  = sc.parallelize([1, 2, 3, 4])
squareRDD = inputRDD.map(lambda x: x * x).collect()          # Map Collect
for num in squareRDD:
        print("%d" % num)

inputRDD  = sc.parallelize(["Coffee Panda", "Happy Panda"])
outputRDD = inputRDD.map(lambda line: line.split(" "))
outputRDD.take(2)
# Displays: [['Coffee', 'Panda'], ['Happy', 'Panda']

outputRDD = inputRDD.flatMap(lambda line: line.split(" "))
outputRDD.take(4)
# Displays: ['Coffee', 'Panda', 'Happy', 'Panda']
```

Resilient Distributed Dataset (RDD) (cont.)

Lazy Loading Principle

- Compute/Retrive RDD only when required (determined through internal metadata)
- For transformation RDD's, maintain **Lineage Graph**
- Recompute again and again, any time you need it, with certain degree of caching (makes sense for large datasets). To override:

```
lines.persist()    # Hold data in memory
lines.count()
lines.first()
```

Other Transformation Operations

- `RDD1.distinct()` returns unique members
- `RDD1.union(RDD2)` returns Union
- `RDD1.intersection(RDD2)` returns Intersection
- `RDD1.subtract(RDD2)` returns RDD1 - RDD2
- `RDD1.cartesian(RDD2)` returns RDD1 $\times$ RDD2

Resilient Distributed Dataset (RDD) (cont.)

Saving RDD's (to HDFS)

- `lines.saveAsTextFile("Directory")`

Spark with Key Value Pairs

```
from pyspark import SparkContext, SparkConf

conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf=conf)

words = sc.textFile("/documenttxt").flatMap(lambda line: line.split(" "))

wc    = words.map(lambda word: (word, 1)).reduceByKey(lambda a,b : a +b)

wc.saveAsTextFile("/sparktest")
```