
CODE OPTIMIZATION & TYPES

Noor Muhammad Sk

Code Optimization

- **Goals of code optimization:**
- Remove redundant code without changing the meaning of program.
- **Objective:**
 - Reduce execution speed
 - Reduce code size
- Achieved through code transformation while preserving semantics

Code Optimization

- The method that is used for the modification of codes in order to improve the efficiency and quality of the code.

As a result of optimization,

- A program may become lesser in size,
 - Consume lesser memory,
 - Execute more rapidly, or
 - Performs fewer operations (input/output).
-
- An optimized program should possess exactly the same outputs and side effects as that of its non-optimized program.

Code Optimization

- Optimization can also be referred to as a *program transformation technique*, performed either by optimizers or programmers.
- An optimizer is a specialized software or an inbuilt unit of a compiler which is also called as an optimizing compiler.
- Modern processors are also used to optimize the order of execution of code instructions.
- Code optimization is generally performed at the end of the developmental stage as it reduces the readability and adds code in order to increase the performance.

Types of Code Optimization

- High-level optimizations
 - Intermediate level optimizations, and
 - Low-level optimizations
-
- ***High-level optimization*** is a language dependent type of optimization that operates at a level in the close vicinity of the source code.
 - High-level optimizations include inlining where a function call is replaced by the function body and partial evaluation which employs reordering of a loop, alignment of arrays, padding, layout.

Intermediate Code Optimization

- Language independent
- **The elimination of common sub Expressions**
- This type of compiler optimization probes for the instances of identical expressions by evaluating to the same value and researches whether it is valuable to replace them with a single variable which holds the computed value.

Example

Code before Optimization

```
S1 = 4 x i  
S2 = a[S1]  
S3 = 4 x j  
S4 = 4 x i // Redundant Expression  
S5 = n  
S6 = b[S4] + S5
```

Code after Optimization

```
S1 = 4 x i  
S2 = a[S1]  
S3 = 4 x j  
S5 = n  
S6 = b[S1] + S5
```

Example

- $A = 2 * (22.0 / 7.0) * r$
- $A = 6.2857 * r$

- $x = 12.4$
- $y = x / 2.3$
- Evaluate $x / 2.3$ as $12.4 / 2.3$

Example

Original Code

```
a = (b + c) * m;  
x = b + c;  
y = (b + c) * z;
```

Optimized Code

```
T1 = b + c;  
a = T1 * m;  
x = T1;  
y = T1 * z;
```

Intermediate Code Optimization

- **Constant propagations**
- Here, expressions which can be evaluated at compile time are identified and replaced with their values.

Example

- $\text{pi} = 3.14$
 - $\text{radius} = 10$
 - Area of circle = $\text{pi} \times \text{radius} \times \text{radius}$
-
- Here, this technique will substitute the value of the variables ‘ pi ’ and ‘ radius ’ at the compile time
 - It will evaluate the expression $3.14 \times 10 \times 10$ at the compile time which will save the time during the program execution.

Intermediate Code Optimization

- **Jump threading**
- This involves an optimization of jump directly into a second one.
- The second condition is eliminated if it is an inverse or a subset of the first which can be done effortlessly in a single pass through the program.
- Acyclic chained jumps are followed till the compiler arrives at a fixed point.

Example

10. a = SomeNumber();

20. IF a > 10 GOTO 50

...

50. IF a > 0 GOTO 100

...

- The jump on line 50 will always be taken if the jump on line 20 is taken.
- Therefore the jump on line 20 may safely be modified to jump directly to line 100.

Intermediate Code Optimization

- **Loop invariant code motion**
- This is also known as hoisting or scalar promotion.
- A loop invariant contains expressions that can be taken outside the body of a loop without any impact on the semantics of the program.
- The above-mentioned movement is performed automatically by loop invariant code motion.

Example

Code before Optimization

```
for ( int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
```

Code after Optimization

```
x = y + z ;
for ( int j = 0 ; j < n ; j ++)
{
a[j] = 6 x j;
}
```

Example

```
a = 200;  
while(a>0)  
{  
    b = x + y;  
    if (a % b == 0}  
    printf("%d", a);  
}
```

```
//This code optimized as  
a = 200;  
b = x + y;  
while(a>0)  
{  
    if (a % b == 0}  
    printf("%d", a);  
}
```

Intermediate Code Optimization

- **Dead code elimination**
- Here, as the name indicates, the codes that do not affect the program results are eliminated.
- It has a lot of benefits including reduction of program size and running time.
- It also simplifies the program structure.
- Dead code elimination is also known as DCE, dead code removal, dead code stripping, or dead code strip.

Example

c = a * b

x = a

till

d = a * b + 4

//After elimination :

c = a * b

till

d = a * b + 4

Example

Code before Optimization

```
i = 0 ;  
if (i == 1)  
{  
    a = x + 5 ;  
}
```

Code after Optimization

```
i = 0 ;
```

Intermediate Code Optimization

- **Strength reduction**
- This compiler optimization replaces expensive operations with equivalent and more efficient ones, but less expensive.
- For example, replace a multiplication within a loop with an addition.

Example

```
i = 1  
While(i<10)  
{  
    y = i * 4;  
    i++;  
}
```

```
//After Reduction  
i = 1  
t = 4  
  
while(t<40)  
{  
    y = t;  
    t = t + 4;  
}
```

Example

Code before Optimization
$B = A \times 2$

Code after Optimization
$B = A + A$

Low-level optimization

- Highly specific to the type of architecture
- **Register allocation** – Here, a big number of target program variables are assigned to a small number of CPU registers.
- This can happen over a local register allocation or a global register allocation or an inter-procedural register allocation.
- **Instruction Scheduling** – This is used to improve an instruction level parallelism that in turn improves the performance of machines with instruction pipelines.
- It will not change the meaning of the code but rearranges the order of instructions to avoid pipeline stalls.
- Semantically ambiguous operations are also avoided.

Low-level optimization

- **Floating-point units utilization** – Floating point units are designed specifically to carry out operations of floating point numbers like addition, subtraction, etc.
- The features of these units are utilized in low-level optimizations which are highly specific to the type of architecture.
- **Branch prediction** – Branch prediction techniques help to guess in which way a branch functions even though it is not known definitively which will be of great help for the betterment of results.
- **Peephole and profile-based optimization** – *Peephole optimization* technique is carried out over small code sections at a time to transform them by replacing with shorter or faster sets of instructions.
- This set is called as a peephole.

Machine-independent optimization and machine-dependent optimization

- ***Machine-independent optimization*** phase tries to improve the intermediate code to obtain a better output.
- The optimized intermediate code does not involve any absolute memory locations or CPU registers.
- ***Machine-dependent optimization*** is done after generation of the target code which is transformed according to target machine architecture.
- This involves CPU registers and may have absolute memory references.
- To conclude, code optimization is certainly required as it provides a cleaner code base, higher consistency, faster sites, better readability and much more.

Unroll loop bodies into equivalent sequential code:

Original code

for i in 1..10 loop

 a[i][i] = 2*i;

end loop

Optimized code

A[1][1] := 2;

A[2][2] := 4;

...

Loop unrolling:

Original Code

```
for i:=1 to 20 do
begin
    for j:= 1 to 2 do
        write (x[i, j])
end
```

Optimized Code

```
for i:=1 to 20 do
begin
    write (x[i, 1], x[i,2])
end
```

If Optimization

```
void f (int *p)
{
    if (p)
    {
        g(1);
        if (p)
            g(2);
        g(3);
    }
    return;
}
```

```
void f (int *p)
{
    if (p)
    {
        g(1);
        g(2);
        g(3);
    }
    return;
}
```

Example

```
void f (int *p)
{
    if (p) g(1);
    if (p) g(2);
    return;
}
```

```
void f (int *p)
{
    if (p)
    {
        g(1);
        g(2);
    }
    return;
}
```

HIGH PERFORMANCE COMPUTING

Dr Noor Mahamamd Sk

Center for High Performance Reconfigurable Computing

Logistics

- Course is @ A slot
- Class Timings
 - MONDAY – 0800 Hrs
 - TUESDAY – 0900 Hrs
 - THURSDAY – 1000 Hrs
- Attendance as per Institute Norms

Evaluation Policy

- Quiz 1 – 20 Marks
- Quiz 2 – 20 Marks
- End Semester – 60 Marks

Motivation

- A few examples of large scale problems for motivation.
Currently that means Tera-scale or Peta-scale
- Proc. speed measured in Gigahertz (hertz = cycles per sec.)
- One operation may take several cycles.
- So a 1 GHz processor does $> 10^8$ operations per second.
- Exascale is a billion times more. (Also flops and bytes.)

Kilo	thousand (10^3)	2^{10}
Mega	million (10^6)	2^{20}
Giga	billion (10^9)	2^{30}
Tera	trillion (10^{12})	2^{40}
Peta	(10^{15})	2^{50}
Exa	(10^{18})	2^{60}

How long does it take to solve a linear system?

- Solving an $n \times n$ linear system $Ax = b$ takes $\approx (1/3)n^3$ flops (using Gaussian elimination for a dense matrix)
- On a 100 MFlops system:

n	flops	time
• 10	3.3×10^2	0.0000033 seconds
• 100	3.3×10^5	0.0033 seconds
• 1000	3.3×10^8	3.33 seconds
• 10000	3.3×10^{11}	3333 seconds = 55.5 minutes
• 100000	3.3×10^{14}	333333 seconds = 926 hours
• 1000000	3.3×10^{17}	926,000 hours = 105 years
- This assumes data transfer not a problem – which it is. Often it's the bottleneck.
- A $10^6 \times 10^6$ matrix has 10^{12} elements – this is 8 terabytes!

Parallel Computing

- **High performance available today only through parallelism**
- Provides alternatives when individual processor speeds reach limits imposed by fundamental physical laws
- Leverages inexpensive commodity parts
- Provides cost-effective means (sometimes only means) of meeting enormous demands of large-scale simulations
- **However:**
 - Relatively immature computing environment, lack of available software
 - Rapid pace of change (code soon obsolete)
 - Complexity of parallel programming (algorithm and software development much harder; lack of standardization)
 - Physical constraints (power, cooling) will limit growth

Some Terminology

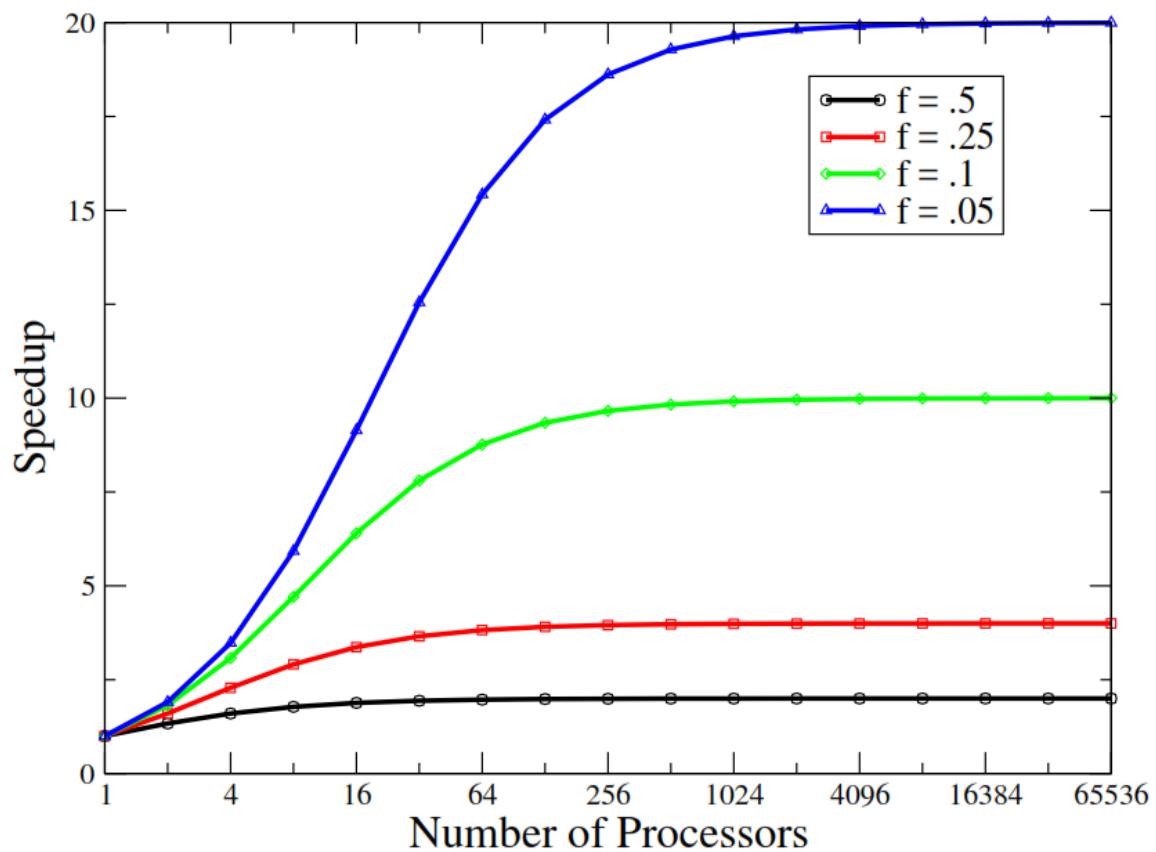
- Core a single computing unit with its own independent control
- Multicore a processor having several cores that can access the same memory concurrently
- A computation is decomposed into several parts called tasks that can be computed in parallel
- Finding enough parallelism is (one of the) crucial steps for high performance (Amdahl's law).

Amdahl's Law

- Used to predict maximum speedup using multiple processors
- Let f = fraction of work performed sequentially
- $(1-f)$ = fraction of work that is parallelizable
- P = number of processors
- On 1 cpu: $T_1 = f + (1-f) = 1.$
- On P processors: $T_P = f + (1-f)/P$
- Speedup $S = T_1/T_P = \{1/(f+(1-f)/P)\} < 1/f$
- Speedup limited by sequential part

Amdahl's Law

- Speedup as a function of sequential fraction

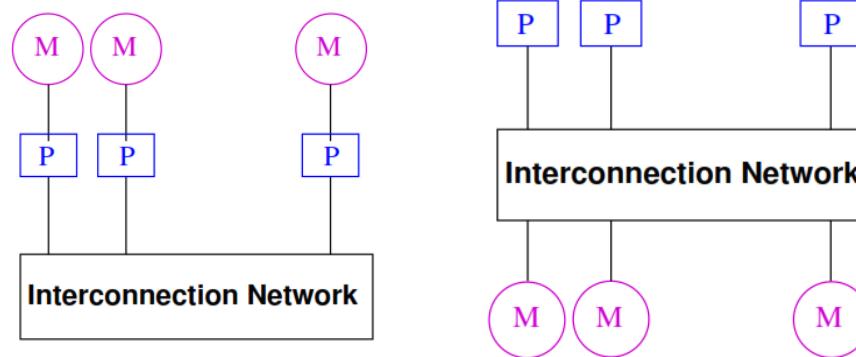


Terminology

- The size of the tasks is called the granularity; tasks may have different sizes
- The assignment of tasks in a given order is scheduling.
- Dependencies between tasks creates constraints for scheduling.
- Mapping: assignment of tasks to physical computation units
- If all tasks take the same amount of time work is load balanced.
(otherwise some cpus are waiting for others finish - this is idle time)
- Parallel programs need synchronization to execute correctly, obey constraints, etc.
- Barrier operations are a form of coordination where all processes or threads wait until all others have also reached that point.
- Some of this done by the OS and some by the programmer. This is what makes parallel programming hard.

Terminology

- One way to classify machines distinguishes between shared memory global memory can be accessed by all processors or cores.
- Information exchanged between threads using shared variables written by one thread and read by another.
- Need to coordinate access to shared variables.
- Distributed memory private memory for each processor, only accessible this processor, so no synchronization for memory accesses needed.
- Information exchanged by sending data from one processor to another via an interconnection network using explicit communication operations.



Terminology

- We will also be concerned with Parallel execution time
- Time for the computation plus time for data exchange (communication), synchronization, other overhead (redundant computation), cost of starting threads/process)
- Quantitative measures of parallel execution time are
- Speedup = parallel execution time / sequential execution time on one processor
- Efficiency = speedup / number of processors

Principles of Parallel Computing

- Finding enough parallelism (Amdahl's law)
- Granularity need large enough amount of work to hide the overhead
- Locality large memories are slow fast memories are small.
- Load balance
- Coordination and synchronization
- Performance modeling
- All these things make parallel programming harder than sequential programming.

Flynn's Taxonomy

- Early classification of parallel computing architectures given by M. Flynn (1972) using number of instruction streams and data streams. Still used.
- **Single Instruction Single Data (SISD)** conventional sequential computer with one processor, single program and data storage.
- **Multiple Instruction Single Data (MISD)** used for fault tolerance (Space Shuttle) - from Wikipedia
- **Single Instruction Multiple Data (SIMD)** each processing element uses same instruction applied synchronously in parallel to different data elements (Connection Machine, GPUs).
- **Multiple Instruction Multiple Data (MIMD)** each processing element loads separate instruction and separate data elements; processors work asynchronously.
- Since 2006 top ten supercomputers of this type
- Update: **Single Program Multiple Data (SPMD)** autonomous processors executing same program but not in lockstep. Most common style of programming.

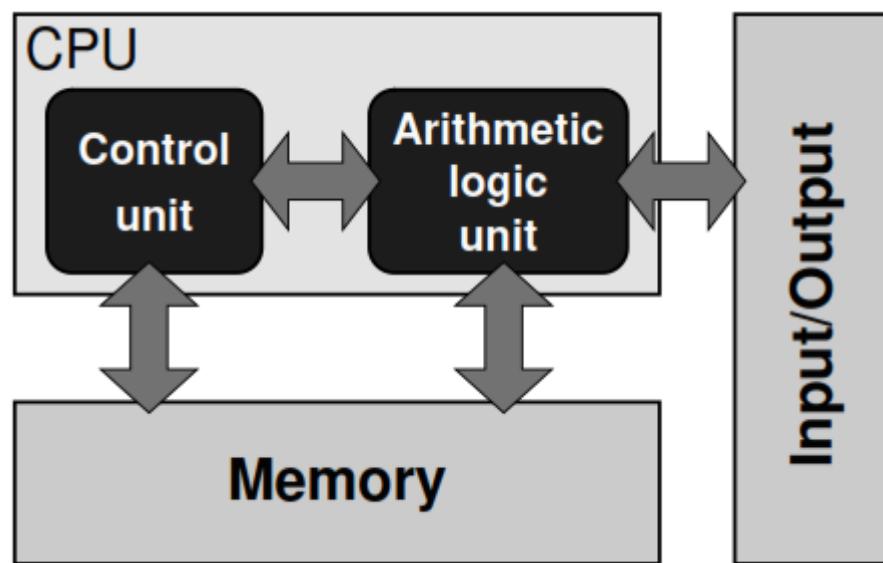
Algorithm Taxonomy - the 7 Dwarfs

- dwarf - coined by Phil Colella, an algorithmic kernel that captures a pattern of computation and communication.
- Specify at a high level of abstraction so that holds across a broad range of applications.
- Dense Linear Algebra
- Sparse Linear Algebra
- FFT (now Spectral Methods)
- Particle Methods (now N-Body Methods)
- Structured Grids (including locally structured, e.g. AMR)
- Unstructured Grids
- Monte Carlo (now MapReduce)
- **6 more subsequently added:**
- Combinatorial Logic, Graph Traversal, Dynamic Programming, Backtracking and Branch and Bound, Graphical Models, and Finite State Machines.

MODERN PROCESSORS

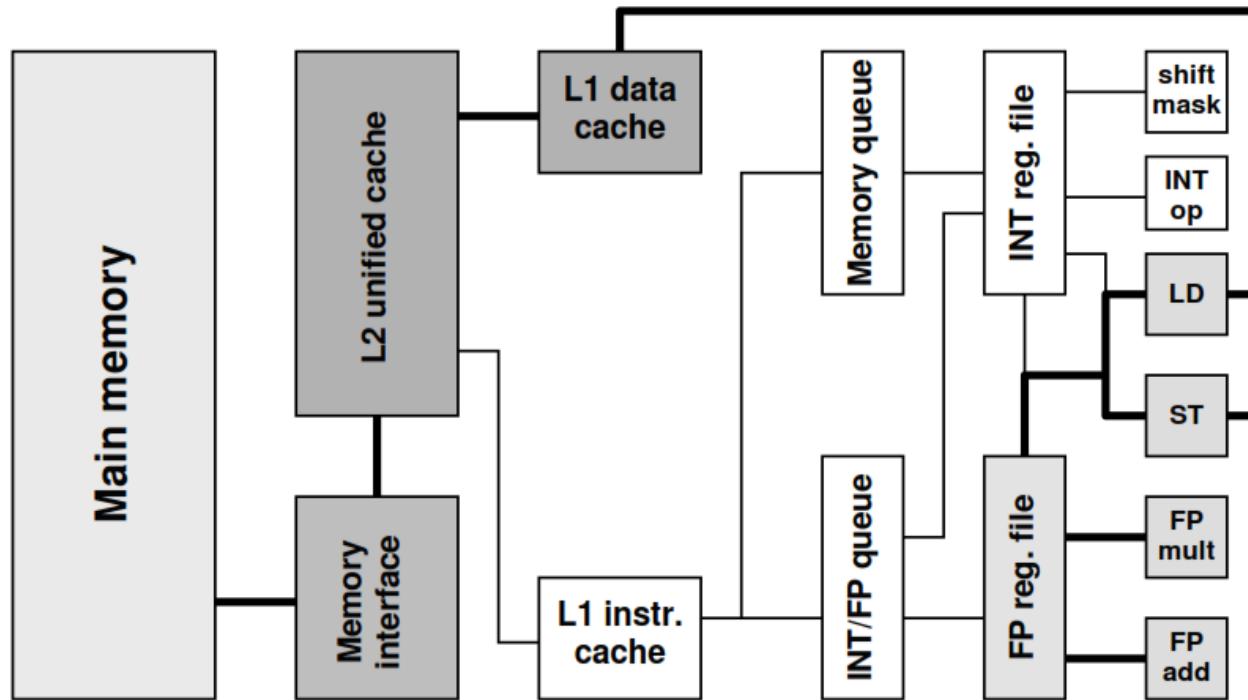
Stored-program computer architecture

- Stored-program computer architectural concept.
- The “program,” which feeds the control unit, is stored in memory together with any data the arithmetic unit requires.



General-purpose cache-based microprocessor architecture

- Simplified block diagram of a typical cache-based microprocessor (one core).
- Other cores on the same chip or package (socket) can share resources like caches or the memory interface.
- The functional blocks and data paths most relevant to performance issues in scientific computing are highlighted.



Performance metrics and benchmarks

- All the components of a CPU core can operate at some maximum speed called peak performance.
- The performance at which the FP units generate results for multiply and add operations is measured in floating-point operations per second (Flops/sec)
- Microprocessors are designed to deliver at most two or four double-precision floating-point results per clock cycle.
- With typical clock frequencies between 2 and 3GHz, this leads to a peak arithmetic performance between 4 and 12GFlops/sec per core.

FLOPS = Sockets x (Cores/Socket) x (Cycles/Second) x (FLOPS/Cycle)

Intel Core i7–970, capable of 4 double-precision or 8 single-precision floating-point operations per cycle.

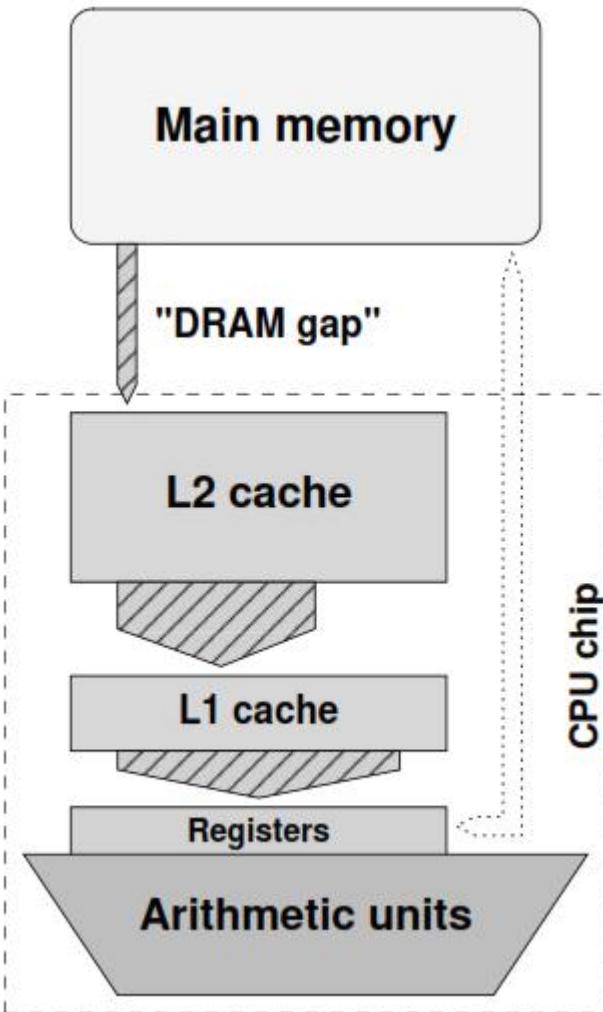
Example

- Intel Core i7–970 has 6 cores. If it is running at 3.46 GHz, the formula would be:
- $1 \text{ (socket)} * 6 \text{ (cores)} * 3,460,000,000 \text{ (cycles per second)} * 8 \text{ (single-precision FLOPs per second)} = 166,080,000,000 \text{ single-precision FLOPs per second} = 166 \text{ GFLOPS}$
- or
- $83,040,000,000 \text{ double-precision FLOPs per second} = 88 \text{ GFLOPS.}$

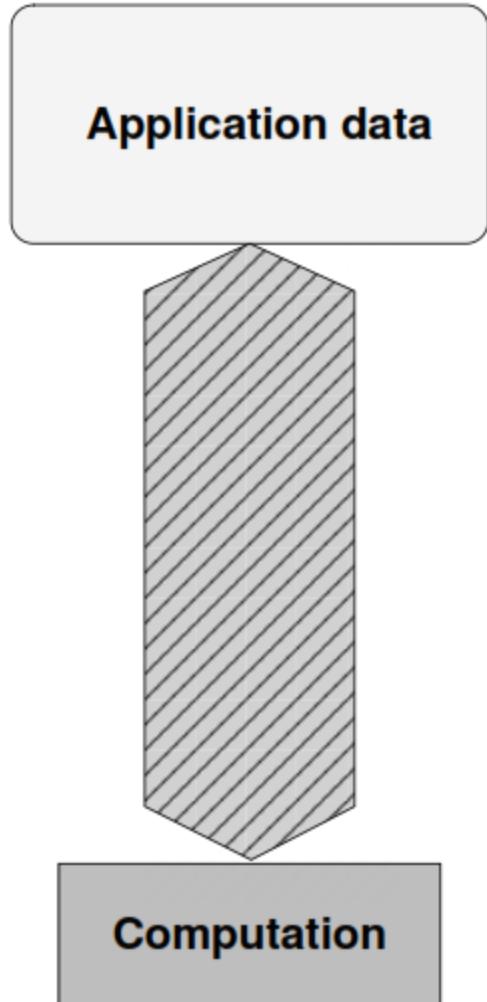
Performance Measurement

- The most sensible time measure in benchmarking is wall clock time, also called elapsed time.

Memory



- The “DRAM gap” denotes the large discrepancy between main memory and cache bandwidths.
- This model must be mapped to the data access requirements of an application.



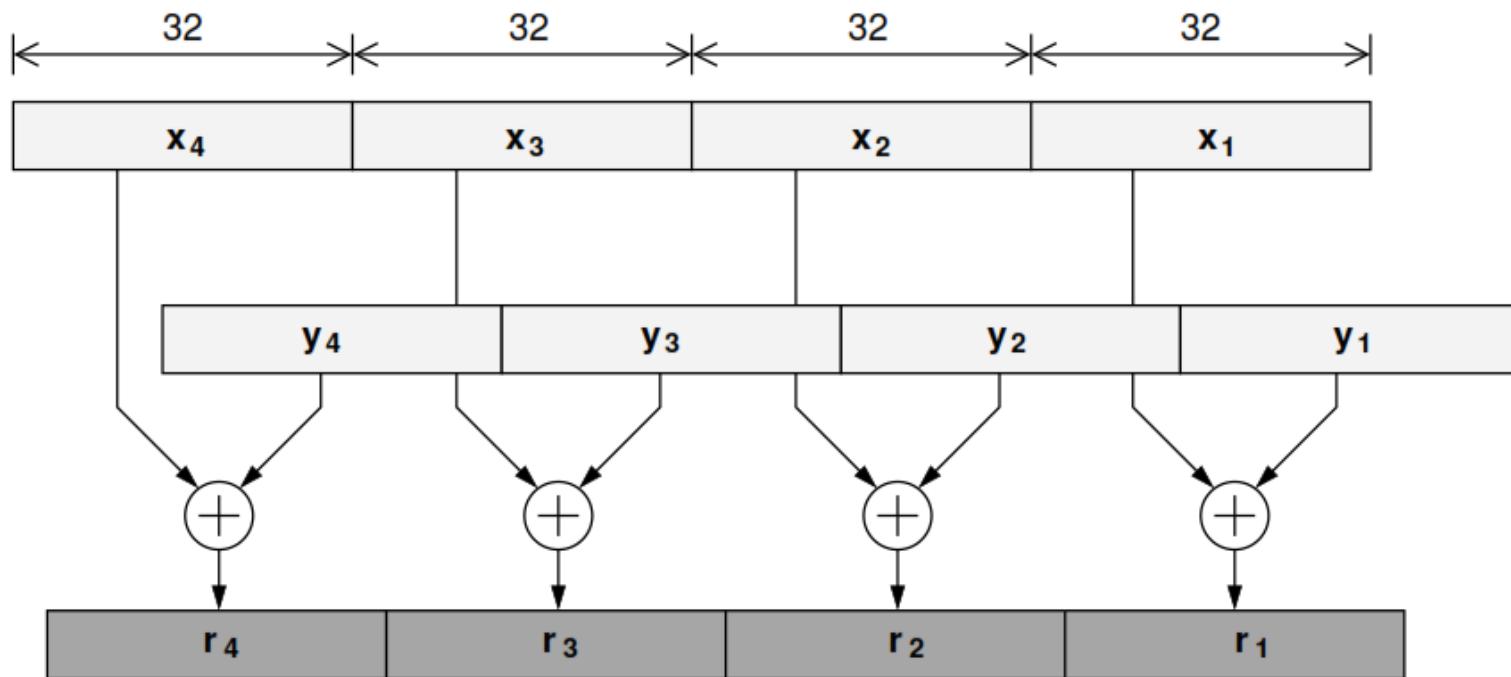
Superscalarity

- If a processor is designed to be capable of executing more than one instruction or, producing more than one “result” per cycle
- Multiple instructions can be fetched and decoded concurrently (3–6 nowadays).
- Address and other integer calculations are performed in multiple integer (add, mult, shift, mask) units (2–6).
- Multiple floating-point pipelines can run in parallel.
- Often there are one or two combined multiply-add pipes that perform $a=b+c * d$ with a throughput of one each.
- Caches are fast enough to sustain more than one load or store operation per cycle
- The number of available execution units for loads and stores reflects that (2–4).

SIMD

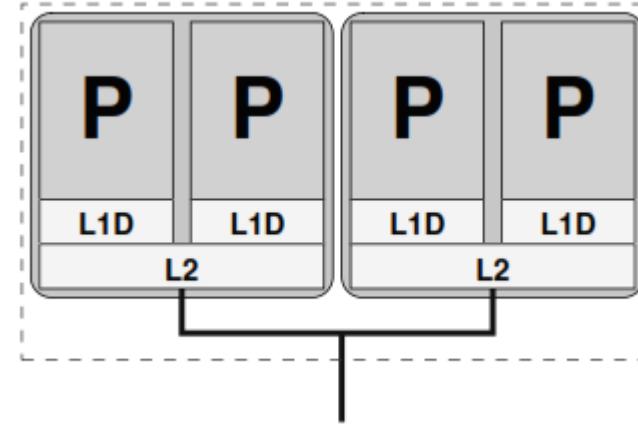
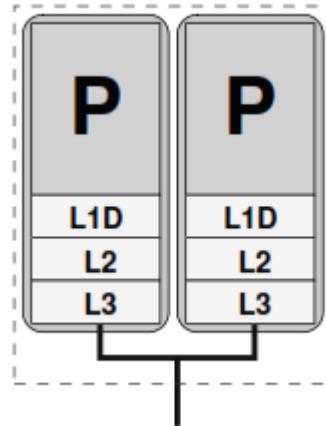
- The SIMD concept became widely known with the first vector supercomputers in the 1970s
- SIMD is the fundamental design principle for the massively parallel *Connection Machines in the 1980s and early 1990s*
- Many recent cache-based processors have instruction set extensions for both integer and floating-point SIMD operations
- They allow the concurrent execution of arithmetic operations on a “wide” register that can hold, e.g., two DP or four SP floating-point words

SIMD Operation



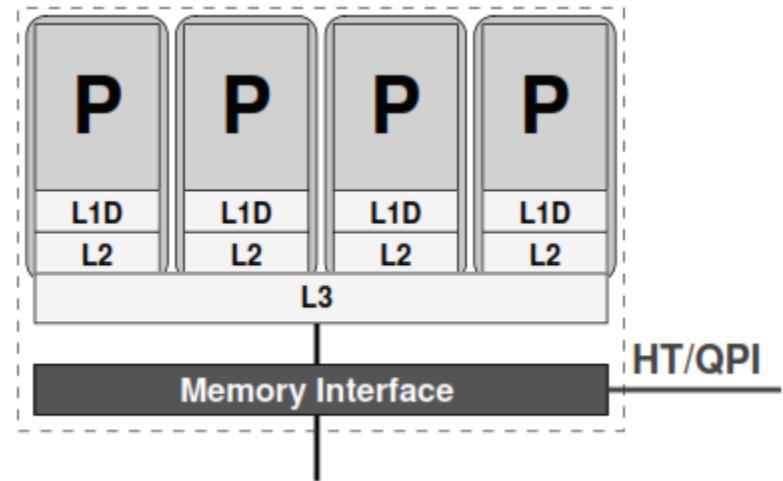
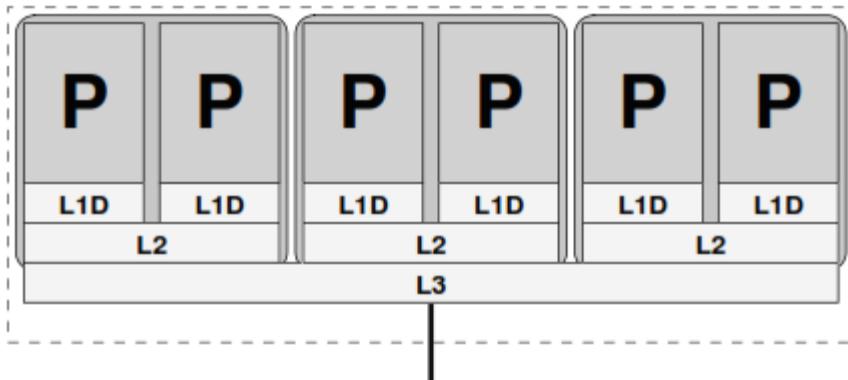
- Two 128-bit registers hold four single-precision floating-point values each.
- A single instruction can initiate four additions at once.

Multicore Processors



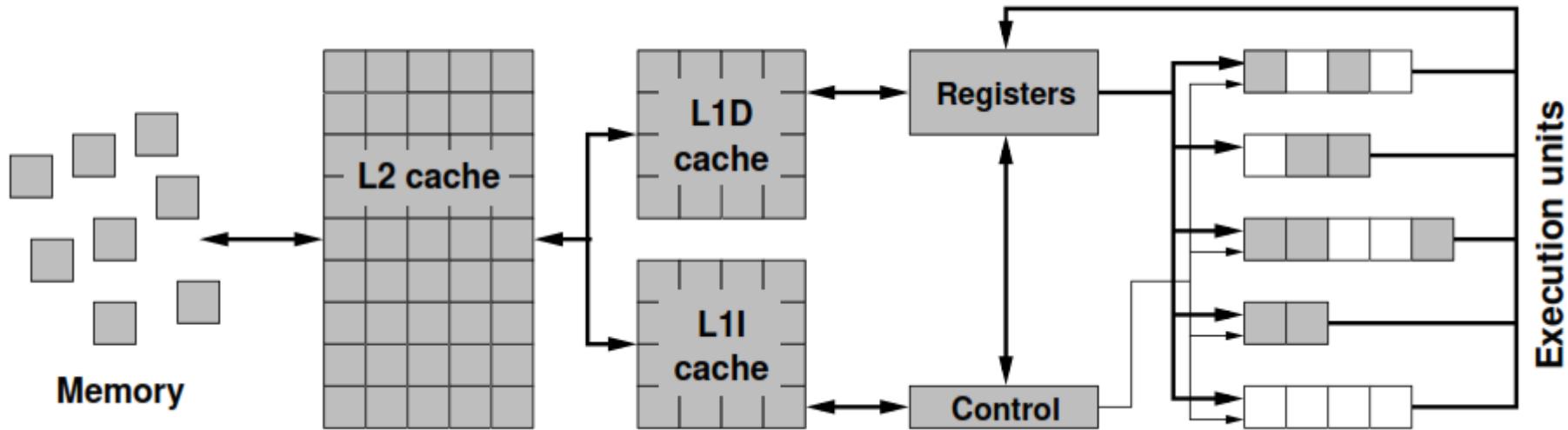
- Dual-core processor chip with separate L1, L2, and L3 caches.
- Each core constitutes its own cache group on all levels.
- Quad-core processor chip, consisting of two dual-cores.
 - Each dual-core has shared L2 and separate L1 caches.
 - There are two dual-core L2 groups.

Multicore Processors



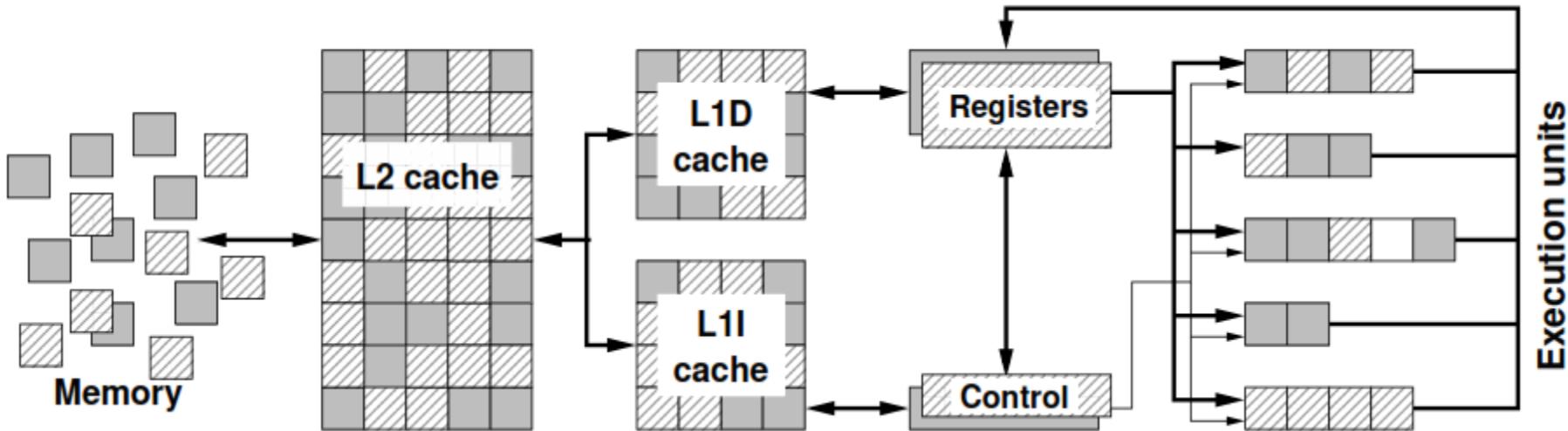
- Hexa-core processor chip with separate L1 caches, shared L2 caches for pairs of cores and a shared L3 cache for all cores.
- L2 groups are dual-cores, and the L3 group is the whole chip.
- Quad-core processor chip with separate L1 and L2 and a shared L3 cache.
 - There are four single-core L2 groups, and the L3 group is the whole chip.
 - A built-in memory interface allows to attach memory and other sockets directly without a chipset.

Multithreaded processors



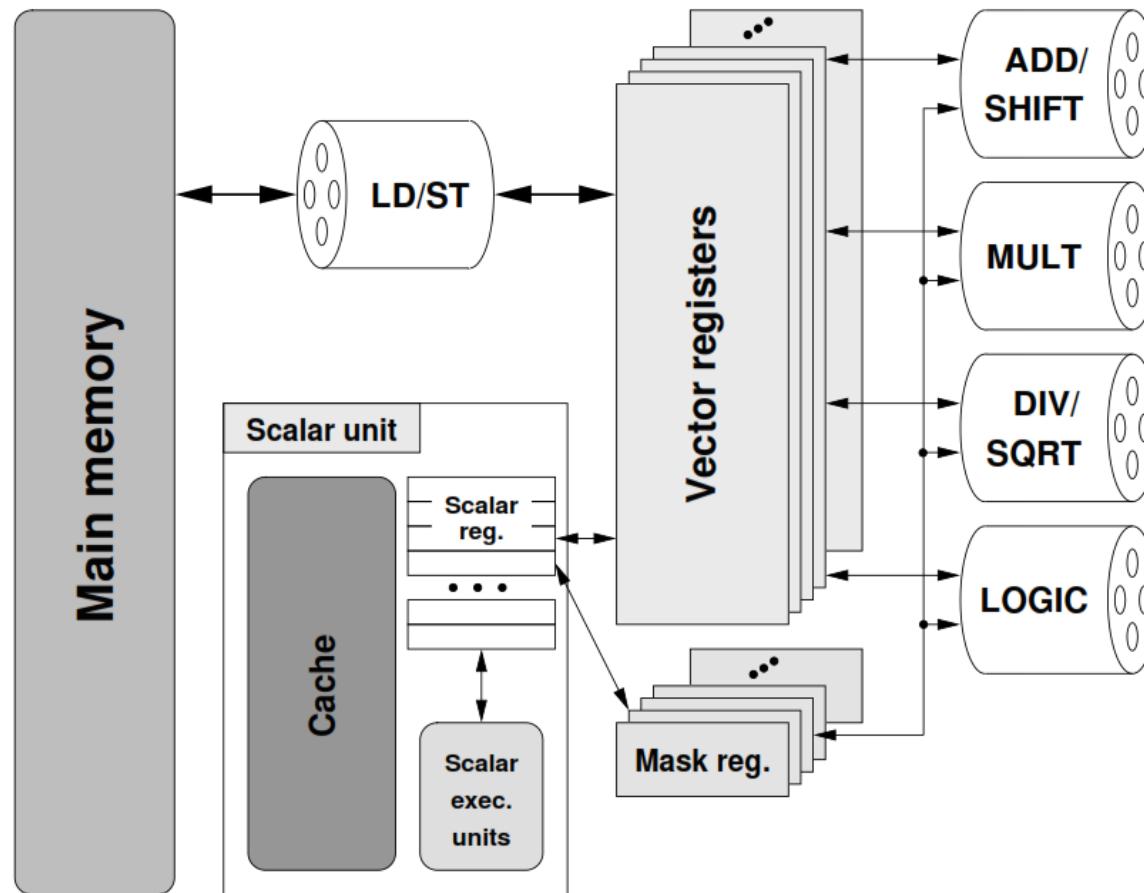
- Simplified diagram of control/data flow in a (multi-)pipelined microprocessor without SMT.
- White boxes in the execution units denote pipeline bubbles (stall cycles).

SMT



- Simplified diagram of control/data flow in a (multi-)pipelined microprocessor with fine-grained two-way SMT.
- Two instruction streams (threads) share resources like caches and pipelines but retain their respective architectural state (registers, control units).

Vector Processors



Reference

- **Georg Hager and Gerhard Wellein**, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.

BASIC OPTIMIZATION TECHNIQUES FOR SERIAL CODE

Dr Noor Muhammad Sk

Center for High Performance Reconfigurable Computing

Scalar Profiling

- Gathering information about a program’s behavior, specifically its use of resources, is called *profiling*.
- The most important “resource” in terms of high performance computing is runtime.
- A common profiling strategy is to find out how much time is spent in the different functions, and lines of a code.
- This will help to identify *hot spots*,
- *The parts of the program that require the dominant fraction of runtime.*
- These hot spots are subsequently analyzed for possible optimization opportunities.

Identifying Bottlenecks in Performance

- Even if hot spots have been determined, it is sometimes not clear from the start what the actual reasons for a particular performance bottleneck
- In such a case whether data access to main memory or pipeline stalls limit performance.
- If data access is the problem, it may not be straightforward to identify which data items accessed in a complex piece of code actually cause the most delay.
- *Hardware performance counters may help to resolve such issues.*
- *They Provide all current processors and allow deep insights into the use of resources within the chip and the system.*
- When nothing can be done any more to accelerate a serial code any further.
- It is essential for the user to be able to identify the point where additional optimization efforts are useless.

Function- and Line-based runtime profiling

- Two technologies are used for function- and line-based profiling: Code *instrumentation and sampling*
- Instrumentation works by letting the compiler modify each function call, inserting some code that logs the call, its caller (or the complete call stack) and probably how much time it required.
- This technique incurs significant overhead, especially if the code contains many functions with short runtime.
- The instrumentation code will try to compensate for that, but there is always some residual uncertainty.

Function- and Line-based runtime profiling

- Sampling is less invasive:
- The program is interrupted at periodic intervals, e.g., 10 milliseconds, and the program counter (and possibly the current call stack) is recorded.
- Necessarily this process is statistical by nature, but the longer the code runs, the more accurate the results will be.
- If the compiler has equipped the object code with appropriate information, sampling can deliver execution time information down to the source line and even machine code level.
- Instrumentation is necessarily limited to functions or *basic blocks* (*code with one entry and one exit point with no calls or jumps in between*) for efficiency reasons.

Function Profiling

- The most widely used profiling tool is **gprof** from the GNU binutils package.
- **gprof** uses both instrumentation and sampling to collect a flat function profile as well as a callgraph profile, also called a *butterfly graph*.
- **In order to activate profiling:**
- The code must be compiled with an appropriate option (many modern compilers can generate gprof-compliant instrumentation; for the GCC, use -pg) and run once.
- This produces a non-human-readable file gmon.out, to be interpreted by the gprof program.

Flat Profile

- The flat profile contains information about execution times of all the program's functions and how often they were called:
 - There is one line for each function.
-

1	%	cumulative	self		self	total	
2	time	seconds	seconds	calls	ms/call	ms/call	name
3	70.45	5.14	5.14	26074562	0.00	0.00	intersect
4	26.01	7.03	1.90	4000000	0.00	0.00	shade
5	3.72	7.30	0.27	100	2.71	73.03	calc_tile

The columns can be interpreted as follows:

- **%time**
- Percentage of the total execution time program spent in this function
- All functions combined should add up to 100%
- **cumulative seconds**
- Cumulative sum of exclusive runtimes of all functions up to and including this one.
- **self seconds**
- Number of seconds used by this function (exclusive).
- By default, the list is sorted according to this field.
- **calls** The number of times this function was called.
- **self ms/call** Average number of milliseconds per call that were spent in this function (exclusive).
- **total ms/call** Average number of milliseconds per call that were spent in this function, including its callees (inclusive).

Analysis - Example

- In the example above, optimization attempts would definitely start with the `intersect()` function, and `shade()` would also deserve a closer look.
- The corresponding exclusive percentages can hint at the maximum possible gain. If, e.g., `shade()` could be optimized to become twice as fast, the whole program would run in roughly $7.3 - 0.95 = 6.35$ seconds, i.e., about 15% faster.

Outcome of a profiling

- The outcome of a profiling run can depend crucially on the ability of the compiler to perform *function inlining*.
- *Inlining* is an optimization technique that replaces a function call by the body of the **callee**, reducing overhead.
- If inlining is allowed, the profiler output may be strongly distorted when some hot spot function gets inlined and its runtime is attributed to the caller.
- If the compiler/profiler combination has no support for correct profiling of inlined functions, it may be required to disallow inlining altogether.
- Of course, this may itself have some significant impact on program performance characteristics.

Flat Profile

- A flat profile contains a lot of information
- It does not reveal how the runtime contribution of a certain function is composed of several different callers
- Which other functions (callees) are called from it, and which contribution to runtime they in turn incur.
- This data is provided by the *butterfly graph, or callgraph profile*:

```
1  index % time      self   children    called      name
2                      0.27   7.03     100/100      main [2]
3  [1]      99.9      0.27   7.03       100      calc_tile [1]
4                      1.90   5.14 4000000/4000000      shade [3]
5  -----
6                      <spontaneous>
7  [2]      99.9      0.00   7.30      100/100      main [2]
8                      0.27   7.03      100/100      calc_tile [1]
9  -----
10                     5517592      shade [3]
11                     1.90   5.14 4000000/4000000      calc_tile [1]
12 [3]      96.2      1.90   5.14 4000000+5517592 shade [3]
13                     5.14   0.00 26074562/26074562      intersect [4]
14                     5517592      shade [3]
15  -----
16                     5.14   0.00 26074562/26074562      shade [3]
17 [4]      70.2      5.14   0.00 26074562      intersect [4]
```

The columns can be interpreted as follows:

- **%time** The percentage of overall runtime spent in this function, including its callees (inclusive time).
- This should be identical to the product of the number of calls and the time per call on the flat profile.
- **self** For each indexed function, this is exclusive execution time (identical to flat profile).
- For its callers (callees), it denotes the inclusive time this function (each callee) contributed to each caller (this function).
- **children** For each indexed function, this is inclusive minus exclusive runtime, i.e., the contribution of all its callees to inclusive time.
- Part of this time contributes to inclusive runtime of each of the function's callers and is denoted in the respective caller rows.
- **called** denotes the number of times the function was called (probably split into recursive plus nonrecursive contributions).
- Which fraction of the number of calls came from each caller is shown in the caller row, whereas the fraction of calls for each callee that was initiated from this function can be found in the callee rows.

Example 2

```
#include<stdio.h>
void func4(void)
{
    printf("\n Inside func4() \n");
    for(int count=0;count<=0xFFFF;count++);
}
void func3(void)
{
    printf("\n Inside func3() \n");
    for(int count=0;count<=0xFFFFFFFF;count++);
}
void func2(void)
{
    printf("\n Inside func2() \n");
    for(int count=0;count<=0xFFFF;count++); func3();
}
void func1(void)
{
    printf("\n Inside func1() \n");
    for(int count=0;count<=0xFFFFFFFF;count++); func2();
}
int main(void)
{
    printf("\n main() starts...\n");
    for(int count=0;count<=0xFFFF;count++;

        func1();
        func4();
        printf("\n main() ends...\n");

        return 0;
}
```

Compile

- `gcc -pg -g program_name.c`
- `./a.out`
- `gprof a.out gmon.out`

Flat profile:

- Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
95.20	0.54	0.54	1	542.66	542.66	func3
5.29	0.57	0.03	1	30.15	572.81	func1
0.00	0.57	0.00	1	0.00	542.66	func2
0.00	0.57	0.00	1	0.00	0.00	func4

Call graph

granularity: each sample hit covers 2 byte(s) for 1.75% of 0.57 seconds

	index & time	self	children	called	name
[1]	100.0	0.03	0.54	1/1	main [2]
		0.03	0.54	1	func1 [1]
		0.00	0.54	1/1	func2 [3]
<hr/>					
[2]	100.0				<spontaneous>
		0.00	0.57		main [2]
		0.03	0.54	1/1	func1 [1]
<hr/>					
[3]	94.7				func1 [1]
		0.00	0.54	1	func2 [3]
		0.54	0.00	1/1	func3 [4]
<hr/>					
[4]	94.7				func2 [3]
		0.54	0.00	1	func3 [4]
<hr/>					
[5]	0.0				main [2]
		0.00	0.00	1	func4 [5]
<hr/>					

Line-based profiling

- Function profiling becomes useless when the program to be analyzed contains large functions (in terms of code lines)
- It contribute significantly to overall runtime:

1	%	cumulative	self	self	total		
2	time	seconds	seconds	calls	s/call	s/call	name
3	73.21	13.47	13.47	1	13.47	18.40	MAIN__
4	6.47	14.66	1.19	21993788	0.00	0.00	mvteil__
5	6.36	15.83	1.17	51827551	0.00	0.00	ran1__
6	6.25	16.98	1.15	35996244	0.00	0.00	gzahl__

- Here the MAIN function in a Fortran program requires over 73% of overall runtime but has about 1700 lines of code.
- If the hot spot in such functions cannot be found by simple common sense, tools for line-based profiling should be used.

Line-based profiling

- The number of sampling hits (first column) and the relative percentage of total program samples (second column):

```
1          :      DO 215 M=1, 3
2 4292  0.9317 :      bremsdir(M) = bremsdir(M) + FH(M)*Z12
3 1462  0.3174 : 215    CONTINUE
4          :
5 682   0.1481 :      U12 = U12 + GCL12 * Upot
6          :
7          :      DO 230 M=1, 3
8 3348  0.7268 :      F(M,I)=F(M,I)+FH(M)*Z12
9 1497  0.3250 :      Fion(M)=Fion(M)+FH(M)*Z12
10 501   0.1088 :230   CONTINUE
```

Line-based profiling

- The compiler-generated symbol tables must be consistent so that a machine instruction's address in memory can be properly matched to the correct source line.
- Modern compilers can reorganize code heavily if high optimization levels are enabled.
- Loops can be fused or split, lines rearranged, variables optimized away, etc., so that the actual code executed may be far from resembling the original source.
- Furthermore, due to the strongly pipelined architecture of modern microprocessors it is usually impossible to attribute a specific moment in time to a particular source line or even machine instruction.
- However, looking at line-based profiling data on a loop-by-loop basis (samples integrated across the loop body) is relatively safe;
- In case of doubt, recompilation with a lower optimization level (and inlining disabled) may provide more insight.

Hardware performance counters

- The first step in performance profiling is concerned with pinpointing the hot spots in terms of runtime, i.e., clock ticks.
- **Clock ticks are insufficient**
- But when it comes to identifying the actual reason for a code to be slow, or
- If one merely wants to know by which resource it is limited,.

- Modern processors feature a small number of *performance counters (<10)*, which are special on-chip registers that get incremented each time a certain event occurs.

Number of bus transactions, i.e., cache line transfers.

- Events like “cache misses” are commonly used instead of bus transactions, however one should be aware that prefetching mechanisms (in hardware or software) can interfere with the number of cache misses counted.
- Bus transactions are often the safer way to account for the actual data volume transferred over the memory bus.
- If the memory bandwidth used by a processor is close to the theoretical maximum, there is no point in trying to optimize further for better bus utilization.
- The number can also be used for checking the correctness of some theoretical model one may have developed for estimating the data transfer requirements of an application

Number of loads and stores

- Together with bus transactions, this can give an indication as to how efficiently cache lines are used for computation.
- If, e.g., the number of DP loads and stores per cache line is less than its length in DP words, this may signal strided memory access.
- One should however take into account how the data is used;
- if, for some reason, the processor pipelines are stalled most of the time, or if many arithmetic operations on the data are performed out of registers, the strided access may not actually be a performance bottleneck.

Number of floating-point operations

- The importance of this very popular metric is often overrated.
- Data transfer is the dominant performance-limiting factor in scientific code.
- Even so, if the number of floating-point operations per clock cycle is somewhere near the theoretical maximum (either given by the CPU's peak performance, or less if there is an asymmetry between MULT and ADD operations), standard code optimization is unlikely to achieve any improvement and algorithmic changes are probably in order.

Mispredicted branches

- This counter is incremented when the CPU has predicted the outcome of a conditional branch and the prediction has proved to be wrong.
- Depending on the architecture, the penalty for a mispredicted branch can be tens of cycles.
- In general, scientific codes tend to be loop-based so that branches are well predictable.
- However, “pointer chasing” and computed branches increase the probability of mispredictions.

Pipeline stalls

- Dependencies between operations running in different stages of the processor pipeline can lead to cycles during which not all stages are busy, so-called *stalls or bubbles*.
- *Often bubbles cannot be avoided*, e.g., when performance is limited by memory bandwidth and the arithmetic units spend their time waiting for data.
- Hence, it is quite difficult to identify the point where there are “too many” bubbles.
- Stall cycle analysis is especially important on in-order architectures (like, e.g., Intel IA64) because bubbles cannot be filled by the hardware if there are no provisions for executing instructions that appear later in the instruction stream but have their operands available.

Number of instructions executed

- Together with clock cycles, this can be a guideline for judging how effectively the superscalar hardware with its multiple execution units is utilized.
- Experience shows that it is quite hard for compiler-generated code to reach more than 2–3 instructions per cycle, even in tight inner loops with good pipelining properties.

lipfpm profile

- In order to get a quick overview of the performance properties of an application, a simple tool can measure overall counts from start to finish and probably calculate some *derived metrics* like “instructions per cycle” or “cache misses per load or store.”
- A typical output of such a tool could look like this if run with some application code (these examples were compiled from output generated by the lipfpm tool,)

1	CPU Cycles.....	8721026107
2	Retired Instructions.....	21036052778
3	Average number of retired instructions per cycle.....	2.398151
4	L2 Misses.....	101822
5	Bus Memory Transactions.....	54413
6	Average MB/s requested by L2.....	2.241689
7	Average Bus Bandwidth (MB/s).....	1.197943
8	Retired Loads.....	694058538
9	Retired Stores.....	199529719
10	Retired FP Operations.....	7134186664
11	Average MFLOP/s.....	1225.702566
12	Full Pipe Bubbles in Main Pipe.....	3565110974
13	Percent stall/bubble cycles.....	40.642963

lipfpm profile

- The large number of retired instructions per cycle indicates that the hardware is well utilized.
- The required bandwidths from the caches and main memory and the relation between retired load/store instructions to L2 cache misses.
- However, there are pipeline bubbles in 40% of all CPU cycles.
- It is hard to tell without some reference whether this is a large or a small value.

Example

- The bandwidth requirements, the low number of instructions per cycle, and the relation between loads/stores and cache misses indicate a memory-bound situation.
- In contrast to the previous case, the percentage of stalled cycles is more than doubled.
- Only an elaborate stall cycle analysis, based on more detailed metrics, would be able to reveal the origin of those bubbles.

1	CPU Cycles.....	28526301346
2	Retired Instructions.....	15720706664
3	Average number of retired instructions per cycle.....	0.551095
4	L2 Misses.....	605101189
5	Bus Memory Transactions.....	751366092
6	Average MB/s requested by L2.....	4058.535901
7	Average Bus Bandwidth (MB/s).....	5028.015243
8	Retired Loads.....	3756854692
9	Retired Stores.....	2472009027
10	Retired FP Operations.....	4800014764
11	Average MFLOP/s.....	252.399428
12	Full Pipe Bubbles in Main Pipe.....	25550004147
13	Percent stall/bubble cycles.....	89.566481

Manual instrumentation

- If the overheads subjected to the application by standard compiler-based instrumentation are too large, or
- If only certain parts of the code should be profiled in order to get a less complex view on performance properties, manual instrumentation may be considered.
- The programmer inserts calls to a wallclock timing routine like `gettimeofday()`
- The results returned by timing routines should be interpreted Carefully.
- The most frequent mistake with code timings occurs when the time periods to be measured are in the same order of magnitude as the timer resolution, i.e., the minimum possible interval that can be resolved.

Common sense optimizations

- Very simple code changes can often lead to a significant performance boost.
- The most important “common sense” guidelines regarding the avoidance of performance pitfalls are summarized:
- **Do less work!**
- **Avoid Expensive Operations**
- **Shrink the workset**

Do less work!

- Rearranging the code such that less work than before is being done will improve performance.
- A very common example is a loop that checks a number of objects to have a certain property, but all that matters in the end is that *any object has the property at all*:

Do less work

```
1 logical :: FLAG
2 FLAG = .false.
3 do i=1,N
4   if(complex_func(A(i)) < THRESHOLD) then
5     FLAG = .true.
6   endif
7 enddo
```

- If `complex_func()` has no side effects, the only information that gets communicated to the outside of the loop is the value of `FLAG`.
- In this case, depending on the probability for the conditional to be true, much computational effort can be saved by leaving the loop as soon as `FLAG` changes.

```
1 logical :: FLAG
2 FLAG = .false.
3 do i=1,N
4   if(complex_func(A(i)) < THRESHOLD) then
5     FLAG = .true.
6     exit
7   endif
8 enddo
```

Avoid expensive operations

- Sometimes, implementing an algorithm is done in a thoroughly “one-to-one” way, translating formulae to code without any reference to performance issues.
- While this is actually good (performance optimization always bears the slight danger of changing numerics, if not results), in a second step all those operations should be eliminated that can be substituted by “cheaper” alternatives.
- Prominent examples for such “strong” operations are trigonometric functions or exponentiation.
- Bear in mind that an expression like $x^{**}2.0$ is often not optimized by the compiler to become $x*x$ but left as it stands, resulting in the evaluation of an exponential and a logarithm.
- The corresponding optimization is called ***strength reduction***.
- *Apart from the simple case described above, strong operations sometimes appear with a limited set of fixed arguments.*

Avoid expensive operations

- This is an example from a simulation code for nonequilibrium spin systems:

```
1 integer :: iL,iR,iU,iO,is,in
2 double precision :: edelz,tt
3 ...
4 edelz = iL+iR+iU+iO+is+in      ! load spin orientations
5 BF    = 0.5d0*(1.d0+TANH(edelz/tt))
```

- The last two lines are executed in a loop that accounts for nearly the whole runtime of the application.
- The integer variables store spin orientations (up or down, i.e., -1 or +1, respectively), so the edelz variable only takes integer values in the range $\{-6, \dots, +6\}$.
- The `tanh()` function is one of those operations that take vast amounts of time (at least tens of cycles), even if implemented in hardware.

Avoid expensive operations – tanh()

- It is easy to eliminate the tanh() call completely by *tabulating* the function over the range of arguments required
- Assuming that tt does not change its value so that the table does only have to be set up once:

```
1 double precision, dimension(-6:6) :: tanh_table
2 integer :: iL,iR,iU,iO,iS,iN
3 double precision :: tt
4 ...
5 do i=-6,6                      ! do this once
6   tanh_table(i) = 0.5d0*(1.d0+TANH(db1e(i)/tt))
7 enddo
8 ...
9 BF = tanh_table(iL+iR+iU+iO+iS+iN) ! loop kernel
```

- The table look-up is performed at virtually no cost compared to the tanh () evaluation since the table will be available in L1 cache at access latencies of a few CPU cycles.
- Due to the small size of the table and its frequent use it will fit into L1 cache and stay there in the course of the calculation.

Shrink the working set

- The *working set of a code is the amount of memory it uses in the course of a calculation, or at least during a significant part of overall runtime.*
- Shrinking the working set by whatever means is a good thing because it raises the probability for cache hits.
- In the above example, the original code used standard four-byte integers to store the spin orientations.
- The working set was thus much larger than the L2 cache of any processor.
- By changing the array definitions to use `integer (kind=1)` for the spin variables, the working set could be reduced by nearly a factor of four, and became comparable to cache size.

Simple measures, large impact: Elimination of common subexpressions

- Common subexpression elimination is an optimization that is often considered a task for compilers.
- Basically one tries to save time by precalculating parts of complex expressions and assigning them to temporary variables before a code construct starts that uses those parts multiple times.
- In case of loops, this optimization is also called *loop invariant code motion*:

```
1 ! inefficient
2 do i=1,N
3   A(i)=A(i)+s+r*sin(x)
4 enddo
```



```
tmp=s+r*sin(x)
do i=1,N
  A(i)=A(i)+tmp
enddo
```

- A lot of compute time can be saved by this optimization, especially where “strong” operations (like sin()) are involved.
- Subexpressions are obstructed by other code and not easily recognizable, compilers are in principle able to detect this situation.
- Compute the subexpression out of the loop if this required employing associativity rules.
- A good strategy is to help the compiler by eliminating common subexpressions by hand.

Avoiding Branches

- “Tight” loops, i.e., loops that have few operations in them, are typical candidates for software pipelining, loop unrolling, and other optimization techniques.
- If for some reason compiler optimization fails or is inefficient, performance will suffer.
- This can easily happen if the loop body contains conditional branches:

```
1 do j=1,N
2   do i=1,N
3     if(i.ge.j) then
4       sign=1.d0
5     else if(i.lt.j) then
6       sign=-1.d0
7     else
8       sign=0.d0
9     endif
10    C(j) = C(j) + sign * A(i,j) * B(i)
11  enddo
12 enddo
```

Avoiding Branches

- In this multiplication of a matrix with a vector, the upper and lower triangular parts get different signs and the diagonal is ignored.
- The if statement serves to decide about which factor to use.
- Each time a corresponding conditional branch is encountered by the processor, some *branch prediction logic tries to guess the most probable* outcome of the test before the result is actually available, based on statistical methods.
- The instructions along the chosen path are then fetched, decoded, and generally fed into the pipeline.
- If the anticipation turns out to be false (this is called a *mispredicted branch or branch miss*), *the pipeline has to be flushed back to the position of the branch*, implying many lost cycles.

Avoiding Branches

- Fortunately, the loop nest can be transformed so that all if statements vanish:

```
1 do j=1,N
2   do i=j+1,N
3     C(j) = C(j) + A(i,j) * B(i)
4   enddo
5 enddo
6 do j=1,N
7   do i=1,j-1
8     C(j) = C(j) - A(i,j) * B(i)
9   enddo
10 enddo
```

Using SIMD instruction sets

- The use of SIMD in microprocessors is often termed “vectorization,”
- It is more similar to the multitrack property of modern vector systems.
- A “vectorizable” loop in this context will run faster if more operations can be performed with a single instruction
- The size of the data type should be as small as possible.
- Switching from DP to SP data could result in up to a twofold speedup, with the additional benefit that more items fit into the cache.
- Preferring SIMD instructions over scalar ones is no guarantee for a performance improvement.
- If the code is strongly limited by memory bandwidth, no SIMD technique can bridge this gap.
- Register-to-register operations will be greatly accelerated, but this will only lengthen the time the registers wait for new data from the memory subsystem.

Using SIMD instruction sets

- A single precision ADD instruction was depicted that might be used in an array addition loop:

```
1 real, dimension(1:N) :: r, x, y
2 do i=1, N
3   r(i) = x(i) + y(i)
4 enddo
```

- All iterations in this loop are independent, there is no branch in the loop body, and the arrays are accessed with a stride of one.
- The use of SIMD requires some rearrangement of a loop kernel like the one above to be applicable:
- A number of iterations equal to the SIMD register size has to be executed as a single “chunk” without any branches in between.
- This is actually a well-known optimization that can pay off even without SIMD and is called *loop unrolling*.

Using SIMD instruction sets

- Since the overall number of iterations is generally not a multiple of the register size, some remainder loop is left to execute in scalar mode.

```
1 ! vectorized part
2 rest = mod(N, 4)
3 do i=1,N-rest,4
4   load R1 = [x(i),x(i+1),x(i+2),x(i+3)]
5   load R2 = [y(i),y(i+1),y(i+2),y(i+3)]
6   ! "packed" addition (4 SP flops)
7   R3 = ADD(R1,R2)
8   store [r(i),r(i+1),r(i+2),r(i+3)] = R3
9 enddo
10 ! remainder loop
11 do i=N-rest+1,N
12   r(i) = x(i) + y(i)
13 enddo
```

- R1, R2, and R3 denote 128-bit SIMD registers here.
- In an optimal situation all this is carried out by the compiler automatically.
- Compiler directives can be used to give hints as to where vectorization is safe and/or beneficial.

SIMD Aligned and Unaligned

- SIMD instruction sets distinguish between *aligned* and *unaligned* data.
- If an aligned load or store is used on a memory address that is not a multiple of 16, an exception occurs.
- In cases where the compiler knows nothing about the alignment of arrays used in a vectorized loop
- Unaligned (or a sequence of scalar) loads and stores must be used, incurring some performance penalty.
- The programmer can force the compiler to assume optimal alignment, but this is dangerous if one cannot make absolutely sure that the assumption is justified.

Loop Dependency – SIMD Vectorization

- A loop with a true dependency cannot be SIMD vectorized.

```
1 do i=2,N  
2   A(i)=s*A(i-1)  
3 enddo
```

- The compiler will revert to scalar operations here, which means that only the lowest operand in the SIMD registers is used (on x86 architectures).

Loop Dependency – SIMD Vectorization

- One possible definition is that all arithmetic within the loop is executed using the full width of SIMD registers.
- The load and store instructions could still be scalar; compilers tend to report such loops as “vectorized” as well.
- On x86 processors with SSE support, the lower and higher 64 bits of a register can be moved independently.

Loop Dependency – SIMD Vectorization

- The vector addition loop above could thus look as follows in double precision:

```
1 rest = mod(N,2)
2 do i=1,N-rest,2
3   ! scalar loads
4   load R1.low = x(i)
5   load R1.high = x(i+1)
6   load R2.low = y(i)
7   load R2.high = y(i+1)
8   ! "packed" addition (2 DP flops)
9   R3 = ADD(R1,R2)
10  ! scalar stores
11  store r(i) = R3.low
12  store r(i+1) = R3.high
13 enddo
14 ! remainder "loop"
15 if(rest.eq.1) r(N) = x(N) + y(N)
```

Loop Dependency – SIMD Vectorization

- This version will not give the best performance if the operands reside in a cache.
- Although the actual arithmetic operations (line 9) are SIMD-parallel, all loads and stores are scalar.
- The only option to identify such a failure is manual inspection of the generated assembly code.
- If the compiler cannot be convinced to properly vectorize a loop even with additional command line options or source code directives, a typical “last resort” before using assembly language altogether is to employ *compiler intrinsics*.
- *Intrinsics are constructs that resemble assembly instructions so closely that they can usually be translated 1:1 by the compiler.*
- The user is relieved from the burden of keeping track of individual registers, because the compiler provides special data types that map to SIMD operands.
- Intrinsics are not only useful for vectorization but can be beneficial in all cases where high-level language constructs cannot be optimally mapped to some CPU functionality.
- Unfortunately, intrinsics are usually not compatible across compilers even on the same architecture.

The role of compilers:

General optimization options

- Every compiler offers a collection of standard optimization options(-O0, -O1, ...).
- What kinds of optimizations are employed at which level is by no means standardized and often (but not always) documented in the manuals.
- All compilers refrain from most optimizations at level -O0, which is hence the correct choice for analyzing the code with a debugger.
- At higher levels, optimizing compilers mix up source lines, detect and eliminate “redundant” variables, rearrange arithmetic expressions, etc.,
- So that any debugger has a hard time giving the user a consistent view on code and data.

Inlining

- Inlining tries to save overhead by inserting the complete code of a function or subroutine at the place where it is called.
- Each function call uses up resources because arguments have to be passed, either in registers or via the stack (depending on the number of parameters and the calling conventions used).

Aliasing

- The compiler, guided by the rules of the programming language and its interpretation of the source, must make certain assumptions that may limit its ability to generate optimal machine code.
- The typical example:

```
1 void scale_shift(double *a, double *b, double s, int n) {  
2     for(int i=1; i<n; ++i)  
3         a[i] = s*b[i-1];  
4 }
```

- Assuming that the memory regions pointed to by a and b do not overlap,
- The ranges $[a, a+n-1]$ and $[b, b+n-1]$ are disjoint, the loads and stores in the loop can be arranged in any order.
- The compiler can apply any software pipelining scheme it considers appropriate, or it could unroll the loop and group loads and stores in blocks, as shown in the following pseudocode:

```
1 loop:  
2     load R1 = b(i+1)  
3     load R2 = b(i+2)  
4     R1 = MULT(s, R1)  
5     R2 = MULT(s, R2)  
6     store a(i) = R1  
7     store a(i+1) = R2  
8     i = i + 2  
9     branch -> loop
```

Aliasing

- The C and C++ standards allow for arbitrary *aliasing of pointers*.
- It must thus be assumed that the memory regions pointed to by a and b do overlap.
- Lacking any further information, the compiler must generate machine instructions according to this scheme.
- Among other things, SIMD vectorization is ruled out.
- The processor hardware allows reordering of loads and stores within certain limits, but this can of course never alter the program's semantics.
- All C/C++ compilers have command line options to control the level of aliasing the compiler is allowed to assume (e.g., `-fno-fnalias` for the Intel compiler and `-fargument-noalias` for the GCC specify that no two pointer arguments for any function ever point to the same location).
- If the compiler is told that argument aliasing does not occur, it can in principle apply the same optimizations as in equivalent Fortran code.

Computational Accuracy

- Compilers sometimes refrain from rearranging arithmetic expressions if this required applying associativity rules, except with very aggressive optimizations turned on.
- The reason for this is the infamous nonassociativity of FP operations $(a+b)+c$ is, in general, not identical to $a+(b+c)$
- if a , b , and c are finite-precision floating-point numbers.
- If accuracy is to be maintained compared to nonoptimized code, associativity rules must not be used and it is left to the programmer to decide whether it is safe to regroup expressions by hand.
- Modern compilers have command line options that limit rearrangement of arithmetic expressions even at high optimization levels.

Using compiler logs

- In order to make the decisions of the compiler’s “intelligence” available to the user
- Many compilers offer options to generate annotated source code listings or at least logs that describe in some detail what optimizations were performed.
- Compiler log for a software pipelined triad loop is shown below.
- “Peak” indicates the maximum possible execution rate for the respective operation type on this architecture (MIPS R14000).

```
1 #<swps> 16383 estimated iterations before pipelining
2 #<swps>      4 unrollings before pipelining
3 #<swps> 20 cycles per 4 iterations
4 #<swps>      8 flops          ( 20% of peak) (madds count as 2)
5 #<swps>      4 flops          ( 10% of peak) (madds count as 1)
6 #<swps>      4 madds          ( 20% of peak)
7 #<swps>      16 mem refs     ( 80% of peak)
8 #<swps>      5 integer ops   ( 12% of peak)
9 #<swps>      25 instructions ( 31% of peak)
10 #<swps>      2 short trip threshold
11 #<swps>      13 integer registers used.
12 #<swps>      17 float registers used.
```

Register optimizations

- It is one of the most vital, but also most complex tasks of the compiler to care about register usage.
- The compiler tries to put operands that are used “most often” into registers and keep them there as long as possible, given that it is safe to do so.
- If, e.g., a variable’s address is taken, its value might be manipulated elsewhere in the program via the address.
- In this case the compiler may decide to write a variable back to memory right after any change on it.
- Loop bodies with lots of variables and many arithmetic expressions are hard for the compiler to optimize because it is likely that there are too few registers to hold all operands at the same time.
- The number of integer and floating-point registers in any processor is strictly limited.
- Today, typical numbers range from 8 to 128.
- If there is a register shortage, variables have to be *spilled*, i.e., written to memory, for later use.
- If the code’s performance is determined by arithmetic operations, register spill can hamper performance quite a bit.
- In such cases it may even be worthwhile **splitting a loop in two to reduce register pressure**.

Reference

- **Georg Hager and Gerhard Wellein**, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.

DATA ACCESS OPTIMIZATION

Dr Noor Muhammad Sk

Center for High Performance Reconfigurable Computing

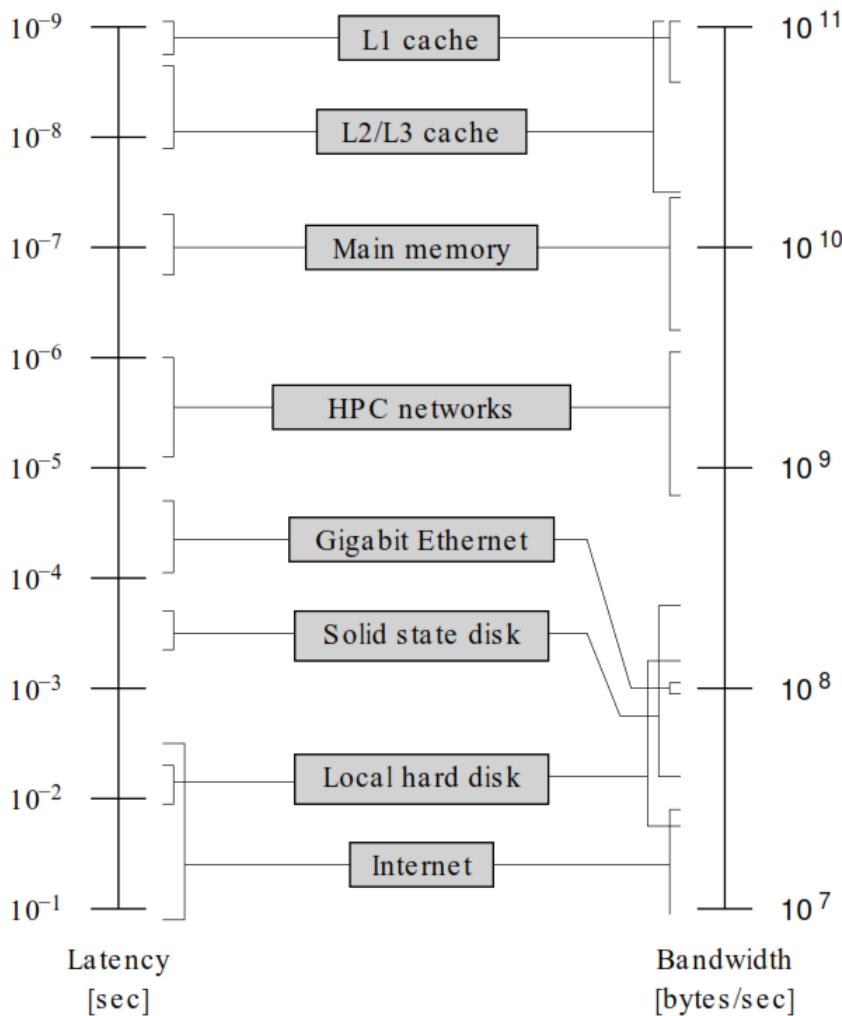
Introduction

- Data access is one of the performance-limiting factors in HPC.
- Microprocessors tend to be inherently “unbalanced” with respect to the relation of **theoretical peak performance** versus **memory bandwidth**.
- Many applications in science and engineering consist of loop-based code that moves large amounts of data in and out of the CPU.
- On-chip resources tend to be underutilized and performance is limited only by the relatively slow data paths to memory or even disks.

Bandwidth vs Latency

- Bandwidth is the amount of data that can be transferred from one point to another normally measured in seconds.
- Latency is the time that a data packet takes to travel from one point to another.

Overview of several data paths present in modern parallel computer systems



- The functional units that perform the computational work, sit at the top of this hierarchy.
- In terms of bandwidth, the slowest data paths are three to four orders of magnitude away
- Eight in terms of latency.
- The deeper a data transfer must reach down through the different levels in order to obtain required operands for some calculation, the harder the impact on performance.
- Any optimization attempt should aim at reducing traffic over slow data paths, or,
- Should at least make data transfer as efficient as possible..

Bandwidth-based Performance Modeling

- If the program at hand is already using the resources in the best possible way
- One can often estimate the theoretical performance of loop-based code that is bound by bandwidth limitations by simple rules of thumb.
- The central concept to introduce here is *balance*.
- *The machine balance* B_m of a processor chip is the ratio of possible memory bandwidth in GWords/sec to peak performance in GFlops/sec:

$$B_m = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak performance [GFlops/sec]}} = \frac{b_{\max}}{P_{\max}}$$

Example

- Consider a dual-core chip with a clock frequency of 3.0 GHz that can perform at most four flops per cycle (per core)
- Has a memory bandwidth of 10.6 GBytes/sec. (64 bit word)
- This processor would have a machine balance of 0.055 W/F.
- At the time of writing, typical values of B_m lie in the range between 0.03 W/F for standard cache-based microprocessors and 0.5 W/F for top of the line vector processors.

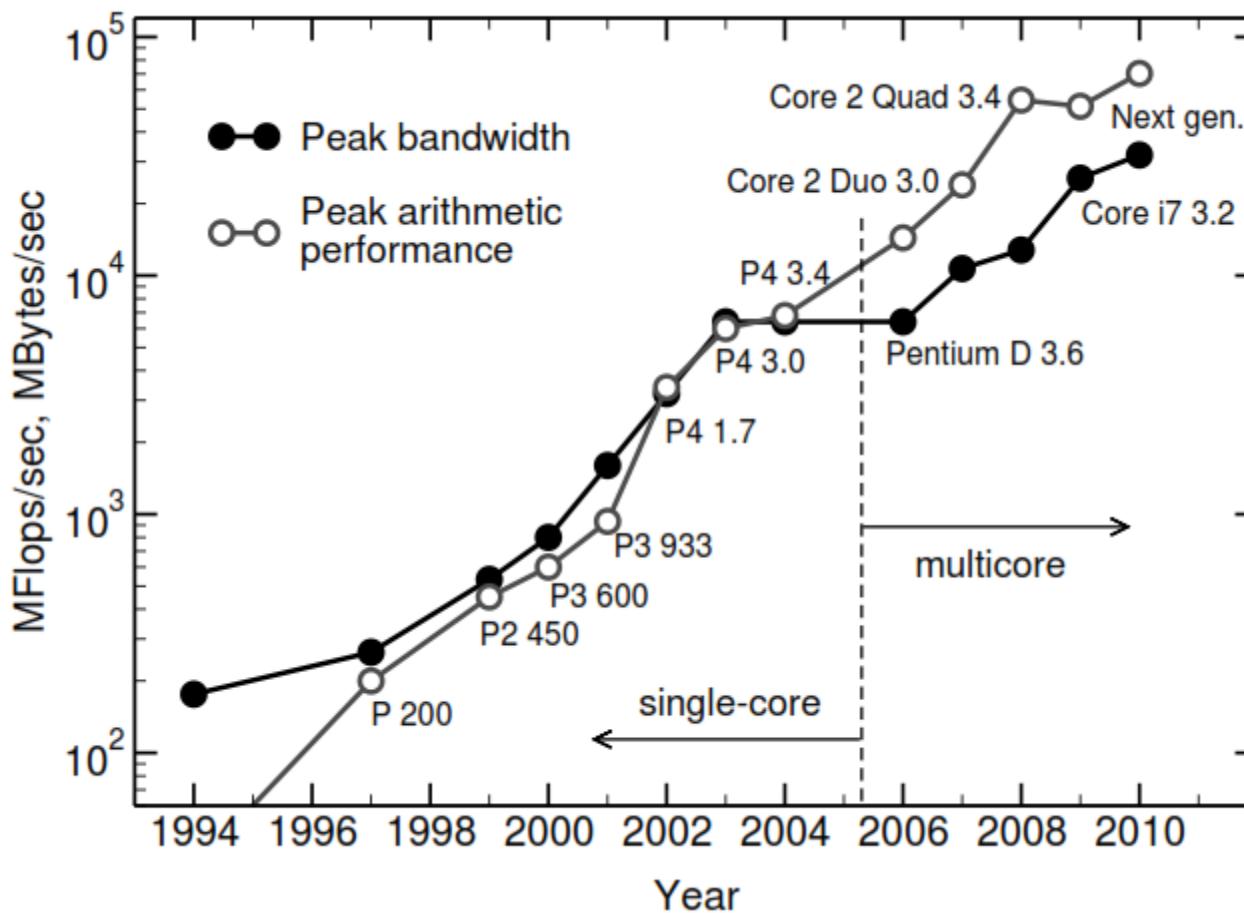
Balance Values

- Typical balance values for operations limited by different transfer paths.
- In case of network and disk connections, the peak performance of typical dual-socket compute nodes was taken as a basis.

data path	balance [W/F]
cache	0.5–1.0
machine (memory)	0.03–0.5
interconnect (high speed)	0.001–0.02
interconnect (GBit ethernet)	0.0001–0.0007
disk (or disk subsystem)	0.0001–0.01

Peak Performance and Memory Bandwidth

- Peak performance did grow faster than memory bandwidth before 2005
- The introduction of the first dual-core chip (Pentium D) really widened the DRAM gap considerably.



Light speed of a Loop

- “Data traffic” refers to all words transferred over the performance-limiting data path
- This metric dependent on the hardware.
- The reciprocal of code balance is often called *computational intensity*.
- The expected maximum fraction of peak performance of a code with balance B_c , on a machine with balance B_m :

$$l = \min \left(1, \frac{B_m}{B_c} \right)$$

- We call this fraction the *lightspeed of a loop*
- Performance in GFlops/sec is

$$P = lP_{\max} = \min \left(P_{\max}, \frac{b_{\max}}{B_c} \right)$$

- If $l \approx 1$, *performance* is not limited by bandwidth but other factors, either inside the CPU or elsewhere.

Simple performance model is based on some Crucial assumptions:

- The loop code makes use of all arithmetic units (MULT and ADD) in an optimal way.
- If this is not the case one must introduce a correction factor that reflects the ratio of “effective” to absolute peak performance
- E.g., if only ADDs are used in the code, effective peak performance would be half of the absolute maximum.
- Similar considerations apply if less than the maximum number of cores per chip are used.
- The loop code is based on double precision floating-point arithmetic.
- Data transfer and arithmetic overlap perfectly.
- The slowest data path determines the loop code’s performance.
- All faster data paths are assumed to be infinitely fast.
- The system is in “throughput mode,” i.e., latency effects are negligible.

Simple performance model is based on some Crucial assumptions:

- It is possible to saturate the memory bandwidth that goes into the calculation of machine balance to its full extent.
- Recent multicore designs tend to underutilize the memory interface if only a fraction of the cores use it.
- This makes performance prediction more complex, since there is a separate “effective” machine balance that is not just proportional to N^{-1} for each core count N .

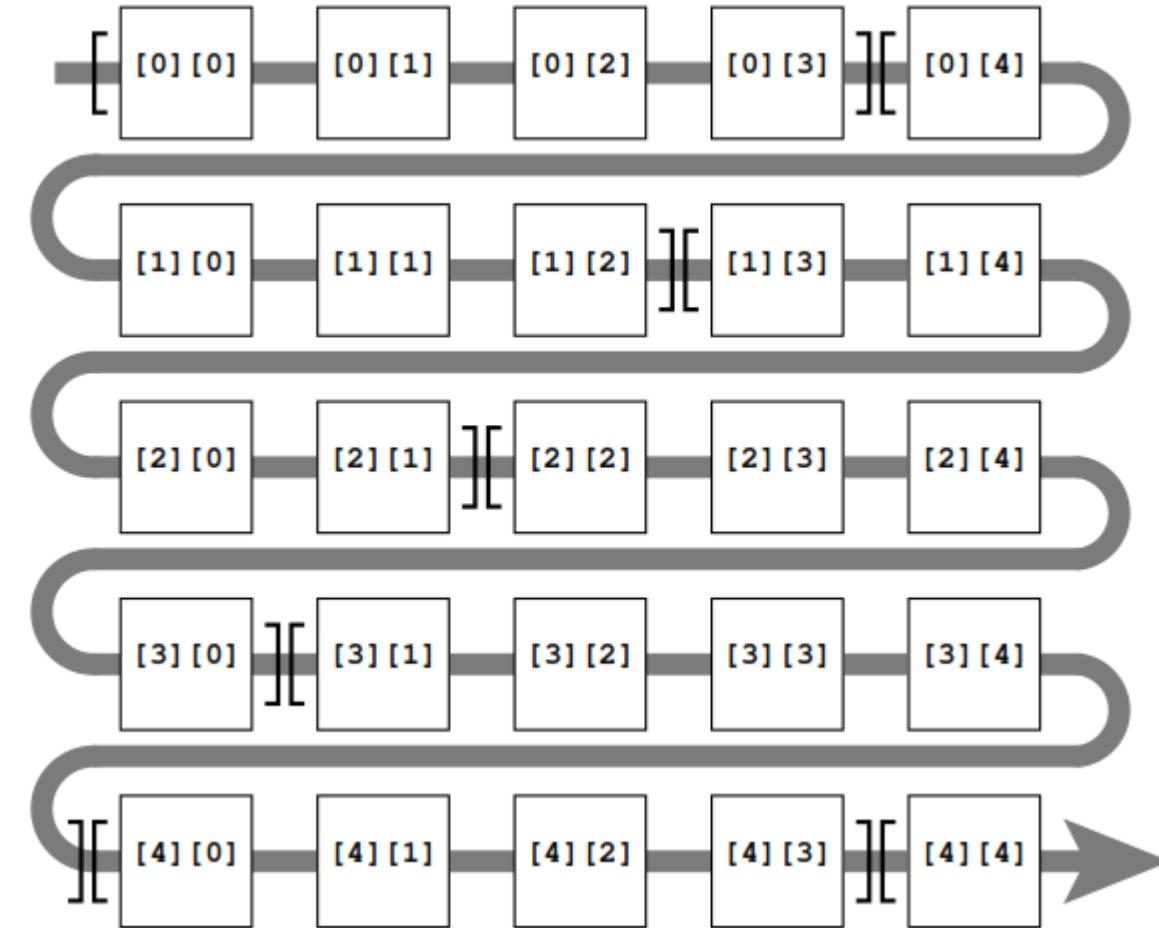
Read and Write Balance

- Maximum bandwidth is often not available in both directions (read and write) concurrently.
- It may be the case, e.g., that the relation from maximum read to maximum write bandwidth is 2:1.
- A write stream cannot utilize the full bandwidth in that case.
- Protocol overhead, deficiencies in chipsets, error correcting memory chips, and large latencies all cut on available bandwidth.
- Data paths inside the processor chip, e.g., connections between L1 cache and registers, can be unidirectional.
- If the code is not balanced between read and write operations, some of the bandwidth in one direction is unused.
- This should be taken into account when applying balance analysis for in-cache situations.

Storage order

- Multidimensional arrays, first and foremost matrices or matrix-like structures, are omnipresent in scientific computing.
- Data access is a crucial topic here as the mapping between the inherently one-dimensional, cache line based memory layout of standard computers.
- Any multidimensional data structure must be matched to the order in which code loads and stores data so that spatial and temporal locality can be employed.
- *Strided access to a one-dimensional array reduces spatial locality*, leading to low utilization of the available bandwidth.

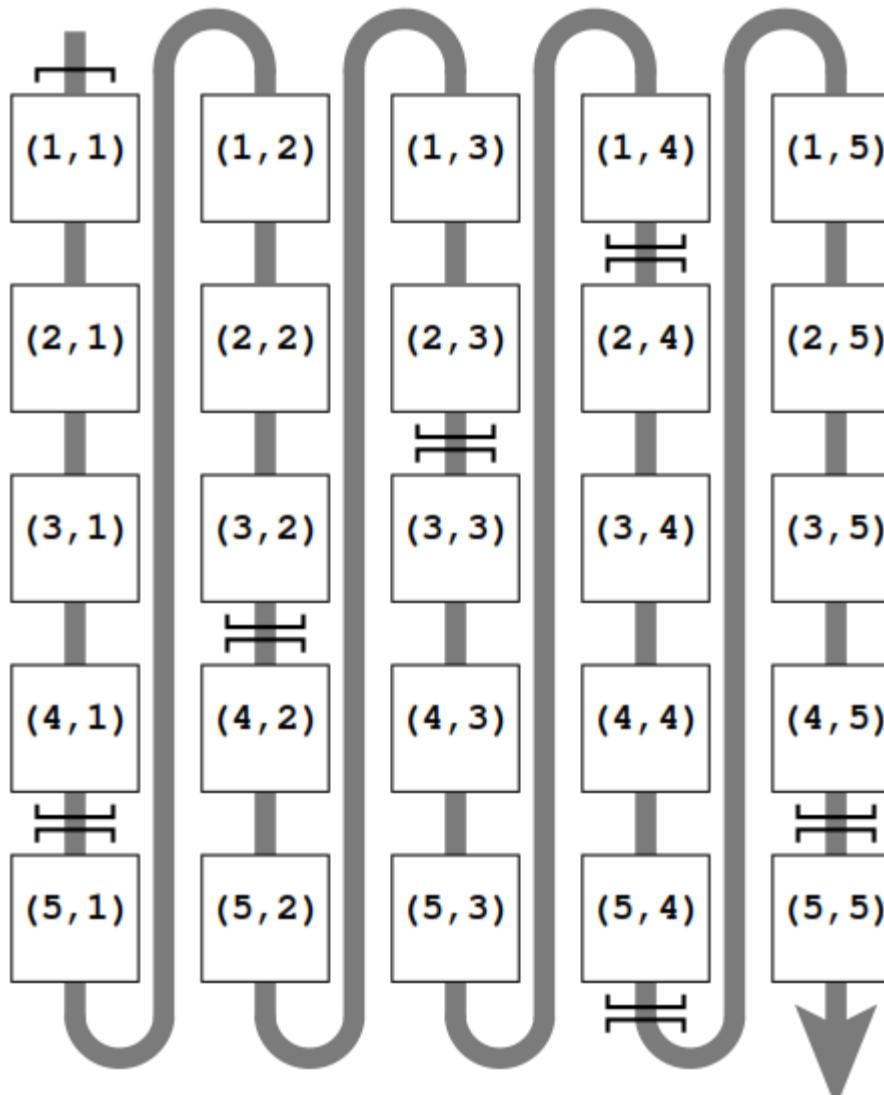
Multidimensional Arrays Access Patterns



```
for(i=0; i<N; ++i) {  
    for(j=0; j<N; ++j) {  
        a[i][j] = i*j;  
    }  
}
```

- Row major order matrix storage scheme, as used by the C programming language.
- Matrix rows are stored consecutively in memory.
- Cache lines are assumed to hold four matrix elements and are indicated by brackets.

Multidimensional Arrays Access Patterns



```
1 do i=1,N  
2   do j=1,N  
3     A(i,j) = i*j  
4   enddo  
5 enddo
```

- Column major order matrix storage scheme, as used by the Fortran programming language.
- Matrix columns are stored consecutively in memory.
- Cache lines are assumed to hold four matrix elements and are indicated by brackets.

Algorithm Classification and Access Optimizations

- The optimization potential of many loops on cache-based processors can easily be estimated just by looking at basic parameters like
 - The scaling behavior of data transfers
 - Arithmetic operations versus problem size.
- It can then be decided whether investing optimization effort would make sense.

$O(N)/O(N)$

- ***Optimization very limited:***
- If both the number of arithmetic operations and the number of data transfers (loads/stores) are proportional to the problem size (or “loop length”) N .
- Examples:
 - Scalar products
 - Vector additions, and
 - Sparse matrix vector multiplication
- They are inevitably memory-bound for large N
- *Compiler-generated code achieves good performance because $O(N)/O(N)$ loops tend to be quite simple and the correct software pipelining strategy is obvious.*

Loop Fusion

- Even if loops are not nested there is sometimes room for improvement

```
1 do i=1,N  
2   A(i) = B(i) + C(i)  
3 enddo  
4 do i=1,N  
5   Z(i) = B(i) + E(i)  
6 enddo
```

loop fusion

```
! optimized  
do i=1,N  
  A(i) = B(i) + C(i)  
  ! save a load for B(i)  
  Z(i) = B(i) + E(i)  
enddo
```

- Loop fusion has achieved an $O(N)$ data reuse for the two-loop constellation so that a complete load stream could be eliminated.*
- This kind optimization is applied by compilers by themselves

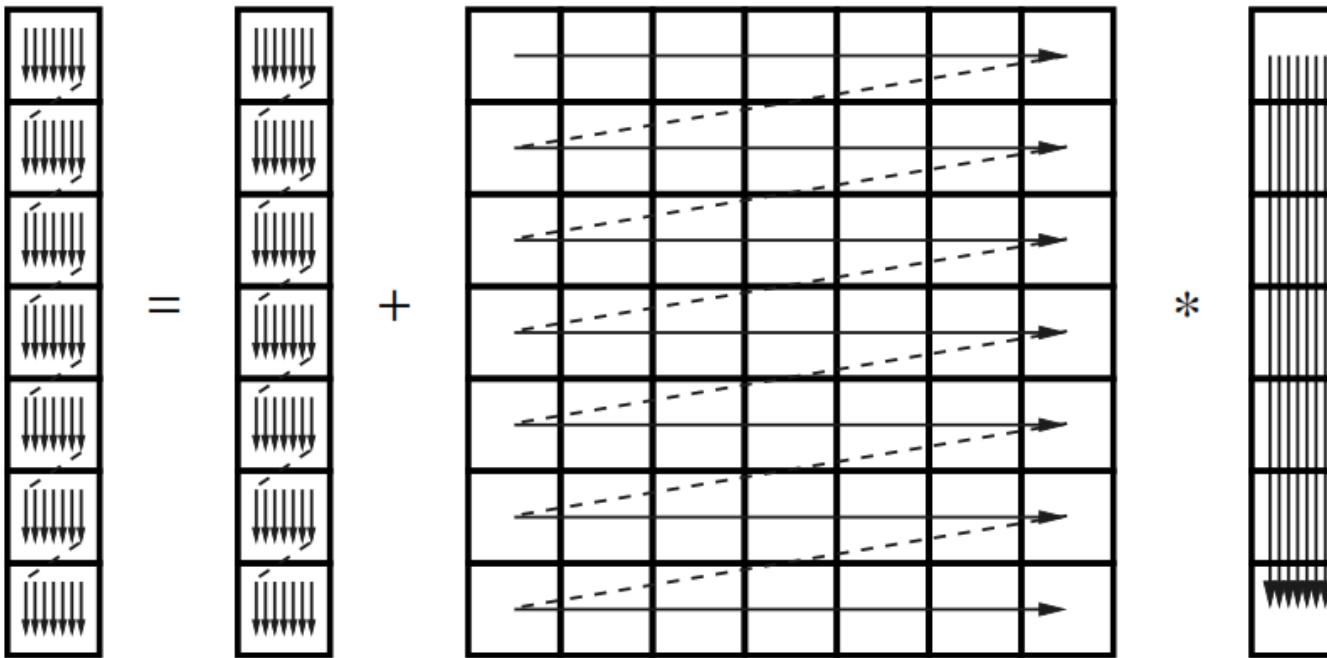
$O(N^2)/O(N^2)$

- In typical two-level loop nests where each loop has a trip count of N , *there are $O(N^2)$ operations for $O(N^2)$ loads and stores.*
- Examples are dense matrix-vector multiply, matrix transpose, matrix addition, etc.
- Although the situation on the inner level is similar to the *$O(N)/O(N)$ case and the problems are generally memory-bound*, the nesting opens new opportunities.
- Optimization, however, is again usually limited to a constant factor of improvement

Unoptimized $N \times N$ dense matrix vector multiply

- The RHS vector is loaded N times.
- As an example we consider dense matrix-vector multiply (MVM)

```
1 do i=1,N  
2   tmp = C(i)  
3   do j=1,N  
4     tmp = tmp + A(j,i) * B(j)  
5   enddo  
6   C(i) = tmp  
7 enddo
```



-
- This code has a balance of 1 W/F (two loads for A and B and two flops).
 - Array C is indexed by the outer loop variable, so updates can go to a register and do not count as load or store streams.
 - Matrix A is only loaded once, but B is loaded *N times, once for each outer loop iteration.*
 - One would like to apply the same fusion trick as above, but there are not just two but *N inner loops to fuse.*

Loop Unrolling

```
1 ! remainder loop
2 do r=1,mod(N,m)
3   do j=1,N
4     C(r) = C(r) + A(j,r) * B(j)
5   enddo
6 enddo
7 ! main loop
8 do i=r,N,m
9   do j=1,N
10    C(i) = C(i) + A(j,i) * B(j)
11  enddo
12  do j=1,N
13    C(i+1) = C(i+1) + A(j,i+1) * B(j)
14  enddo
15 ! m times
16 ...
17 do j=1,N
18   C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
19 enddo
20 enddo
```

- *The outer loop is traversed with a stride m and the inner loop is replicated m times.*
- *We thus have to deal with the situation that the outer loop count might not be a multiple of m .*
- *This case has to be handled by a remainder loop:*

Unroll and jam

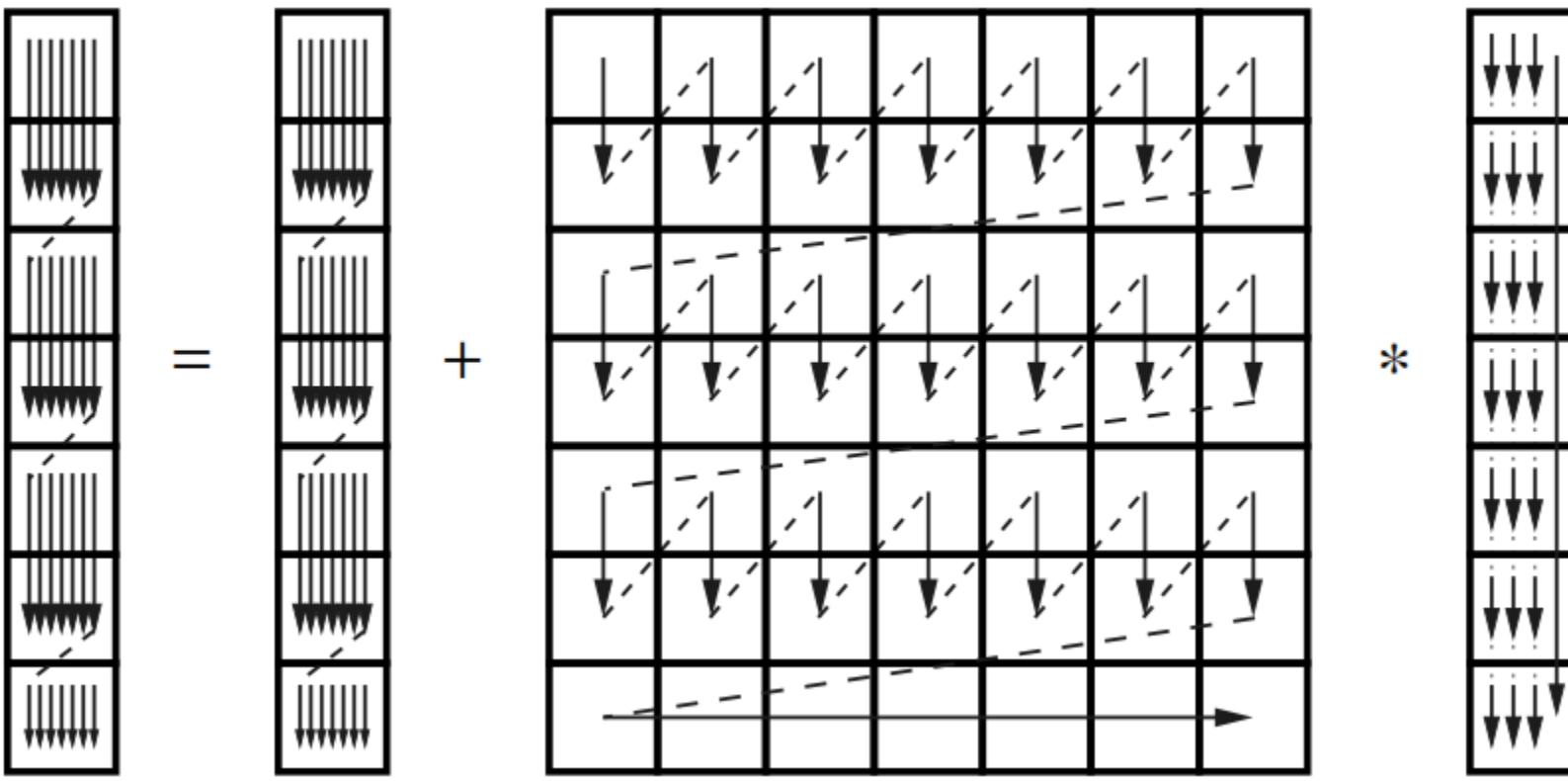
```
1 ! remainder loop ignored
2 do i=1,N,m
3   do j=1,N
4     C(i) = C(i) + A(j,i) * B(j)
5     C(i+1) = C(i+1) + A(j,i+1) * B(j)
6     ! m times
7     ...
8     C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
9   enddo
10 enddo
```

unroll and jam

- The combination of outer loop unrolling and fusion is often called *unroll and jam*.
- *By m-way unroll and jam we have achieved an m-fold reuse of each element of B from register so that code balance reduces to $(m+1)/2m$ which is clearly smaller than one for $m > 1$.*
- *If m is very large, the performance gain can get close to a factor of two.*
- *In this case array B is only loaded a few times or, ideally, just once from memory.*
- As A is always loaded exactly once and has size N^2 , the total memory traffic with *m-way unroll and jam* amounts to $N^2(1 + 1/m) + N$.

The memory access pattern for two-way unrolled dense matrix-vector multiply

- Two-way unrolled dense matrix vector multiply.
- The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two.
- The remainder loop is only a single (outer) iteration in this example.



Loop Unroll

- Assumes, that register pressure is not too large,
- i.e., the CPU has enough registers to hold all the required operands used inside the now quite sizeable loop body.
- If this is not the case, the compiler must spill register data to cache, slowing down the computation.
- Unroll and jam can be carried out automatically by some compilers at high optimization levels.

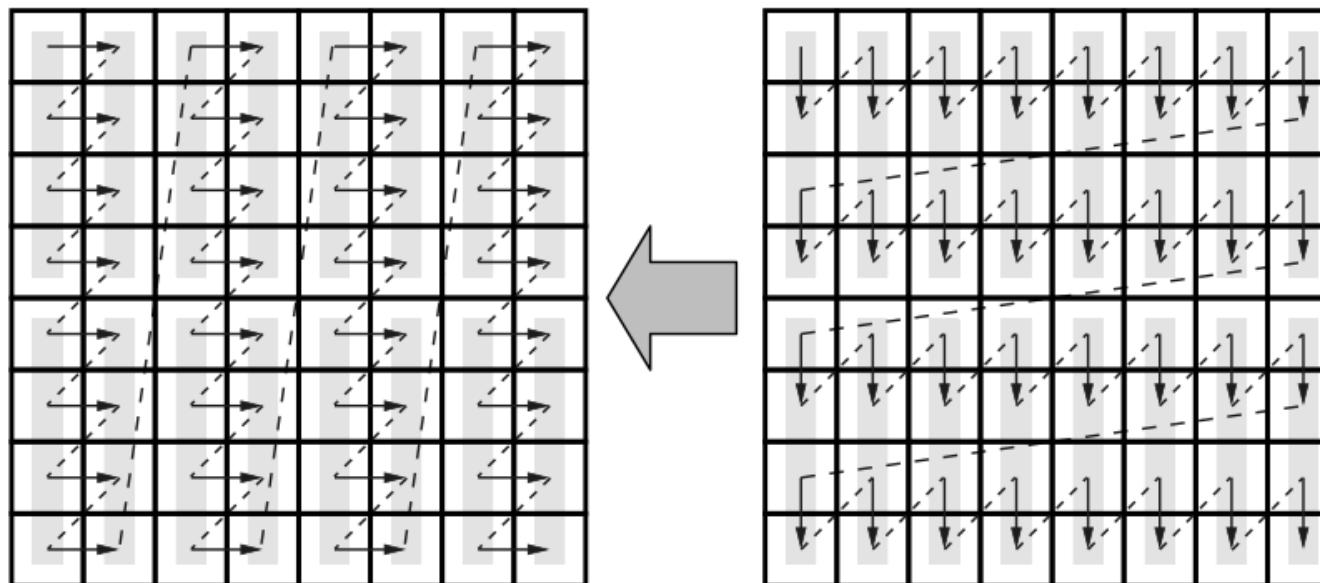
Matrix Transpose Code

- The matrix transpose code is another typical example for an $O(N^2)/O(N^2)$ problem
- There is no direct opportunity for saving on memory traffic;
- Both matrices have to be read or written exactly once.
- Nevertheless, by using unroll and jam on the “flipped” version a significant performance boost of nearly 50% is observed

```
1 do j=1,N,m
2   do i=1,N
3     A(i,j)      = B(j,i)
4     A(i,j+1)    = B(j+1,i)
5     ...
6     A(i,j+m-1) = B(j+m-1,i)
7   enddo
8 enddo
```

Two-way unrolled “flipped” matrix transpose

- Naively one would not expect any effect at $m = 4$ because the basic analysis stays the same:
- In the mid- N region the number of available cache lines is large enough to hold up to L columns of the store stream.
- Figure shows the situation for $m = 2$.
- However, the fact that m words in each of the load stream's cache lines are now accessed in direct succession reduces the TLB misses by a factor of m , although the TLB is still way too small to map the whole working set.



-
- Cutting down on TLB misses does not remedy the performance breakdown for large N *when the cache gets too small to hold N cache lines.*
 - *It would be nice to have a strategy which reuses the remaining $L-m$ words of the strided stream's cache lines right away so that each line may be evicted soon and would not have to be reclaimed later.*
 - A “brute force” method is L_c -way unrolling, but this approach leads to large-stride accesses in the store stream and is not a general solution as large unrolling factors raise register pressure in loops with arithmetic operations.

Loop Blocking

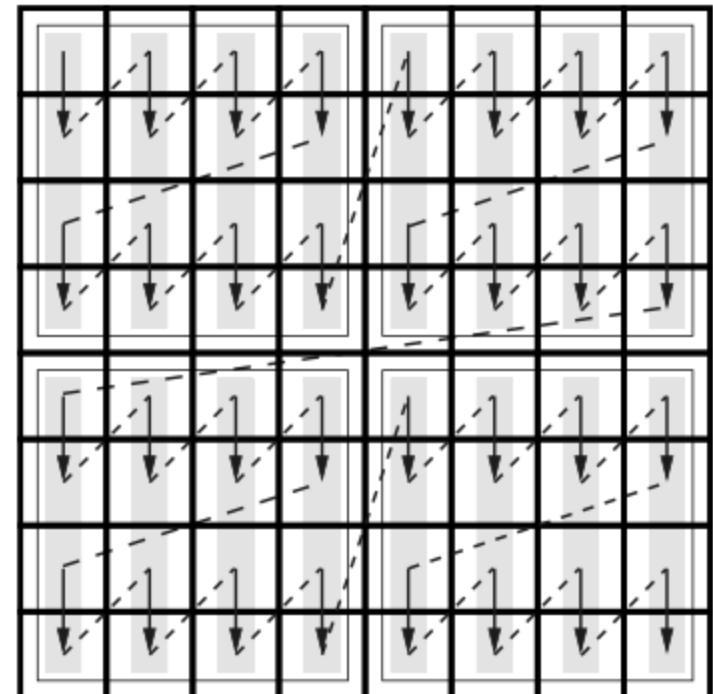
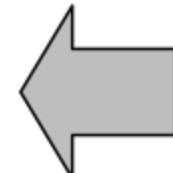
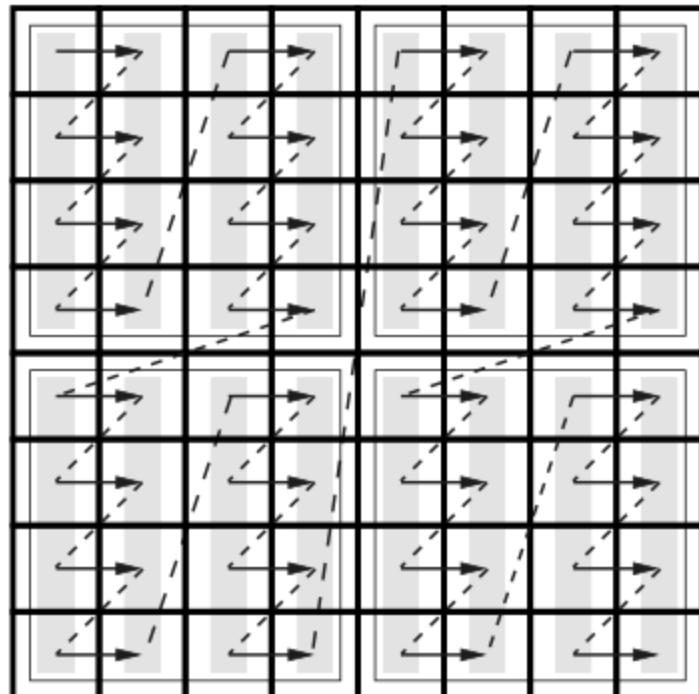
- Loop blocking can achieve optimal cache line use without additional register pressure.
- It does not save load or store operations but increases the cache hit ratio.
- For a loop nest of depth d , blocking introduces up to d additional outer loop levels that cut the original inner loops into chunks:

```
1 do jj=1,N,b
2   jstart=jj; jend=jj+b-1
3   do ii=1,N,b
4     istart=ii; iend=ii+b-1
5     do j=jstart,jend,m
6       do i=istart,iend
7         a(i,j) = b(j,i)
8         a(i,j+1) = b(j+1,i)
9         ...
10        a(i,j+m-1) = b(j+m-1,i)
11      enddo
12    enddo
13  enddo
14 enddo
```

Blocking

- In this example we have used *two-dimensional blocking with identical blocking factors b for both loops in addition to m-way unroll and jam.*
- *This change does not alter the loop body so the number of registers needed to hold operands stays the same.*
- However, the cache line access characteristics are much improved
- A combination of two-way unrolling and 4×4 blocking.
- If the blocking factors are chosen appropriately, the cache lines of the strided stream will have been used completely at the end of a block and can be evicted “soon.”
- Hence, we expect the large- N *performance breakdown to disappear.*

4×4 blocked and two-way unrolled “flipped” matrix transpose



Loop Blocking

- Loop blocking is a very general and powerful optimization that can often not be performed by compilers.
- The correct blocking factor to use should be determined experimentally through careful benchmarking, but one may be guided by typical cache sizes,
- i.e., when blocking for L1 cache the aggregated working set size of all blocked inner loop nests should not be much larger than half the cache.
- Which cache level to block for depends on the operations performed and there is no general recommendation.

$O(N^3)/O(N^2)$

- If the number of operations is larger than the number of data items by a factor that grows with problem size
- By the techniques described above (unroll and jam, loop blocking) it is sometimes possible for these kinds of problems to render the implementation cache-bound.
- Examples for algorithms that show $O(N^3)/O(N^2)$ characteristics are dense matrix-matrix multiplication (MMM) and dense matrix diagonalization.
- It is beyond the scope of this book to develop a well-optimized MMM,
- Let alone eigenvalue calculation, but we can demonstrate the basic principle by means of a simpler example which is actually of the $O(N^2)/O(N^2)$ type:

```
1 do i=1,N
2   do j=1,N
3     sum = sum + foo(A(i),B(j))
4   enddo
5 enddo
```

$O(N^3)/O(N^2)$

- The complete data set is $O(N)$ here but $O(N^2)$ operations (calls to `foo()`, additions) are performed on it.
- In the form shown above, array B is loaded from memory N times, so the total memory traffic amounts to $N(N + 1)$ words.
- *m-way unroll and jam* is possible and will immediately reduce this to $N(N/m+1)$, but the disadvantages of large unroll factors have been pointed out already.
- **Blocking the inner loop with a blocksize of b , however, has two effects:**
 - Array B is now loaded only once from memory, provided that b is small enough so that b elements fit into cache and stay there as long as they are needed.
 - Array A is loaded from memory N/b times instead of once.

```
1 do jj=1,N,b
2   jstart=jj; jend=jj+b-1
3   do i=1,N
4     do j=jstart,jend
5       sum = sum + foo(A(i),B(j))
6   enddo
7 enddo
8 enddo
```

-
- Although A is streamed through cache N/b times,
 - *The probability that the current block of B will be evicted is quite low*
 - The reason being that those cache lines are used very frequently and thus kept by the LRU replacement algorithm.
 - This leads to an effective memory traffic of $N(N/b+1)$ words.
 - As b can be made much larger than typical unrolling factors, blocking is the best optimization strategy here.
 - Unroll and jam can still be applied to enhance in-cache code balance.
 - The basic N^2 dependence is still there, but with a prefactor that can make the difference between memory bound and cache-bound behavior.
 - A code is cache-bound if main memory bandwidth and latency are not the limiting factors for performance any more.
 - Whether this goal is achievable on a certain architecture depends on the cache size, cache and memory speeds, and the algorithm, of course.

-
- Algorithms of the $O(N^3)/O(N^2)$ type are typical candidates for optimizations that can potentially lead to performance numbers close to the theoretical maximum.
 - If blocking and unrolling factors are chosen appropriately, dense matrix-matrix multiply, e.g., is an operation that usually achieves over 90% of peak for $N \times N$ matrices if N is not too small.
 - *It is provided in highly optimized versions by system vendors* as, e.g., contained in the BLAS (Basic Linear Algebra Subsystem) library.
 - One might ask why unrolling should be applied at all when blocking already achieves the most important task of making the code cache-bound.
 - The reason is that even if all the data resides in a cache, many processor architectures do not have the capability for sustaining enough loads and stores per cycle to feed the arithmetic units continuously.
 - For instance, the current x86 processors from Intel can sustain one load and one store operation per cycle, which makes unroll and jam mandatory if the kernel of a loop nest uses more than one load stream, especially in cache-bound situations like the blocked $O(N^2)/O(N)$ example above.

Reference

- **Georg Hager and Gerhard Wellein**, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.

PARALLEL COMPUTERS

Dr Noor Mahamamd Sk

Introduction

- *Parallel computing is a number of “compute elements” (cores) solve a problem in a cooperative way.*
- All modern supercomputer architectures depend heavily on parallelism, and the number of CPUs in large-scale supercomputers increases steadily.
- A common measure for supercomputer “speed”

Taxonomy of Parallel Computing Paradigms

- A widely used taxonomy for describing the amount of **concurrent control** and **data streams** present in a parallel architecture was proposed by Flynn
- **SIMD Single Instruction, Multiple Data.**
- A single instruction stream, either on a single processor (core) or on multiple compute elements, provides parallelism by operating on multiple data streams concurrently.
- Examples: Vector processors
- The SIMD capabilities of modern superscalar microprocessors, and Graphics Processing Units (GPUs).

Taxonomy of Parallel Computing Paradigms

- **MIMD *Multiple Instruction, Multiple Data.***
- *Multiple instruction streams on multiple* processors (cores) operate on different data items concurrently.
- The shared memory and distributed-memory parallel computers are typical examples for the MIMD paradigm.

Shared-memory computers

- A *shared-memory parallel computer* is a system in which a number of CPUs work on a common shared physical address space.
- Although transparent to the programmer as far as functionality is concerned
- There are two varieties of shared memory systems that have very different performance characteristics in terms of main memory access:
 - *Uniform Memory Access (UMA)*
 - *Cache-coherent Nonuniform Memory Access (ccNUMA)*

Uniform Memory Access (UMA)

- Systems exhibit a “flat” memory model:
- Latency and bandwidth are the same for all processors and all memory locations.
- This is also called *symmetric multiprocessing (SMP)*.
- *At the time of writing*, single multicore processor chips are “UMA machines.”
- However, “cluster on a chip” designs that assign separate memory controllers to different groups of cores on a die are already beginning to appear.

Non Uniform Memory Access (NUMA)

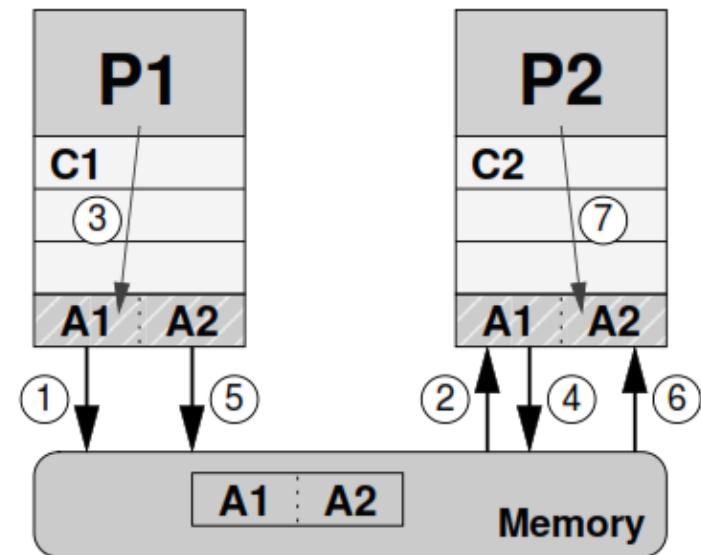
- Memory is *physically distributed but logically shared*.
- *The physical layout of such systems is quite similar to the distributed-memory case, but network logic makes the aggregated memory of the whole system appear as one single address space.*
- Due to the distributed nature, memory access performance varies depending on which CPU accesses which parts of memory (“local” vs. “remote” access).

Cache Coherence

- Cache coherence mechanisms are required in all cache-based multiprocessor systems, whether they are of the UMA or the ccNUMA kind.
- This is because copies of the same cache line could potentially reside in several CPU caches.
- If, e.g., one of those gets modified and evicted to memory, the other caches' contents reflect outdated data.
- Cache coherence protocols ensure a consistent view of memory under all circumstances.

MESI

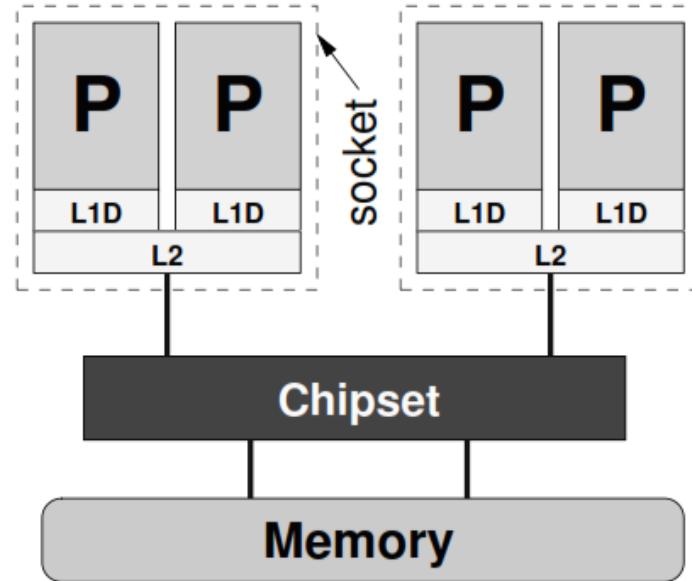
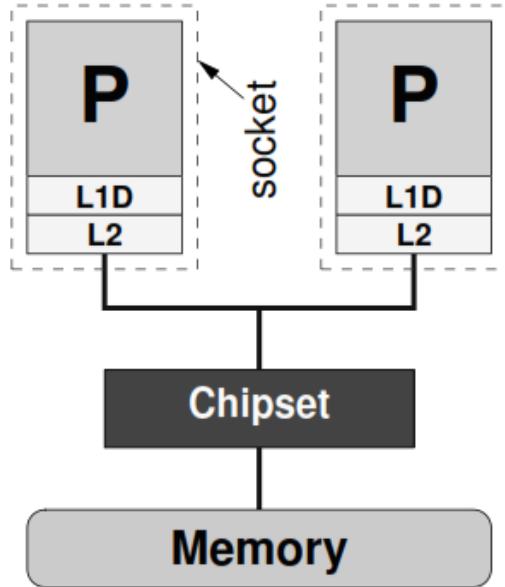
- Two processors P1, P2 modify the two parts A1, A2 of the same cache line in caches C1 and C2.
- The MESI coherence protocol ensures consistency between cache and memory.
- 1. C1 requests exclusive CL ownership
- 2. set CL in C2 to state I
- 3. CL has state E in C1 → modify A1 in C1 and set to state M
- 4. C2 requests exclusive CL ownership
- 5. evict CL from C1 and set to state I
- 6. load CL to C2 and set to state E
- 7. modify A2 in C2 and set to state M in C2



MESI

- **M modified:**
 - *The cache line has been modified in this cache, and it resides in no other cache than this one.*
 - Only upon eviction will memory reflect the most current state.
- **E exclusive:**
 - *The cache line has been read from memory but not (yet) modified.*
 - However, it resides in no other cache.
- **S shared:**
 - *The cache line has been read from memory but not (yet) modified.*
 - There may be other copies in other caches of the machine.
- **I invalid:**
 - *The cache line does not reflect any sensible data.*
 - *Under normal circumstances* this happens if the cache line was in the shared state and another processor has requested exclusive ownership.

UMA



- A UMA system with two single core CPUs that share a common front side bus (FSB).
- A UMA system in which the FSBs of two dual-core chips are connected separately to the chipset.

UMA

- The simplest implementation of a UMA system is a dual-core processor, in which two CPUs on one chip share a single path to memory.
- It is very common in high performance computing to use more than one chip in a compute node, be they single core or multicore.
- Two (single-core) processors, each in its own socket, communicate and access memory over a common bus, the so-called *front side bus (FSB)*.
- *All arbitration* protocols required to make this work are already built into the CPUs.
- The chipset (often termed “northbridge”) is responsible for driving the memory modules and connects to other parts of the node like I/O subsystems.
- This kind of design is outdated and is not used any more in modern systems.

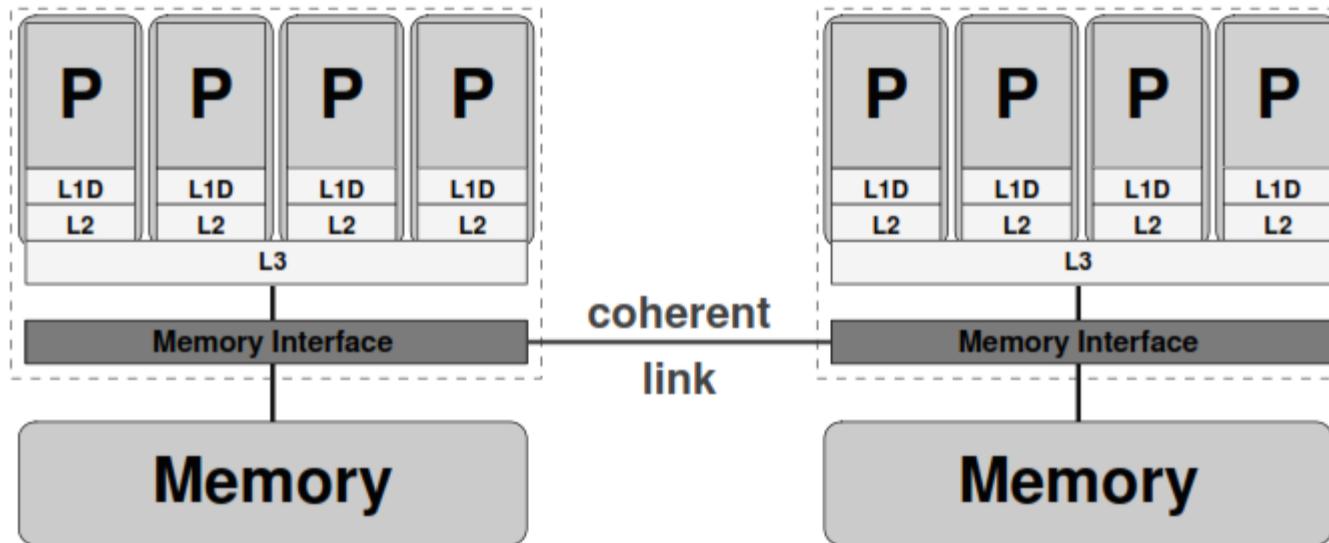
UMA

- Two dual-core chips connect to the chipset, each with its own FSB.
- The chipset plays an important role in enforcing cache coherence and also mediates the connection to memory.
- In principle, a system like this could be designed so that the bandwidth from chipset to memory matches the aggregated bandwidth of the front side buses.
- Each chip features a separate L1 on each core and a dual-core L2 group.
- The arrangement of cores, caches, and sockets make the system inherently *anisotropic*, i.e., the “distance” between one core and another varies depending on whether they are on the same socket or not.
- With large many-core processors comprising multilevel cache groups, the situation gets more complex still.

Problems of UMA

- The general problem of UMA systems is that bandwidth bottlenecks are bound to occur when the number of sockets (or FSBs) is larger than a certain limit.
- In very simple designs a common *memory bus is used that can only transfer data to one CPU at a time.*
- In order to maintain scalability of memory bandwidth with CPU number, nonblocking *crossbar switches can be built that establish point-to-point connections between sockets and memory modules*
- Due to the very large aggregated bandwidths those become very expensive for a larger number of sockets.
- At the time of writing, the largest UMA systems with scalable bandwidth (the NEC SX-9 vector nodes) have sixteen sockets.
- This problem can only be solved by giving up the UMA principle.

ccNUMA



- A ccNUMA system with two locality domains (one per socket) and eight cores.

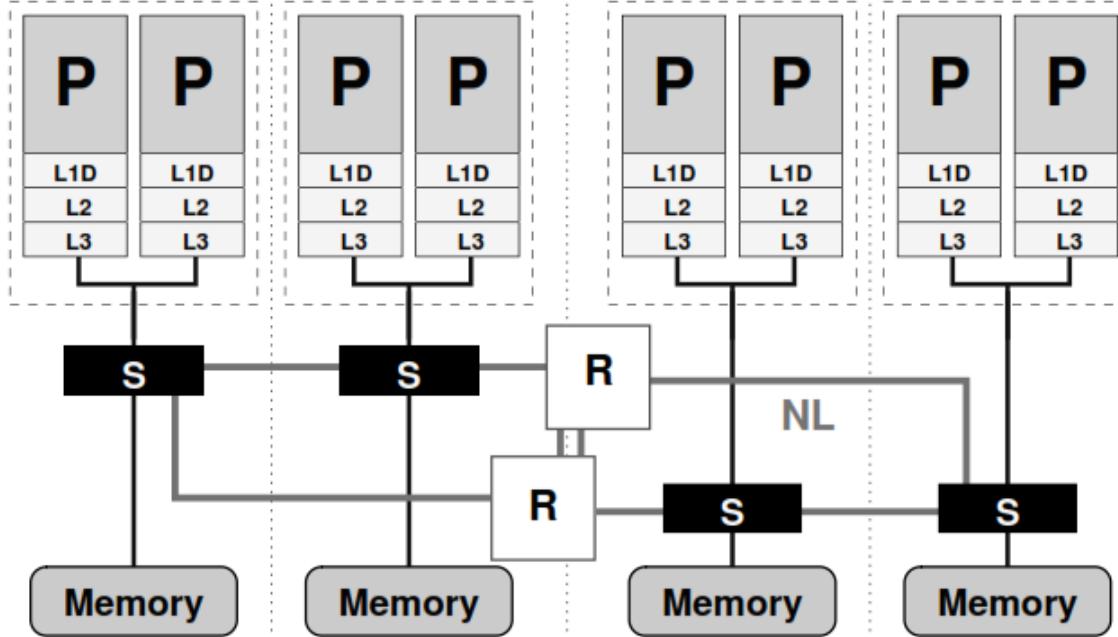
ccNUMA

- A *locality domain (LD)* is a set of processor cores together with locally connected memory.
- This memory can be accessed in the most efficient way, i.e., without resorting to a network of any kind.
- Multiple LDs are linked via a *coherent* interconnect, which allows transparent access from any processor to any other processor's memory.
- In this sense, a locality domain can be seen as a UMA "building block."
- The whole system is still of the shared-memory kind, and runs a single OS instance.
- Although the ccNUMA principle provides scalable bandwidth for very large processor counts, it is also found in inexpensive small two- or four-socket nodes frequently used for HPC clustering.

ccNUMA

- In this particular example two locality domains, i.e., quad-core chips with separate caches and a common interface to local memory, are linked using a high-speed connection.
- *HyperTransport (HT) and QuickPath (QPI) are the current technologies favored by AMD and Intel, respectively, but other solutions do exist.*
- Apart from the minor peculiarity that the sockets can drive memory directly, making separate interface chips obsolete, the inter socket link can mediate direct, cache-coherent memory accesses.
- From the programmer's point of view this mechanism is transparent: All the required protocols are handled by hardware.

ccNUMA



- A ccNUMA system (SGI Altix) with four locality domains, each comprising one socket with two cores.
- The LDs are connected via a routed NUMALink (NL) network using routers (R).

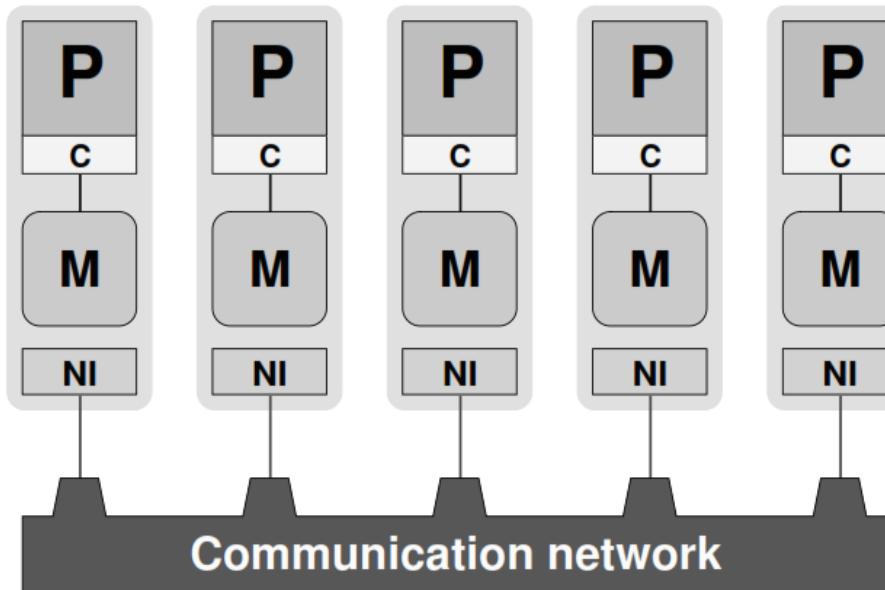
bandwidth and latency

- Network connections must have bandwidth and latency characteristics that are at least the same order of magnitude as for local memory.
- ***Locality Problem:***
- It occurs even if there is only one serial program running on a ccNUMA machine.
- The second problem is potential *contention if two processors from different locality domains access memory in the same locality domain, fighting for memory bandwidth.*
- Even if the network is nonblocking and its performance matches the bandwidth and latency of local access, contention can occur.
- Both problems can be solved by carefully observing the data access patterns of an application and restricting data access of each processor to its own locality domain.

ccNUMA

- In inexpensive ccNUMA systems I/O interfaces are often connected to a single LD.
- Although I/O transfers are usually slow compared to memory bandwidth
- There are high-speed network interconnects that feature multi-GB bandwidths between compute nodes.

Distributed-Memory Computers



- Simplified programmer's view, or “programming model,” of a distributed-memory parallel computer: Separate processes run on processors (P), communicating via interfaces (NI) over some network.
- No process can access another process' memory (M) directly, although processors may reside in shared memory.

Distributed-Memory Computers

- Each processor P is connected to exclusive local memory, i.e., no other CPU has direct access to it.
- Nowadays there are actually no distributed-memory systems any more that implement such a layout.
- *For price/performance reasons all parallel machines today, first and foremost the popular PC clusters, consist of a number of shared-memory “compute nodes” with two or more CPUs*
- The “distributed-memory programmer’s” view does not reflect that.
- It is even possible to use distributed-memory programming on pure shared memory machines.

Distributed-Memory Computers

- Each node comprises at least one network interface (NI) that mediates the connection to a *communication network*.
- A *serial process runs on each CPU that can communicate with other processes on other CPUs by means of the network.*
- It is easy to envision how several processors could work together on a common problem in a shared-memory parallel computer
- But as there is no remote memory access on distributed-memory machines
- The problem has to be solved cooperatively by sending messages back and forth between processes.

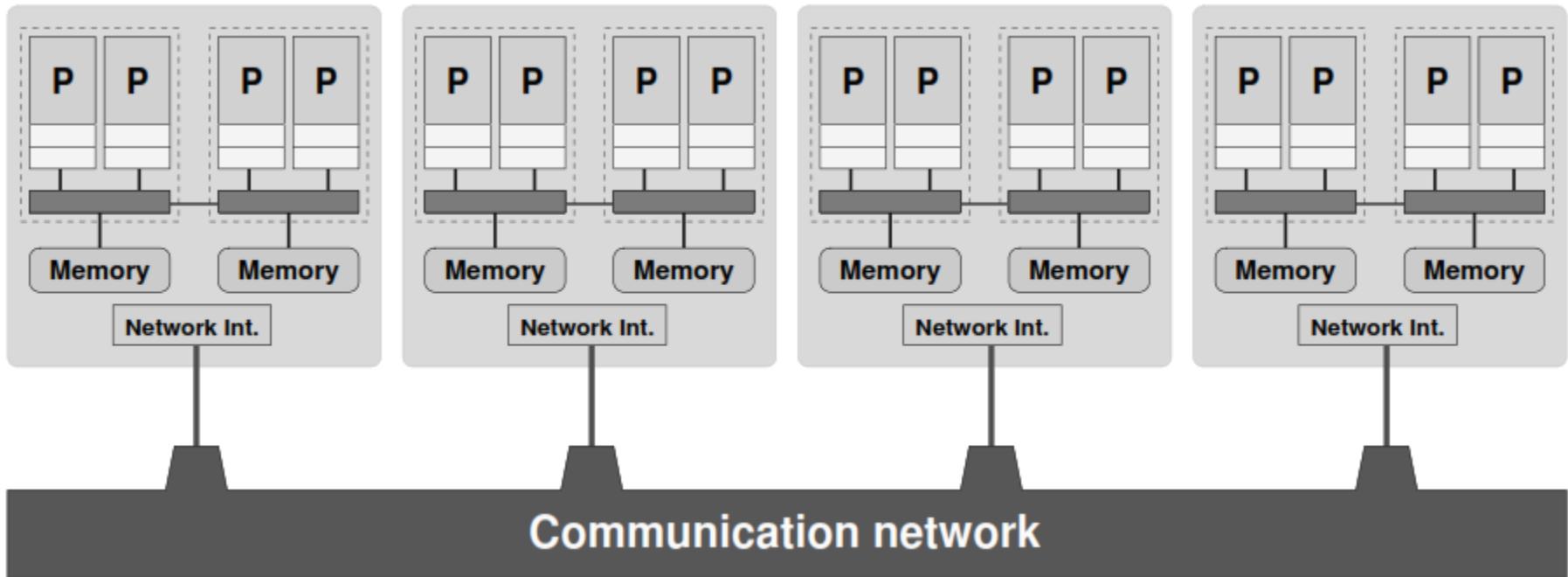
NORMA

- The distributed-memory architecture outlined here is also named *No Remote Memory Access (NORMA)*.
- *Some vendors provide libraries and sometimes hardware* support for limited remote memory access functionality even on distributed-memory machines.
- Since such features are strongly vendor-specific, and there is no widely accepted standard available

Hierarchical (hybrid) systems

- As already mentioned, large-scale parallel computers are neither of the purely shared-memory nor of the purely distributed-memory type
- But a mixture of both, i.e., there are shared-memory building blocks connected via a fast network.
- This makes the overall system design even more anisotropic than with multicore processors and cNUMA nodes, because the network adds another level of communication characteristics
- The concept has clear advantages in terms of price vs. performance;
- It is cheaper to build a shared-memory node with two sockets instead of two nodes with one socket each, as much of the infrastructure can be shared.
- Moreover, with more cores or sockets sharing a single network connection, the cost for networking is reduced.

Hierarchical (hybrid) systems



- Typical hybrid system with shared-memory nodes (ccNUMA type).
- Two-socket building blocks represent the price vs. performance “sweet spot” and are thus found in many commodity clusters.

Hybrid Systems

- Two-socket building blocks are currently the “sweet spot” for inexpensive *commodity clusters*,
- *i.e., systems built from standard components that were not specifically designed for high performance computing.*
- Depending on which applications are run on the system, this compromise may lead to performance limitations due to the reduced available network bandwidth per core.
- Moreover, it is *per se unclear how* the complex hierarchy of cores, cache groups, sockets and nodes can be utilized efficiently.
- The only general consensus is that the optimal programming model is highly application- and system-dependent.

Networks

- The communication overhead can have significant impact on application performance.
- The characteristics of the network that connects the “execution units,” “processors,” “compute nodes,” or whatever play a dominant role here.
- A large variety of network technologies and topologies are available on the market, some proprietary and some open.

Basic performance characteristics of Networks

- The simplest and cheapest solution is Gigabit Ethernet
- Which will suffice for many throughput applications but is far too slow for parallel programs with any need for fast communication.
- *InfiniBand* is the dominating distributed-memory interconnect in commodity clusters.

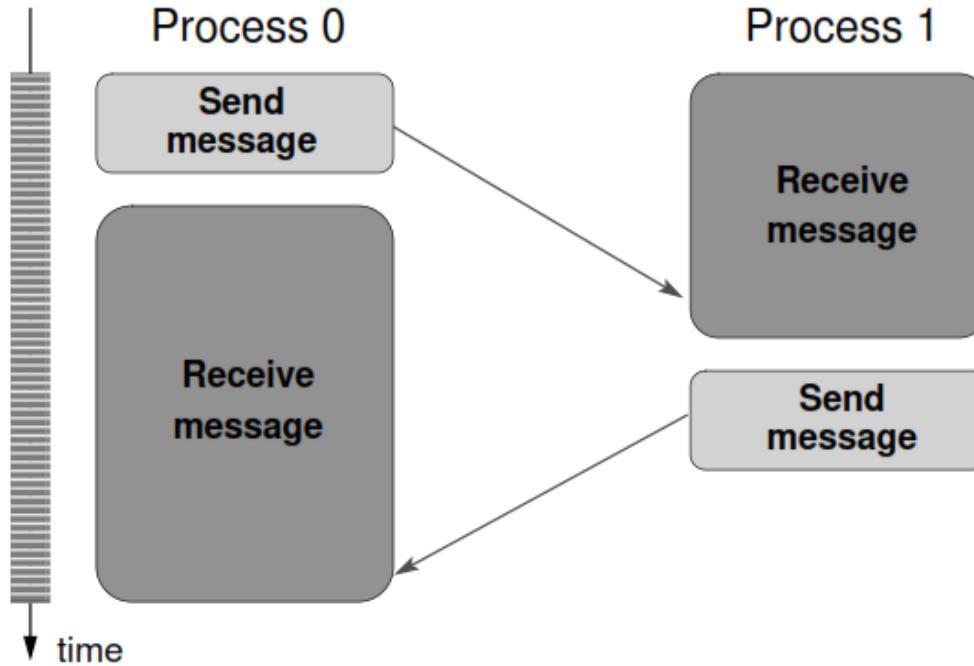
Latency

- All data transmission protocols have some overhead in the form of administrative data like message headers, etc.
- Some protocols (like, e.g., TCP/IP as used over Ethernet) define minimum message sizes, so even if the application sends a single byte, a small “frame” of $N > 1$ bytes is transmitted.
- Initiating a message transfer is a complicated process that involves multiple software layers, depending on the complexity of the protocol.
- Each software layer adds to latency.
- Standard PC hardware as frequently used in clusters is not optimized towards low-latency I/O

high-performance networks

- High-performance networks try to improve latency by reducing the influence of all of the above.
- Lightweight protocols, optimized drivers, and communication devices directly attached to processor buses are all employed by vendors to provide low latency.

PingPong

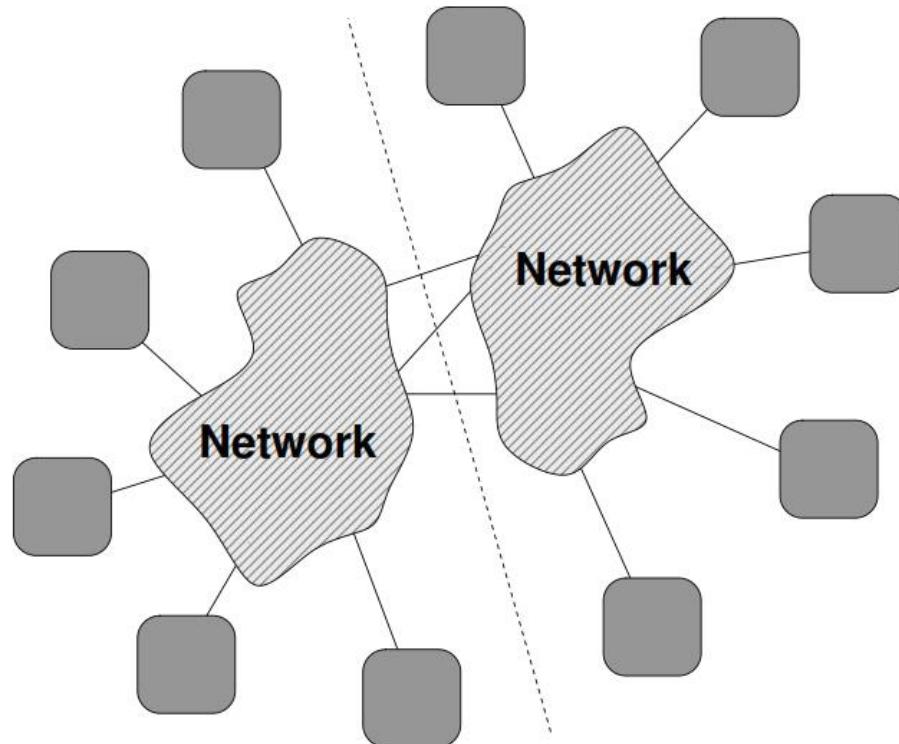


- Timeline for a “PingPong” data exchange between two processes.
- PingPong reports the time it takes for a message of length N bytes to travel from process 0 to process 1 and back.

Bisection bandwidth

- Note that the simple PingPong algorithm described above cannot pinpoint “global” saturation effects:
- If the network fabric is not completely nonblocking and all nodes transmit or receive data at the same time,
- Aggregated bandwidth, i.e., the sum over all effective bandwidths for all point-to-point connections, is lower than the theoretical limit.
- This can severely throttle the performance of applications on large CPU numbers as well as overall throughput of the machine.
- One helpful metric to quantify the maximum aggregated communication capacity across the whole network is its *bisection bandwidth* B_b
- It is the sum of the bandwidths of the minimal number of connections cut when splitting the system into two equal-sized parts.
- In hybrid/hierarchical systems, a more meaningful metric is actually the available bandwidth per core, i.e., bisection bandwidth divided by the overall number of compute cores.
- It is one additional adverse effect of the multicore transition that bisection bandwidth per core goes down.

Bisection Bandwidth



- The bisection bandwidth B_b is the sum of the bandwidths of the minimal number of connections cut (three in this example) when dividing the system into two equal parts.

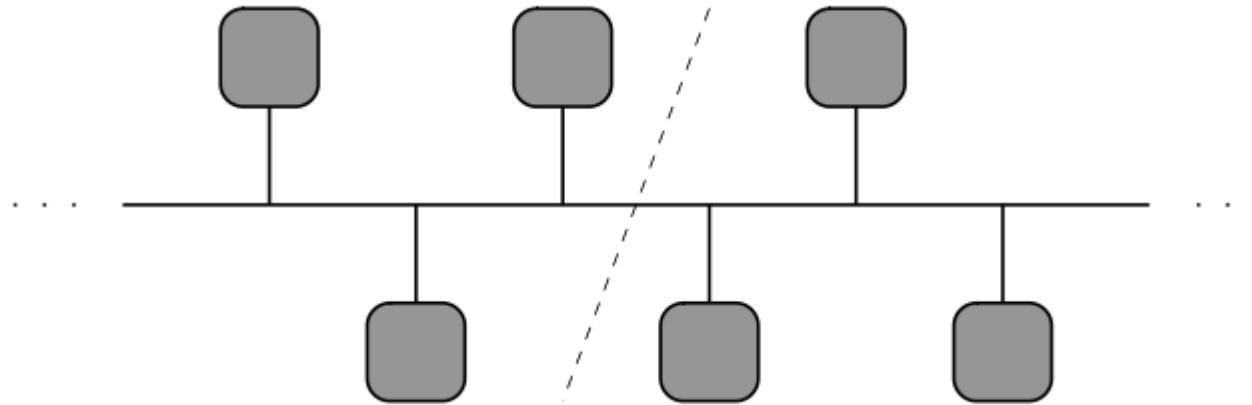
Buses

- A bus is a shared medium that can be used by exactly one communicating device at a time.
- Some appropriate hardware mechanism must be present that detects collisions (i.e., attempts by two or more devices to transmit concurrently).
- Buses are very common in computer systems.
- They are easy to implement, feature lowest latency at small utilization, and ready-made hardware components are available that take care of the necessary protocols.
- A typical example is the PCI bus, which is used in many commodity systems to connect I/O components.
- In some current multicore designs, a bus connects separate CPU chips in a common package with main memory.

Buses

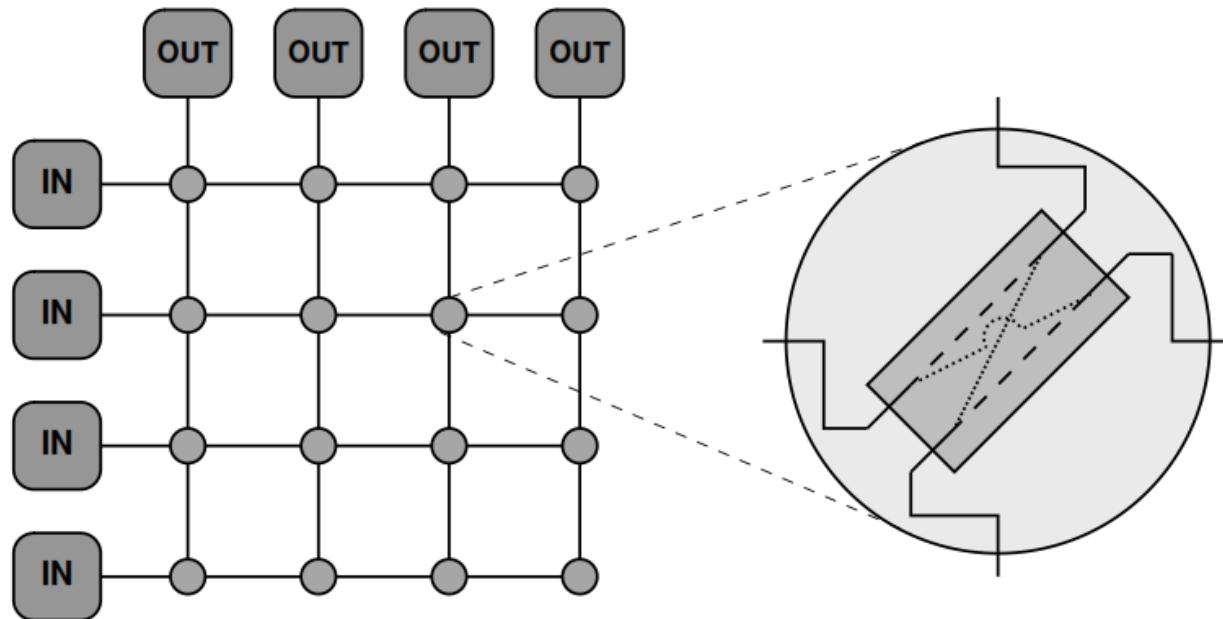
- The most important drawback of a bus is that it is *blocking*.
- *All devices share* a constant bandwidth, which means that the more devices are connected, the lower the average available bandwidth per device.
- Moreover, it is technically involved to design fast buses for large systems as capacitive and inductive loads limit transmission speeds.
- And finally, buses are susceptible to failures because a local problem can easily influence all devices.
- In high performance computing the use of buses for high-speed communication is usually limited to the processor or socket level, or to diagnostic networks.

Bus



- A bus network (shared medium).
- Only one device can use the bus at any time, and bisection bandwidth is independent of the number of nodes.

2D Crossbar Network



- A flat, fully nonblocking two-dimensional crossbar network.
- Each circle represents a possible connection between two devices from the “IN” and “OUT” groups, respectively, and is implemented as a 2×2 switching element.
- The whole circuit can act as a four-port nonblocking switch.

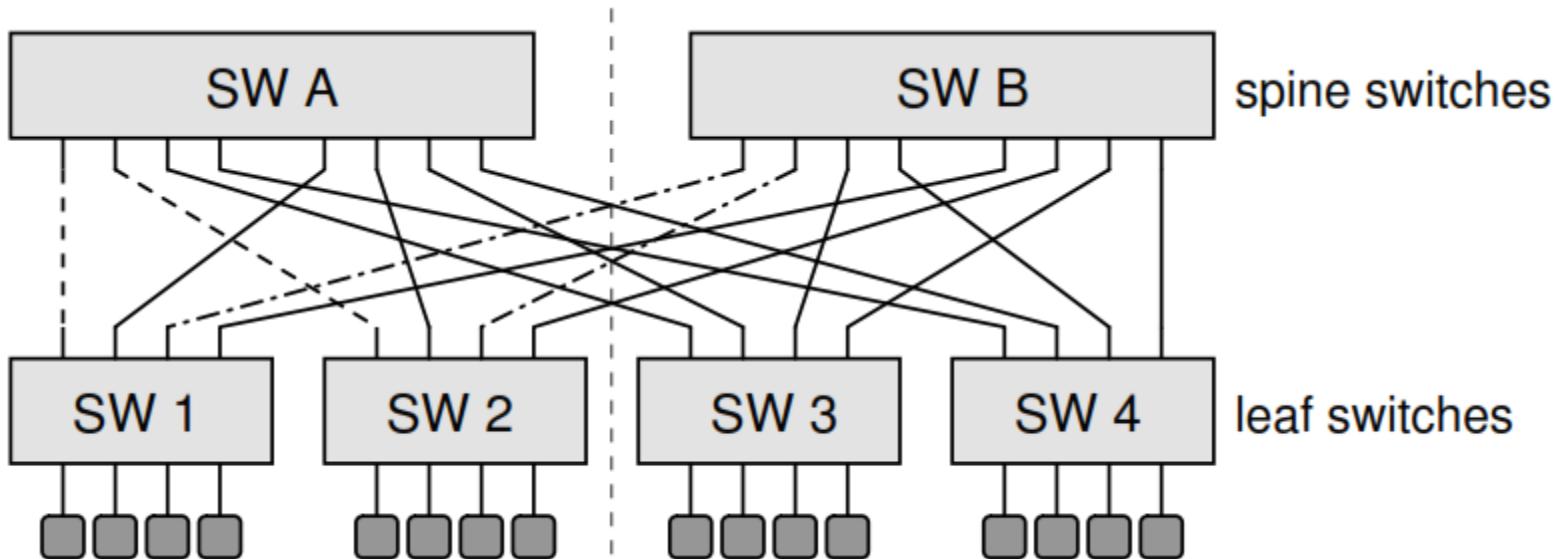
Switched and fat-tree networks

- A switched network subdivides all communicating devices into groups.
- The devices in one group are all connected to a central network entity called a *switch* in a star-like manner.
- Switches are then connected with each other or using additional switch layers.
- In such a network, the distance between two communicating devices varies according to how many “hops” a message has to sustain before it reaches its destination.
- Therefore, a multiswitch hierarchy is necessarily heterogeneous with respect to latency.
- The maximum number of hops required to connect two arbitrary devices is called the *diameter of the network*.

Switched and fat-tree networks

- A single switch can either support a fully *nonblocking operation*,
 - Which means that all pairs of ports can use their full bandwidth concurrently, or
 - It can have partly or completely — a bus-like design where bandwidth is limited.
- One possible implementation of a fully nonblocking switch is a *crossbar*.
- Such building blocks can be combined and cascaded to form a *fat tree switch hierarchy*
- In this case the bisection bandwidth per compute element is less than half the leaf switch bandwidth per port, and contention will occur even if static routing is itself not a problem.
- Note that the network infrastructure must be capable of (dynamically or statically) balancing the traffic from all the leaves over the thinly populated higher-level connections.
- If this is not possible, some node-to-node connections may be faster than others even if the network is lightly loaded.
- On the other hand, maximum latency between two arbitrary compute elements usually depends on the number of switch hierarchy layers only.

full-bandwidth fat-tree network

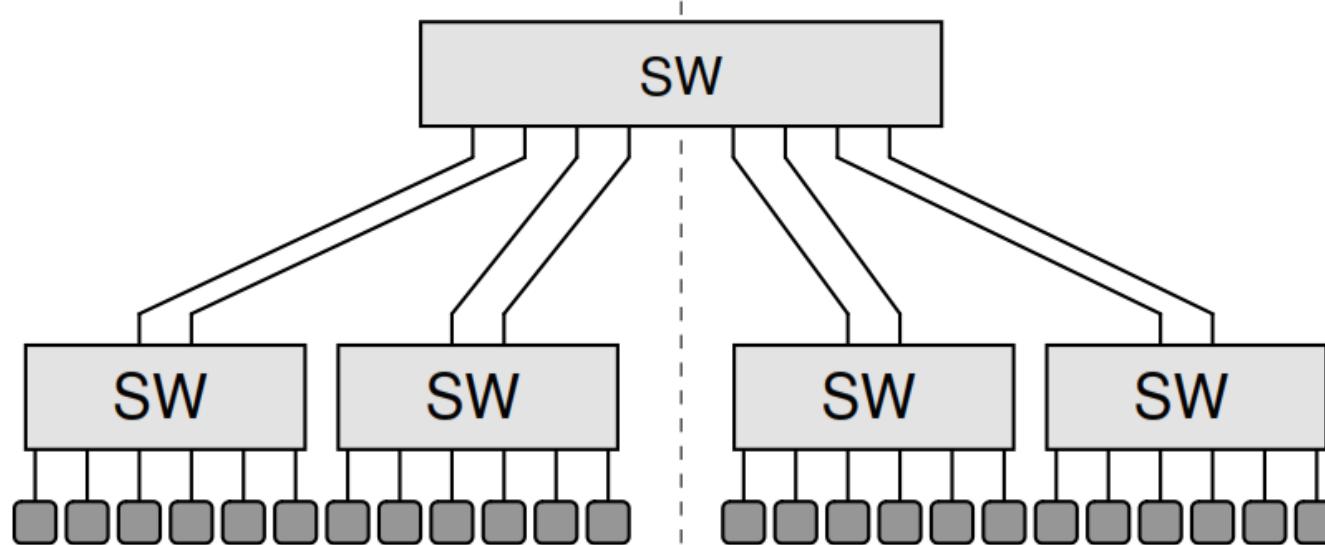


- A fully nonblocking full-bandwidth fat-tree network with two switch layers.
- The switches connected to the actual compute elements are called *leaf switches*, whereas the upper layers form the *spines of the hierarchy*.

Static and Adaptive Routing

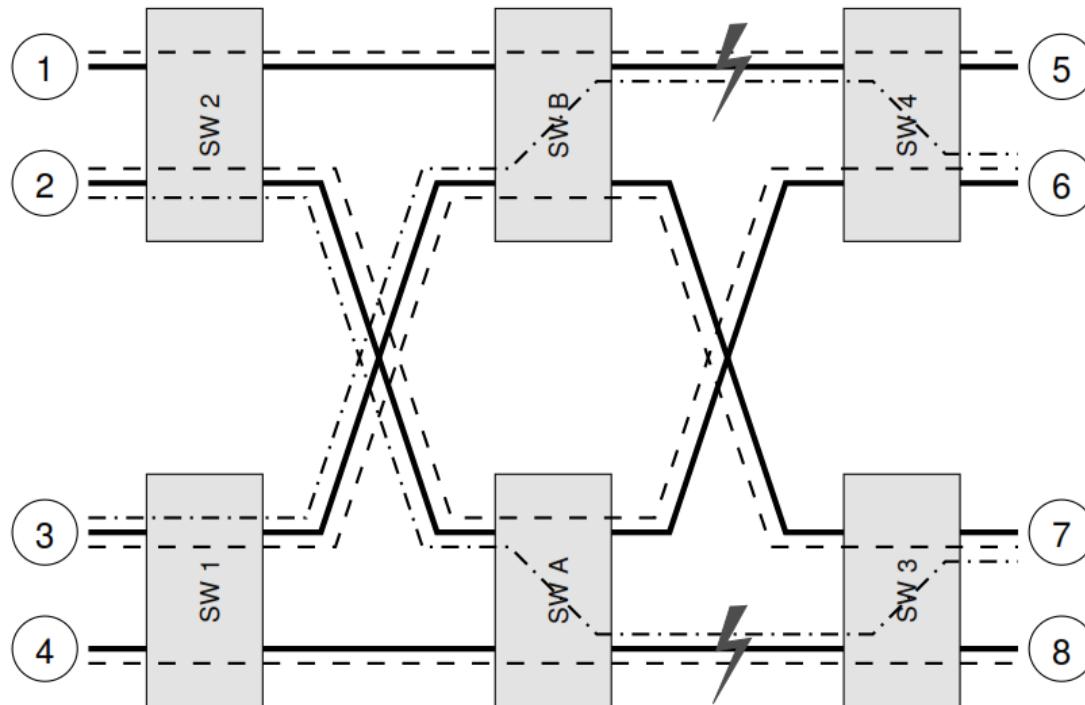
- If *static routing is used, i.e., if connections between compute elements are “hardwired” in the sense that there is one and only one chosen data path (sequence of switches traversed) between any two*
- One can easily encounter situations where the utilization of spine switch ports is unbalanced, leading to collisions when the load is high.
- Many commodity switch products today use static routing tables.
- In contrast, *adaptive routing selects data paths depending on the network load and thus avoids collisions.*
- Only adaptive routing bears the potential of making full use of the available bisection bandwidth for all communication patterns.

Flat Free Network



- A fat-tree network with a bottleneck due to “1:3 oversubscription” of communication links to the spine.
- By using a single spine switch, the bisection bandwidth is cut in half as compared to the layout in last Figure because only four nonblocking pairs of connections are possible.
- Bisection bandwidth per compute element is even lower.

Fat Tree Switch Hierarchy



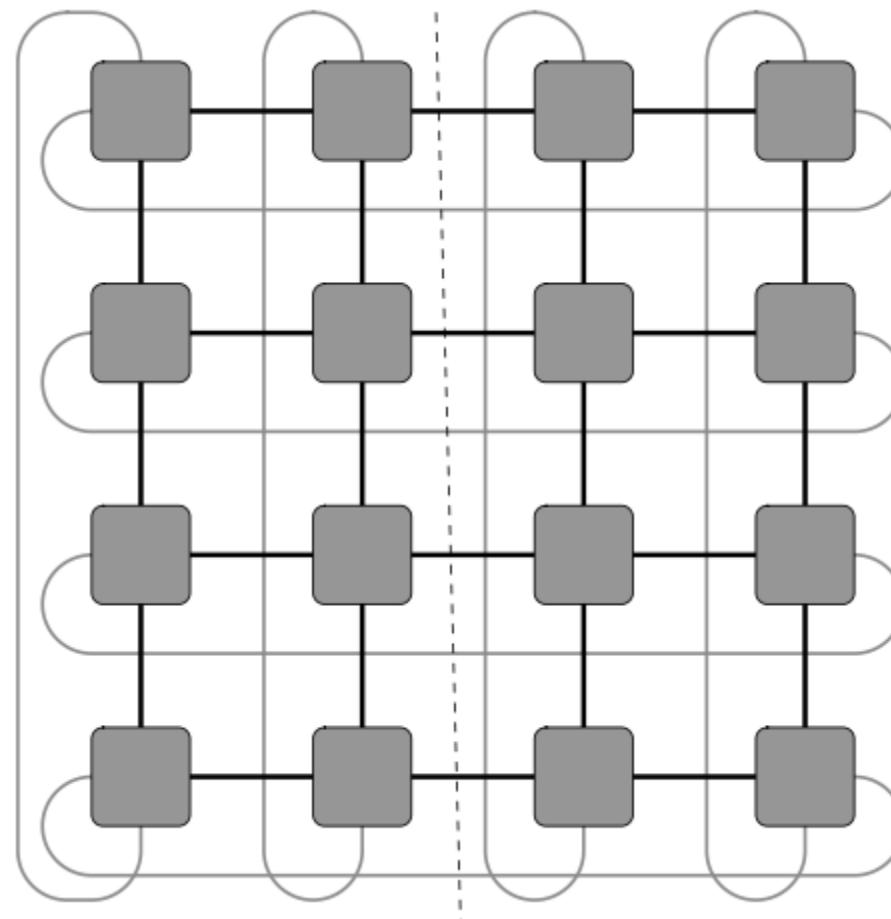
- Even in a fully nonblocking fat-tree switch hierarchy (network cabling shown as solid lines), not all possible combinations of $N/2$ point-to-point connections allow collisionfree operation under static routing.
- When, starting from the collision-free connection pattern shown with dashed lines, the connections $2 \leftrightarrow 6$ and $3 \leftrightarrow 7$ are changed to $2 \leftrightarrow 7$ and $3 \leftrightarrow 6$, respectively (dotted-dashed lines), collisions occur, e.g., on the highlighted links if connections $1 \leftrightarrow 5$ and $4 \leftrightarrow 8$ are not re-routed at the same time.

Mesh networks

- Fat-tree switch hierarchies have the disadvantage of limited scalability in very large systems, mostly in terms of price vs. performance.
- The cost of active components and the vast amount of cabling are prohibitive and often force compromises like the reduction of bisection bandwidth per compute element.
- In order to overcome those drawbacks and still arrive at a controllable scaling of bisection bandwidth, large MPP machines like the IBM Blue Gene or the Cray XT use *mesh networks, usually in the form of multidimensional (hyper-)cubes*.
- *Each* compute element is located at a Cartesian grid intersection.
- Usually the connections are wrapped around the boundaries of the hypercube to form a torus topology.
- There are no direct connections between elements that are not next neighbors.
- The task of routing data through the system is usually accomplished by special ASICs (application specific integrated circuits), which take care of all network traffic, bypassing the CPU whenever possible.
- The network diameter is the sum of the system's sizes in all three Cartesian directions.

A two-dimensional (square) torus network

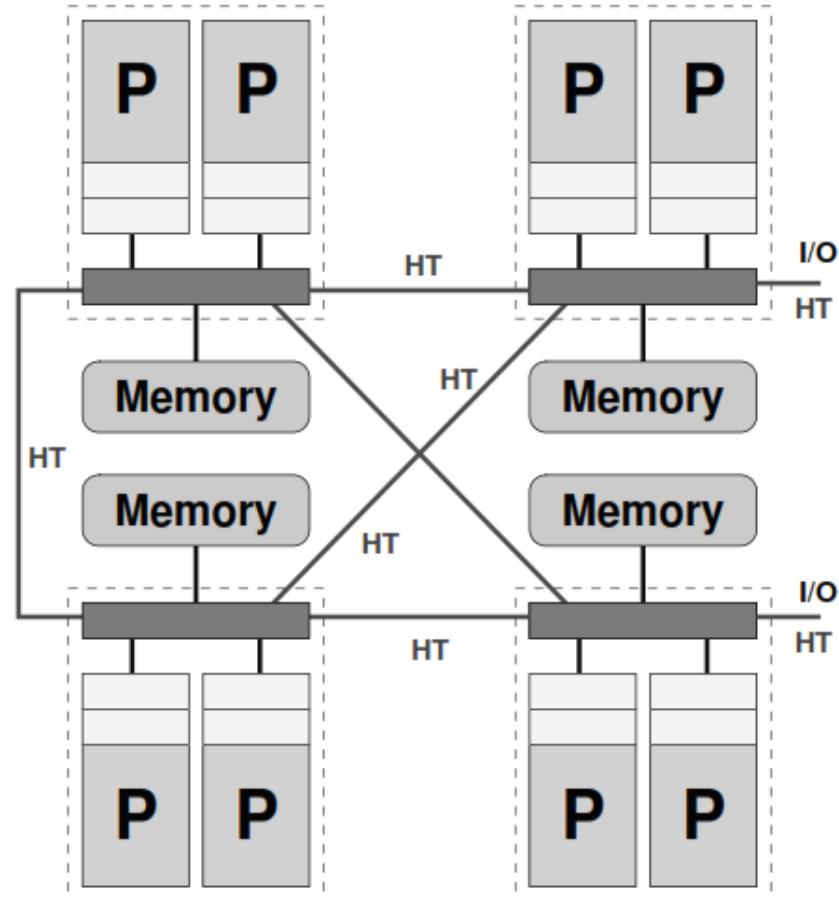
- Bisection bandwidth scales like $\text{Sqrt}(N)$ *in this case.*



-
- Maximum latency scales like $N^{1/d}$.
 - Although these properties appear unfavorable at first sight, the torus topology is an acceptable and extremely cost-effective compromise for the large class of applications that are dominated by nearest-neighbor communication.
 - If the maximum bandwidth per link is substantially larger than what a single compute element can “feed” into the network (its *injection bandwidth*), *there is enough headroom to support* more demanding communication patterns as well.
 - Another advantage of a cubic mesh is that the amount of cabling is limited, and most cables can be kept short.
 - As with fat-tree networks, there is some heterogeneity in bandwidth and latency behavior, but if compute elements that work in parallel to solve a problem are located close together, these characteristics are well predictable.
 - Moreover, there is no “arbitrary” system size at which bisection bandwidth per node suddenly has to drop due to cost and manageability concerns.

Four-socket ccNUMA system

- Figure A four-socket ccNUMA system with a HyperTransport-based mesh network.
- Each socket has only three HT links, so the network has to be heterogeneous in order to accommodate I/O connections and still utilize all provided HT ports.
- On smaller scales, simple mesh networks are used in shared-memory systems for ccNUMA-capable connections between locality domains.
- Figure shows an example of a four-socket server with HyperTransport interconnect.
- This node actually implements a heterogeneous topology (in terms of intersocket latency) because two HT connections are used for I/O connectivity:
- Any communication between the two locality domains on the right incurs an additional hop via one of the other domains.



Hybrids

- If a network is built as a combination of at least two of the topologies described above, it is called *hybrid*.
- *In a sense, a cluster of shared-memory nodes* implements a hybrid network even if the internode network itself is not hybrid.
- This is because intranode connections tend to be buses (in multicore chips) or simple meshes (for ccNUMA-capable fabrics like HyperTransport or QuickPath).
- On the large scale, using a cubic topology for node groups of limited size and a nonblocking fat tree further up reduces the bisection bandwidth problems of pure cubic meshes.

Reference

- **Georg Hager and Gerhard Wellein**, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.

BASICS OF PARALLELIZATION

Dr Noor Mahamamd Sk

Why Parallelize?

- A single core may be too slow to perform the required task(s) in a “tolerable” amount of time.
- The definition of “**tolerable**” certainly varies, but “overnight” is often a reasonable estimate.
- The memory requirements cannot be met by the amount of main memory which is available on a single system, because larger problems (with higher resolution, more physics, more particles, etc.) need to be solved.

Parallelism

- Writing a parallel program must always start by identifying the parallelism inherent in the algorithm at hand.
- Different variants of parallelism induce different methods of parallelization.
- We will restrict ourselves to methods for exploiting parallelism using multiple cores or compute nodes.
- The fine-grained concurrency implemented with superscalar processors and SIMD capabilities.

Data parallelism

- Many problems in scientific computing involve processing of large quantities of data stored on a computer.
- If this manipulation can be performed in parallel i.e., by multiple processors working on different parts of the data, is called *data parallelism*.
- As a matter of fact, this is the dominant parallelization concept in scientific computing on MIMD-type computers.
- It also goes under the name of *SPMD* (*Single Program Multiple Data*), as usually the same code is executed on all processors, with independent instruction pointers.
- It is thus not to be confused with SIMD parallelism.

Example: Medium-grained loop parallelism

- Processing of array data by loops or loop nests is a central component in most scientific codes.
- A typical example are linear algebra operations on **vectors** or **matrices**.
- Often the computations performed on individual array elements are independent of each other and are hence typical candidates for parallel execution by several processors in shared memory.
- The distribution of work across processors is flexible and easily changeable down to the single data element

Example: Medium-grained loop parallelism

- The iterations of a loop are distributed to two processors P1 and P2 (in shared memory) for concurrent execution.

P1	do i=1,500 a(i)=c*b(i) enddo	do i=1,1000 a(i)=c*b(i) enddo
P2	do i=501,1000 a(i)=c*b(i) enddo	

Example: Coarse-grained parallelism by domain decomposition

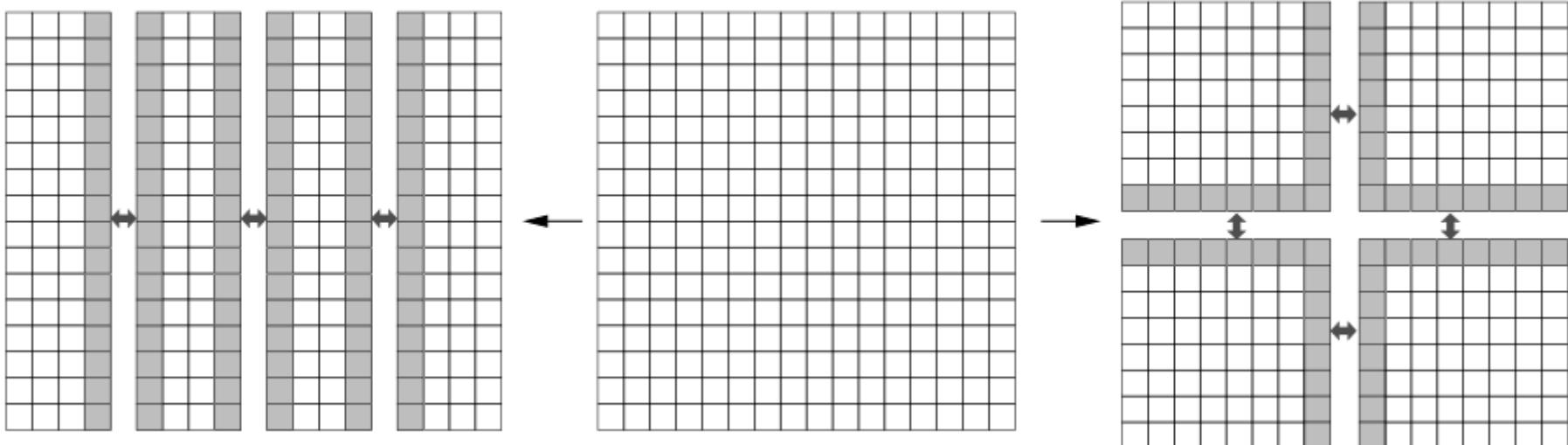
- Simulations of physical processes (like, e.g., fluid flow, mechanical stress, quantum fields) often work with a simplified picture of reality in which a *computational domain*,
- *E.g., some volume of a fluid, is represented as a grid that defines discrete positions for the physical quantities under consideration*
- The goal of the simulation is usually the computation of observables on this grid.
- A straightforward way to distribute the work involved across workers, i.e., processors, is to assign a part of the grid to each worker.
- This is called *domain decomposition*.

Coarse-grained parallelism by domain decomposition

- *load balancing*: The computational effort should be equal for all domains to prevent some workers from idling while others still update their own domains.
- After load imbalance has been eliminated one should care about reducing the communication overhead.
- The data volume to be communicated is proportional to the overall area of the domain cuts.
- Note that the calculation of communication overhead depends crucially on the *locality of data dependencies*
- *In the sense that communication cost grows linearly* with the distance that has to be bridged in order to calculate observables at a certain site of the grid.

-
- Domain decomposition has the attractive property that domain boundary area grows more slowly than volume if the problem size increases with N constant.
 - One can sometimes alleviate communication bottlenecks just by choosing a larger problem size.

Domain decomposition



- Domain decomposition of a two-dimensional Jacobi solver, which requires next neighbor interactions.
- Cutting into stripes (left) is simple but incurs more communication than optimal decomposition (right).
- Shaded cells participate in network communication.

Functional Parallelism

- Sometimes the solution of a “big” numerical problem can be split into more or less disparate subtasks, which work together by data exchange and synchronization.
- In this case, the subtasks execute completely different code on different data items, which is why functional parallelism is also called *MPMD (Multiple Program Multiple Data)*.
- This does not rule out, however, that each subtask could be executed in parallel by several processors in an SPMD fashion.

Pros and Cons of Functional Parallelism

- When different parts of the problem have different performance properties and hardware requirements, bottlenecks and load imbalance can easily arise.
- On the other hand, overlapping tasks that would otherwise be executed sequentially could accelerate execution considerably.

Master-worker scheme

- Reserving one compute element for administrative tasks while all others solve the actual problem is called the *master-worker scheme*.
- *The master distributes work and collects results.*
- A typical example is a parallel ray tracing program:
- A ray tracer computes a photorealistic image from a mathematical representation of a scene.
- For each pixel to be rendered, a “ray” is sent from the imaginary observer’s eye into the scene, hits surfaces, gets reflected, etc., picking up color components.
- If all compute elements have a copy of the scene, all pixels are independent and can be computed in parallel.
- Due to efficiency concerns, the picture is usually divided into “work packages” (rows or tiles).
- Whenever a worker has finished a package, it requests a new one from the master, who keeps lists of finished and yet to be completed tiles.
- In case of a distributed-memory system, the finished tile must also be communicated over the network.

Drawback of the master-worker scheme

- The potential communication and performance bottleneck that may appear with a single master when the number of workers is large.

Functional decomposition

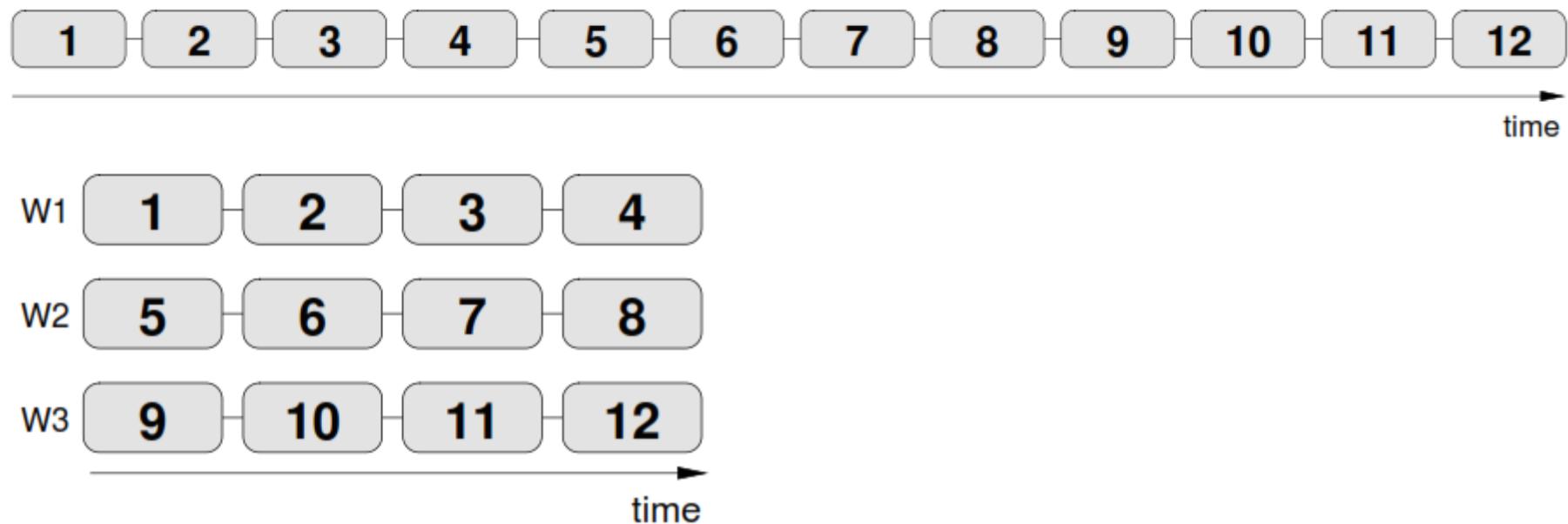
- Multiphysics simulations are prominent applications for parallelization by functional decomposition.
- For instance, the airflow around a racing car could be simulated using a parallel CFD (Computational Fluid Dynamics) code.
- On the other hand, a parallel finite element simulation could describe the reaction of the flexible structures of the car body to the flow, according to their geometry and material properties.
- Both codes have to be coupled using an appropriate communication layer.
- **Load balancing problem** because it is hard in practice to dynamically shift resources between the different functional domains.

Parallel scalability:

Factors that limit parallel execution

- Finding parallelism is not only a common problem in computing but also in many other areas like manufacturing, traffic flow and even business processes.
- In a very simplistic view, all execution units (workers, assembly lines, waiting queues, CPUs, . . .) execute their assigned work in exactly the same amount of time.
- Under such conditions, using N workers, a problem that takes a time T to be solved sequentially will now ideally take only T/N .
- We call this a speedup of N .

Speedup



- Parallelizing a sequence of tasks using three workers (W1 . . . W3) with perfect speedup

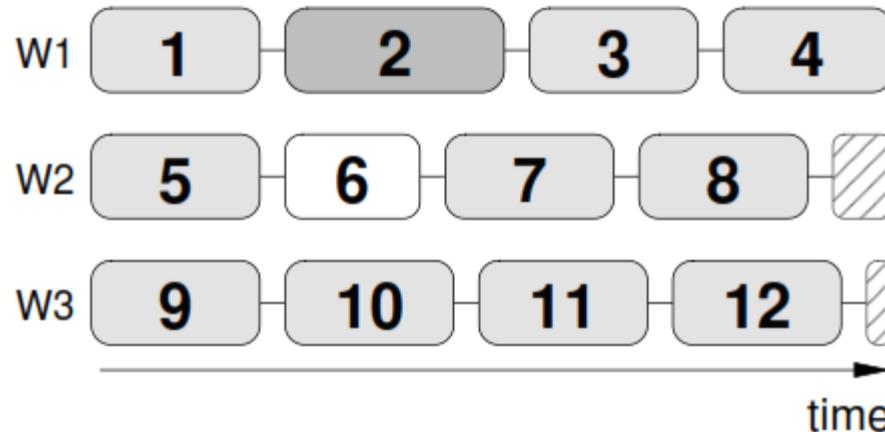
Factors that limit parallel execution

- Not all workers might execute their tasks in the same amount of time because the problem was not (or could not) be partitioned into pieces with equal complexity.
- Hence, there are times when all but a few have nothing to do but wait for the latecomers to arrive.
- This *load imbalance hampers performance because* some resources are underutilized.
- Moreover there might be shared resources like, e.g., tools that only exist once but are needed by all workers.
- This will effectively *serialize part of the concurrent execution*.
- *And finally, the parallel workflow* may require some communication between workers, adding overhead that would not be present in the serial case.
- All these effects can impose limits on speedup.

Use of Scalability Metric

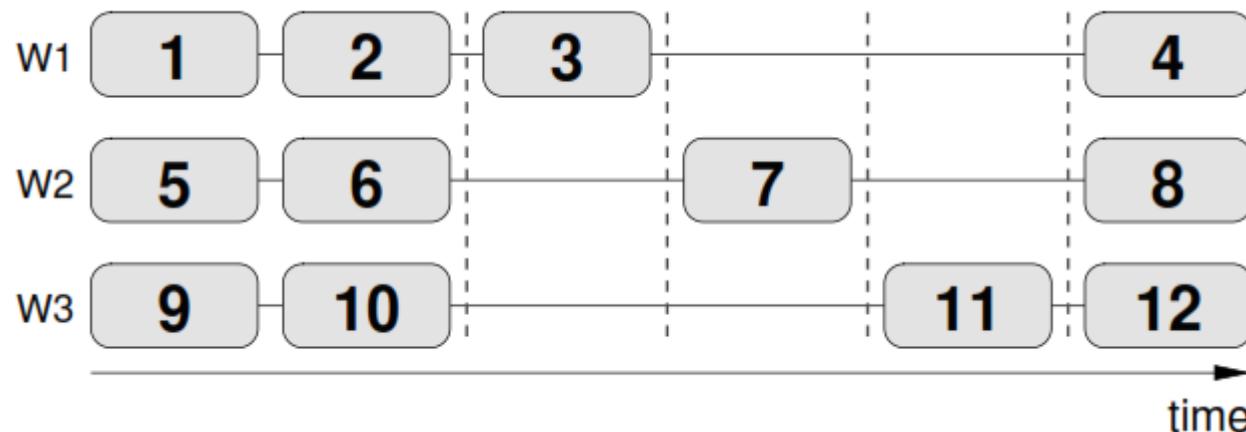
- How well a task can be parallelized is usually quantified by some *scalability metric*.
- Using such metrics, one can answer questions like:
- How much faster can a given problem be solved with N workers instead of one?
- How much more work *can be done with N workers instead of one?*
- What impact do the communication requirements of the parallel application have on performance and scalability?
- What fraction of the resources is actually used productively for solving the problem?

Load Imbalance.



- Some tasks executed by different workers at different speeds lead to *load imbalance*.
- *Hatched regions indicate unused resources.*

Effectively serialize part of the concurrent execution



- Parallelization with a bottleneck.
- Tasks 3, 7 and 11 cannot overlap with anything else across the dashed “barriers.”

Scalability metrics

- The overall problem size (“amount of work”) shall be $s + p = 1$,
- *Where s is the serial (nonparallelizable) part and p is the perfectly parallelizable fraction.*
- *There can be many reasons for a nonvanishing serial part:*
- *Algorithmic limitations.*
- *Bottlenecks*
- *Startup overhead*
- *Communication*

Algorithmic limitations

- Operations that cannot be done in parallel
- Example:
- Mutual dependencies,
- Can only be performed one after another, or even in a certain order.

Bottlenecks

- *Shared resources are common in computer systems:*
- *Execution units in the core, shared paths to memory in multicore chips, I/O devices.*
- Access to a shared resource *serializes execution.*
- *Even if the algorithm itself could be performed completely in parallel, concurrency may be limited by bottlenecks.*

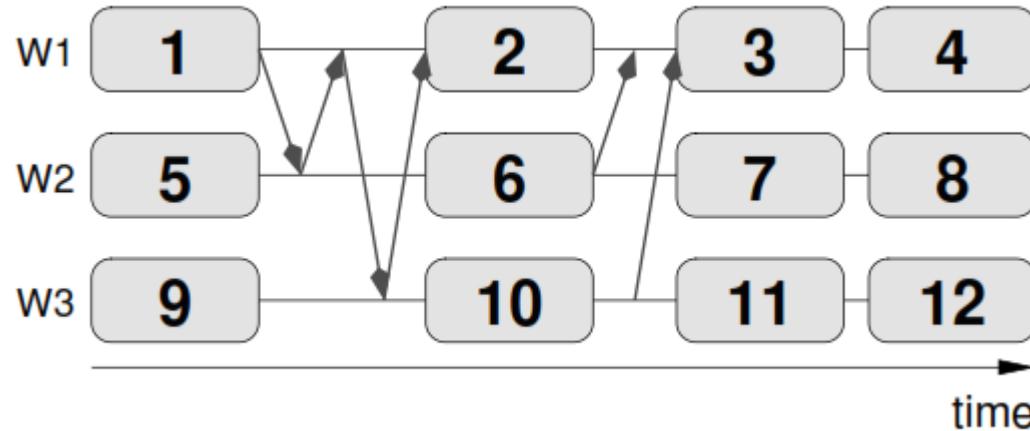
Startup overhead

- *Starting a parallel program, regardless of the technical details,* takes time.
- Of course, system designs try to minimize startup time, especially in massively parallel systems, but there is always a nonvanishing serial part.
- If a parallel application's overall runtime is too short, startup will have a strong impact.

Communication

- *Fully concurrent communication between different parts of a parallel system cannot be taken for granted.*
- If solving a problem in parallel requires communication, some serialization is usually unavoidable.

Communication



- Communication processes (arrows represent messages) limit scalability if they cannot be overlapped with each other or with calculation.

Strong Scaling

- First we assume a fixed problem, which is to be solved by N workers.
- We normalize the single-worker (serial) runtime
- $T_f^s = s + p$
- Solving the same problem on N workers will require a runtime of
$$T_f^p = s + (P/N)$$
- This is called **strong scaling** because the amount of work stays constant no matter how many workers are used.
- Here the goal of parallelization is minimization of time to solution for a given problem.

Weak Scaling

- If time to solution is not the primary objective because larger problem sizes (for which available memory is the limiting factor) are of interest
- It is appropriate to scale the problem size with some power of N so that the total amount of work is $s + pN^\alpha$
- Where α is a positive but otherwise free parameter.
- Here we use the implicit assumption that the serial fraction s is a constant.
- We define the serial runtime for the scaled (variably-sized) problem as
- $T_v^s = s + pN^\alpha$
- Consequently, the parallel runtime is
- $T_v^p = s + pN^{\alpha-1}$

Parallel Efficiency

- Parallel efficiency is then defined as

$$\varepsilon = \frac{\text{performance on } N \text{ CPUs}}{N \times \text{performance on one CPU}} = \frac{\text{speedup}}{N}$$

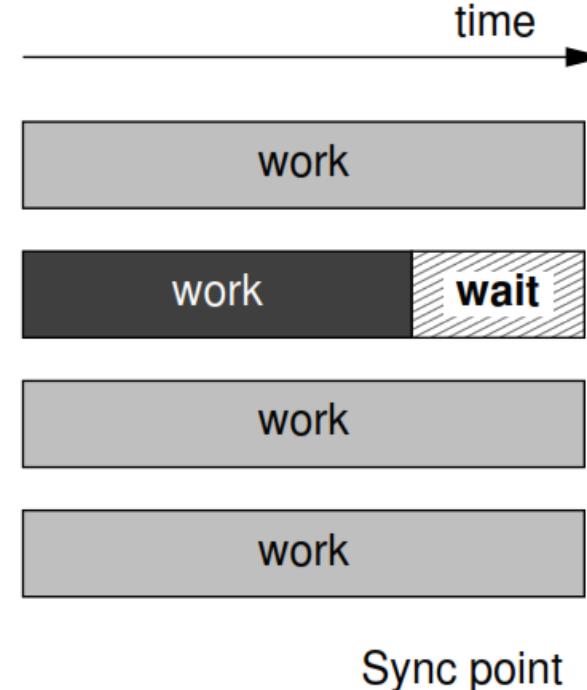
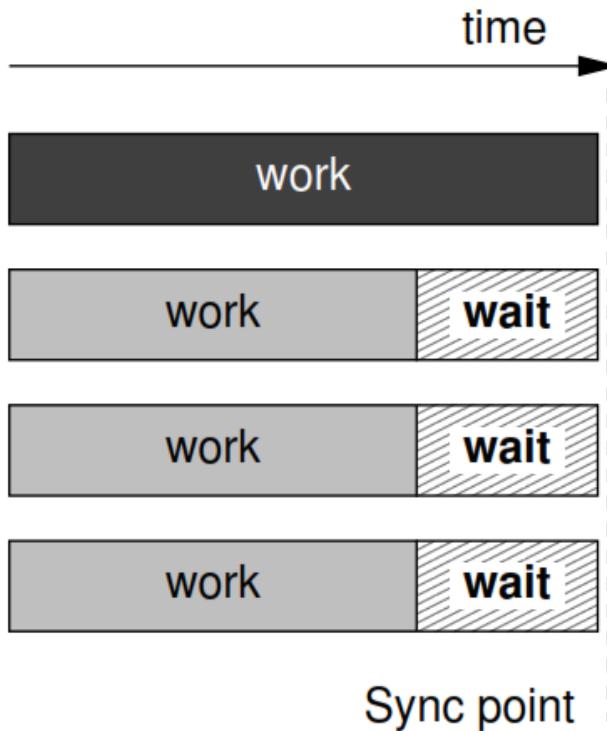
Load imbalance

- Inexperienced HPC users usually try to find the reasons for bad scalability of their parallel programs in the hardware details of the platform used and the specific drawbacks of the chosen parallelization method:
- Communication overhead, synchronization loss, false sharing, NUMA locality, bandwidth bottlenecks, etc.
- While all these are possible reasons for bad scalability (and are covered in due detail elsewhere in this book), load imbalance is often overlooked.
- Load imbalance occurs when synchronization points are reached by some workers earlier than by others, leading to at least one worker idling while others still do useful work.
- As a consequence, resources are underutilized.

Load imbalance

- The consequences of load imbalance are hard to characterize in a simple model without further assumptions about the work distribution.
- Also, the actual impact on performance is not easily judged:
- Having a few workers that take longer to reach the synchronization point (“laggers”) leaves the rest,
- i.e., the majority of workers, idling for some time, incurring significant loss.
- On the other hand, a few “speeders,” i.e., workers that finish their tasks early, may be harmless because the accumulated waiting time is negligible.

Load imbalance



- Load imbalance with few (one in this case) “laggers”: A lot of resources are underutilized (hatched areas).

- Load imbalance with few (one in this case) “speeders”: Underutilization may be acceptable.

Reference

- **Georg Hager and Gerhard Wellein**, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.

SHARED-MEMORY PARALLEL PROGRAMMING WITH OPENMP

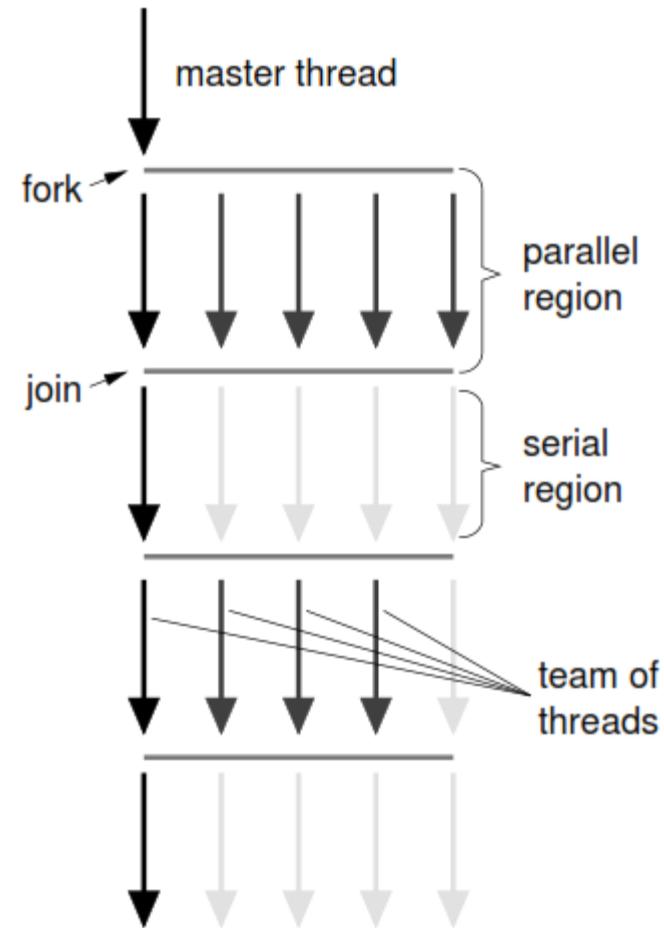
Dr Noor Muhammad Sk

OpenMP

- Shared memory opens the possibility to have immediate access to all data from all processors without explicit communication.
- OpenMP is a set of *compiler directives* that a non-OpenMP-capable compiler would just regard as comments and ignore.
- A well-written parallel OpenMP program is also a valid serial program (with Compiler Directives of OpenMP).
- The central entity in an OpenMP program is not a process but a *Thread*.
- Threads are also called “**lightweight processes**” because several of them can share a common address space and mutually access data.
- Spawning a thread is much less costly than forking a new process, because threads share everything but instruction pointer (the address of the next instruction to be executed), stack pointer and register state.

Uses Fork and Join Model

- The master thread “forks” team of threads, which work on shared memory in a parallel region.
- After the parallel region, the threads are “joined,” i.e., terminated or put to sleep, until the next parallel region starts.
- The number of running threads may vary among parallel regions.



Thread – Private Variable

- Each thread can, by means of its local stack pointer, also have “private” variables,
- Since all data is accessible via the common address space
- It is only a matter of taking the address of an item to make it accessible to all other threads as well.
- However, the OpenMP standard actually *forbids making a private object available* to other threads via its address.

Parallel Execution

- A single thread is called the *master thread*, runs immediately after startup.
- Truly parallel execution happens inside *parallel regions*, of which an arbitrary number can exist in a program.
- Between two parallel regions, no thread except the master thread executes any code.
- Inside a parallel region, a *team of threads executes instruction streams* concurrently.
- The number of threads in a team may vary among parallel regions.

Example

- OpenMP is a layer that adapts the raw OS thread interface to make it more usable with the typical structures that numerical software tends to employ.

```
1 #include <omp.h>
2
3 std::cout << "I am the master, and I am alone";
4 #pragma omp parallel
5 {
6     do_work_package(omp_get_thread_num(),omp_get_num_threads());
7 }
```

```
1 $ export OMP_NUM_THREADS=4
2 $ ./a.out
```

Data Scoping

- Any variables that existed before a parallel region still exist inside, and are by default shared between all threads.
- True work sharing, however, makes sense only if each thread can have its own, *private variables*.
- *OpenMP supports this concept* by defining a separate stack for every thread.
- There are three ways to make private variables.

Private variable

- A variable that exists before entry to a parallel construct can be privatized, i.e., made available as a private instance for every thread, by a PRIVATE clause to the OMP PARALLEL directive.
- The private variable's scope extends until the end of the parallel construct.
- The index variable of a work sharing loop is automatically made private.
- Local variables in a subroutine called from a parallel region are private to each calling thread.

Private Clause

- The PRIVATE clause to the PARALLEL directive privatizes all specified variables, i.e., each thread gets its own instance of each variable on its local stack, with an undefined initial value

OpenMP worksharing for loops

- Loops are natural candidates for parallelization if individual iterations are independent.
- This corresponds to the medium-grained data parallelism
- The iterations of the loop are distributed among the threads (which are running because we are in a parallel region).
- Each thread gets its own iteration space, i.e., is assigned to a different set of i values.
- How threads are mapped to iterations is implementation-dependent by default, but can be influenced by the programmer.
- Although shared in the enclosing parallel region, the loop counter i is privatized automatically.
- Loop counters are restricted to integers (signed or unsigned), pointers, or random access iterators.

Example

```
1  double precision :: pi,w,sum,x
2  integer :: i,N=1000000
3
4  pi = 0.d0
5  w = 1.d0/N
6  sum = 0.d0
7 !$OMP PARALLEL PRIVATE (x) FIRSTPRIVATE (sum)
8 !$OMP DO
9   do i=1,n
10     x = w*(i-0.5d0)
11     sum = sum + 4.d0/(1.d0+x*x)
12   enddo
13 !$OMP END DO
14 !$OMP CRITICAL
15   pi= pi + w*sum
16 !$OMP END CRITICAL
17 !$OMP END PARALLEL
```

Race Condition

- In a parallel loop, each thread executes “its” share of the loop’s iteration space, accumulating into its private *sum* variable.
- After the loop, and still inside the parallel region, the partial sums must be added to get the final result
- Because the private instances of *sum* will be gone once the region is left.
- **There is a problem**, however: Without any countermeasures, threads would write to the result variable *pi* concurrently.
- The result would depend on the exact order the threads access *pi*, and it would most probably be wrong.
- This is called a *race condition*.

Synchronization - Critical Regions

- Concurrent write access to a shared variable or, in more general terms, a shared resource, must be avoided by all means to circumvent race conditions.
- *Critical regions* solve this problem by making sure that at most one thread at a time executes some piece of code.
- If a thread is executing code inside a critical region, and another thread wants to enter, the latter must wait (block) until the former has left the region.
- Note that the order in which threads enter the critical region is undefined, and can change from run to run.
- Consequently, the definition of a “correct result” must encompass the possibility that the partial sums are accumulated in a random order, and the usual reservations regarding floating-point accuracy do apply.

Deadlock

- Critical regions hold the danger of *deadlocks when used inappropriately.*
- A *deadlock* arises when one or more “agents” (threads in this case) wait for resources that will never become available, a situation that is easily generated with badly arranged CRITICAL directives.
- When a thread encounters a CRITICAL directive inside a critical region, it will block forever.
- Since this could happen in a deeply nested subroutine, deadlocks are sometimes hard to pin down.

Solution to Deadlock

- OpenMP has a simple solution for this problem:
- A critical region may be given a *name* that distinguishes it from others.
- Without the names on the two different critical regions, this code would deadlock because a thread that has just called `func()`,
- Already in a critical region, would immediately encounter the second critical region and wait for itself indefinitely to free the resource.
- With the names, the second critical region is understood to protect a different resource than the first.

```
1 !$OMP PARALLEL DO PRIVATE (x)
2   do i=1,N
3     x = SIN(2*PI*x/N)
4   !$OMP CRITICAL (psum)
5     sum = sum + func(x)
6   !$OMP END CRITICAL (psum)
7   enddo
8 !$OMP END PARALLEL DO
9 ...
10 double precision func(v)
11 double precision :: v
12 !$OMP CRITICAL (prand)
13   func = v + random_func()
14 !$OMP END CRITICAL (prand)
15 END SUBROUTINE func
```

Disadvantage of Named Critical Regions

- The names are unique identifiers.
- It is not possible to have them indexed by an integer variable, for instance.
- There are OpenMP API functions that support the use of *locks for protecting shared resources*.
- The advantage of locks is that they are ordinary variables that can be arranged as arrays or in structures.
- That way it is possible to protect each single element of an array of resources individually, even if their number is not known at compile time.

Synchronization - Barriers

- If, at a certain point in the parallel execution, it is necessary to synchronize *all* threads.
- The barrier is a *synchronization point, which guarantees that all threads have reached* it before any thread goes on executing the code below it.
- Certainly it must be ensured that every thread hits the barrier, or a deadlock may occur.
- Note also that every parallel region executes an implicit barrier at its end, which cannot be removed.
- There is also a default implicit barrier at the end of work sharing loops and some other constructs to prevent race conditions.

Reductions

- the loop implements a *reduction* operation:
- Many contributions (the updated elements of `a()`) are accumulated into a single variable.
- We have previously solved this problem with a critical region, but OpenMP provides a more elegant alternative by supporting reductions directly via the REDUCTION clause.
- It automatically privatizes the specified variable(s) and initializes the private instances with a sensible starting value.
- At the end of the construct, all partial results are accumulated into the shared instance of `s`, using the specified operator (+ here) to get the final result.
- The most common cases (addition, subtraction, multiplication, logical, etc.) are covered.

Example

- Example with reduction clause for adding noise to the elements of an array and calculating its vector norm.

```
1  double precision :: r,s
2  double precision, dimension(N) :: a
3
4  call RANDOM_SEED()
5 !$OMP PARALLEL DO PRIVATE(r) REDUCTION(+:s)
6  do i=1,N
7      call RANDOM_NUMBER(r)    ! thread safe
8      a(i) = a(i) + func(r)   ! func() is thread safe
9      s = s + a(i) * a(i)
10 enddo
11 !$OMP END PARALLEL DO
12
13 print *, 'Sum = ', s
```

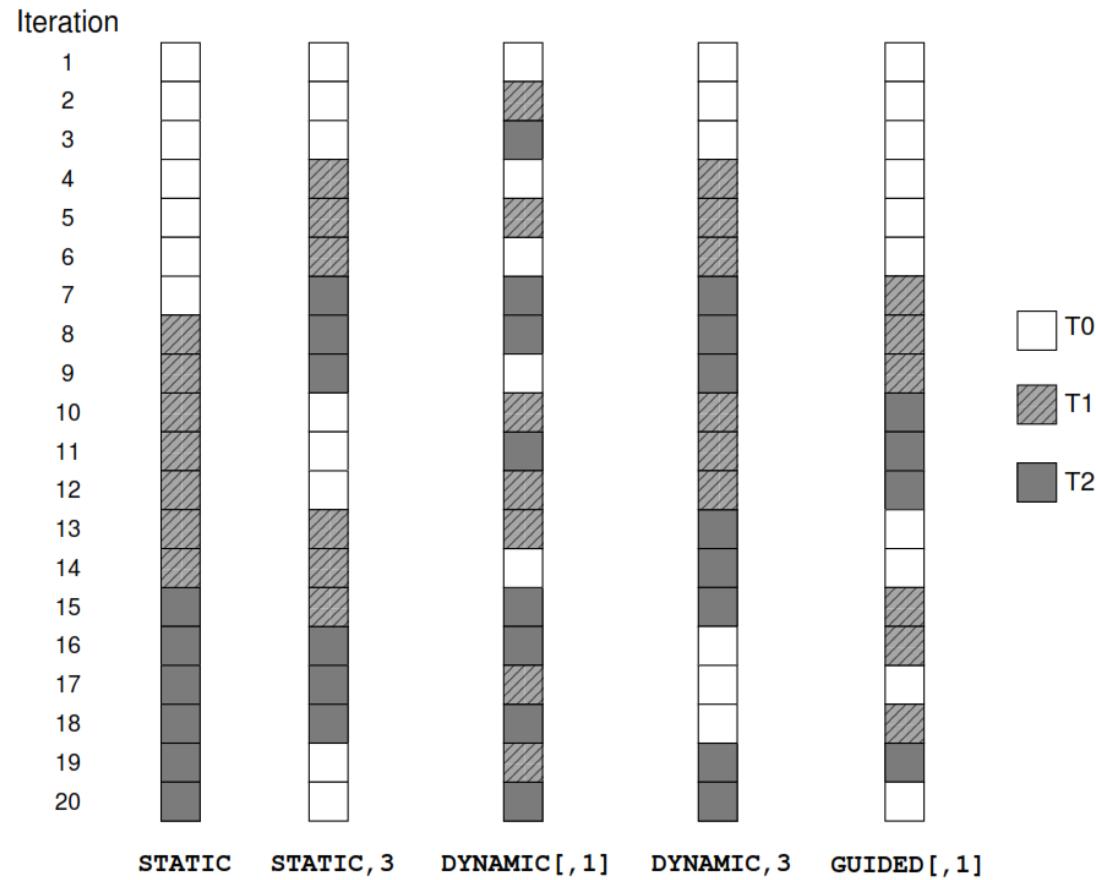
Loop Scheduling

```
1 !$OMP DO SCHEDULE (STATIC)
2   do i=1,N
3     a(i) = calculate(i)
4   enddo
5 !$OMP END DO
```

- As mentioned earlier, the mapping of loop iterations to threads is configurable.
- It can be controlled by the argument of a SCHEDULE clause to the loop work sharing directive:
- The simplest possibility is STATIC, which divides the loop into contiguous chunks of (roughly) equal size.
- Each thread then executes on exactly one chunk.
- If for some reason the amount of work per loop iteration is not constant but, e.g., decreases with loop index, this strategy is suboptimal because different threads will get vastly different workloads, which leads to load imbalance.
- One solution would be to use a *chunk size like in “STATIC,1,” dictating that chunks of size 1 should be distributed across threads in a round-robin manner.*
- The chunk size may not only be a constant but any valid integer-valued expression.

Loop schedules in OpenMP

- The example loop has 20 iterations and is executed by three threads (T0, T1, T2).
- The default chunksize for DYNAMIC and GUIDED is one.
- If a chunkszie is specified, the last chunk may be shorter.
- Note that only the STATIC schedules guarantee that the distribution of chunks among threads stays the same from run to run.



Dynamic Scheduling

- *Dynamic scheduling assigns a chunk of work, whose size is defined by the chunk size, to the next thread that has finished its current chunk.*
- This allows for a very flexible distribution which is usually not reproduced from run to run.
- Threads that get assigned “easier” chunks will end up completing more of them, and load imbalance is greatly reduced.
- The dynamic scheduling generates significant overhead if the chunks are too small in terms of execution time.
- This is why it is often desirable to use a moderately large chunk size on tight loops, which in turn leads to more load imbalance.
- In cases where this is a problem, the *guided schedule may help.*

DATA RACE

Example: Helloworld with OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char argv[]){
    int omp_rank;
#pragma omp parallel private(omp_rank)
{
    omp_rank = omp_get_thread_num();
    printf("Hello world! by
        thread %d", omp_rank);
}
}
```

```
> cc -h omp omp_hello.c -o omp
> aprun -n 1 -d 4 -e OMP_NUM_THREADS=4
./omp
Hello world! by thread 2
Hello world! by thread 3
Hello world! by thread 0
Hello world! by thread 1
```

Race Condition

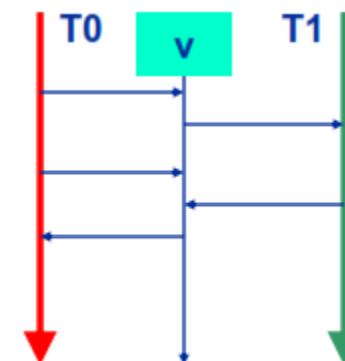
- Race conditions take place when multiple threads read and write a variable simultaneously, for example

```
asum = 0.0d0
!$OMP PARALLEL DO SHARED(x,y,n,asum) PRIVATE(i)
do i = 1, n
    asum = asum + x(i)*y(i)
end do
!$OMP END PARALLEL DO
```

- Random results depending on the order the threads access **asum**
- We need some mechanism to control the access

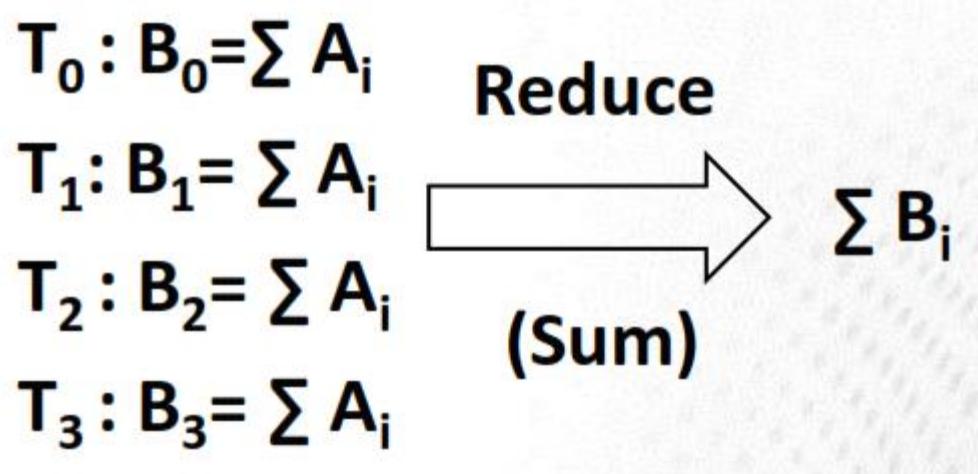
Race Condition

- Two threads access the same shared variable and at least one thread modifies the variable and the sequence of the accesses is undefined, i.e. unsynchronized
- The result of a program depends on the detailed timing of the threads in the team.
- This is often caused by unintended sharing of data
- Consequence: Wrong results



Reductions

- Summing elements of array is an example of reduction operation



- OpenMP provides support for common reductions with the reduction clause

Reduction Clause

- **reduction(operator:var_list)**
- Performs reduction on the (scalar) variables in list
- Private reduction variable is created for each thread's partial result
- Private reduction variable is initialized to operator's initial value
- After parallel region the reduction operation is applied to private variables and result is aggregated to the shared variable

Reduction Operators

Operator	Initial value
+	0
-	0
*	1

Operator	Initial value
&	~ 0
	0
^	0
&&	1
	0

C/C++ only

Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEGV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.
MIN	max pos.
MAX	min neg.

Fortran only

Race Condition with Reduction

```
!$OMP PARALLEL DO SHARED(x,y,n) PRIVATE(i) REDUCTION(:asum)
do i = 1, n
    asum = asum + x(i)*y(i)
end do
!$OMP END PARALLEL DO
```

Execution Controls

- Sometimes a part of parallel region should be executed only by the master thread or by a single thread at time
 - IO, initializations, updating global values, etc.
 - Synchronization
- OpenMP provides clauses for controlling the execution of code blocks

Execution Controls: barrier

- Synchronizes all threads at this point
- When a thread reaches a barrier it only continues after **ALL** threads have reached it
- Implicit barrier at the end of parallel do/for, single
- **Restrictions:**
- Each barrier must be encountered by all threads in a team, or none at all
- The sequence of work-sharing regions and barrier regions encountered must be same for all threads in team

Execution Controls

- **master**
 - Specifies a region that should be executed only by the master thread
 - Note that there is no implicit barrier at end
-
- **single**
 - Specifies that a regions should be executed only by a single (arbitrary) thread
 - Other threads wait (implicit barrier)

Execution Controls

- **critical [(name)]**
 - A section that is executed by only one thread at a time
 - Optional name specifies different critical section
 - Unnamed critical sections are treated as the **same section**
-
- **flush [(name)]**
 - Synchronizes the memory of all threads
 - Makes sure each thread has a consistent view of memory
 - Implicit flush at barrier, critical

Execution Controls

- **atomic**
- Strictly limited construct to update a single value, can not be applied to code blocks
- Can be faster on hardware platforms that support atomic updates

Example: Reduction using Critical Section

```
!$OMP PARALLEL SHARED(x,y,n,asum) PRIVATE(i, psum)
  psum = 0.0d
  !$OMP DO
    do i = 1, n
      psum = psum + x(i)*y(i)
    end do
  !$OMP END DO
  !$OMP CRITICAL(dosum)
    asum = asum + psum
  !$OMP END CRITICAL(dosum)
  !$OMP END PARALLEL DO
```

Example: Initialization and Output

```
#pragma omp parallel
while (err > tolerance) {
    #pragma omp master
    {
        err = 0.0;
    }
    #pragma omp barrier
    // Compute err
    ...
    #pragma omp single
    printf("Error is now: %5.2f\n", err);
}
```

Example: Updating Global Variable

```
int global_max = 0;
int local_max = 0;
#pragma omp parallel firstprivate(local_max) private(i)
{
#pragma omp for
for (i=0; i < 100; i++) {
    local_max = MAX(local_max, a[i]);
}
#pragma omp critical(domax)
global_max = MAX(local_max, global_max);
}
```

Deadlock in Critical Construct OpenMP

- The code inside a CRITICAL region is executed by only one thread at a time.
- The order is not specified.
- This means that if a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.

```
#pragma omp critical  
x = x + 1;
```

Example

```
subroutine foo
    !$OMP PARALLEL
    !$OMP CRITICAL
        print*, 'Hallo i am just a single thread and I like it that way'
    !$OMP END CRITICAL
    !$OMP END PARALLEL
end subroutine foo

program deadlock
implicit none
integer :: i,sum = 0
    !$OMP PARALLEL
    !$OMP DO
        do i = 1, 100
        !$OMP CRITICAL
            sum = sum + i
            call foo()
        !$OMP END CRITICAL
        enddo
    !$OMP END DO
    !$OMP END PARALLEL

    print*, sum
end program deadlock
```

Deadlock

- In OpenMP deadlock can happen if inside a critical region a function is called which contains another critical region.
- In this case the critical region of the called function will wait for the first critical region to terminate - which will never happen.
- *When a thread encounters a CRITICAL directive inside a critical region, it will block forever.*
- Two possible ways - deadlock occurs in a nested critical region

First Approach

- If two threads arrive at a nested critical construct (one critical region inside another), thread one enters the "outer" critical region and thread two waits.
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name.
- Now, thread one DOES NOT enter the nested critical region, because it is a synchronization point where threads wait for all other threads to arrive before proceeding.
- And since the second thread is waiting for the first thread to exit the "outer" critical region they are in a deadlock.

Second Approach

- Both threads arrive at the "outer" critical construct.
- Thread one enters the "outer" critical construct, thread two waits.
- Now, thread one ENTERS the "inner" critical construct and stops at it's implied barrier, because it waits for thread two.
- Thread two on the other hand waits for thread one to exit to "outer" thread and so both are waiting forever.

Deadlock - Example

```
!$OMP PARALLEL DO PRIVATE(x)
  do i=1,N
    x = SIN(2*PI*x/N)
  !$OMP CRITICAL
    sum = sum + func(x)
  !$OMP END CRITICAL
  enddo
!$OMP END PARALLEL DO
...
→ double precision function func(v)
double precision :: v
!$OMP CRITICAL
  func = v + random_func()
!$OMP END CRITICAL
end function func
```

Nested critical regions
deadlock even a *single* thread,
since they are understood to
protect the same resource

Deadlock – Example: Remedies

- Named critical regions decouple independent resources:

```
!$OMP PARALLEL DO PRIVATE (x)
  do i=1,N
    x = SIN(2*PI*x/N)
    !$OMP CRITICAL (psum)
    sum = sum + func(x)
    !$OMP END CRITICAL (psum)
  enddo
!$OMP END PARALLEL DO
...
→ double precision function func(v)
double precision :: v
✓ !$OMP CRITICAL (prand)
  func = v + random_func()
!$OMP END CRITICAL (prand)
end function func
```

Names make the critical regions protect disjoint resources
→ deadlock gone

OPENMP SCHEDULING

OpenMP: Scheduling

- OpenMP specialty is a parallelization of loops.
- The loop construct enables the parallelization.

#pragma omp parallel for

for (...)

{ ... }

- OpenMP then takes care about all the details of the parallelization.
- It creates a team of threads and distributes the iterations between the threads.
- OpenMP schedules the iterations between the threads

Scheduling: Explicit

- If the loop construct has explicit schedule clause
- OpenMP uses **scheduling-type** for scheduling the iterations of the for loop.

```
#pragma omp parallel for schedule(scheduling-type)
```

```
for (...)
```

```
{ ... }
```

Scheduling: Runtime

- If the scheduling-type is equal to runtime then OpenMP determines the scheduling by the internal control variable run-sched-var.
- We can set this variable by setting the environment variable OMP_SCHEDULE to the desired scheduling type.
- For example, in bash-like terminals:
- **\$ export OMP_SCHEDULE=scheduling-type**
- Another way to specify run-sched-var is to set it with **omp_set_schedule** function.

...

```
omp_set_schedule(scheduling-type);
```

...

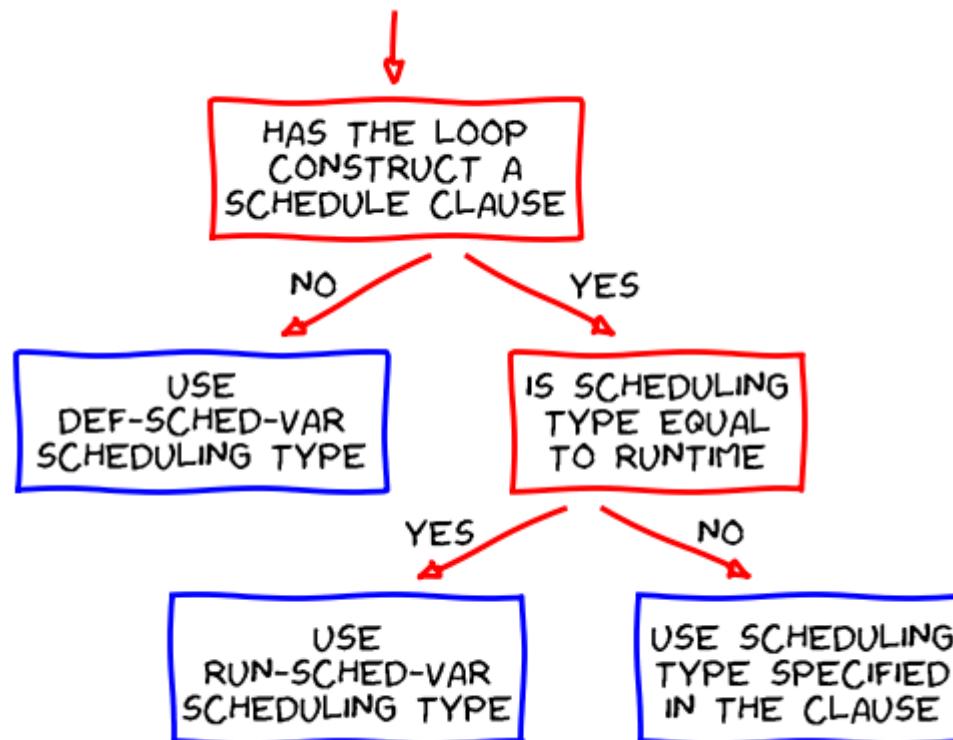
Schedule: Default

- If the loop construct does not have an explicit schedule clause then OpenMP uses the default scheduling type.
- It is defined by the internal control variable def-sched-var and it is implementation dependent.

```
#pragma omp parallel for  
for (...)  
{ ... }
```

Scheduling

- OpenMP determines a scheduling type of a for loop.



JAKASCORNER.COM

The scheduling types

- We can choose between five different scheduling types:
- static,
- dynamic,
- guided,
- auto and
- runtime.

Scheduling: Static

- The `schedule(static, chunk-size)` clause of the loop construct specifies that the for loop has the static scheduling type.
- OpenMP divides the iterations into chunks of size **chunk-size** and it distributes the chunks to threads in a circular order.
- When no **chunk-size** is specified, OpenMP divides iterations into chunks that are approximately equal in size and
- It distributes at most one chunk to each thread.

Examples of Static Scheduling

```
schedule(static):
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
schedule(static, 4):
```

```
**** **** **** ****  
**** **** **** ****  
**** **** **** ****  
**** **** **** ****
```

```
schedule(static, 8):
```

```
***** *****  
***** *****  
***** *****  
***** *****
```

Example of Static Scheduling

- We parallelized a for loop with 64 iterations and we used four threads to parallelize the for loop.
- Each row of stars in the examples represents a thread.
- Each column represents an iteration.
- The first example (`schedule(static)`) has 16 stars in the first row.
- This means that the first thread executes iterations 1, 2, 3, ..., 15 and 16.
- The second row has 16 blanks and then 16 stars.
- This means that the second thread executes iterations 17, 18, 19, ..., 31, 32. Similar applies to the threads three and four.
- We see that for `schedule(static)` OpenMP divides iterations into four chunks of size 16 and it distributes them to four threads.
- For `schedule (static, 4)` and `schedule (static, 8)` OpenMP divides iterations into chunks of size 4 and 8, respectively.
- The static scheduling type is appropriate when all iterations have the same computational cost.

Scheduling: Dynamic

- The **schedule(dynamic, chunk-size)** clause of the loop construct specifies that the for loop has the dynamic scheduling type.
- OpenMP divides the iterations into chunks of size chunk-size.
- Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.
- There is no particular order in which the chunks are distributed to the threads.
- The order changes each time when we execute the for loop.
- If we do not specify **chunk-size**, it defaults to one.

Examples of dynamic scheduling

schedule(dynamic):

```
schedule(dynamic, 1):
```

- We can see that for **schedule(dynamic)** and **schedule(dynamic, 1)** OpenMP determines similar scheduling.
 - The size of chunks is equal to one in both instances.
 - The distribution of chunks between the threads is arbitrary.

Examples of dynamic scheduling

```
schedule(dynamic, 4):
```

```
    ****          ****          ****  
****      ****      ****      ****      ****  
    ****      ****      ****      ****      ****  
    ****          ****          ****          ****
```

```
schedule(dynamic, 8):
```

```
    *****          *****          *****  
    *****          *****          *****  
*****      *****      *****  
    *****          *****          *****
```

- For **schedule(dynamic, 4)** and **schedule(dynamic, 8)** OpenMP divides iterations into chunks of size four and eight, respectively.
- The distribution of chunks to the threads has no pattern.

Scheduling: Dynamic

- The dynamic scheduling type is appropriate when the iterations require different computational costs.
- This means that the iterations are poorly balanced between each other.
- The dynamic scheduling type has higher overhead than the static scheduling type because it dynamically distributes the iterations during the runtime.

Scheduling: Guided

- The guided scheduling type is similar to the dynamic scheduling type.
- OpenMP again divides the iterations into chunks.
- Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.
- The difference with the dynamic scheduling type is in the size of chunks.
- The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads.
- Therefore the size of the chunks decreases.

Scheduling: Guided

- The minimum size of a chunk is set by **chunk-size**.
- We determine it in the scheduling clause: **schedule(guided, chunk-size)**.
- However, the chunk which contains the last iterations may have smaller size than **chunk-size**.
- If we do not specify chunk-size, it defaults to one.

Examples of the Guided Scheduling

schedule(guided):

```
schedule(guided, 2):
```


- We can see that the size of the chunks is decreasing.
 - First chunk has always 16 iterations.
 - This is because the for loop has 64 iterations and we use 4 threads to parallelize the for loop.
 - If we divide $64 / 4$, we get 16.

Examples of the Guided Scheduling

```
schedule(guided, 4):
```

```
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

```
schedule(guided, 8):
```

```
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

Scheduling: Guided

- We can also see that the minimum chunk size is determined in the schedule clause.
- The only exception is the last chunk. Its size might be lower than the prescribed minimum size.
- The guided scheduling type is appropriate when the iterations are poorly balanced between each other.
- The initial chunks are larger, because they reduce overhead.
- The smaller chunks fills the schedule towards the end of the computation and improve load balancing.
- This scheduling type is especially appropriate when poor load balancing occurs toward the end of the computation.

Scheduling: Auto

- The auto scheduling type delegates the decision of the scheduling to the compiler and/or runtime system.
- In the following example, the compiler/system determined the static scheduling.

```
schedule(auto):  
*****  
*****  
*****
```

```
*****
```

```
*****
```

Scheduling: Runtime

- The runtime scheduling type defers the decision about the scheduling until the runtime.
- We already described different ways of specifying the scheduling type in this case.
- One option is with the environment variable **OMP_SCHEDULE** and the other option is with the function **omp_set_schedule**.

Scheduling: Default

- If we do not specify the scheduling type in a for loop

```
#pragma omp parallel for  
for (...)  
{ ... }
```

- OpenMP uses the default scheduling type (defined by the internal control variable def-sched-var).
- If we do not specify the scheduling type in my machine, we get the following result.

```
default:
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

DISTRIBUTED-MEMORY PARALLEL PROGRAMMING WITH MPI

Noor Muhammad Sk

Advantages of Parallel Programming

Need to solve larger problems

- more memory intensive
- more computation
- more data intensive

Parallel programming provides

- more CPU resources
- more memory resources
- solve problems that were not possible with serial program
- solve problems more quickly

Two Basic Architectures

Distributed Memory (ex. Compute cluster)

- collection of nodes which have multiple cores
- each node uses its own local memory
- work together to solve a problem
- communicate between nodes and cores via messages
- nodes are networked together

Shared Memory Computer

- multiple cores
- share a global memory space
- cores can efficiently exchange/share data

Parallel Programming Models

Directives-based parallel programming language

- OpenMP (most widely used)
- High Performance Fortran (HPF)
- Directives tell processor how to distribute data and work across the processors
- Directives appear as comments in the serial code
- Implemented on shared memory architectures

Message Passing (MPI)

- pass messages to send/receive data between processes
- each process has its own local variables
- can be used on either shared or distributed memory architectures
- outgrowth of PVM software

Pros and Cons of OpenMP/MPI

Pros of OpenMP

- easier to program and debug than MPI
- directives can be added incrementally - gradual parallelization
- can still run the program as a serial code
- serial code statements usually don't need modification
- code is easier to understand and maybe more easily maintained

Cons of OpenMP

- can only be run in shared memory computers
- requires a compiler that supports OpenMP
- mostly used for loop parallelization

Pros and Cons of OpenMP/MPI

Pros of MPI

- Runs on either shared or distributed memory architectures
- Can be used on a wider range of problems than OpenMP
- Each process has its own local variables
- Distributed memory computers are less expensive than large shared memory computers

Cons of MPI

- Requires more programming changes to go from serial to parallel version
- Can be harder to debug
- Performance is limited by the communication network between the nodes

Parallel Programming Issues

Goal is to reduce execution time

- computation time
- idle time - waiting for data from other processors
- communication time - time the processors take to send and receive messages

Load Balancing

- divide the work equally among the available processors

Minimizing Communication

- reduce the number of messages passed
- reduce amount of data passed in messages
- Where possible - overlap communication and computation
- Many problems scale well to only a limited number of processors

MPI

- MPI, the Message Passing Interface.
- The use of explicit *message passing (MP)*, i.e., *communication between processes*
- *MPI is the* most tedious and complicated but also the most flexible parallelization method.
- MPI is a *programming model and used* on shared-memory or hybrid systems

Message passing

- Message passing is required if a parallel computer is of the distributed-memory type,
- In Distributed Memory there is no way for one processor to directly access the address space of another processor.

MPI Working Approach

- The same program runs on all processes (Single Program Multiple Data, or *SPMD*).
- *This is no restriction compared to the more general MPMD (Multiple Program Multiple Data) model as all processes taking part in a parallel calculation can be distinguished by a unique identifier called *rank* (see below).*
- The program is written in a sequential language like Fortran, C or C++.
- Data exchange, i.e., sending and receiving of messages, is done via calls to an appropriate library.
- All variables in a process are local to this process.
- There is no concept of shared memory.

Message Passing Program

- Messages carry data between processes.
- Those processes could be running on separate compute nodes, or different cores inside a node, or
- Even on the same processor core, time-sharing its resources.
- A message can be as simple as a single item (like a DP word) or
- Even a complicated structure, perhaps scattered all over the address space.
- For a message to be transmitted in an orderly manner, some parameters have to be fixed in advance.

MPI Parameters that need to be fixed are

- Which process is sending the message?
 - Where is the data on the sending process?
 - What kind of data is being sent?
 - How much data is there?
 - Which process is going to receive the message?
 - Where should the data be left on the receiving process?
 - What amount of data is the receiving process prepared to accept?
- **Note:**
- The above parameters strictly relate to point-to-point communication, where there is always exactly one sender and one receiver

MPI

- MPI supports much more than just sending a single message between two processes;
- There is a similar set of parameters for those more complex cases as well.
- MPI is a very broad standard with a huge number of library routines.
- Fortunately, most applications merely require less than a dozen of those.

“Hello World” MPI program in C

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(int argc, char** argv) {
5     int rank, size;
6
7     MPI_Init(&argc, &argv);
8     MPI_Comm_size(MPI_COMM_WORLD, &size);
9     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11    printf("Hello World, I am %d of %d\n", rank, size);
12
13    MPI_Finalize();
14    return 0;
15 }
```

MPI

- **`MPI_Init()`** - This initializes the parallel environment
- **`MPI_COMM_WORLD`** – MPI World Communicator
- A communicator defines a group of MPI processes that can be referred to by a communicator *handle*.
- *The `MPI_COMM_WORLD` handle describes all processes that have been started as part of the parallel program.*
- If required, other communicators can be defined as subsets of `MPI_COMM_WORLD`.
- Nearly all MPI calls require a communicator as an argument.

MPI

- The calls to **`MPI_Comm_size()`** and **`MPI_Comm_rank()`** is to determine the number of processes (size) in the parallel program and the unique identifier (rank) of the calling process, respectively.
- Note that the C bindings require output arguments (like rank and size above) to be specified as pointers.
- The ranks in a communicator, in this case **`MPI_COMM_WORLD`**, are consecutive, starting from zero.
- The parallel program is shut down by a call to **`MPI_Finalize()`**.
- Note that no MPI process except rank 0 is guaranteed to execute any code beyond **`MPI_Finalize()`**.

MPI Compilation

- `mpirun -np 4 ./hello.exe`
- This would compile the code and start it with four processes.
- Be aware that processors may have to be allocated from some resource management (batch) system before parallel programs can be launched.
- How exactly MPI processes are started is entirely up to the implementation.
- Ideally, the start mechanism uses the resource manager's infrastructure (e.g., daemons running on all nodes) to launch processes.
- The same is true for process-to-core affinity;
- if the MPI implementation provides no direct facilities for affinity control.

MPI Program Output

```
1 Hello World, I am 3 of 4
2 Hello World, I am 0 of 4
3 Hello World, I am 2 of 4
4 Hello World, I am 1 of 4
```

Messages and point-to-point communication

- The “Hello World” example did not contain any real communication apart from starting and stopping processes.
- An MPI message is defined as an array of elements of a particular MPI data type.
- Data types can either be basic types or *derived types*, which must be defined by appropriate MPI calls.
- The reason why MPI needs to know the data types of messages is that it supports heterogeneous environments where it may be necessary to do on-the-fly data conversions.
- For any message transfer to proceed, the data types on sender and receiver sides must match.

point-to-point communication

- If there is exactly one sender and one receiver we speak of *point-to-point communication*.
- Both ends are identified uniquely by their ranks.
- Each point-to-point message can carry an additional integer label, the so-called *tag*, which may be used to identify the type of a message, and which must match on both ends.
- It may carry any accompanying information, or just be set to some constant value if it is not needed.

Details of Message Passing

- For a Communication to Succeed
- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver's buffer must be large enough

MPI C Data Types

MPI Data type	C Data Type
MPI_CHAR	Signed Char
MPI_SHORT	Signed Short Int
MPI_INT	Signed Int
MPI_LONG	Signed Long Int
MPI_UNSIGNED_CHAR	Unsigned Char
MPI_UNSIGNED_SHORT	Unsigned Short Int
MPI_UNSIGNED_INT	Unsigned Int
MPI_UNSIGNED_LONG	Unsigned Long Int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long Double
MPI_BYTE	
MPI_PACKED	

MPI Point-to-Point Communication Modes

Send Modes	MPI function	Completion Condition
Synchronous send	MPI_Ssend	only completes when the receive has completed
Buffered send	MPI_Bsend	always completes (unless an error occurs) irrespective of the receiver
**Standard send	MPI_Send	message sent (receive state unknown)
Ready send	MPI_Rsend	may be used only when the a matching receive has already been posted

Receive Mode	MPI function	Completion Condition
Receive	MPI_Recv	Complete when a message has arrived

MPI_Send()

- The basic MPI function to send a message from one process to another is **MPI_Send()**
-

```
1 <type> buf(*)
2 integer :: count, datatype, dest, tag, comm, ierror
3 call MPI_Send(buf,           ! message buffer
4                  count,        ! # of items
5                  datatype,    ! MPI data type
6                  dest,         ! destination rank
7                  tag,          ! message tag (additional label)
8                  comm,         ! communicator
9                  ierror)      ! return value
```

MPI_Recv()

- A message may be received with the **MPI_Recv()** function:
-

```
1 <type> buf(*)
2 integer :: count, datatype, source, tag, comm,
3 integer :: status(MPI_STATUS_SIZE), ierror
4 call MPI_Recv(buf,           ! message buffer
5                 count,        ! maximum # of items
6                 datatype,    ! MPI data type
7                 source,       ! source rank
8                 tag,         ! message tag (additional label)
9                 comm,         ! communicator
10                status,      ! status object (MPI_Status* in C)
11                ierror)     ! return value
```

- Compared with MPI_Send(), this function has an additional output argument, the status object.
- After MPI_Recv() has returned, the status object can be used to determine parameters that have not been fixed by the call's arguments.
- Primarily, this pertains to the length of the message, because the count parameter is only a maximum value at the receiver side;
- The message may be shorter than count elements.

Status

- The MPI_Get_count() function can retrieve the real number:

```
1 integer :: status(MPI_STATUS_SIZE), datatype, count, ierror
2 call MPI_Get_count(status,          ! status object from MPI_Recv()
3                      datatype,    ! MPI data type received
4                      count,       ! count (output argument)
5                      ierror)      ! return value
```

- However, the status object also serves another purpose.
- The source and tag arguments of MPI_Recv() may be equipped with the special constants (“wildcards”) MPI_ANY_SOURCE and MPI_ANY_TAG, respectively.
- MPI_ANY_SOURCE specifies that the message may be sent by anyone
- MPI_ANY_TAG determines that the message tag should not matter.
- After MPI_Recv() has returned, status(MPI_SOURCE) and status (MPI_TAG) contain the sender’s rank and the message tag, respectively.
- In C, the status object is of type struct MPI_Status, and access to source and tag information works via the “.” operator.

Simple program can be improved in several ways:

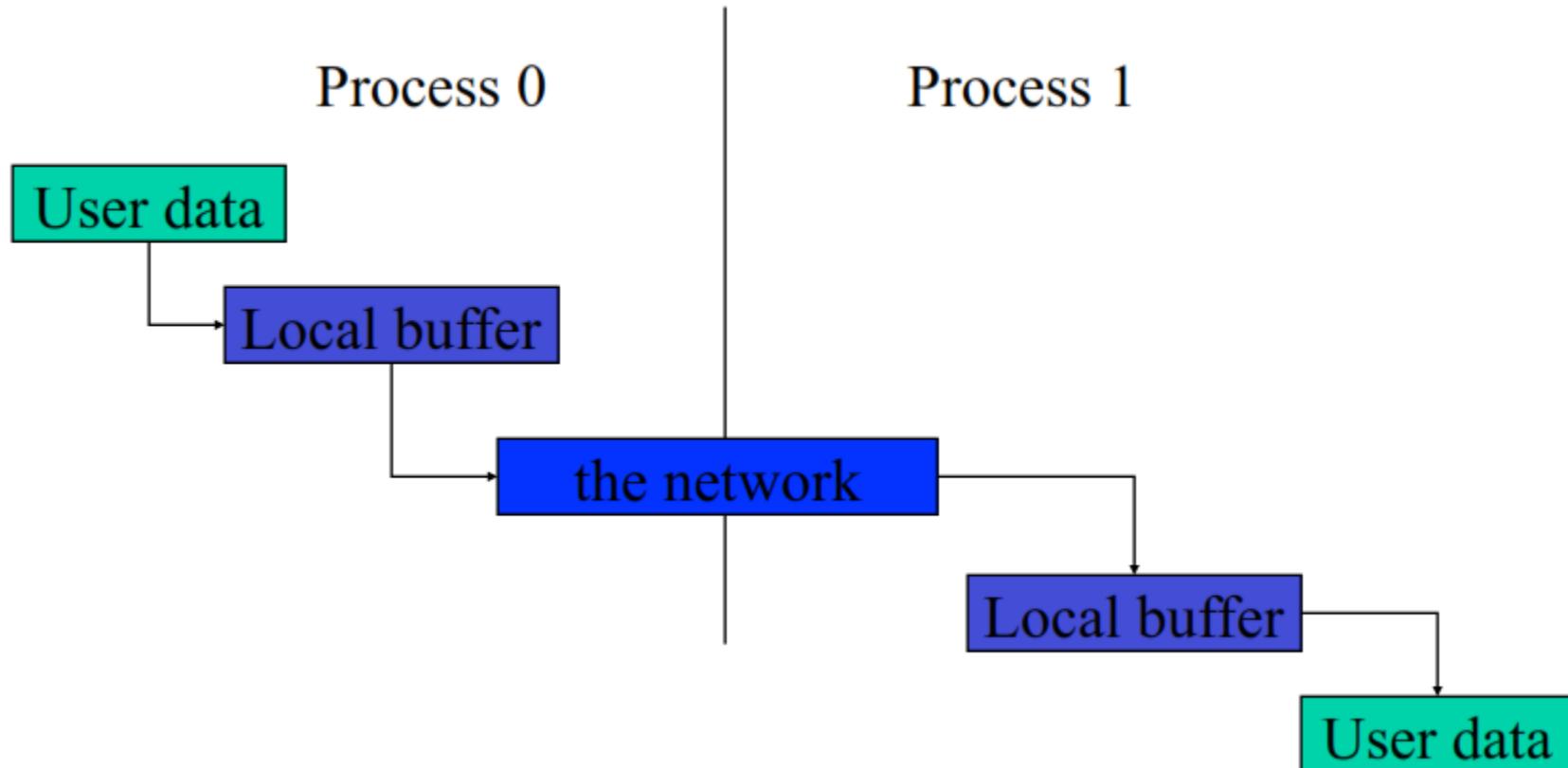
- MPI does not preserve the temporal order of messages unless they are transmitted between the same sender/receiver pair (and with the same tag).
- To allow the reception of partial results at rank 0 without delay due to different execution times of the integrate() function
- It may be better to use the MPI_ANY_SOURCE wildcard instead of a definite source rank in line 23.
- Rank 0 does not call MPI_Recv() before returning from its own execution of integrate().
- If other processes finish their tasks earlier, communication cannot proceed, and it cannot be overlapped with computation.
- The MPI standard provides *nonblocking point-to-point communication facilities* that allow multiple outstanding receives (and sends), and even let implementations support asynchronous messages.

Simple program can be improved in several ways:

- Since the final result is needed at rank 0, this process is necessarily a communication bottleneck if the number of messages gets large.
- We will demonstrate optimizations that can significantly reduce communication overhead in those situations.
- Fortunately, nobody is required to write explicit code for this.
- In fact, the global sum is an example for a *reduction operation* and is well supported within MPI.
- Vendor implementations are assumed to provide optimized versions of such global operations.

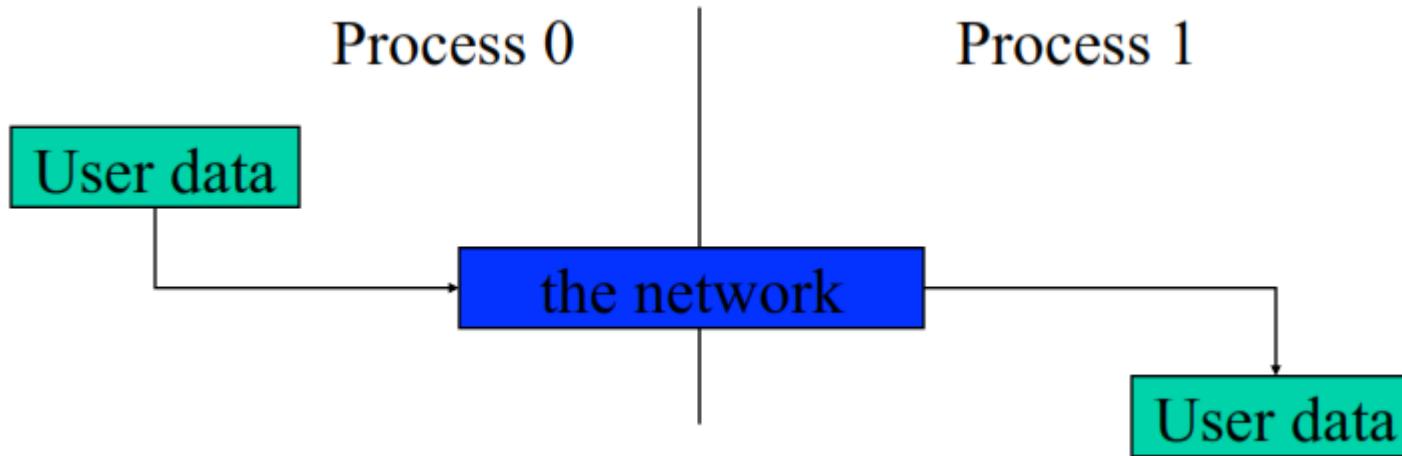
Buffers

- When you send data, where does it go? One possibility is:



Avoiding Buffering

- It is better to avoid copies:



- This requires that **MPI_Send** wait on delivery, or that **MPI_Send** return before transfer is complete, and we wait later.

Sources of Deadlocks

- Send a large message from process 0 to process 1
- If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
Send (1)	Send (0)
Recv (1)	Recv (0)

- This is called “unsafe” because it depends on the availability of system buffers in which to store the data sent until it can be received

Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

Process 0	Process 1
Send(1)	Recv(0)
Recv(1)	Send(0)

- Supply receive buffer at same time as send:

Process 0	Process 1
Sendrecv(1)	Sendrecv(0)

More Solutions to the “unsafe” Problem

- Supply own space as buffer for send

Process 0	Process 1
Bsend(1)	Bsend(0)
Recv(1)	Recv(0)

- Use non-blocking operations:

Process 0	Process 1
Isend(1)	Isend(0)
Irecv(1)	Irecv(0)
Waitall	Waitall

Communication Modes

- MPI provides multiple *modes for sending messages*:
- **Synchronous mode (MPI_Ssend)**: the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
- **Buffered mode (MPI_Bsend)**: the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.)
- **Ready mode (MPI_Rsend)**: user guarantees that a matching receive has been posted.
 - Allows access to fast protocols
 - undefined behavior if matching receive not posted
- Non-blocking versions (MPI_Issend, etc.)
- MPI_Recv receives messages sent in any mode.

Buffered Mode

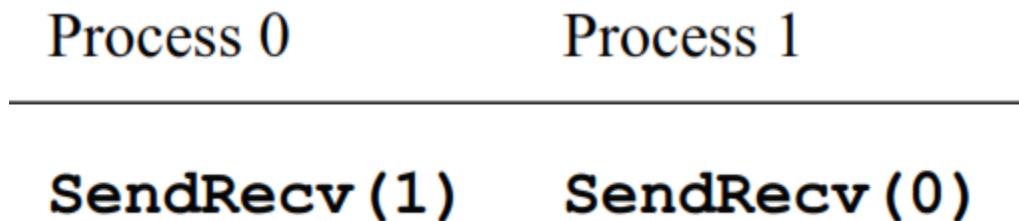
- When MPI_Isend is awkward to use (e.g. lots of small messages), the user can provide a buffer for the system to store messages that cannot immediately be sent.
- int bufsize;
- char *buf = malloc(bufsize);
- MPI_Buffer_attach(buf, bufsize);
- ...
- MPI_Bsend(... same as MPI_Send ...)
- ...
- MPI_Buffer_detach(&buf, &bufsize);
- MPI_Buffer_detach waits for completion.
- Performance depends on MPI implementation and size of message.

Other Point-to Point Features

- **MPI_Sendrecv**
- **MPI_Sendrecv_replace**
- **MPI_Cancel(request)**
 - Cancel posted Isend or Irecv
- Persistent requests
 - Useful for repeated communication patterns
 - Some systems can exploit to reduce latency and increase performance
 - **MPI_Send_init(..., &request)**
 - **MPI_Start(request)**

MPI_Sendrecv

- Allows simultaneous send and receive
- Everything else is general.
 - Send and receive datatypes (even type signatures) may be different
 - Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, ...)
 - More general than “send left”



Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of
 - if (has down nbr)
 Call MPI_Send(... down ...)
 - if (has up nbr)
 Call MPI_Recv(... up ...)
 - ...
- sequentializes (all except the bottom process blocks)

Sequentialization

Start	Start	Start	Start	Start	Start	Send	Recv
Send	Send	Send	Send	Send	Send		
						Send	Recv
						Send	Recv
						Send	Recv
						Send	Recv
						Send	Recv
						Send	Recv
Send	Recv						

Deadlock or Race Conditions

- Deadlock or race conditions occur when the message passing cannot be completed.
- Consider the following and assume that the MPI_Send does not complete until the corresponding MPI_Recv is posted and visa versa.
- The MPI_Send commands will never be completed and the program will deadlock.

```
if (rank == 0) {  
    MPI_Send(..., 1, tag, MPI_COMM_WORLD);  
    MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);  
} else if (rank == 1) {  
    MPI_Send(..., 0, tag, MPI_COMM_WORLD);  
    MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);  
}
```

Avoid Deadlock

- There are a couple of ways to fix this problem.
- One way is to reverse the order of one of the send/receive pairs:

```
if (rank == 0) {  
    MPI_Send(..., 1, tag, MPI_COMM_WORLD);  
    MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);  
} else if (rank == 1) {  
    MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);  
    MPI_Send(..., 0, tag, MPI_COMM_WORLD);  
}
```

Avoiding MPI Deadlock or Race Conditions

- Another way is to make the send be a non-blocking one (MPI_Isend)

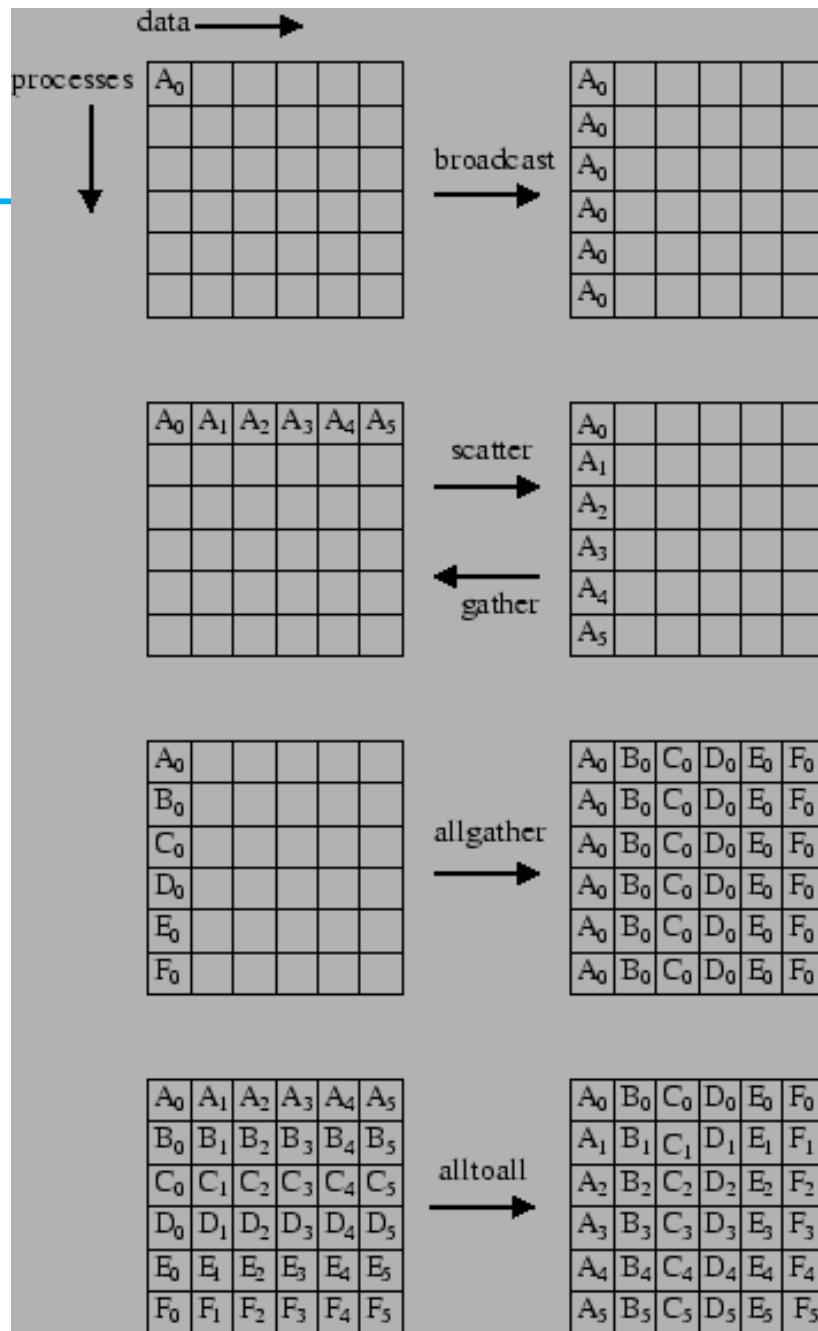
```
if (rank == 0) {
    MPI_Isend(..., 1, tag, MPI_COMM_WORLD, &req);
    MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
    MPI_Wait(&req, &status);
} else if (rank == 1) {
    MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
    MPI_Send(..., 0, tag, MPI_COMM_WORLD);
}
```

MPI Error Handling

- MPI provides two predefined error handlers
 - **MPI_ERRORS_ARE_FATAL** (the default): causes MPI to abort
 - **MPI_ERRORS_RETURN**: causes MPI to return an error values instead of aborting
- MPI Error Handling Functions
 - **MPI_Errhandler_set**: set the error handler
 - **MPI_Error_class**: convert an error code into an error class
 - **MPI_Error_string**: returns a string for a given error code

MPI Collective Communication Routines

MPI_Function	Function Description
MPI_Bcast	Broadcast a message from one process to all others
MPI_Barrier	blocks until all processes have reached this routine
MPI_Reduce	Reduce values from all processes to a single value (add,mult, min, max, etc.)
MPI_Gather	Gathers together values from a group of processes
MPI_Scatter	Sends data from process to the other processes in a group
MPI_Allgather	Gathers data from all tasks and distributes it to all
MPI_Allreduce	Reduces values from all processes and distributes the result back to all processes

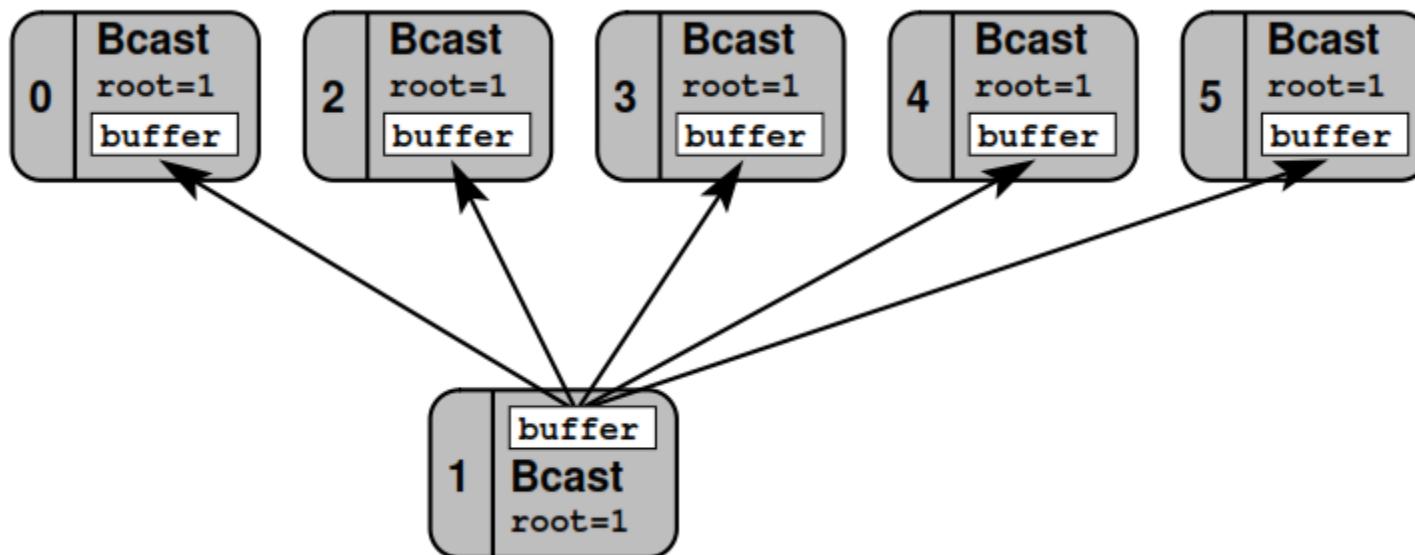


Broadcast

```
1 <type> buf(*)
2 integer :: count, datatype, root, comm, ierror
3 call MPI_Bcast(buffer,           ! send/receive buffer
                 count,          ! message length
                 datatype,        ! MPI data type
                 root,           ! rank of root process
                 comm,            ! communicator
                 ierror)         ! return value
```

MPI Broadcast

- The “root” process (rank 1 in this example) sends the same message to all others.
- Every rank in the communicator must call `MPI_Bcast()` with the same root argument.



Broadcast

- The buffer argument to `MPI_Bcast()` is a send buffer on the root and a receive buffer on any other process.
- Every process in the communicator must call the routine.
- A broadcast is needed whenever one rank has information that it must share with all others;
- E.g., there may be one process that performs some initialization phase after the program has started, like reading parameter files or command line options.
- This data can then be communicated to everyone else via `MPI_Bcast()`.

Advanced Collective Calls

- Advanced Collective Calls are used for global data distribution
- MPI_Gather() collects the send buffer contents of all processes and concatenates them in rank order into the receive buffer of the root (Rank 0) process.
- MPI_Scatter() does the reverse, distributing equal-sized chunks of the root's send buffer.
- Both exist in variants (with a “v” appended to their names) that support arbitrary per-rank chunk sizes.
- MPI_Allgather() is a combination of MPI_Gather() and MPI_Bcast().

MPI_Reduce()

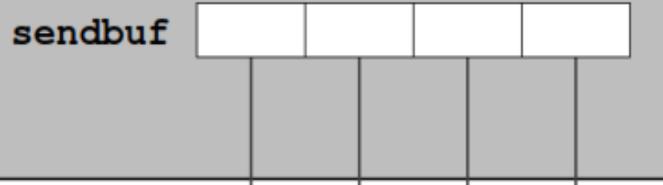
- MPI_Reduce() combines the contents of the sendbuf array on all processes, element-wise, using an operator encoded by the op argument, and stores the result in recvbuf on root.
- There are twelve predefined operators, the most important being MPI_MAX, MPI_MIN, MPI_SUM and MPI_PROD, which implement the global maximum, minimum, sum, and product, respectively.
- User-defined operators are also supported.

```
1 <type> sendbuf(*), recvbuf(*)
2 integer :: count, datatype, op, root, comm, ierror
3 call MPI_Reduce(sendbuf,      ! send buffer
4                   recvbuf,      ! receive buffer
5                   count,       ! number of elements
6                   datatype,    ! MPI data type
7                   op,          ! MPI reduction operator
8                   root,        ! root rank
9                   comm,        ! communicator
10                  ierror)     ! return value
```

MPI Reduce

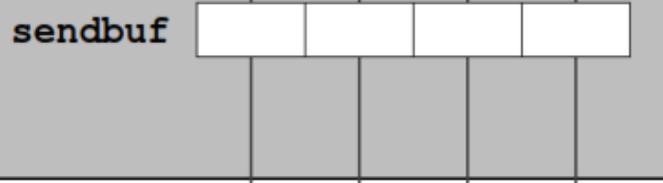
3

Reduce
root=1
count=4
op=MPI_SUM



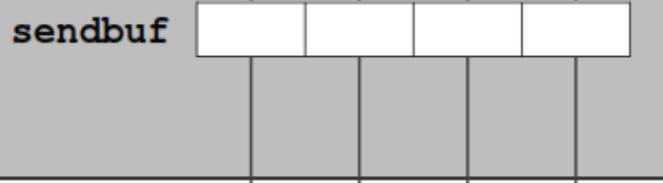
2

Reduce
root=1
count=4
op=MPI_SUM



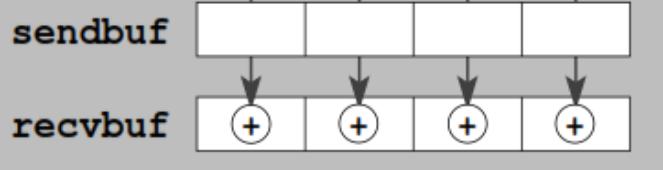
0

Reduce
root=1
count=4
op=MPI_SUM



1

Reduce
root=1
count=4
op=MPI_SUM



- A reduction on an array of length count (a sum in this example) is performed by MPI_Reduce().
- Every process must provide a send buffer.
- The receive buffer argument is only used on the root process.
- The local copy on root can be prevented by specifying MPI_IN_PLACE instead of a send buffer address.

MPI_Allreduce()

- MPI_Allreduce() is a fusion of a reduction with a broadcast, and MPI_Reduce_scatter()
- MPI_Reduce_scatter() combines MPI_Reduce() with MPI_Scatter()
- Collective communication are prone to deadlock hazards as blocking point-to-point communication.
- This means, e.g., that collectives must be executed by all processes in the same order.

Note Key Point

- It is a good idea to prefer collectives over point-to-point constructs or
- Combinations of simpler collectives that “emulate” the same semantics.
- Good MPI implementations are optimized for data flow on collective communication and
- Should also have some knowledge about network topology built in.

MPI Blocking Communication

- Received data has arrived completely and sent data has left the buffer so that it can be safely modified without inadvertently changing the message.
- Collective communication in MPI is always blocking in the current MPI standard
- Point-to-point communication can be performed with *nonblocking semantics as well*.

MPI nonblocking Communication

- A nonblocking point-to-point call merely initiates a message transmission and returns very quickly to the user code.
- In an efficient implementation, waiting for data to arrive and the actual data transfer occur in the background, leaving resources free for computation. Synchronization is ruled out.
- Nonblocking MPI is a way in which communication may be overlapped with computation if implemented efficiently.
- The message buffer must not be used as long as the user program has not been notified that it is safe to do so (which can be checked by suitable MPI calls).
- Nonblocking and blocking MPI calls are mutually compatible, i.e., a message sent via a blocking send can be matched by a nonblocking receive.

nonblocking send is MPI_Isend()

- As opposed to the blocking send.
- MPI_Isend() has an additional output argument, the *request handle*.
- *It serves as an identifier by which the program can later refer to the “pending” communication request (in C, it is of type struct MPI_Request).*

```
1 <type> buf(*)
2 integer :: count, datatype, dest, tag, comm, request, ierror
3 call MPI_Isend(buf,           ! message buffer
4                  count,        ! # of items
5                  datatype,    ! MPI data type
6                  dest,         ! destination rank
7                  tag,          ! message tag
8                  comm,         ! communicator
9                  request,      ! request handle (MPI_Request* in C)
10                 ierror)     ! return value
```

MPI_Irecv() initiates a nonblocking Receive

```
1 <type> buf(*)
2 integer :: count, datatype, source, tag, comm, request, ierror
3 call MPI_Irecv(buf,           ! message buffer
4                  count,        ! # of items
5                  datatype,    ! MPI data type
6                  source,       ! source rank
7                  tag,         ! message tag
8                  comm,         ! communicator
9                  request,     ! request handle
10                 ierror)      ! return value
```

- The status object known from MPI_Recv() is missing here, because it is not needed
- No actual communication has taken place when the call returns to the user code.

Checking Pending Communication

- Checking a pending communication for completion can be done via the MPI_Test() and MPI_Wait() functions.
- MPI_Test() only tests for completion and returns a flag.
- MPI_Wait() blocks until the buffer can be used:

```
1 logical :: flag
2 integer :: request, status(MPI_STATUS_SIZE), ierror
3 call MPI_Test(request,          ! pending request handle
               flag,            ! true if request complete (int* in C)
               status,          ! status object
               ierror)          ! return value
4
5 call MPI_Wait(request,          ! pending request handle
                 status,        ! status object
                 ierror)          ! return value
6
7
```

The status object contains useful information only if the pending communication is a completed receive (i.e., in the case of MPI_Test() the value of flag must be true).

MPI's communication modes

	Point-to-point	Collective
Blocking	<code>MPI_Send()</code> <code>MPI_Ssend()</code> <code>MPI_Bsend()</code> <code>MPI_Recv()</code>	<code>MPI_Barrier()</code> <code>MPI_Bcast()</code> <code>MPI_Scatter() /</code> <code>MPI_Gather()</code> <code>MPI_Reduce()</code> <code>MPI_Reduce_scatter()</code> <code>MPI_Allreduce()</code>
Nonblocking	<code>MPI_Isend()</code> <code>MPI_Irecv()</code> <code>MPI_Wait() / MPI_Test()</code> <code>MPI_Waitany() /</code> <code>MPI_Testany()</code> <code>MPI_Waitsome() /</code> <code>MPI_Testsome()</code> <code>MPI_Waitall() /</code> <code>MPI_Testall()</code>	N/A

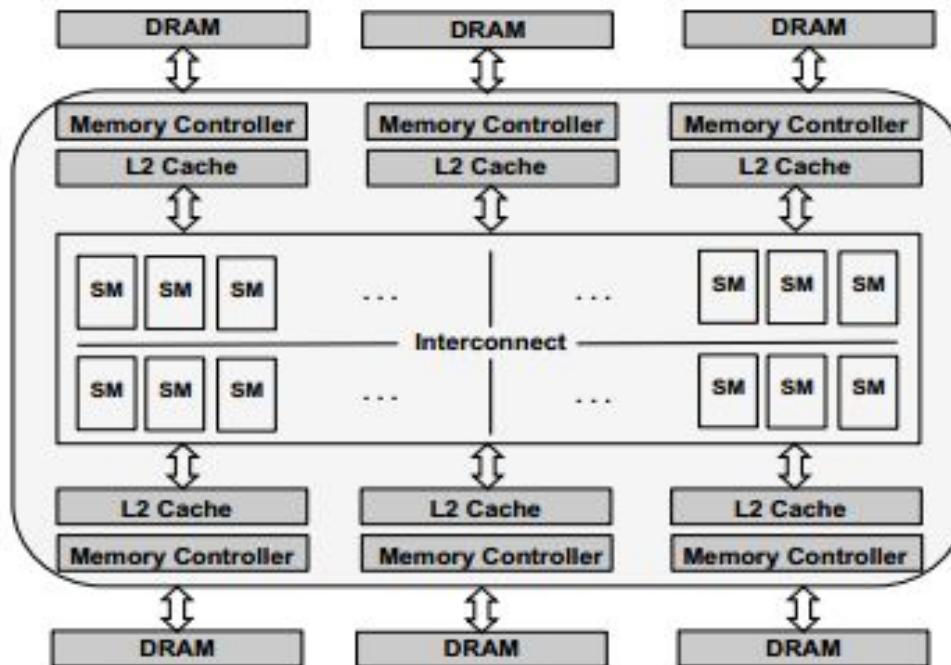
PROGRAMMING GPGPU ARCHITECTURE

Dr Noor Muhammad Sk

Goals

- Data Parallelism: What is it, and how to exploit it?
 - Workload characteristics
- Execution Models/ GPU Architectures
 - MIMD (SPMD), SIMD, SIMT
- GPU Programming Models
 - Terminology translations: CPU ↪ AMD GPU ↪ Nvidia GPU
 - Intro to OpenCL
- Modern GPU Microarchitectures
 - i.e., programmable GPU pipelines, not their fixed-function predecessors

Architecture of GPU

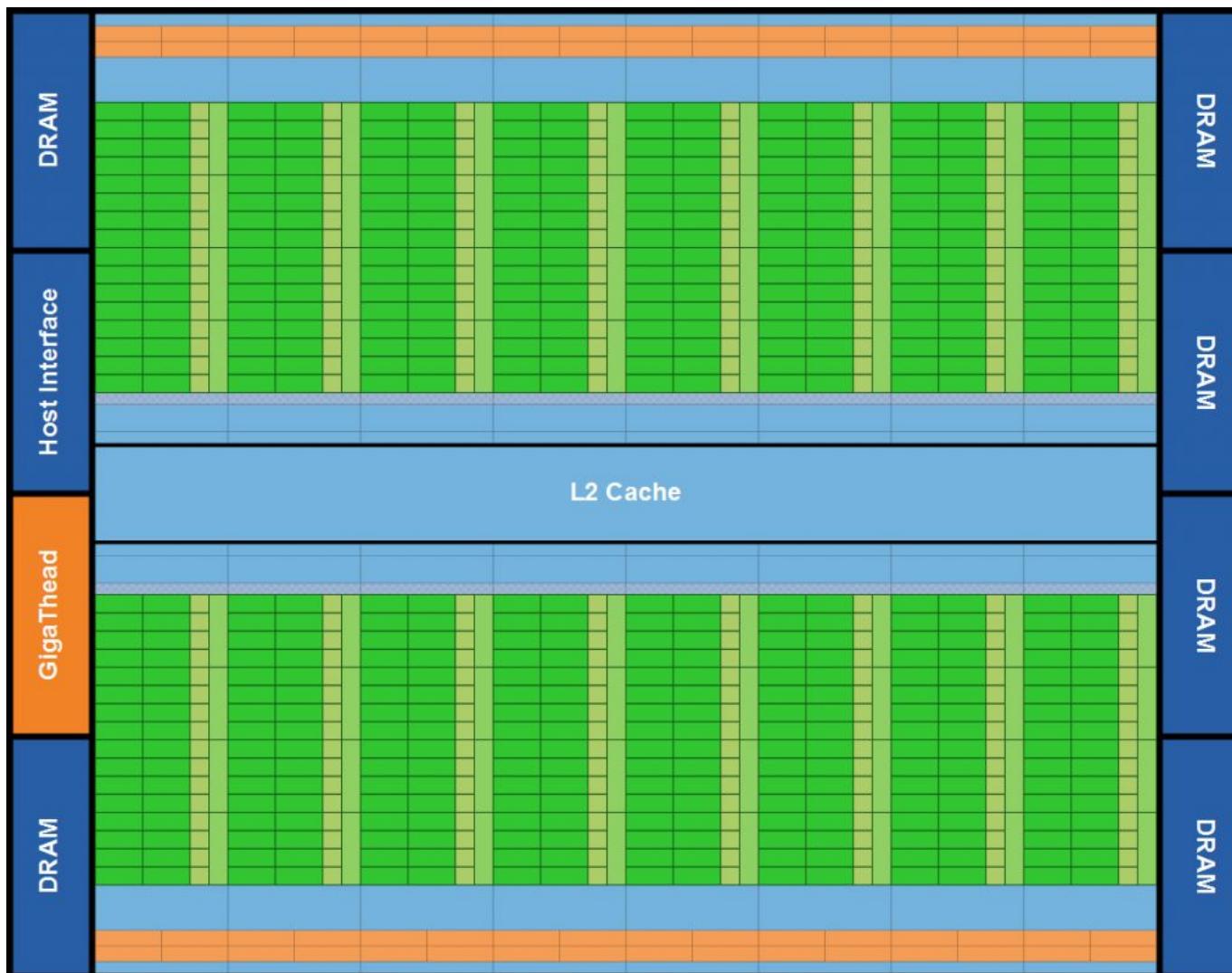


- Many streaming multiprocessors (SMs)
- Each SM typically having a SIMT width of 8 to 32
- Each SM is associated with
 - A private L1 Data Cache and
 - Read-only texture and
 - Constant caches
 - Along with a low latency shared memory (scratchpad memory)
- Every MC is associated with a slice of the shared L2 cache for faster access to the cached data.
- Both MC and L2 are on-chip.

Fermi based GPU

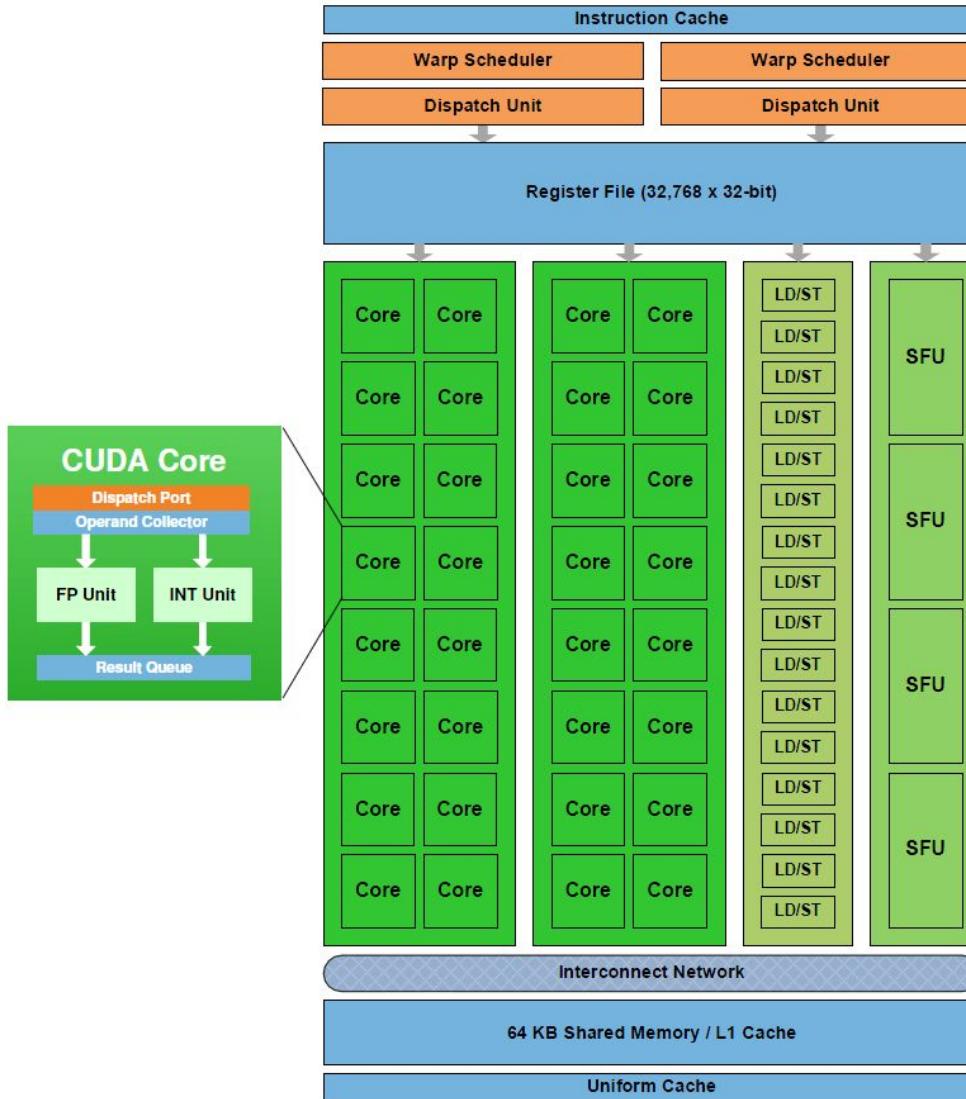
- Fermi based GPU has 512 CUDA cores.
- A CUDA core executes a floating point or integer instruction per clock for a thread.
- The 512 cores are organized in 16 Streaming Multiprocessors (SM) of 32 cores each.
- The SMs are supported by a L2, host interface, GigaThread scheduler and multiple DRAM interfaces.
- The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6GB of GDDR5 DRAM memory.
- A host interface connects the GPU to the CPU via PCI-Express.
- The GigaThread global scheduler distributes thread blocks to SM thread schedulers.

Fermi based GPU



- Fermi's 16 SM are positioned around a common L2 Cache.
- Each SM is a vertical rectangular strip that contain an orange portion (Scheduler and dispatch)
- A green portion (execution units), and light blue portions (Register file and L1 Cache)

Architecture of Fermi Streaming Multiprocessor

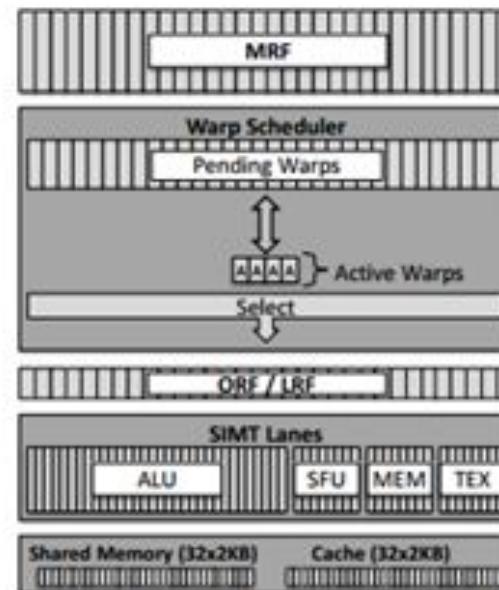
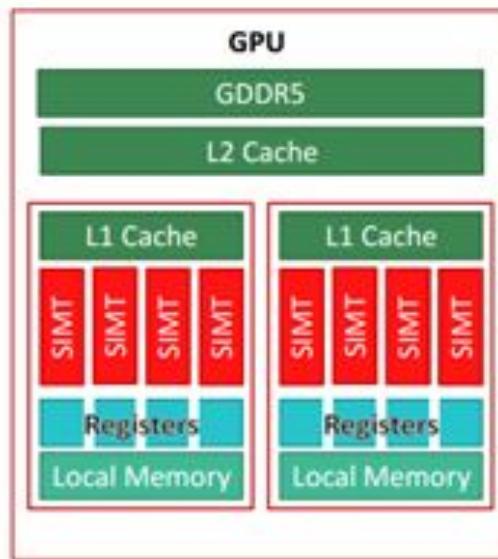


- For Fermi, a SM can have at most 8 Thread Blocks, 48 Warps (32 Threads/Warp) and 1536 Threads.

32 Streaming Processor:

- Each core has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).
- SP has no independent units of Fetch, Decode or Dispatch, and thus it can only receive the instructions issued by SM.
- Given that one SP has one ALU, thus we can use ALU to denote one SP or one Lane.
- Right figure shows an SM containing 32 SIMT lanes that each executes up to one thread instruction per cycle.
- The SM has up to 1024 resident threads, and a 32-entry, single issue, in-order warp scheduler selects one warp per cycle to issue an instruction.

Stream Multiprocessor



(b) Streaming multiprocessor

-
- ***16 Load/Store Units:***
 - Each SM has 16 l/s units, allowing source and destination addresses to be calculated for 16 threads per clock.
 - Supporting units load and store the data at each address to cache or DRAM.
 - ***Four Special Function Units:***
 - Each SFU executes one instruction per thread, per clock;
 - A warp executes over eight clocks.
 - The SFU pipeline is decoupled from the dispatch unit, allowing the dispatch unit to issue to other execution units while the SFU is occupied.

GigaThread Thread Scheduler

- One of the most important technologies of Fermi architecture is its two-level, distributed thread scheduler.
- At the chip level, a global work distribution engine schedules thread blocks to various SMs, while at the SM level, each warp scheduler distributes warps of 32 threads to its execution units.
- The first generation GigaThread engine introduced in G80 managed up to 12,288 threads in realtime.

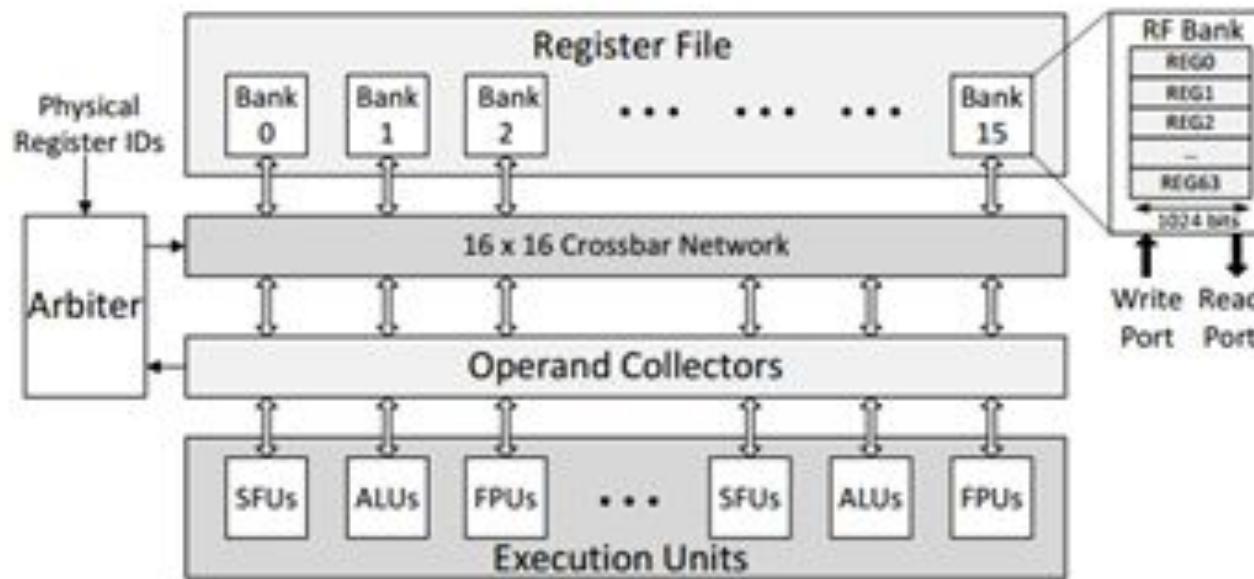
Dual Warp Scheduler

- Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently.
- Fermi's dual warp scheduler selects two warps, and issues one instruction from each warp to a group of sixteen cores, sixteen I/s units, or four SFUs.
- Because warps execute independently, Fermi's scheduler does not need to check for dependencies from within the instruction stream.

32768 Registers

- Each SM has a large and unified register file that is shared across warps executing in the same SM.
- Register size is namely 128KB, and 21 registers/thread, 32768/1536, at full scheduler occupancy.
- The total register file capacity across the chip is 2MB, substantially exceeding the size of the L2 cache.
- GPUs adopt a multibanked register file architecture to avoid the area and power overhead of a multiported register file.
- Crossbar networks and operand collectors are used to transport operands from different banks to the appropriate SIMD execution lane.
- The register file can be modeled as 16 dual-ported (one read and one write port) banks, each of which provides 2048 logical 32-bit registers.
- Physically, these registers are accessed as 64, 1024-bit registers.
- Thus, each operand read from a register file bank provides operands for 32 SIMT threads.
- A crossbar interconnection network is used to transfer operands from register file banks to operand collectors.
- Furthermore, in the case of bank conflicts, the arbiter is responsible for serializing the register file access, and the operand collectors are used to buffer the data already read from the register file.
- Each thread complete owns certain registers, and releases only when the thread block finishes.

Register File structure in Fermi Architecture



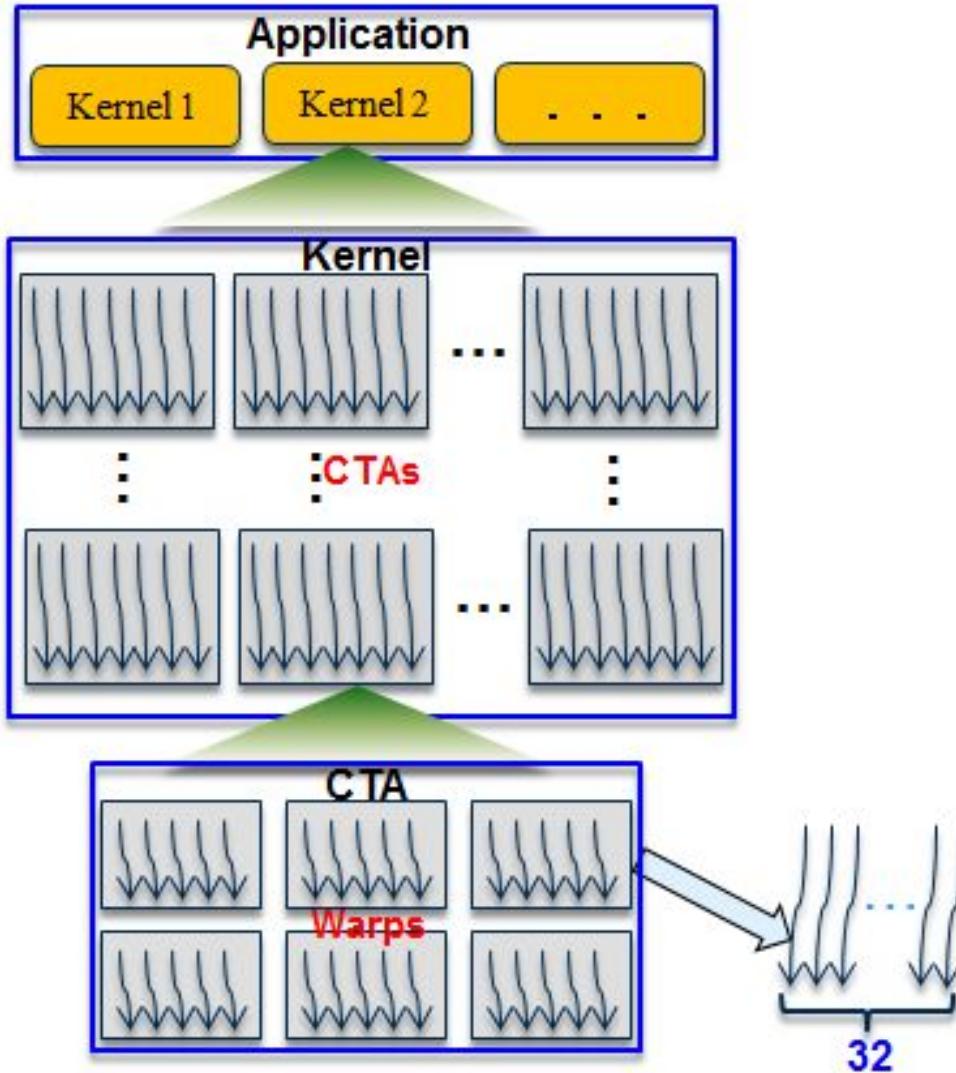
Register file structure in the Fermi architecture.

64 KB Configurable Shared Memory and L1 Cache

- on-chip shared memory enable threads within the same thread block to cooperate, facilitates extensive reuse of on-chip data, and greatly reduces off-chip traffic.
- On one hand, the shared memory is used by multiple thread blocks;
- on the other hand, each thread block exclusively uses part of the shared memory, which is shared by all thread of the block.

CUDA Hierarchy of Threads, Blocks and Grids

- A GPU executes one or more kernel grids;
 - An SM executes one or more thread blocks; and
 - CUDA cores and other execution units in the SM execute threads.
-
- A program may consist of one or more kernels, each consisting of one or more co-operative thread arrays (CTAs), and each CTA consists of multiple warps.

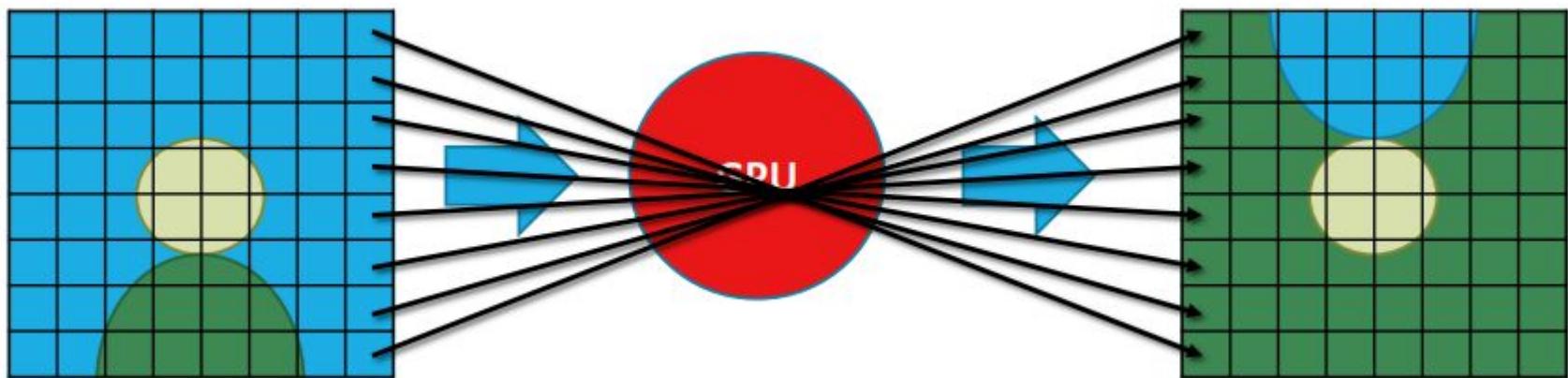


Thread, Thread Block, Grid

- ***Thread:***
- Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results.
- ***Thread Block (also called Cooperative Thread Array, CTA):***
- A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory.
- A thread block has a block ID within its grid. (For Fermi, Max Thread/Block is 1024)
- ***Grid (Kernel):***
- A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls.

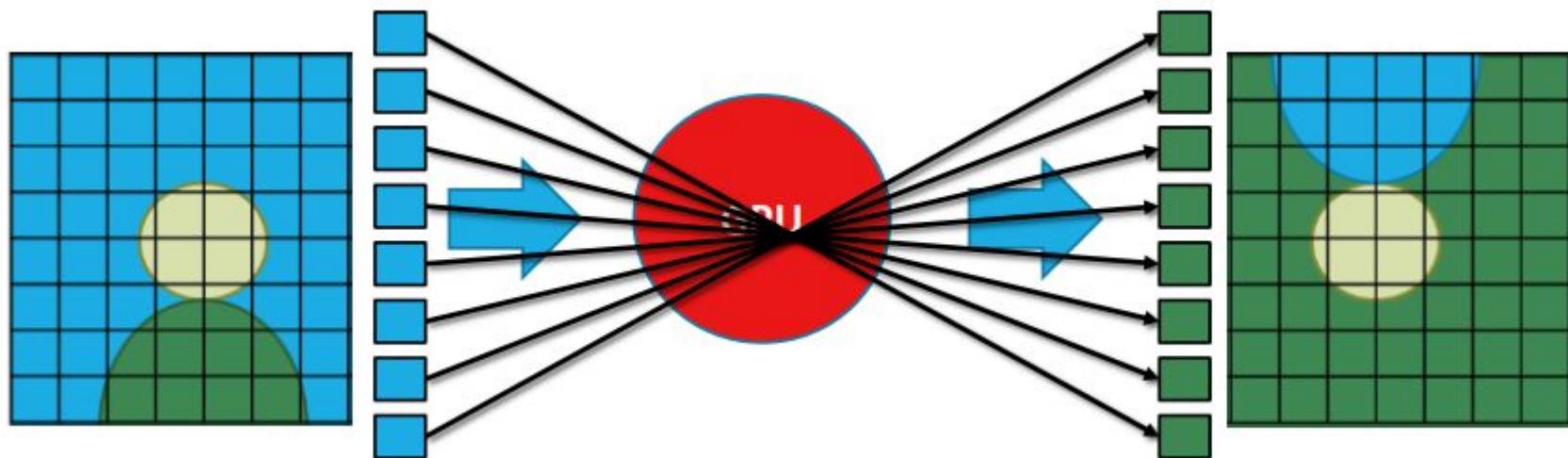
Graphics Workloads

- *Identical, Streaming computation on pixels*



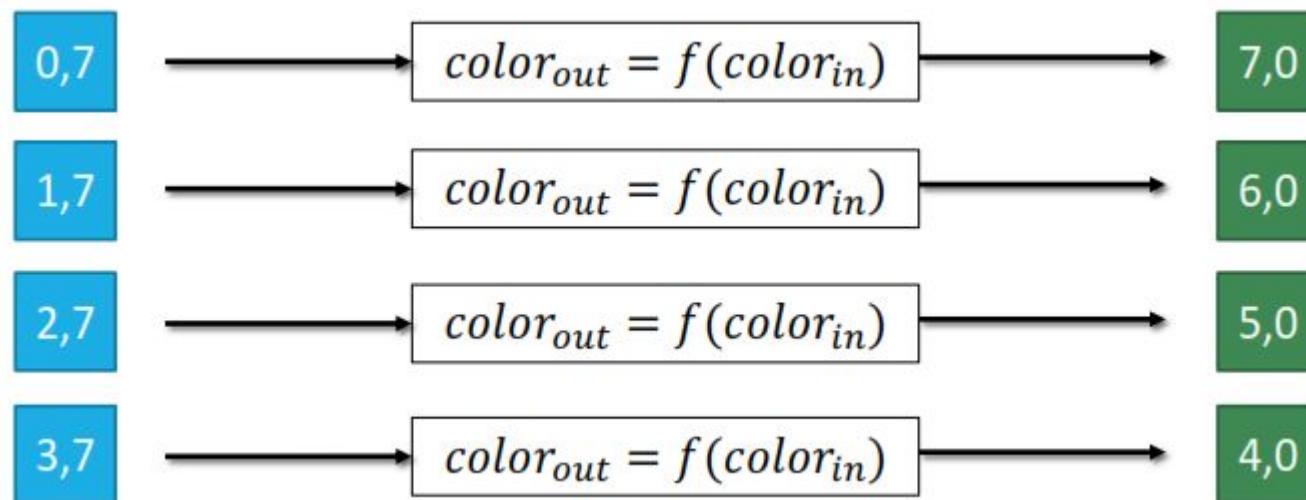
Graphics Workloads

- *Identical, Independent, Streaming computation on pixels*



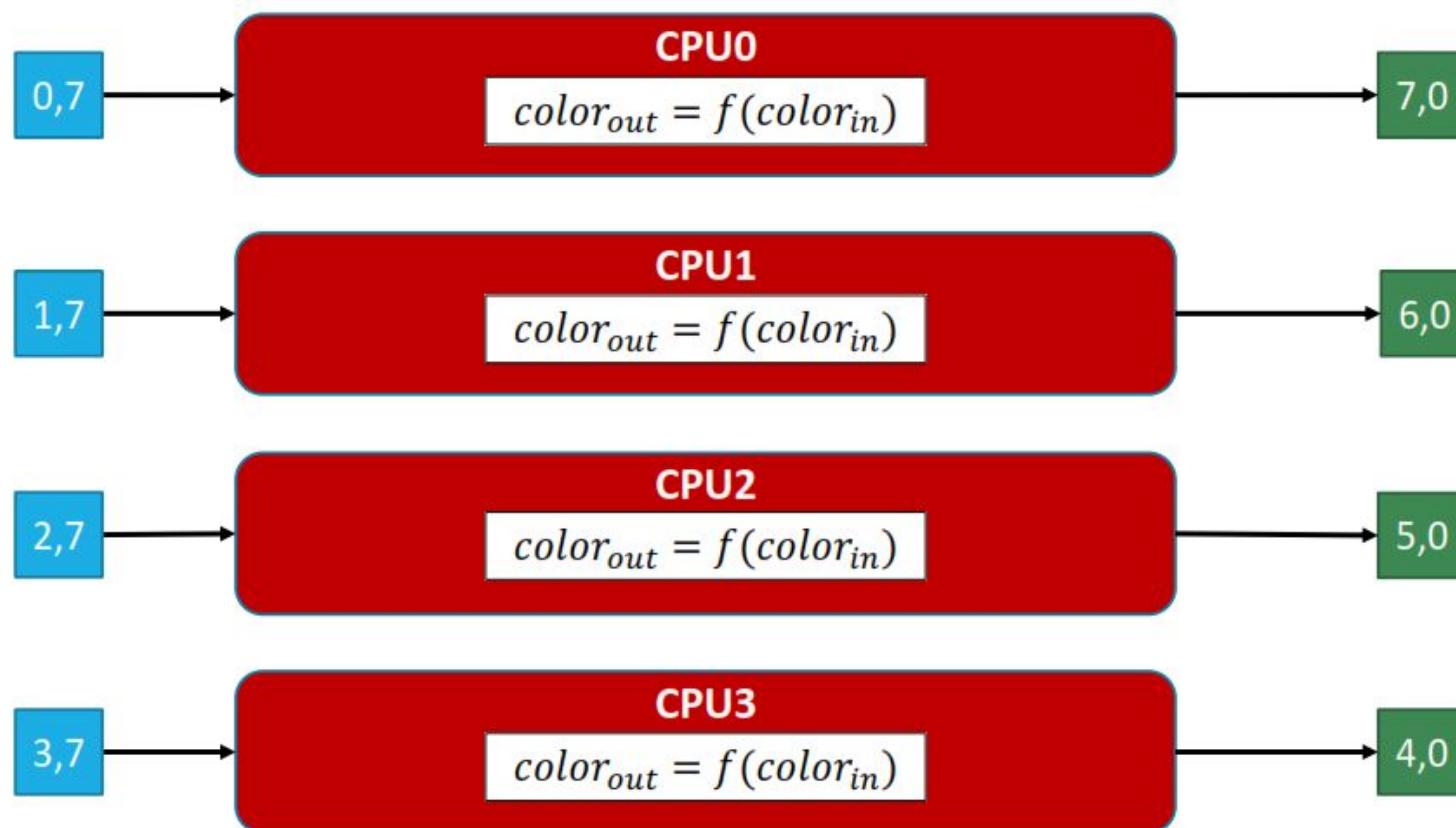
Generalize: Data Parallel Workloads

- *Identical, Independent computation on multiple data inputs*



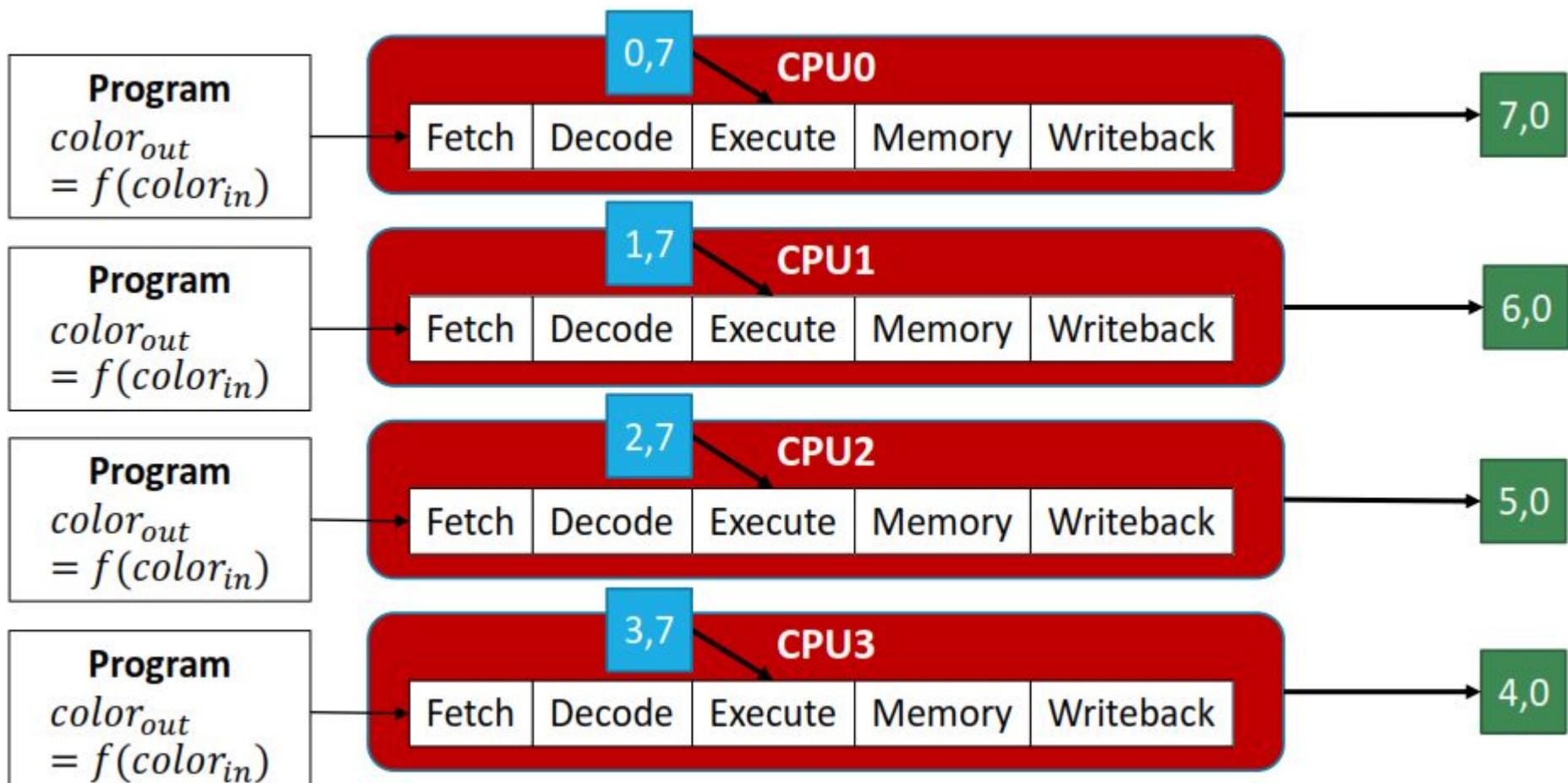
Naïve Approach

- Split **independent** work over **multiple** processors



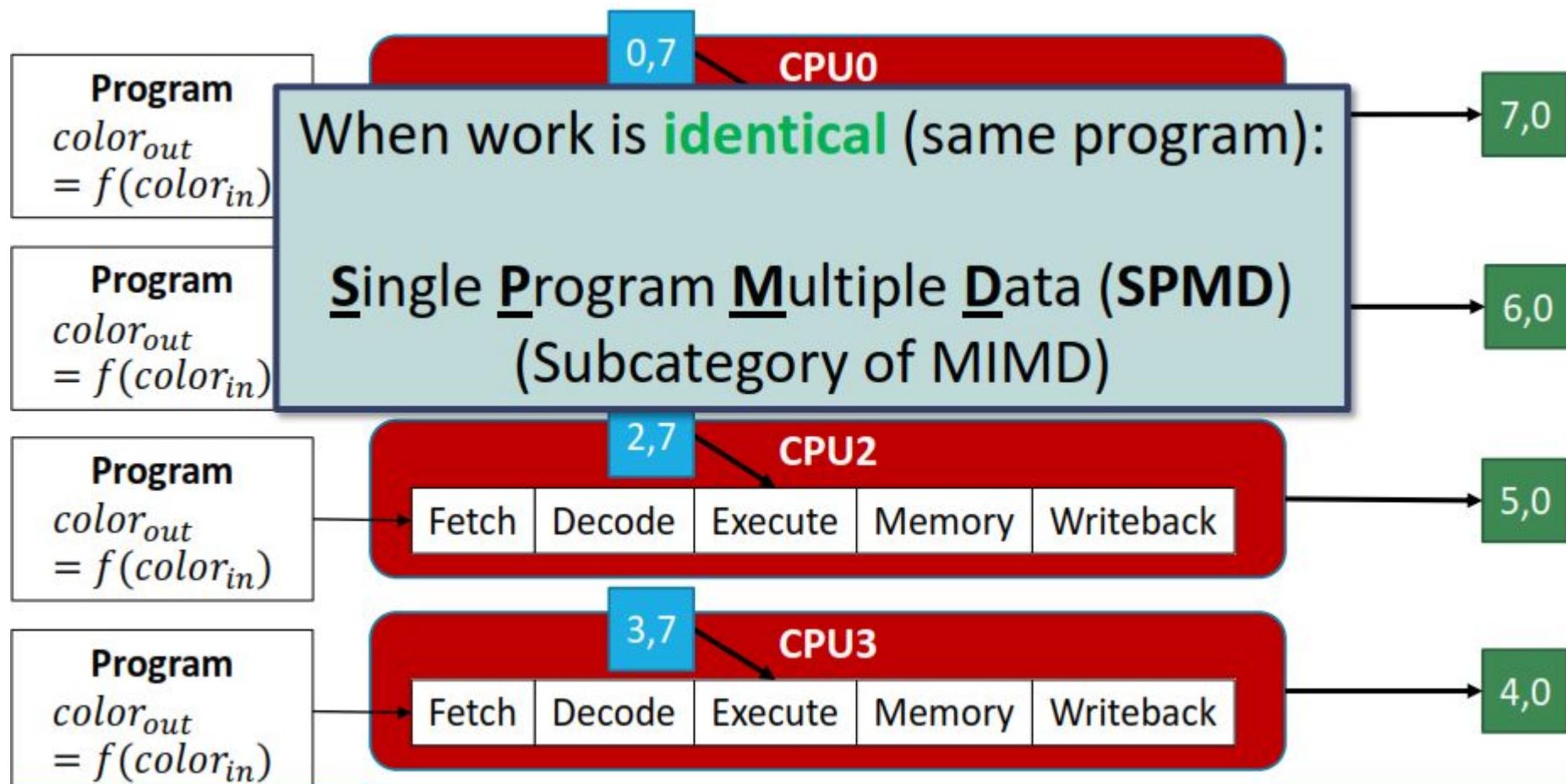
Data Parallelism: A MIMD Approach

- Multiple Instruction Multiple Data
- Split **independent** work over **multiple processors**



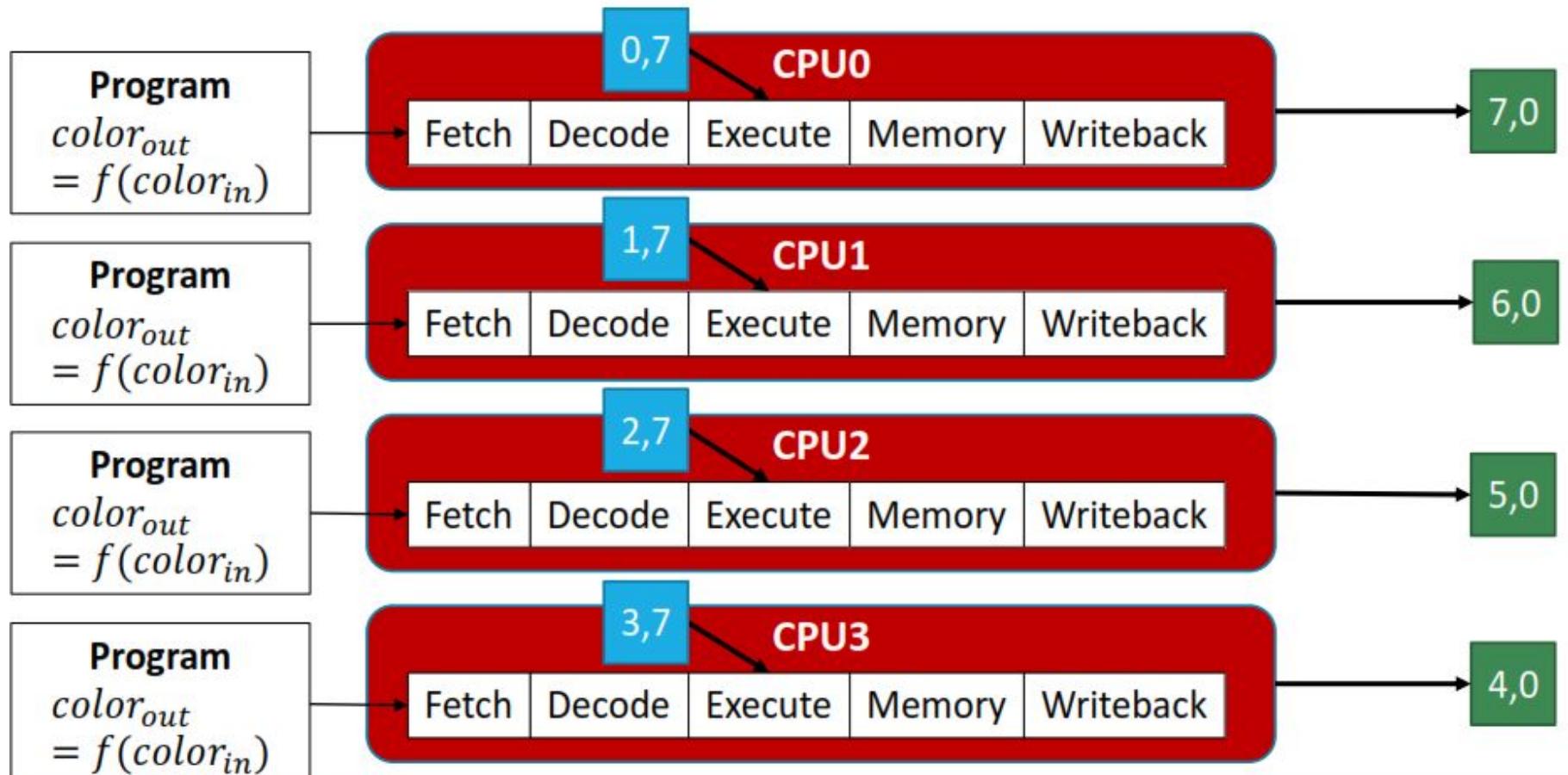
Data Parallelism: A MIMD Approach

- Multiple Instruction Multiple Data
- Split independent work over multiple processors



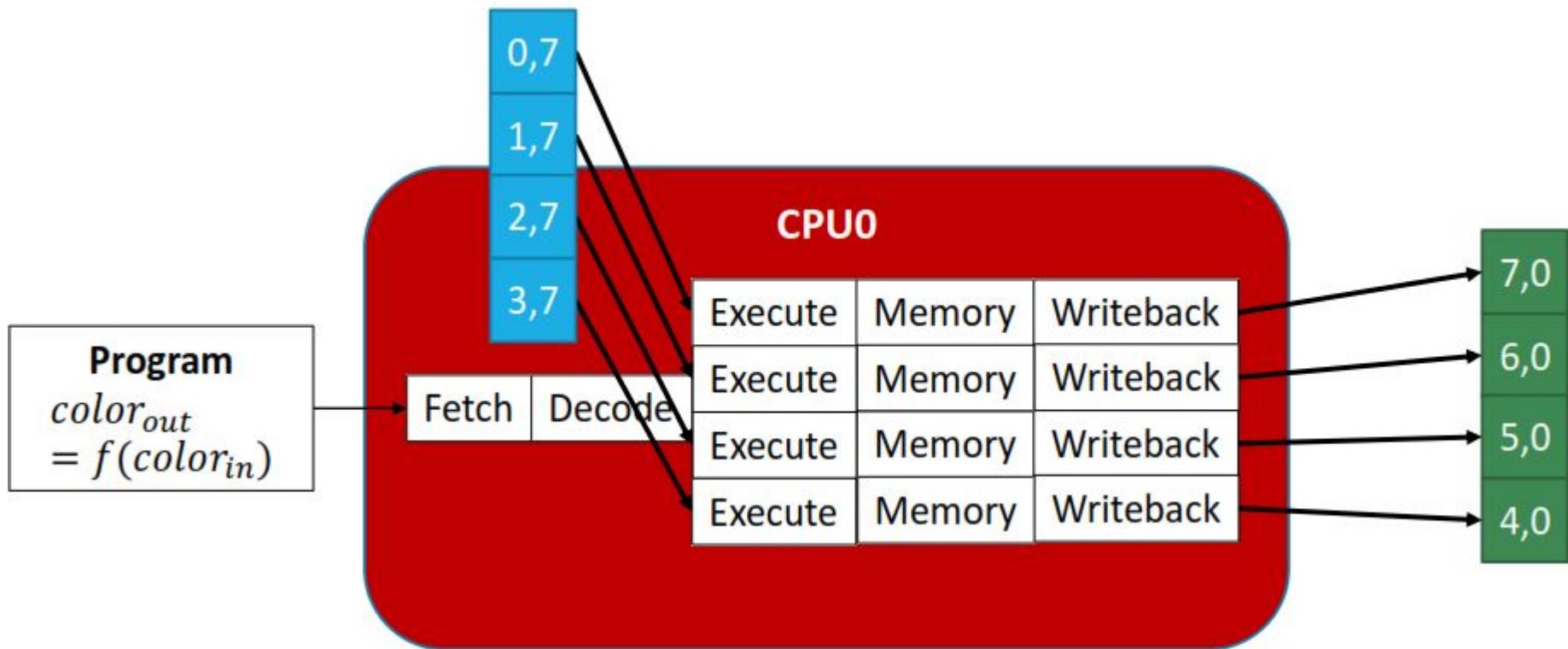
Data Parallelism: An SPMD Approach

- Single Program Multiple Data
- Split identical, independent work over multiple processors



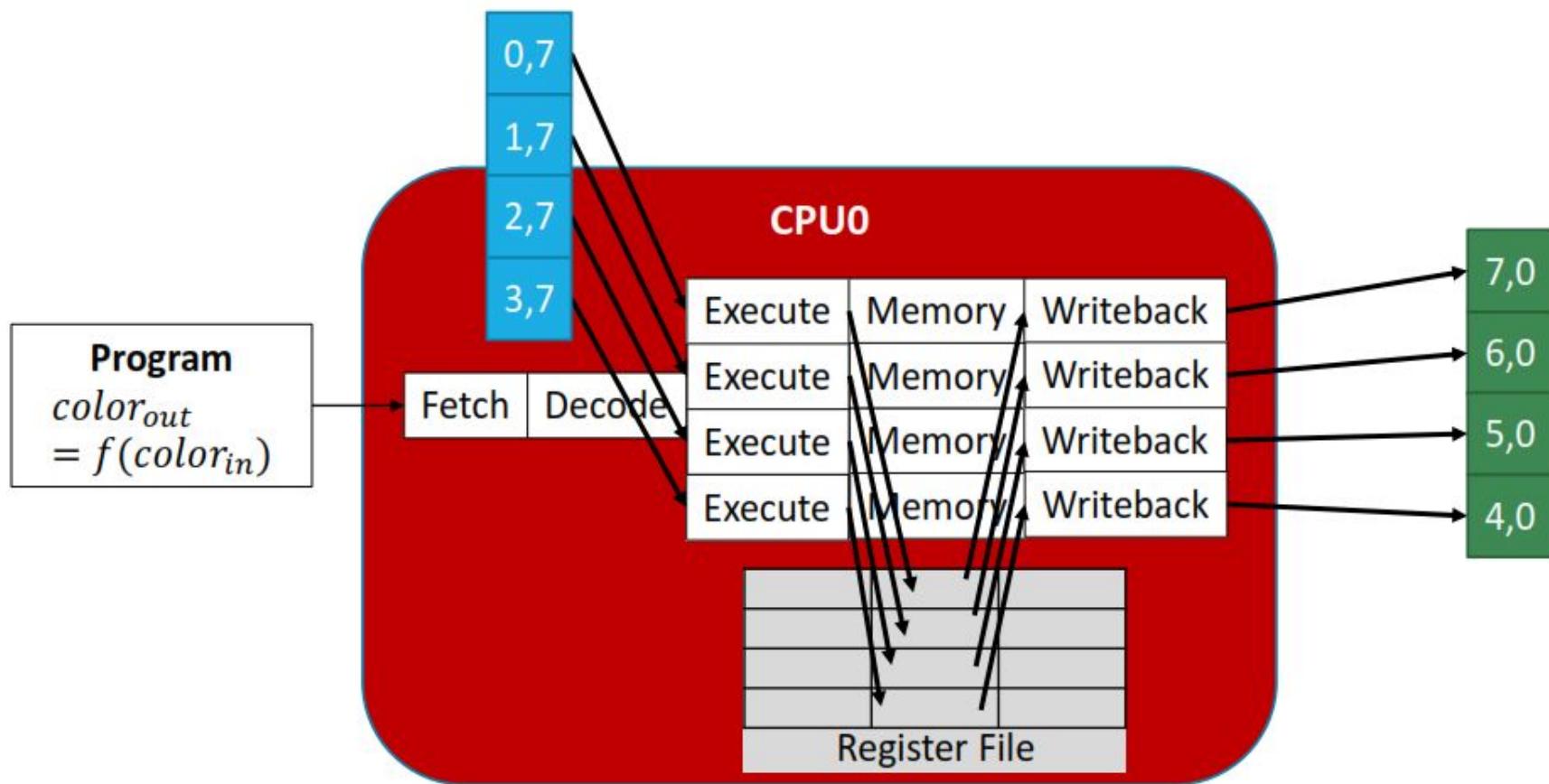
Data Parallelism: A SIMD Approach

- Single Instruction Multiple Data
- Split identical, independent work over multiple execution units (lanes)
- More efficient: Eliminate redundant fetch/decode



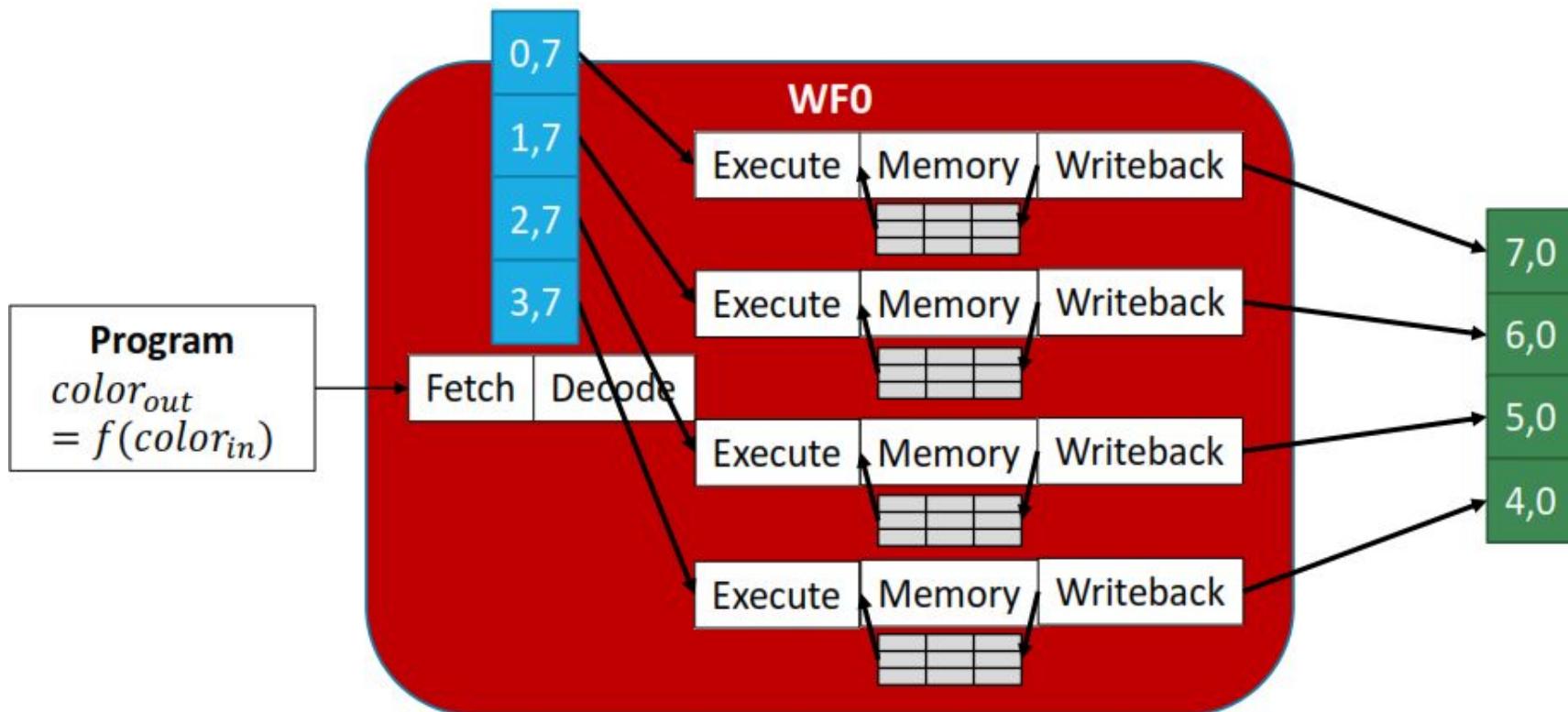
SIMD: A Closer Look

- One Thread + Data Parallel Ops ↳ Single PC, single register file



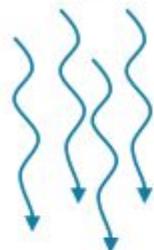
Data Parallelism: A SIMT Approach

- Single Instruction Multiple Thread
- Split identical, independent work over multiple *lockstep threads*
- Multiple Threads + Scalar Ops ↳ One PC, Multiple register files



Data Parallel Execution Models

MIMD/SPMD



Multiple **independent** threads

SIMD/Vector



One thread with wide execution datapath

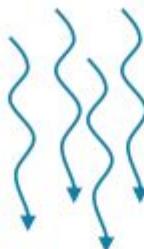
SIMT



Multiple **lockstep** threads

Execution Model Comparison

MIMD/SPMD



SIMD/Vector



SIMT



Example Architecture

Multicore CPUs

x86 SSE/AVX

GPUs

Pros

More general:
supports TLP

Can mix sequential
& parallel code

Easier to program
Gather/Scatter
operations

Cons

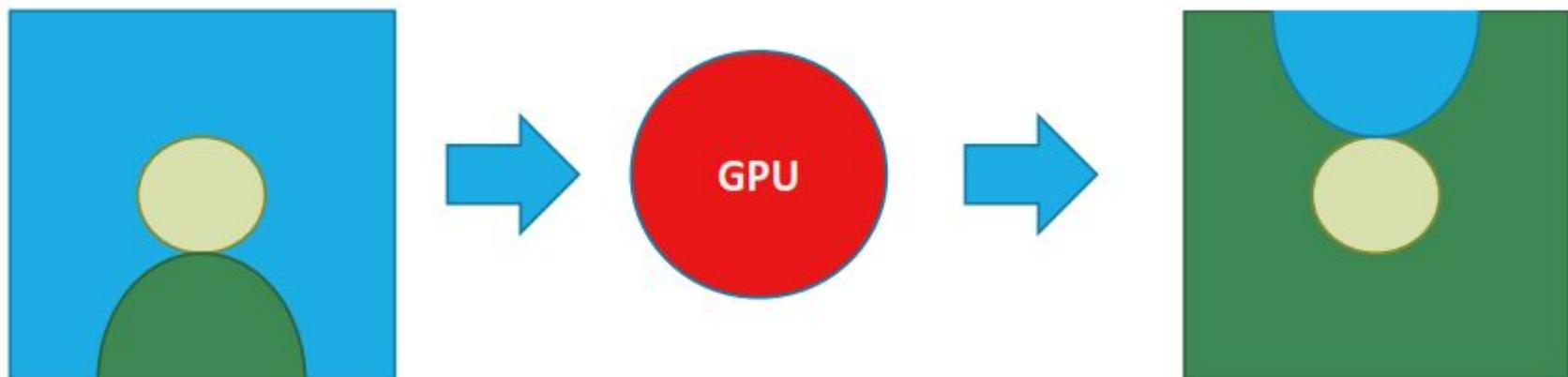
Inefficient for data
parallelism

Gather/Scatter can
be awkward

Divergence kills
performance

GPUs and Memory

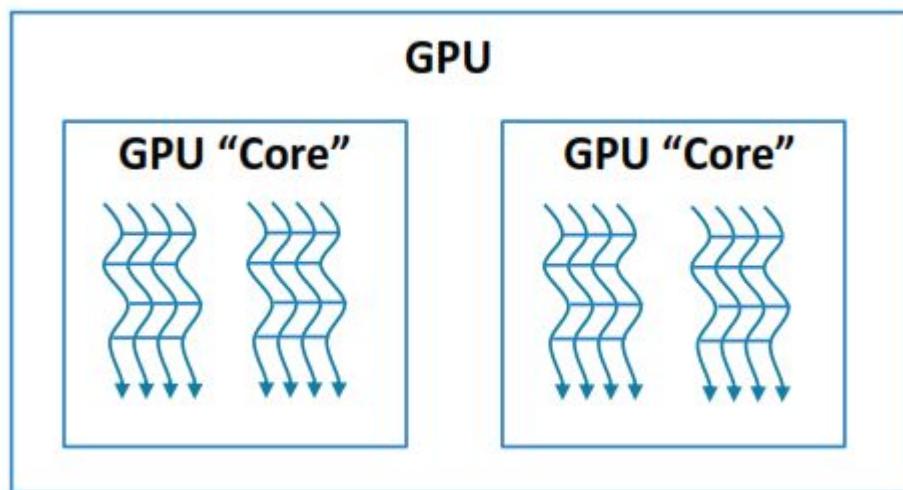
- Recall: GPUs perform *Streaming computation* ?
Streaming memory access



DRAM latency: 100s of GPU cycles
How do we keep the GPU busy (*hide memory latency*)?

Multicore Multithreaded SIMT

- Many SIMT “threads” grouped together into GPU “Core”
- SIMT threads in a group == SMT threads in a CPU core
 - Unlike CPU, groups are exposed to programmers
- Multiple GPU “Cores”



Terminology

Nvidia/CUDA	AMD/OpenCL	Derek's CPU Analogy
CUDA Processor	Processing Element	Lane
CUDA Core	SIMD Unit	Pipeline
Streaming Multiprocessor	Compute Unit	Core
GPU Device	GPU Device	Device

GPU "Core"

The diagram shows a box labeled "GPU 'Core'" containing two separate groups of four wavy arrows pointing downwards, representing multiple CUDA cores.

GPU

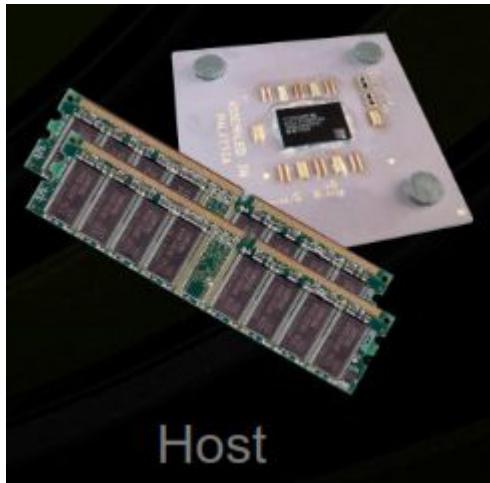
The diagram shows a box labeled "GPU" containing two groups of four wavy arrows pointing downwards, representing multiple streaming multiprocessors.

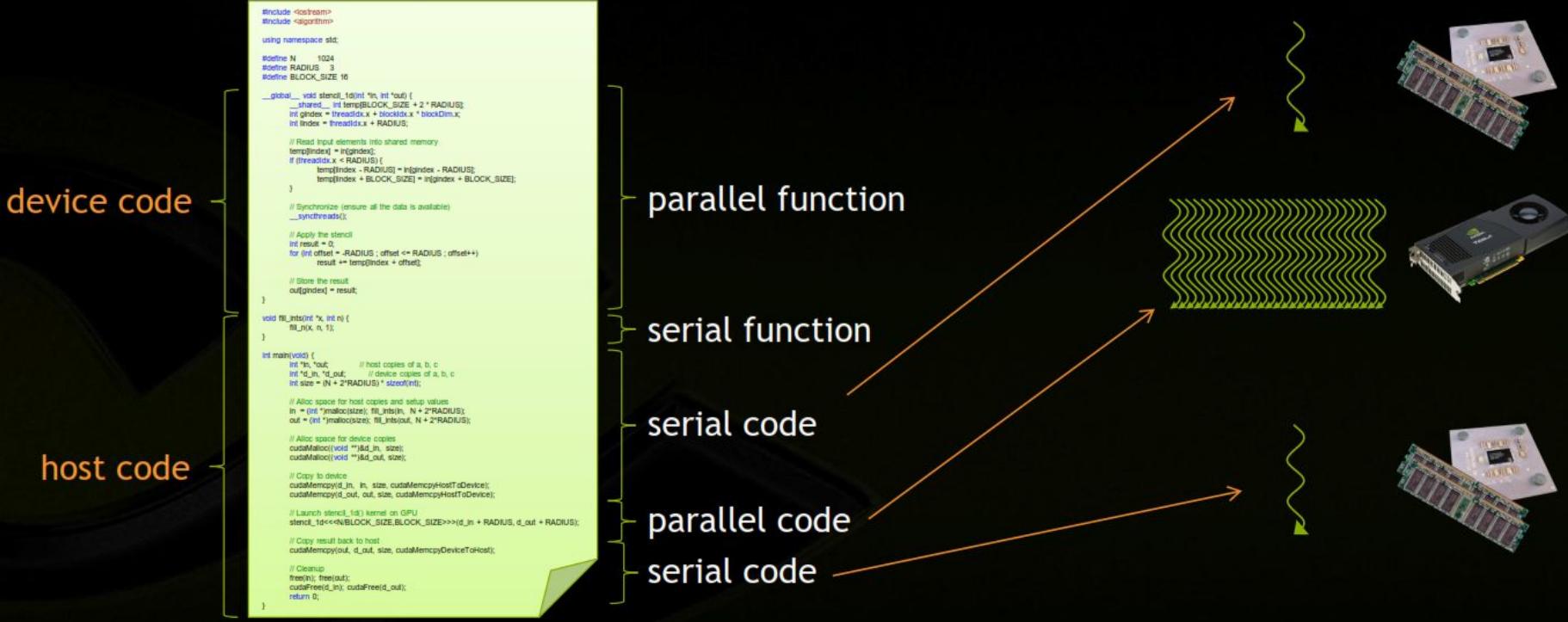
GPU Programming Models

- **CUDA – Compute Unified Device Architecture**
 - Developed by Nvidia -- proprietary
 - First serious GPGPU language/environment
- **OpenCL – Open Computing Language**
 - From makers of OpenGL
 - Wide industry support: AMD, Apple, Qualcomm, Nvidia, etc.
- **C++ AMP – C++ Accelerated Massive Parallelism**
 - Microsoft
 - Much higher abstraction than CUDA/OpenCL
- **OpenACC – Open Accelerator**
 - Like OpenMP for GPUs (semi-auto-parallelize serial code)
 - Much higher abstraction than CUDA/OpenCL

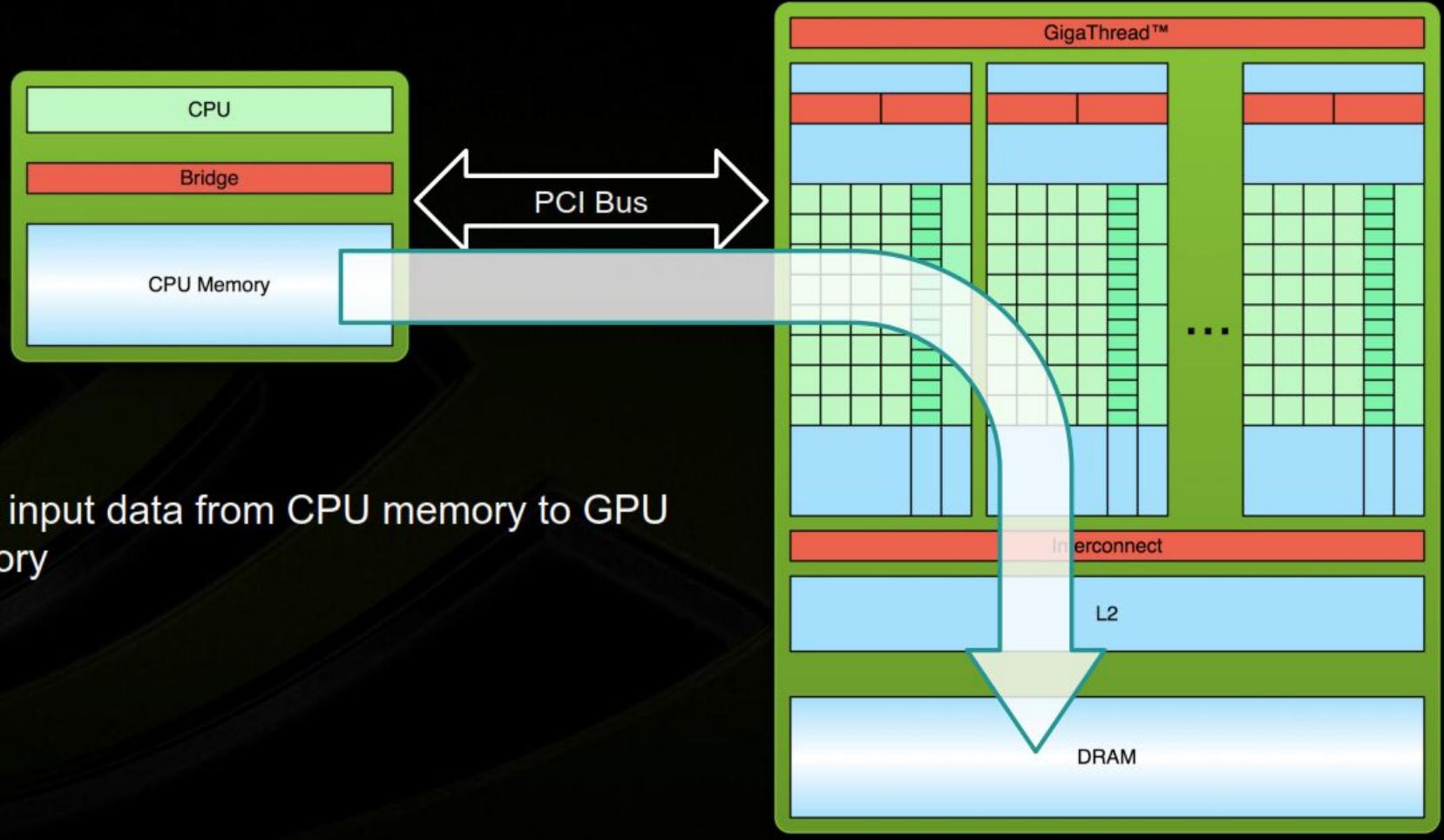
Terminology

- *Host The CPU and its memory (host memory)*
- *Device The GPU and its memory (device memory)*

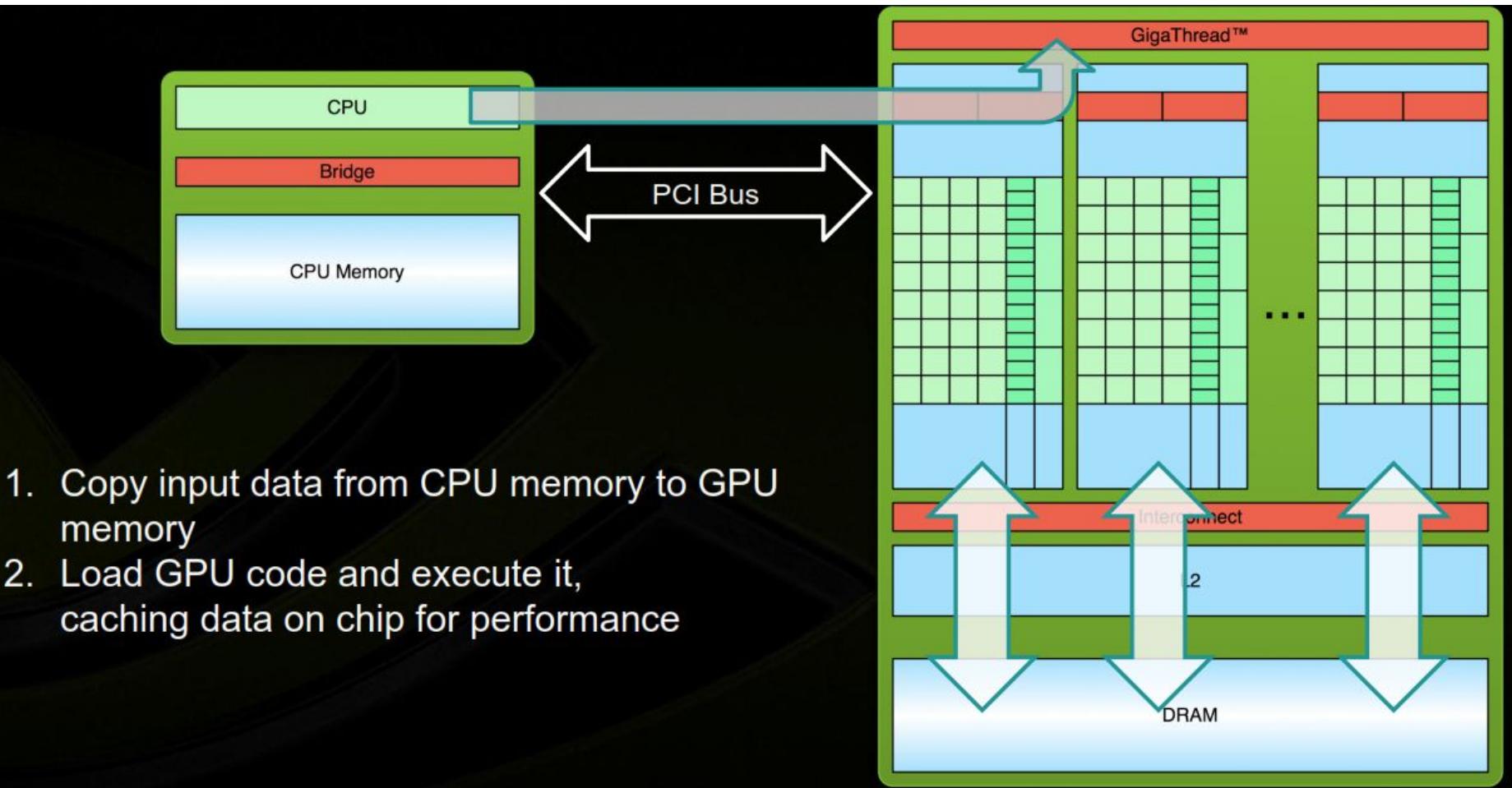




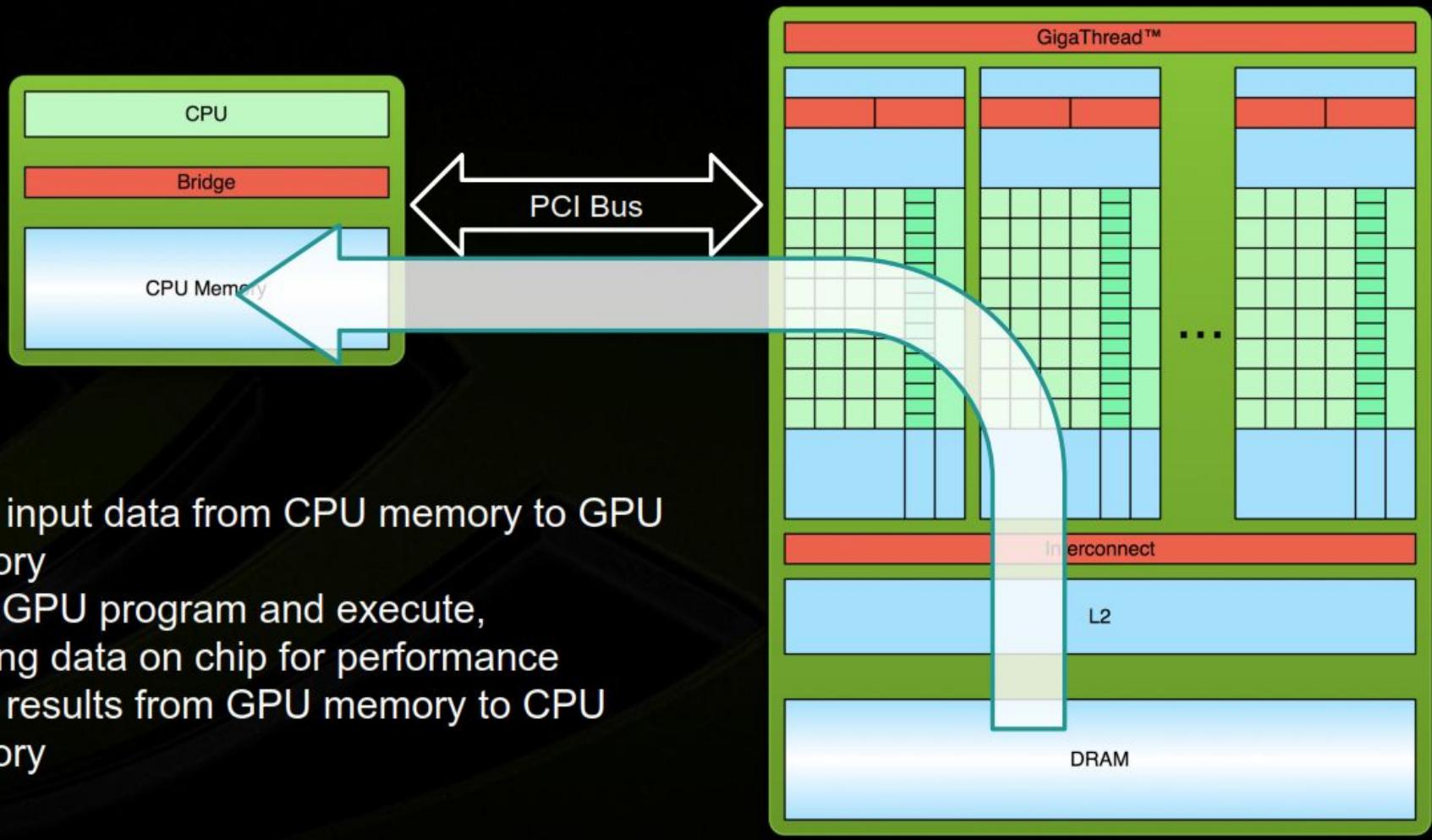
Simple Processing Flow



Simple Processing Flow



Simple Processing Flow



Introduction to CUDA C



What is CUDA?

- CUDA Architecture
 - Expose general-purpose GPU computing as first-class capability
 - Retain traditional DirectX/OpenGL graphics performance
- CUDA C
 - Based on industry-standard C
 - A handful of language extensions to allow heterogeneous programs
 - Straightforward APIs to manage devices, memory, etc.
- This talk will introduce you to CUDA C

Introduction to CUDA C

- What will you learn today?
 - Start from “Hello, World!”
 - Write and launch CUDA C kernels
 - Manage GPU memory
 - Run parallel kernels in CUDA C
 - Parallel communication and synchronization
 - Race conditions and atomic operations

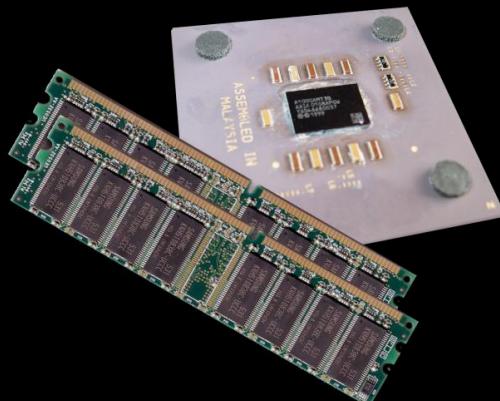
CUDA C Prerequisites

- You (probably) need experience with C or C++
- You do not need any GPU experience
- You do not need any graphics experience
- You do not need any parallel programming experience

CUDA C: The Basics

- Terminology
 - *Host* - The CPU and its memory (host memory)
 - *Device* - The GPU and its memory (device memory)

Host



Device



Note: Figure Not to Scale

Hello, World!

```
int main( void ) {  
    printf( "Hello, World!\n" );  
    return 0;  
}
```

- This basic program is just standard C that runs on the *host*
- NVIDIA's compiler (`nvcc`) will not complain about CUDA programs with no *device* code
- At its simplest, CUDA C is just C!

Hello, World! with Device Code

```
__global__ void kernel( void ) {  
}  
  
int main( void ) {  
    kernel<<<1,1>>>();  
    printf( "Hello, World!\n" );  
    return 0;  
}
```

- Two notable additions to the original “Hello, World!”

Hello, World! with Device Code

```
__global__ void kernel( void ) {  
}
```

- CUDA C keyword `__global__` indicates that a function
 - Runs on the device
 - Called from host code
- nvcc splits source file into host and device components
 - NVIDIA's compiler handles device functions like `kernel()`
 - Standard host compiler handles host functions like `main()`
 - gcc
 - Microsoft Visual C

Hello, World! with Device Code

```
int main( void ) {
    kernel<<< 1, 1 >>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- Triple angle brackets mark a call from *host* code to *device* code
 - Sometimes called a “kernel launch”
 - We’ll discuss the parameters inside the angle brackets later
- This is all that’s required to execute a function on the GPU!
- The function `kernel()` does nothing, so this is fairly anticlimactic...

A More Complex Example

- A simple kernel to add two integers:

```
__global__ void add( int *a, int *b, int *c ) {  
    *c = *a + *b;  
}
```

- As before, `__global__` is a CUDA C keyword meaning
 - `add()` will execute on the device
 - `add()` will be called from the host

A More Complex Example

- Notice that we use pointers for our variables:

```
__global__ void add( int *a, int *b, int *c ) {  
    *c = *a + *b;  
}
```

- `add()` runs on the device...so `a`, `b`, and `c` must point to device memory
- How do we allocate memory on the GPU?

Memory Management

- Host and device memory are distinct entities
 - Device pointers point to GPU memory
 - May be passed to and from host code
 - May not be dereferenced from host code
 - Host pointers point to CPU memory
 - May be passed to and from device code
 - May not be dereferenced from device code
- Basic CUDA API for dealing with device memory
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar to their C equivalents, `malloc()`, `free()`, `memcpy()`



A More Complex Example: add()

- Using our add() kernel:

```
__global__ void add( int *a, int *b, int *c ) {  
    *c = *a + *b;  
}
```

- Let's take a look at main() ...

A More Complex Example: main()

```
int main( void ) {
    int a, b, c;                                // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;                  // device copies of a, b, c
    int size = sizeof( int );                    // we need space for an integer

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = 2;
    b = 7;
```

A More Complex Example: main() (cont)

```
// copy inputs to device
cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );

// launch add() kernel on GPU, passing parameters
add<<< 1, 1 >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );

cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
}
```

Parallel Programming in CUDA C

- But wait...GPU computing is about massive parallelism
- So how do we run code in parallel on the device?
- Solution lies in the parameters between the triple angle brackets:

```
add<<< 1, 1 >>>( dev_a, dev_b, dev_c );  
|  
add<<< N, 1 >>>( dev_a, dev_b, dev_c );
```

- Instead of executing `add()` once, `add()` executed `N` times in parallel

Parallel Programming in CUDA C

- With `add()` running in parallel...let's do vector addition
- Terminology: Each parallel invocation of `add()` referred to as a ***block***
- Kernel can refer to its block's index with the variable `blockIdx.x`
- Each block adds a value from `a[]` and `b[]`, storing the result in `c[]` :

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

- By using `blockIdx.x` to index arrays, each block handles different indices

Parallel Programming in CUDA C

- We write this code:

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

- This is what runs in parallel on the device:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

Parallel Addition: add()

- Using our newly parallelized add() kernel:

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

- Let's take a look at main() ...

Parallel Addition: main()

```
#define N 512

int main( void ) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;                      // device copies of a, b, c
    int size = N * sizeof( int );                   // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition: main() (cont)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with N parallel blocks
add<<< N, 1 >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Review

- Difference between “host” and “device”
 - Host = CPU
 - Device = GPU
- Using `__global__` to declare a function as device code
 - Runs on device
 - Called from host
- Passing parameters from host code to a device function

Review (cont)

- Basic device memory management
 - `cudaMalloc()`
 - `cudaMemcpy()`
 - `cudaFree()`
- Launching parallel kernels
 - Launch N copies of `add()` with: `add<<< N, 1 >>>();`
 - Used `blockIdx.x` to access block's index

Threads

- Terminology: A block can be split into parallel *threads*
- Let's change vector addition to use parallel threads instead of parallel blocks:

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

- We use `threadIdx.x` instead of `blockIdx.x` in `add()`
- `main()` will require one change as well...

Parallel Addition (Threads): main()

```
#define N 512

int main( void ) {
    int *a, *b, *c;                                //host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;                      //device copies of a, b, c
    int size = N * sizeof( int );                   //we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition (Threads): main() (cont)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with N blocks
add<<< N, N >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

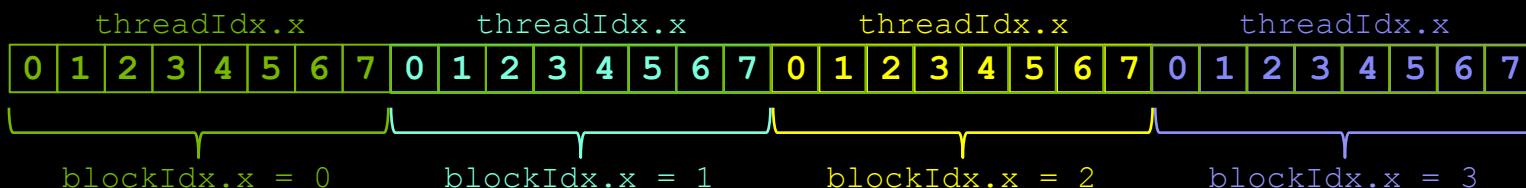
free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Using Threads And Blocks

- We've seen parallel vector addition using
 - Many blocks with 1 thread apiece
 - 1 block with many threads
- Let's adapt vector addition to use lots of *both* blocks and threads
- After using threads and blocks together, we'll talk about *why* threads
- First let's discuss data indexing...

Indexing Arrays With Threads And Blocks

- No longer as simple as just using `threadIdx.x` or `blockIdx.x` as indices
- To index array with 1 thread per entry (using 8 threads/block)

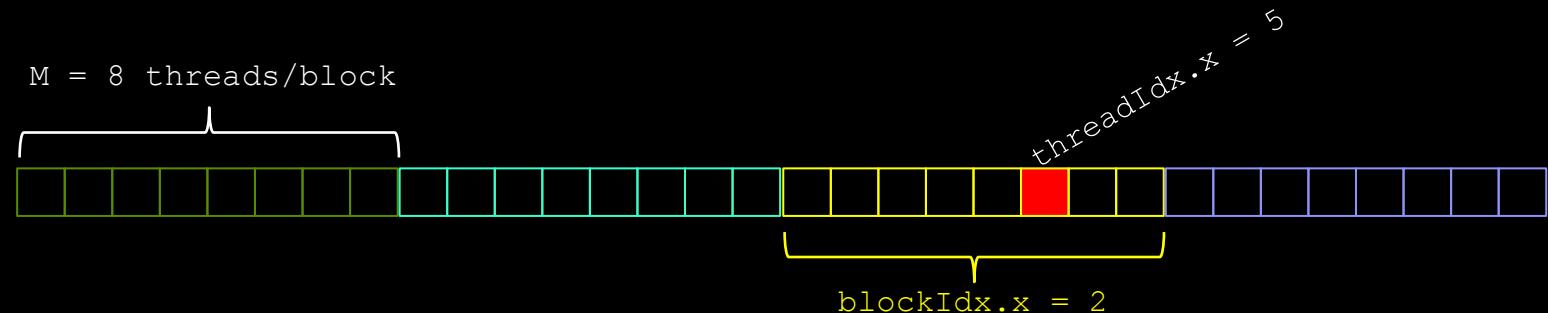


- If we have M threads/block, a unique array index for each entry given by

```
int index = threadIdx.x + blockIdx.x * M;  
int index =     x      +      y      * width;
```

Indexing Arrays: Example

- In this example, the red entry would have an index of 21:



```
int index = threadIdx.x + blockIdx.x * M;  
= 5 + 2 * 8;  
= 21;
```

Addition with Threads and Blocks

- The `blockDim.x` is a built-in variable for threads per block:

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

- A combined version of our vector addition kernel to use blocks *and* threads:

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

- So what changes in `main()` when we use both blocks and threads?

Parallel Addition (Blocks/Threads): main()

```
#define N    (2048*2048)
#define THREADS_PER_BLOCK 512

int main( void ) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;                      // device copies of a, b, c
    int size = N * sizeof( int );                   // we need space for N integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition (Blocks/Threads): main()

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with blocks and threads
add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

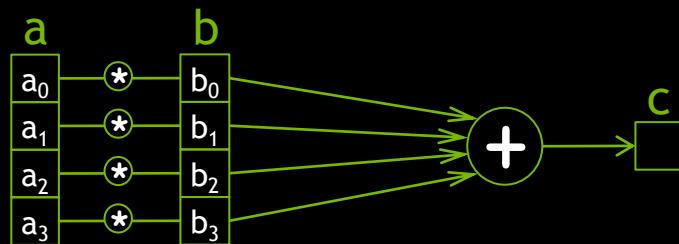
return 0;
}
```

Why Bother With Threads?

- Threads seem unnecessary
 - Added a level of abstraction and complexity
 - What did we gain?
- Unlike parallel blocks, parallel threads have mechanisms to
 - Communicate
 - Synchronize
- Let's see how...

Dot Product

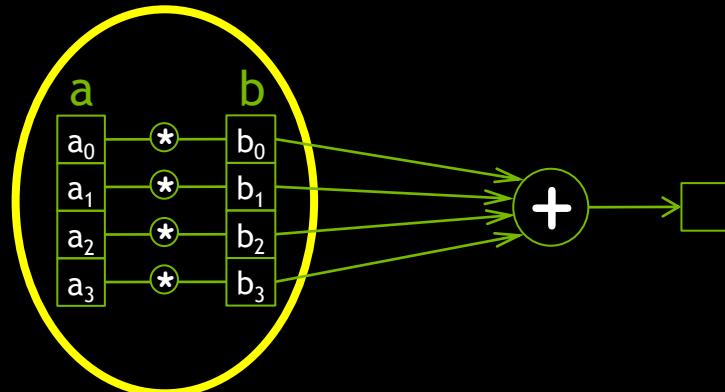
- Unlike vector addition, dot product is a *reduction* from vectors to a scalar



$$\begin{aligned} c &= \vec{a} \cdot \vec{b} \\ &= (a_0, a_1, a_2, a_3) \cdot (b_0, b_1, b_2, b_3) \\ &= a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3 \end{aligned}$$

Dot Product

- Parallel threads have no problem computing the pairwise products:

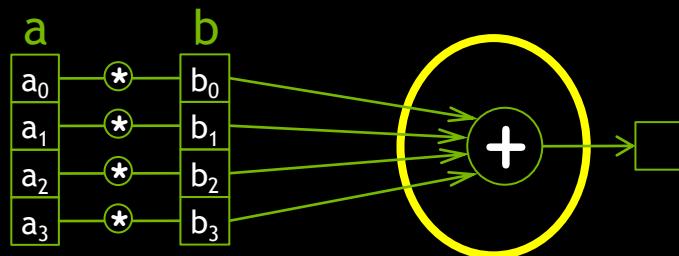


- So we can start a dot product CUDA kernel by doing just that:

```
__global__ void dot( int *a, int *b, int *c ) {  
    // Each thread computes a pairwise product  
    int temp = a[threadIdx.x] * b[threadIdx.x];
```

Dot Product

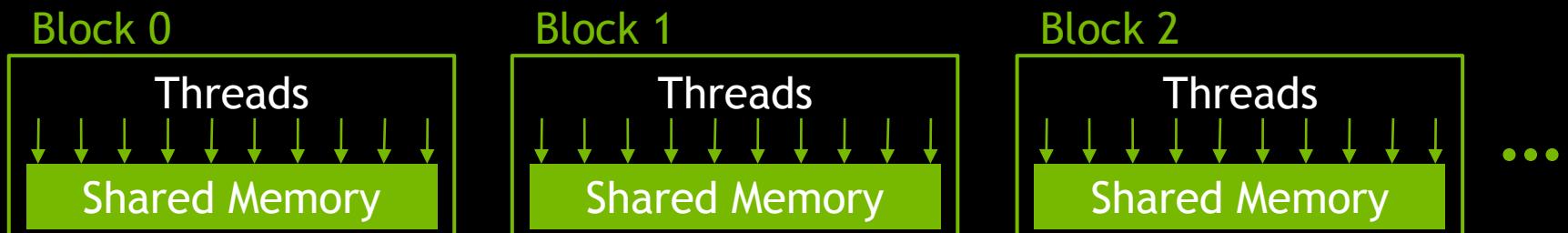
- But we need to share data between threads to compute the final sum:



```
__global__ void dot( int *a, int *b, int *c ) {  
    // Each thread computes a pairwise product  
    int temp = a[threadIdx.x] * b[threadIdx.x];  
  
    // Can't compute the final sum  
    // Each thread's copy of 'temp' is private  
}
```

Sharing Data Between Threads

- Terminology: A block of threads shares memory called...*shared memory*
- Extremely fast, on-chip memory (user-managed cache)
- Declared with the `__shared__` CUDA keyword
- Not visible to threads in other blocks running in parallel



Parallel Dot Product: `dot()`

- We perform parallel multiplication, serial addition:

```
#define N 512
__global__ void dot( int *a, int *b, int *c ) {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

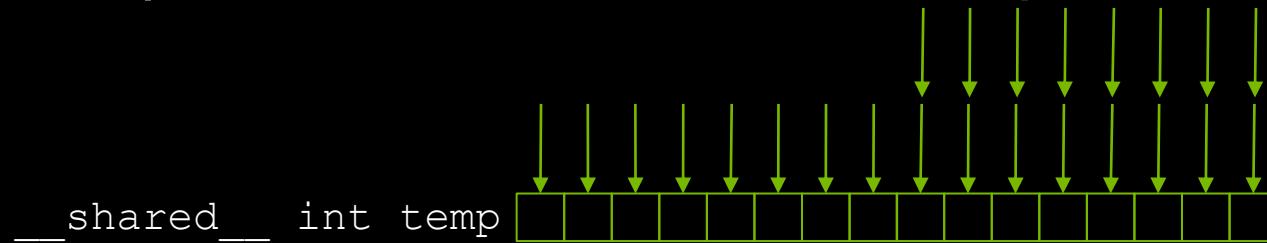
    // Thread 0 sums the pairwise products
    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
            sum += temp[i];
        *c = sum;
    }
}
```

Parallel Dot Product Recap

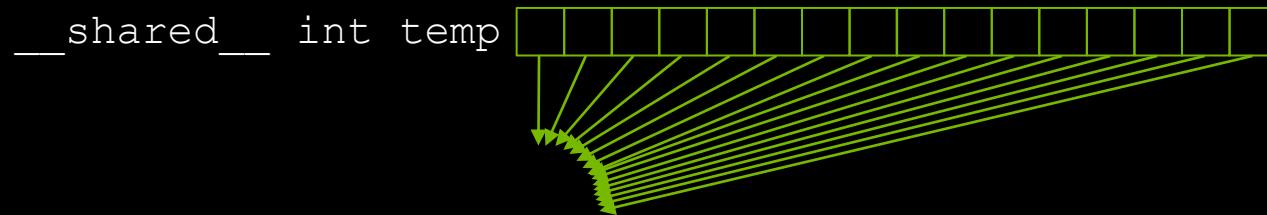
- We perform parallel, pairwise multiplications
- Shared memory stores each thread's result
- We sum these pairwise products from a single thread
- Sounds good...but we've made a huge mistake

Faulty Dot Product Exposed!

- Step 1: In parallel, each thread writes a pairwise product



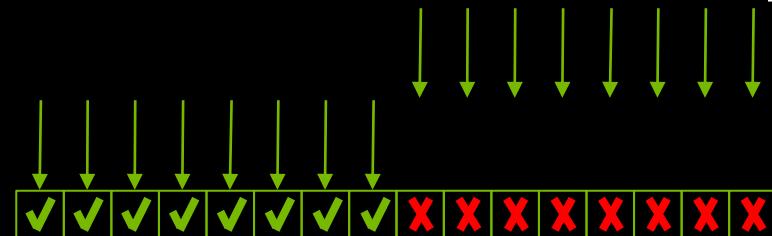
- Step 2: Thread 0 reads and sums the products



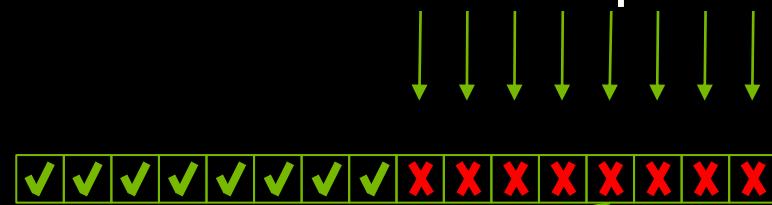
- But there's an assumption hidden in Step 1...

Read-Before-Write Hazard

- Suppose thread 0 finishes its write in step 1

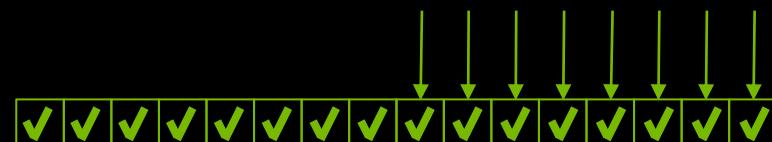


- Then thread 0 reads index 12 in step 2



 *This read returns garbage!*

- Before thread 12 writes to index 12 in step 1?



Synchronization

- We need threads to wait between the sections of `dot()`:

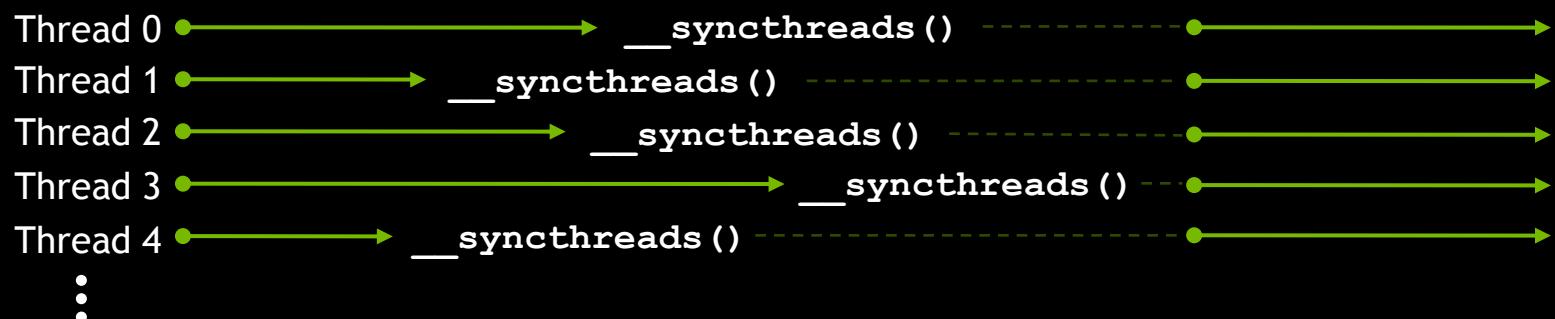
```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // * NEED THREADS TO SYNCHRONIZE HERE *
    // No thread can advance until all threads
    // have reached this point in the code

    // Thread 0 sums the pairwise products
    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
            sum += temp[i];
        *c = sum;
    }
}
```

__syncthreads ()

- We can synchronize threads with the function `__syncthreads ()`
- Threads in the block wait until *all* threads have hit the `__syncthreads ()`



- Threads are *only* synchronized within a block

Parallel Dot Product: `dot()`

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
            sum += temp[i];
        *c = sum;
    }
}
```

- With a properly synchronized `dot()` routine, let's look at `main()`

Parallel Dot Product: main()

```
#define N 512

int main( void ) {
    int *a, *b, *c;                                // copies of a, b, c
    int *dev_a, *dev_b, *dev_c;                      // device copies of a, b, c
    int size = N * sizeof( int );                   // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof( int ) );

    a = (int *)malloc( size );
    b = (int *)malloc( size );
    c = (int *)malloc( sizeof( int ) );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Dot Product: main()

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch dot() kernel with 1 block and N threads
dot<<< 1, N >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, sizeof( int ) , cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
}
```

Review

- Launching kernels with parallel threads
 - Launch `add()` with N threads: `add<<< 1, N >>>();`
 - Used `threadIdx.x` to access thread's index
- Using both blocks and threads
 - Used `(threadIdx.x + blockIdx.x * blockDim.x)` to index input/output
 - $N/\text{THREADS_PER_BLOCK}$ blocks and THREADS_PER_BLOCK threads gave us N threads total

Review (cont)

- Using `__shared__` to declare memory as shared memory
 - Data shared among threads in a block
 - Not visible to threads in other parallel blocks

- Using `__syncthreads()` as a barrier
 - No thread executes instructions after `__syncthreads()` until all threads have reached the `__syncthreads()`
 - Needs to be used to prevent *data hazards*

Multiblock Dot Product

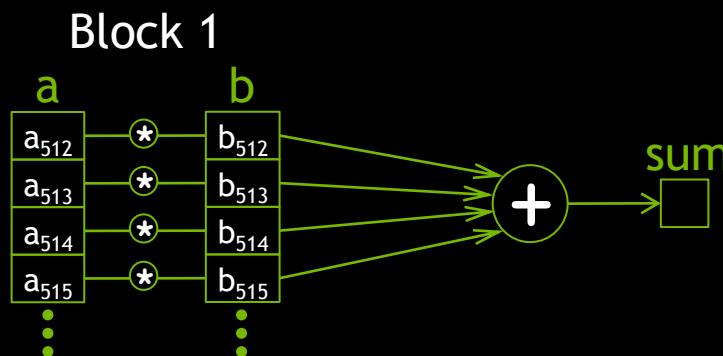
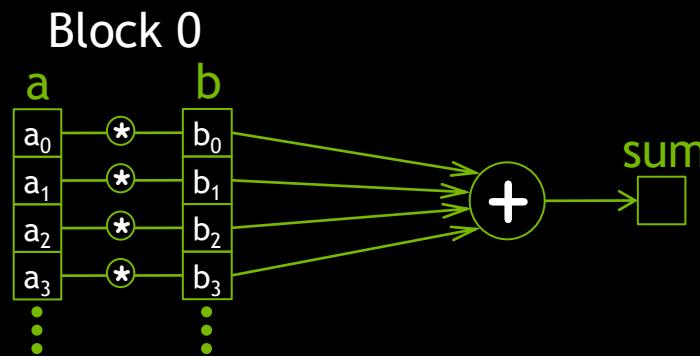
- Recall our dot product launch:

```
// launch dot() kernel with 1 block and N threads  
dot<<< 1, N >>>( dev_a, dev_b, dev_c );
```

- Launching with one block will not utilize much of the GPU
- Let's write a multiblock version of dot product

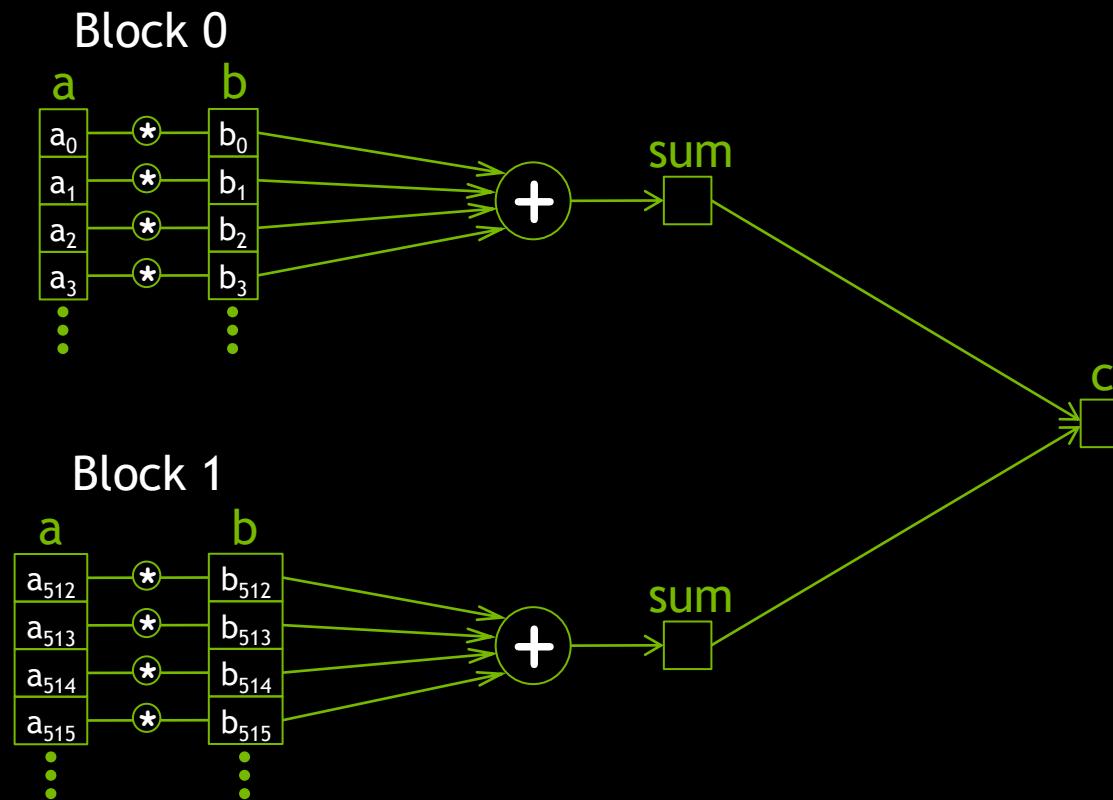
Multiblock Dot Product: Algorithm

- Each block computes a sum of its pairwise products like before:



Multiblock Dot Product: Algorithm

- And then contributes its sum to the final result:



Multiblock Dot Product: dot()

```
#define N    (2048*2048)
#define THREADS_PER_BLOCK 512
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < THREADS_PER_BLOCK; i++ )
            sum += temp[i];
        atomicAdd( c, sum );
    }
}
```

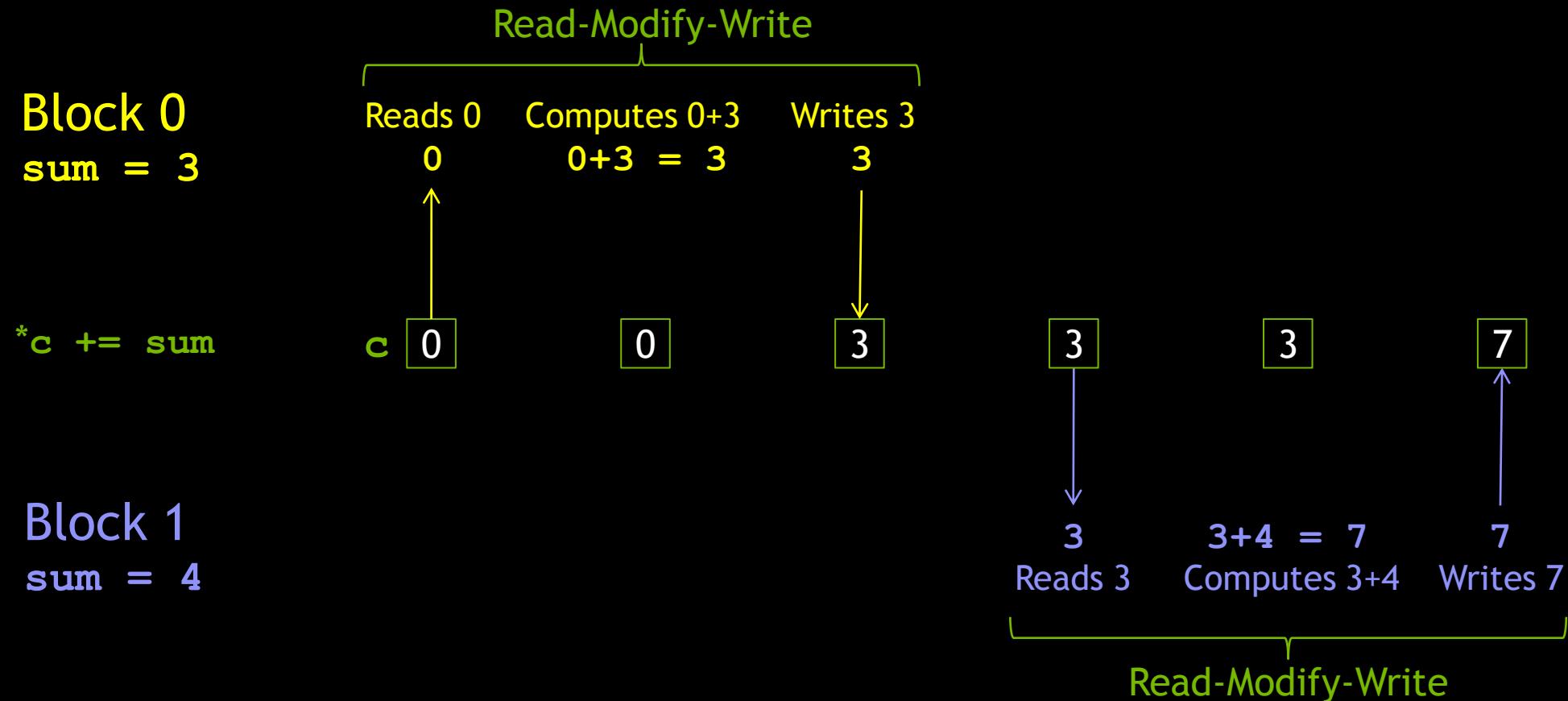
- But we have a race condition...
- We can fix it with one of CUDA's atomic operations

Race Conditions

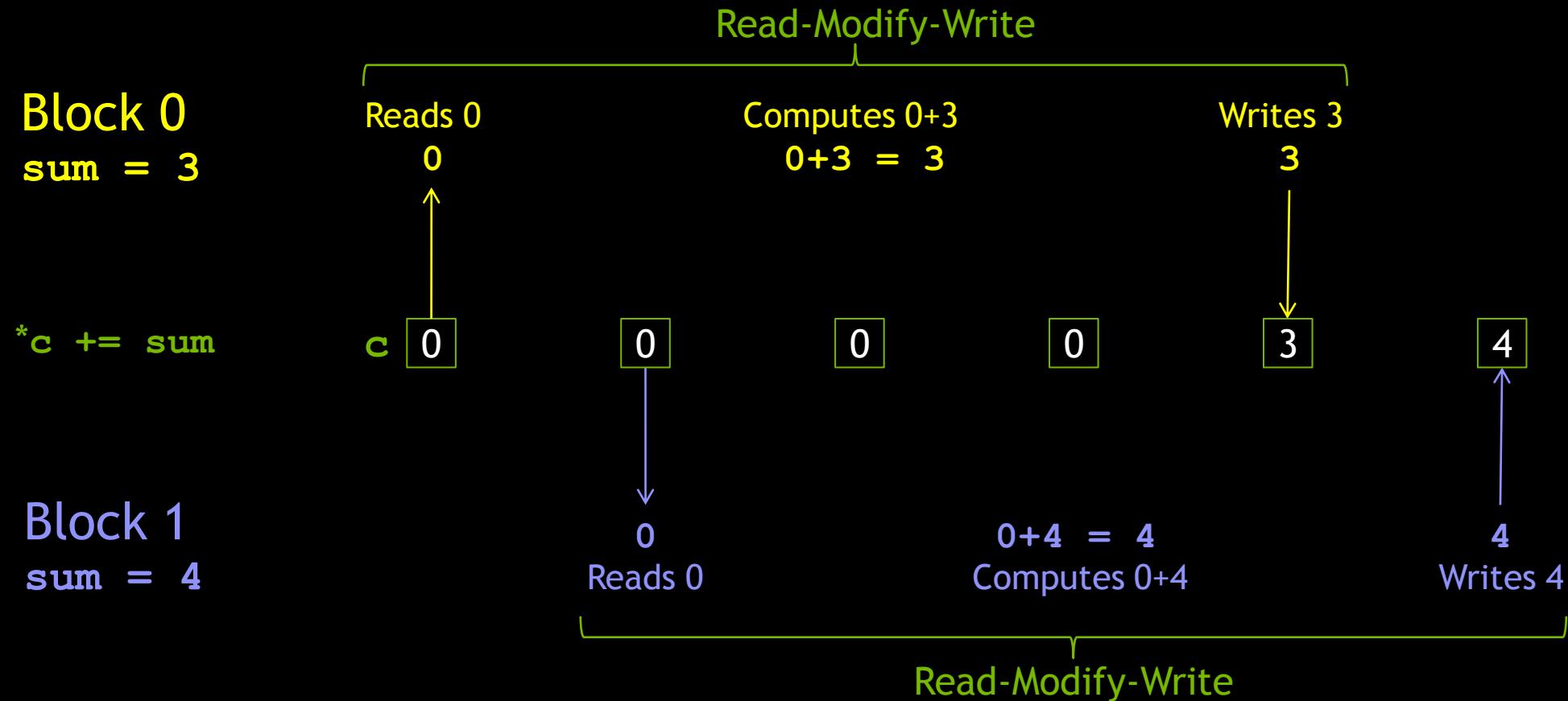
- Terminology: A *race condition* occurs when program behavior depends upon relative timing of two (or more) event sequences
- What actually takes place to execute the line in question: `*c += sum;`
 - Read value at address `c`
 - Add `sum` to value
 - Write result to address `c`

} Terminology: *Read-Modify-Write*
- What if two threads are trying to do this at the same time?
 - Thread 0, Block 0
 - Read value at address `c`
 - Add `sum` to value
 - Write result to address `c`
 - Thread 0, Block 1
 - Read value at address `c`
 - Add `sum` to value
 - Write result to address `c`

Global Memory Contention



Global Memory Contention



Atomic Operations

- Terminology: Read-modify-write uninterruptible when *atomic*
- Many *atomic operations* on memory available with CUDA C
 - `atomicAdd()`
 - `atomicSub()`
 - `atomicMin()`
 - `atomicMax()`
 - `atomicInc()`
 - `atomicDec()`
 - `atomicExch()`
 - `atomicCAS()`
- Predictable result when simultaneous access to memory required
- We need to atomically add `sum` to `c` in our multiblock dot product

Multiblock Dot Product: dot()

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < THREADS_PER_BLOCK; i++ )
            sum += temp[i];
        atomicAdd( c, sum );
    }
}
```

- Now let's fix up `main()` to handle a multiblock dot product

Parallel Dot Product: main()

```
#define N    (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;                      // device copies of a, b, c
    int size = N * sizeof( int );                   // we need space for N ints

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof( int ) );

    a = (int *)malloc( size );
    b = (int *)malloc( size );
    c = (int *)malloc( sizeof( int ) );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Dot Product: main ()

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch dot() kernel
dot<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, sizeof( int ), cudaMemcpyDeviceToHost );

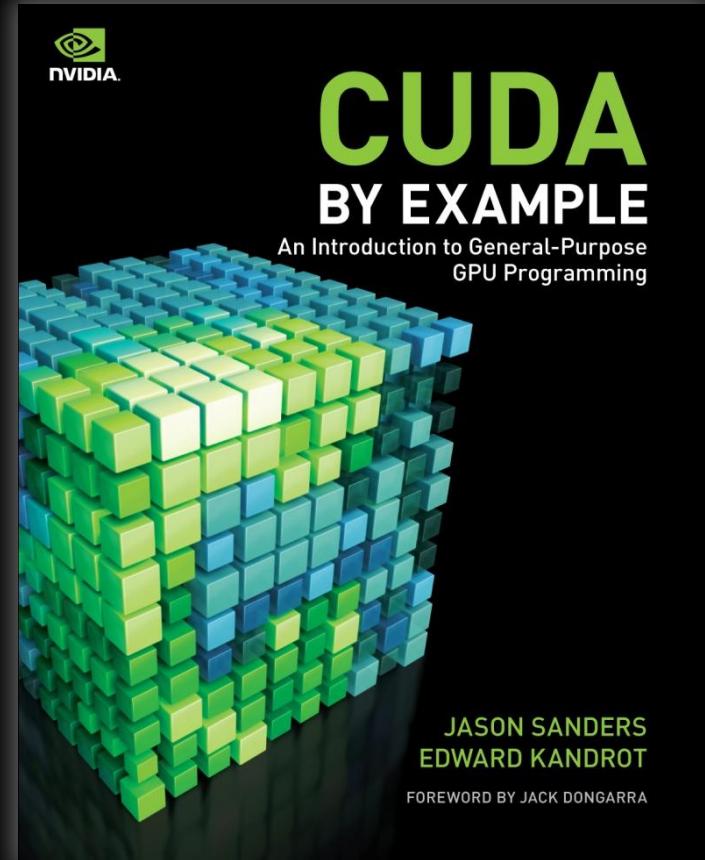
free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Review

- Race conditions
 - Behavior depends upon relative timing of multiple event sequences
 - Can occur when an implied read-modify-write is interruptible
- Atomic operations
 - CUDA provides read-modify-write operations guaranteed to be atomic
 - Atomics ensure correct results when multiple threads modify memory

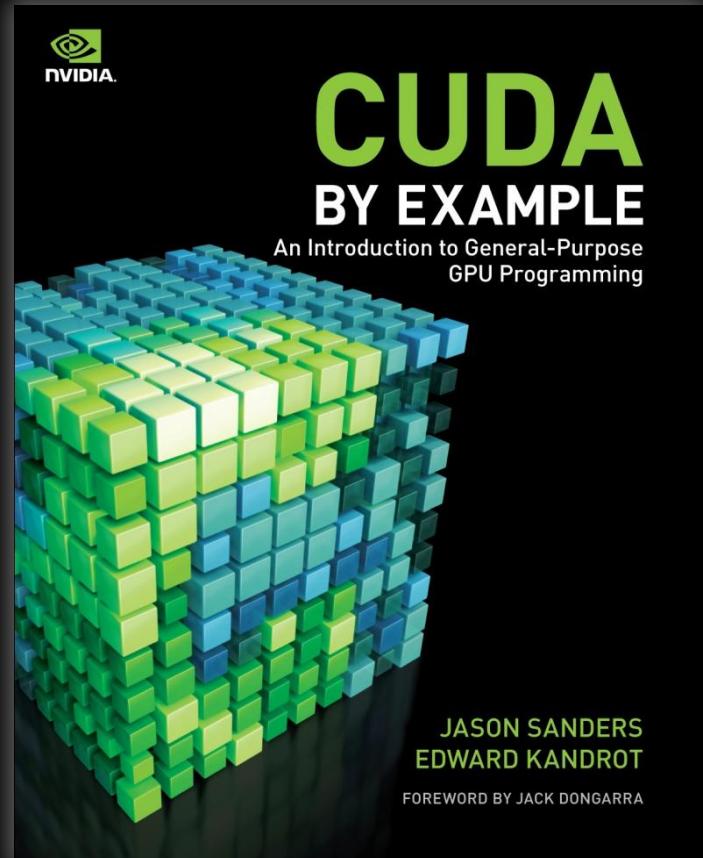
To Learn More CUDA C

- Check out *CUDA by Example*
 - Parallel Programming in CUDA C
 - Thread Cooperation
 - Constant Memory and Events
 - Texture Memory
 - Graphics Interoperability
 - Atomics
 - Streams
 - CUDA C on Multiple GPUs
 - Other CUDA Resources
- <http://developer.nvidia.com/object/cuda-by-example.html>



Questions

- First my questions
- Now your questions...





CUDA C/C++ Basics

Supercomputing 2011 Tutorial

Cyril Zeller, NVIDIA Corporation





What is CUDA?

- CUDA Architecture
 - Expose GPU computing for general purpose
 - Retain performance
- CUDA C/C++
 - Based on industry-standard C/C++
 - Small set of extensions to enable heterogeneous programming
 - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C/C++

Introduction to CUDA C/C++



- What will you learn in this session?
 - Start from “Hello World!”
 - Write and execute C code on the GPU
 - Manage GPU memory
 - Manage communication and synchronization

Prerequisites



- You (probably) need experience with C or C++
- You don't need GPU experience
- You don't need parallel programming experience
- You don't need graphics experience

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

HELLO WORLD!

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

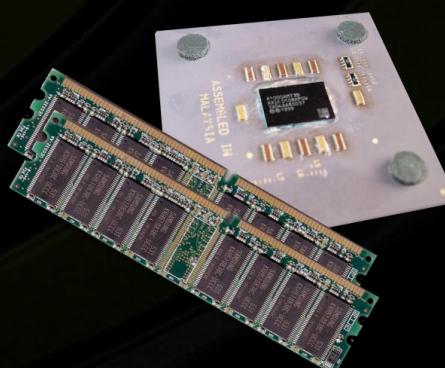
Handling errors

Managing devices

Heterogeneous Computing



- Terminology:
 - *Host* The CPU and its memory (host memory)
 - *Device* The GPU and its memory (device memory)



Host



Device

Heterogeneous Computing



device code

host code

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N      1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    shared_int temp[BLOCK_SIZE / 2 + RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int index = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[index] = in[gindex];
    #if (threadIdx.x < RADIUS)
        temp[index - RADIUS] = in[gindex - RADIUS];
        temp[index + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    #endif

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[index + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;           // host copies of a, b, c
    int *d_in, *d_out;       // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **) &d_in, size);
    cudaMalloc((void **) &d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<N</BLOCK_SIZE,BLOCK_SIZE>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

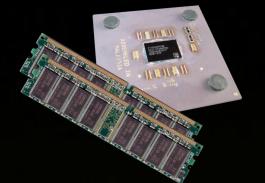
parallel function

serial function

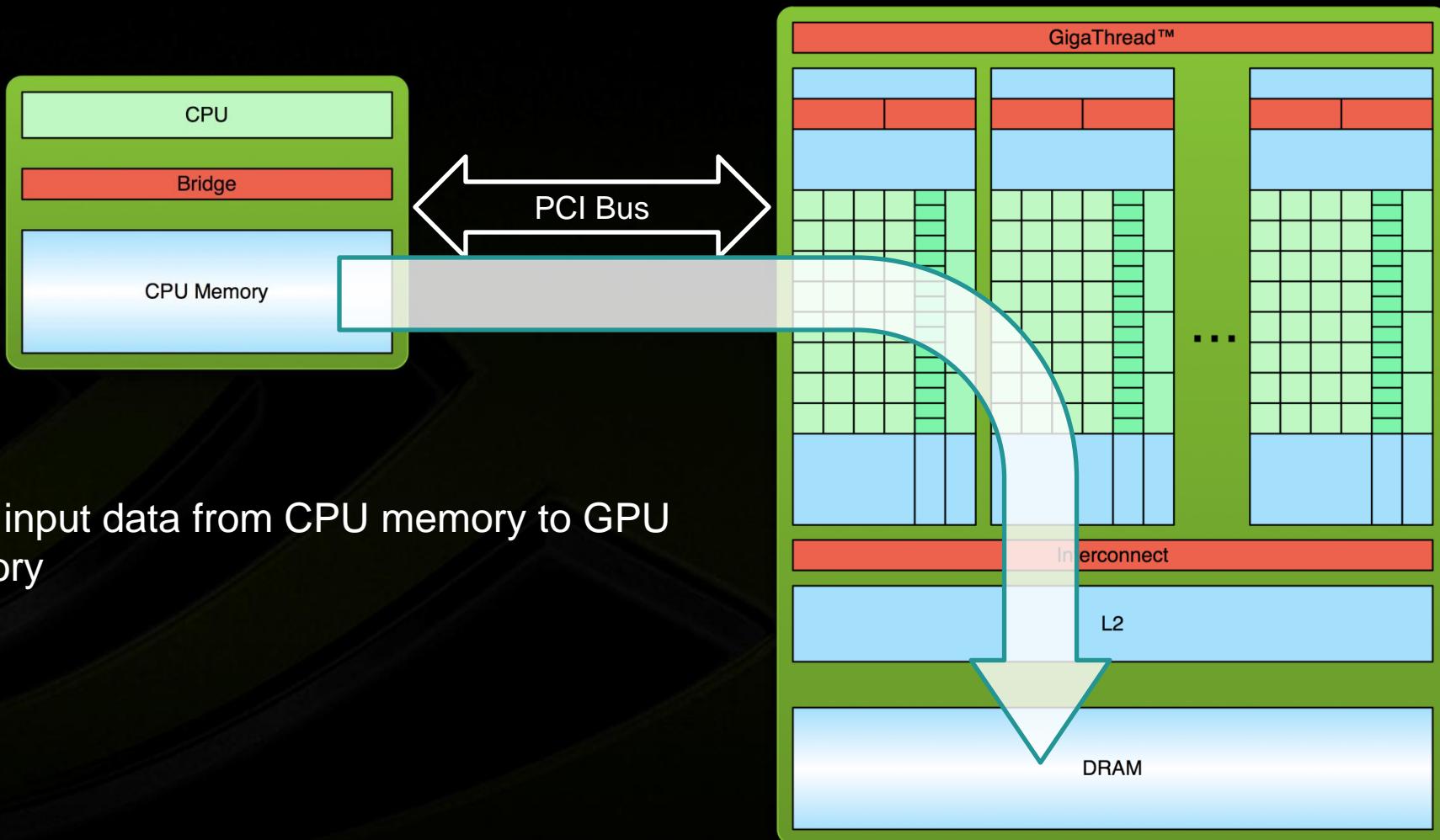
serial code

parallel code

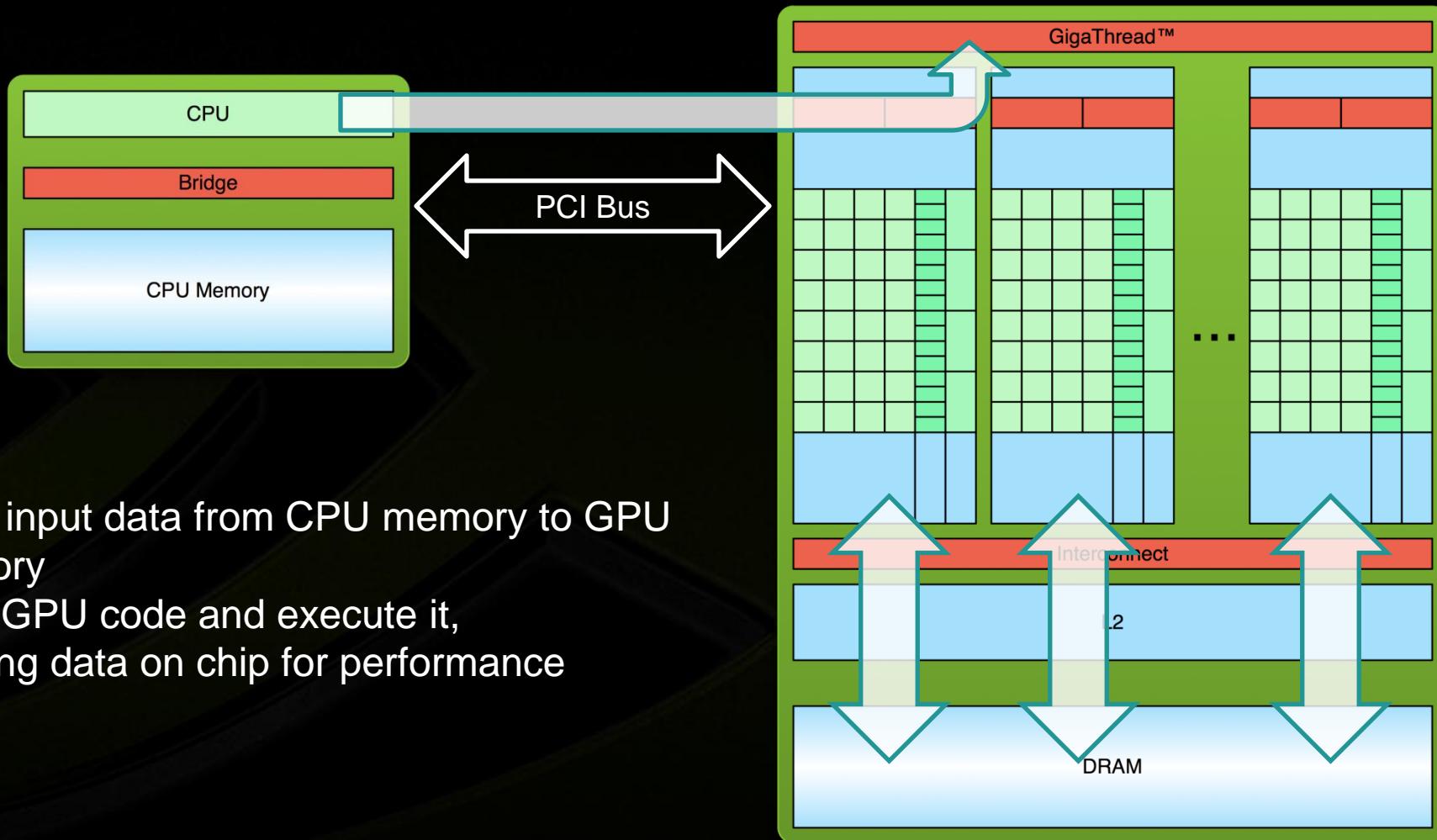
serial code



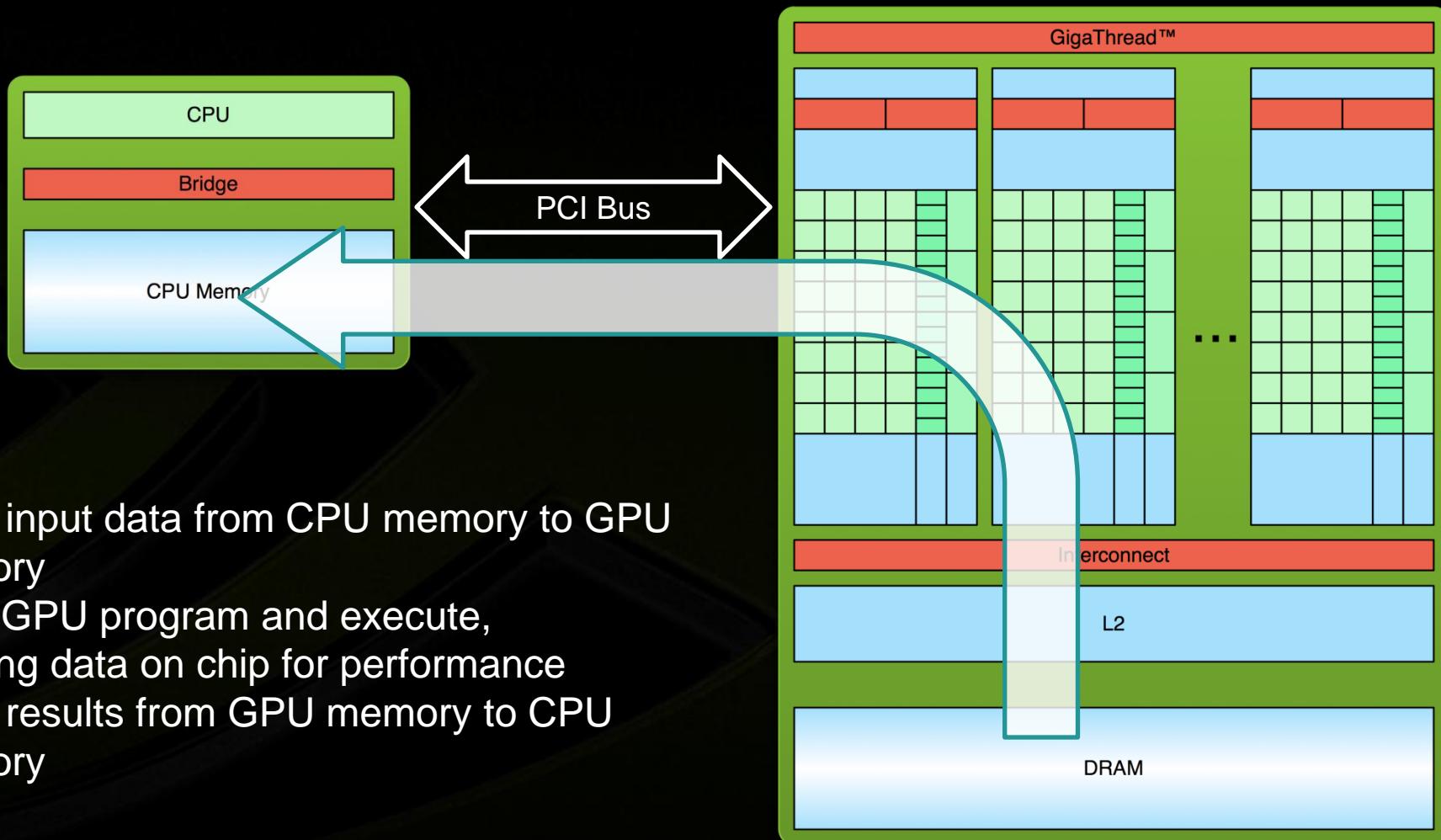
Simple Processing Flow



Simple Processing Flow



Simple Processing Flow



Hello World!



```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc  
hello_world.cu  
$ a.out  
Hello World!  
$
```



Hello World! with Device Code

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Two new syntactic elements...

Hello World! with Device Code



```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
 - Runs on the device
 - Is called from host code
- nvcc separates source code into host and device components
 - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
 - Host functions (e.g. `main()`) processed by standard host compiler
 - `gcc`, `cl.exe`



Hello World! with Device Code

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
 - Also called a “kernel launch”
 - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!



Hello World! with Device Code

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

Output:

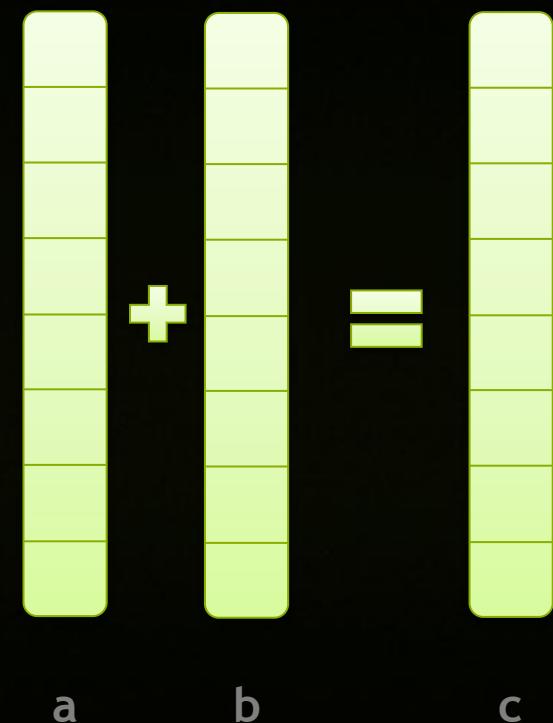
```
$ nvcc hello.cu  
$ a.out  
Hello World!  
$
```

- `mykernel()` does nothing, somewhat anticlimactic!

Parallel Programming in CUDA C/C++



- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We'll start by adding two integers and build up to vector addition





Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
 - `add()` will execute on the device
 - `add()` will be called from the host



Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU

Memory Management

- Host and device memory are separate entities
 - *Device* pointers point to GPU memory
 - May be passed to/from host code
 - May *not* be dereferenced in host code
 - *Host* pointers point to CPU memory
 - May be passed to/from device code
 - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`





Addition on the Device: add()

- Returning to our add() kernel

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Let's take a look at main()...

Addition on the Device: main ()



```
int main(void) {  
    int a, b, c;                                // host copies of a, b, c  
    int *d_a, *d_b, *d_c;                        // device copies of a, b, c  
    int size = sizeof(int);  
  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **) &d_a, size);  
    cudaMalloc((void **) &d_b, size);  
    cudaMalloc((void **) &d_c, size);  
  
    // Setup input values  
    a = 2;  
    b = 7;
```

Addition on the Device: main()



```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

RUNNING IN PARALLEL

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

Moving to Parallel

- GPU computing is about massive parallelism
 - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();  
          ^  
          N  
add<<< N, 1 >>>();
```



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

Vector Addition on the Device



- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **block**
 - The set of blocks is referred to as a **grid**
 - Each invocation can refer to its block index using `blockIdx.x`

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

- By using `blockIdx.x` to index into the array, each block handles a different element of the array

Vector Addition on the Device



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

- On the device, each block can execute in parallel:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```



Vector Addition on the Device: add()

- Returning to our parallelized add() kernel

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

- Let's take a look at main()...

Vector Addition on the Device: main()



```
#define N 512

int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: main()



```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```



Review (1 of 2)

- Difference between *host* and *device*
 - *Host* CPU
 - *Device* GPU
- Using `__global__` to declare a function as device code
 - Executes on the device
 - Called from the host
- Passing parameters from host code to a device function



Review (2 of 2)

- Basic device memory management
 - `cudaMalloc()`
 - `cudaMemcpy()`
 - `cudaFree()`
- Launching parallel kernels
 - Launch N copies of `add()` with `add<<<N, 1>>>(...);`
 - Use `blockIdx.x` to access block index

INTRODUCING THREADS

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

CUDA Threads



- Terminology: a block can be split into parallel **threads**
- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

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

- We use `threadIdx.x` instead of `blockIdx.x`
- Need to make one change in `main()`...

Vector Addition Using Threads: main()



```
#define N 512

int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition Using Threads: main()



```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

COMBINING THREADS AND BLOCKS

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices



Combining Blocks and Threads

- We've seen parallel vector addition using:
 - Several blocks with one thread each
 - One block with several threads
- Let's adapt vector addition to use both *blocks* and *threads*
- Why? We'll come to that...
- First let's discuss data indexing...

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
 - Consider indexing an array with one element per thread (8 threads/block)



- With M threads per block, a unique index for each thread is given by:

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

Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;  
=      5      +      2      *  8;  
= 21;
```

Vector Addition with Blocks and Threads



- Use the built-in variable `blockDim.x` for threads per block

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

- Combined version of `add()` to use parallel threads *and* parallel blocks

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

- What changes need to be made in `main()`?

Addition with Blocks and Threads: main()



```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: main()



```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```



Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        c[index] = a[index] + b[index];  
}
```

- Update the kernel launch:

```
add<<< (N + M-1) / M>>>(d_a, d_b, d_c, N);
```



Why Bother with Threads?

- Threads seem unnecessary
 - They add a level of complexity
 - What do we gain?
- Unlike parallel blocks, threads have mechanisms to efficiently:
 - Communicate
 - Synchronize
- To look closer, we need a new example...

COOPERATING THREADS

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
 - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



Implementing Within a Block

- Each thread processes one output element
 - `blockDim.x` elements per block
- Input elements are read several times
 - With radius 3, each input element is read seven times



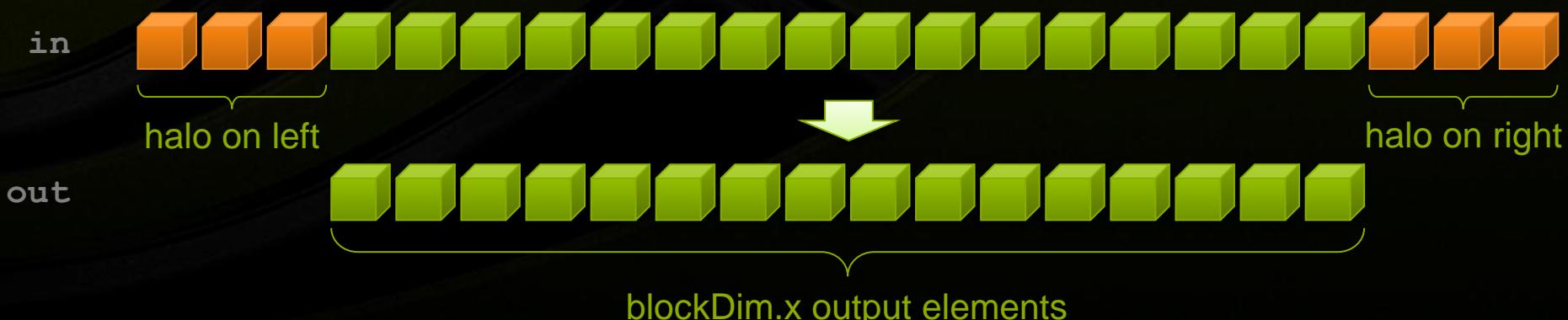
Sharing Data Between Threads



- Terminology: within a block, threads share data via **shared memory**
- Extremely fast on-chip memory
 - By opposition to device memory, referred to as **global memory**
 - Like a user-managed cache
- Declare using **__shared__**, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

- Cache data in shared memory
 - Read $(blockDim.x + 2 * radius)$ input elements from global memory to shared memory
 - Compute $blockDim.x$ output elements
 - Write $blockDim.x$ output elements to global memory
- Each block needs a halo of $radius$ elements at each boundary

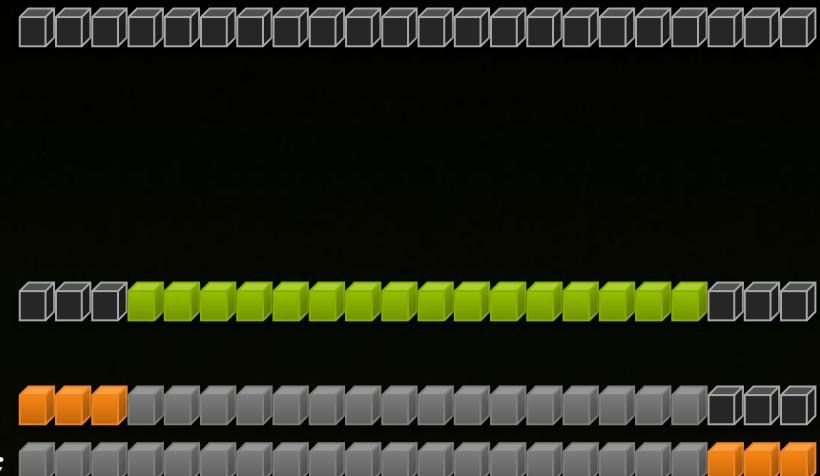


Stencil Kernel



```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
}
```



Stencil Kernel



```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```

Data Race!

- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```
...
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
```

Store at temp[18]



Skipped since threadIdx.x > RADIUS





__syncthreads()

- `void __syncthreads();`
- Synchronizes all threads within a block
 - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
 - In conditional code, the condition must be uniform across the block

Stencil Kernel



```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
```

Stencil Kernel



```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```

Review (1 of 2)

- Launching parallel threads
 - Launch N blocks with M threads per block with `kernel<<<N, M>>>(...);`
 - Use `blockIdx.x` to access block index within grid
 - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

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

Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
 - Data is shared between threads in a block
 - Not visible to threads in other blocks

- Use `__syncthreads()` as a barrier
 - Use to prevent data hazards

MANAGING THE DEVICE

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

Coordinating Host & Device



- Kernel launches are **asynchronous**
 - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

`cudaMemcpy()`

Blocks the CPU until the copy is complete
Copy begins when all preceding CUDA calls have completed

`cudaMemcpyAsync()`

Asynchronous, does not block the CPU

`cudaDeviceSynchronize()`

Blocks the CPU until all preceding CUDA calls have completed



Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
 - Error in the API call itself
 - OR
 - Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```
- Get a string to describe the error:

```
char *cudaGetString(cudaError_t)
```

```
printf("%s\n", cudaGetString(cudaGetLastError()));
```

Device Management



- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)  
cudaSetDevice(int device)  
cudaGetDevice(int *device)  
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple host threads can share a device
- A single host thread can manage multiple devices

```
cudaSetDevice(i) to select current device  
cudaMemcpy(...) for peer-to-peer copies
```

Introduction to CUDA C/C++



- What have we learned?
 - Write and launch CUDA C/C++ kernels
 - `__global__`, `<<<>>>`, `blockIdx`, `threadIdx`, `blockDim`
 - Manage GPU memory
 - `cudaMalloc()`, `cudaMemcpy()`, `cudaFree()`
 - Manage communication and synchronization
 - `__shared__`, `__syncthreads()`
 - `cudaMemcpy()` **vs** `cudaMemcpyAsync()`, `cudaDeviceSynchronize()`



Topics we skipped

- We skipped some details, you can learn more:
 - CUDA Programming Guide
 - CUDA Zone – tools, training, webinars and more
 - <http://developer.nvidia.com/cuda>
- Need a quick primer for later:
 - Compute capability
 - Multi-dimensional indexing
 - Textures

Compute Capability

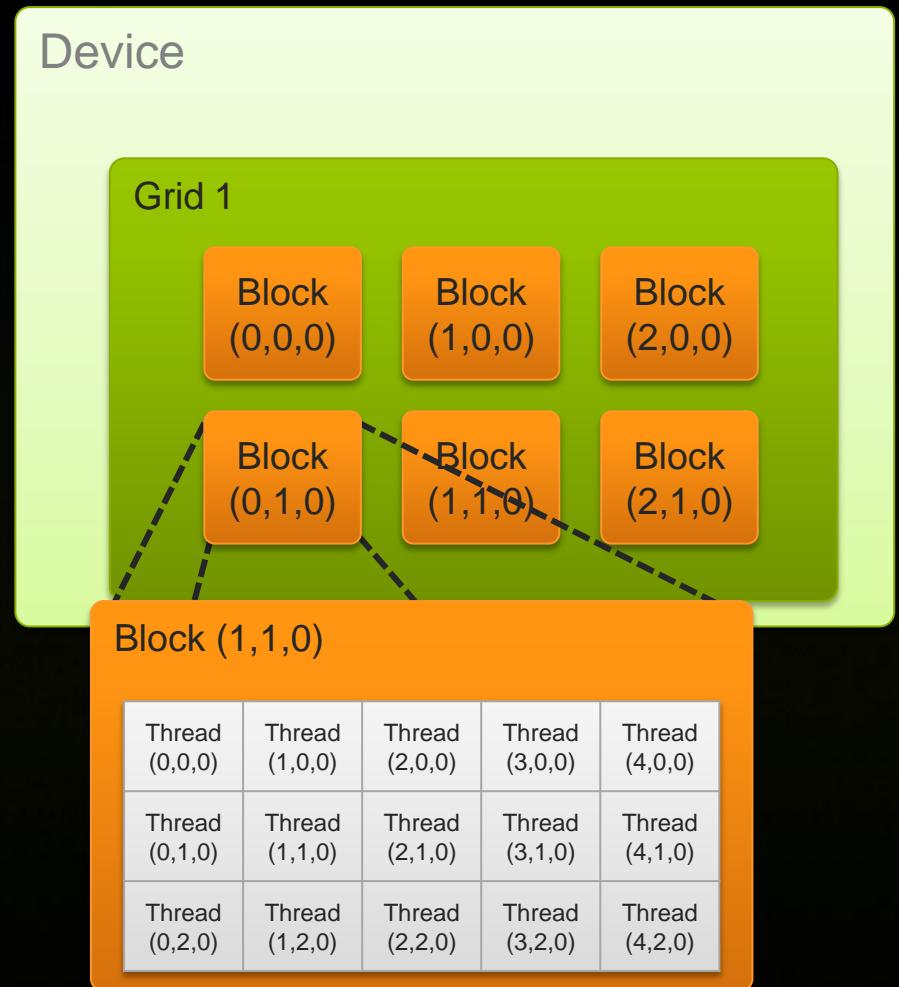
- The **compute capability** of a device describes its architecture, e.g.
 - Number of registers
 - Sizes of memories
 - Features & capabilities

Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

- The following presentations concentrate on Fermi devices
 - Compute Capability ≥ 2.0

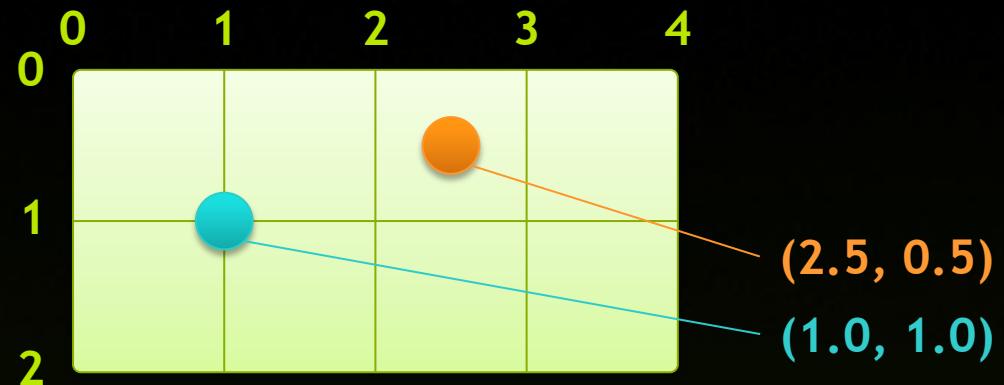
IDs and Dimensions

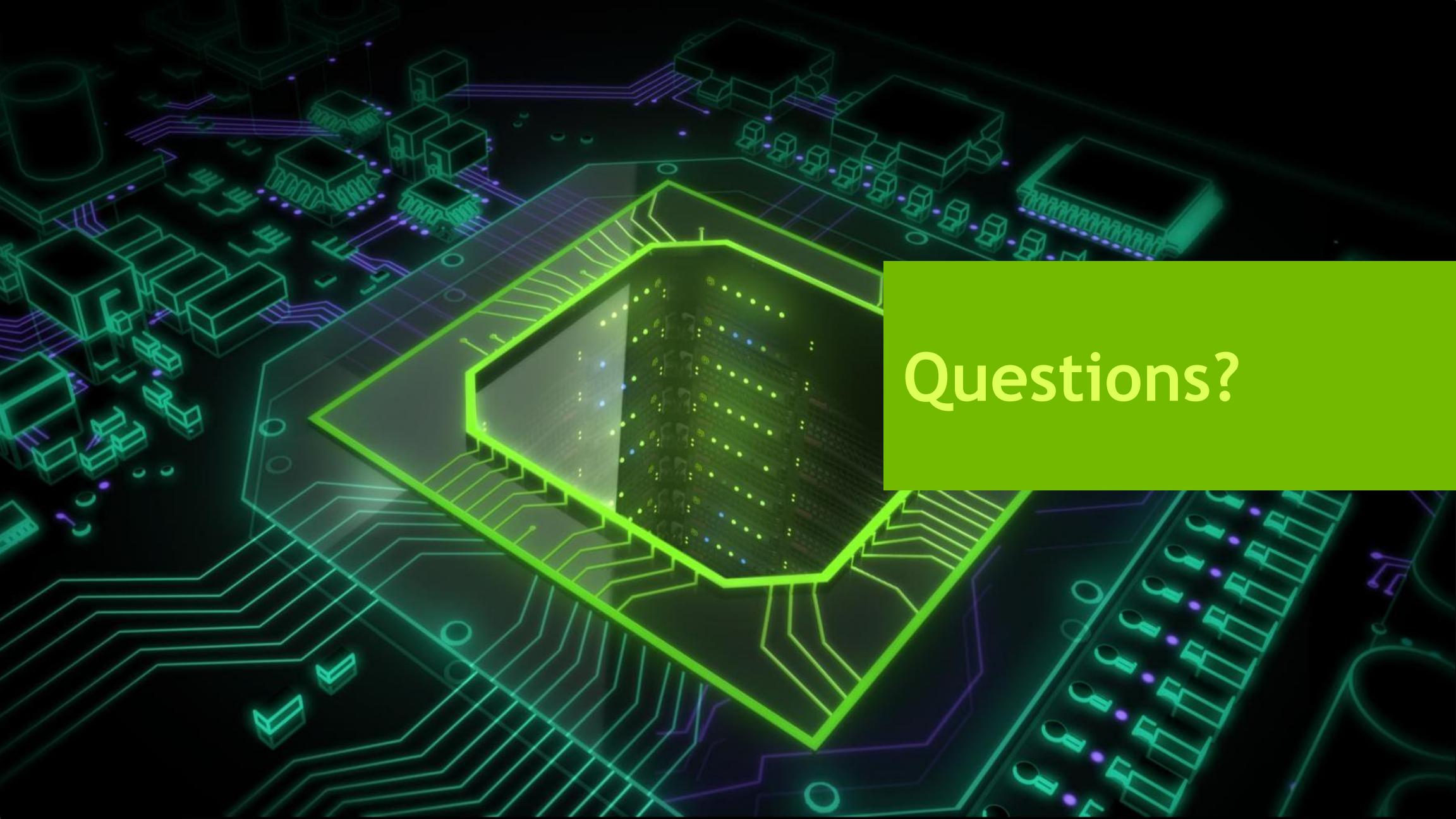
- A kernel is launched as a grid of blocks of threads
 - `blockIdx` and `threadIdx` are 3D
 - We showed only one dimension (x)
- Built-in variables:
 - `threadIdx`
 - `blockIdx`
 - `blockDim`
 - `gridDim`



Textures

- Read-only object
 - Dedicated cache
- Dedicated filtering hardware
(Linear, bilinear, trilinear)
- Addressable as 1D, 2D or 3D
- Out-of-bounds address handling
(Wrap, clamp)



A glowing green cube sits atop a complex circuit board, which is itself set against a dark background with glowing purple lines and components. The cube is illuminated from within, casting a bright glow. It has a smooth, reflective surface and is positioned centrally. The circuit board beneath it is densely packed with various electronic components like resistors, capacitors, and integrated circuits, all connected by a network of glowing purple lines.

Questions?