

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 01

01/22/2020

# Quote of the day

“Human beings only use ten percent of their brains. Ten percent! Can you imagine how much we could accomplish if we used the other sixty percent?”

Ellen DeGeneres, American comedian, television host, actress, writer, and producer [1958 - ]

# And how “Quote of the Day” applies to this class

In most cases, we use 10% of the compute power at our disposal. Ten percent!  
Can you imagine what we could accomplish if we used the other 60 %?  
**That's what ME759 is all about – using the other 60%.**

Dan Negruț, Romanian comedian, television host, actor, writer, and producer [1968 - ]

# Instructor: Dan Negruț

- Polytechnic Institute of Bucharest, Romania
  - B.S. – Aerospace Engineering (1992)
- University of Iowa
  - Ph.D. – Mechanical Engineering (1998)
- MSC.Software
  - Product Development Engineer 1998-2005
- University of Michigan
  - Adjunct Assistant Professor, Dept. of Mathematics (2004)
- Division of Mathematics and Computer Science, Argonne National Laboratory
  - Visiting Scientist 2004-2005, 2006, 2010
- University of Wisconsin-Madison, Joined in Nov. 2005
  - Research Focus: Computational Dynamics
  - Technical lead, Simulation-Based Engineering Lab (<http://sbel.wisc.edu>)

# Before we get started...

- Today
  - 30,000 feet perspective of ME759
    - All sorts of disjoint information; and lots of it
    - Note: everything in blue and underlined, like [syllabus](#), contains a link that you can follow to a relevant doc/page
- Other tidbits:
  - Reading assigned: check the on-line [syllabus](#)
  - First assignment, goes out tomorrow, due on Th, 01/30, at 9 pm
    - Information on homework submission provided in the text of the assignment

# Good to Know...

- **Lecture Time** 11:00-12:15 PM → Mo & Wd & Fr (75 minutes lectures)
- **Office** 4150ME
- **Phone** 608 772 0914
- **E-Mail** [negrut@wisc.edu](mailto:negrut@wisc.edu) (because my last name is Negrut)
- **How people call me** Dan (because my name is Dan)

# Support crew

- TA: Nic Olsen, Department of Mechanical Engineering
  - [nicholas.olsen@wisc.edu](mailto:nicholas.olsen@wisc.edu)



- TA: Lijing Yang, Department of Mechanical Engineering
  - [lyang296@wisc.edu](mailto:lyang296@wisc.edu)



- Sysadmin: Colin Vanden Heuvel
  - Will take care of the hardware & software side of things
  - The person taking care of Euler



# The Course Webpage

- Course website managed under Canvas: <https://canvas.wisc.edu/courses/190689>
- Canvas page contains links to
  - Piazza discussion forum: <https://piazza.com/class/k5crmnymtx95c0>
  - Conferencing tool
  - Drop box
  - Grades
  - Etc.

# Course Related Information

- A link to the “today’s slides” emailed to you 15 mins before the beginning of each lecture
  - “slide deck A”
- Link to PPT slides is also available online at class website (Canvas)
  - “slide deck B”
- The official deck of slides is “slide deck B”
  - Rationale: I correct/augment the slides in the “slide deck A”. What’s in Canvas (“slide deck B”) it’s what you’ll probe on in the comprehensive exam

# Course Related Information

- Syllabus will contain info about
  - Reading assignments ⇐ these are very important
  - Topics we cover
  - Homework due dates
  - Syllabus will likely change to reflect the day-to-day progress
- Both the [syllabus](#) & [course description](#) available at the Canvas course website

# Course Related Information

- Lectures are video recorded. Available online, probably 60 mins after class is over
  - Download the recording here: [https://mediasite.engr.wisc.edu/Mediasite/Catalog/catalogs/me\\_759\\_ccd](https://mediasite.engr.wisc.edu/Mediasite/Catalog/catalogs/me_759_ccd)

# F2F Students: On your presence in the lecture room

- The course capacity filled up in less than 24 hours
- Room assigned: 48 seats
- Only way to take more students was to let them in but ask to not necessarily show up
- If you watch the recording, you are in business

# ME759: Two Sections

- Face-to-face (F2F):
  - 107 students (as of 01/21/2020)
- On-line programs
  - 6 online students
- IMPORTANT semantics aspect:
  - “online students” are students who are registered for an online degree
  - “online students” are not students who prefer to watch the lectures on-line

# Office Hours

- All times CST

- **F2F students**
  - Monday: 1:00 – 2:30 PM (Nic, 4150ME)
  - Tuesday: 12:30 – 2:00 PM (Nic, 4150ME)
  - Wednesday: 2:00 – 3:30 PM (Lijing, 4150ME)
  - Thursday: 12:30 – 2:00 PM (Lijing, 4150ME)
  - Friday: 2:00 – 3:30 PM (Dan, 4150ME)
- **Online students (see NOTE below)**
  - Monday: 7:00 – 8:00 PM (Nic, Canvas)
  - Tuesday: 7:00 – 8:00 PM (Lijing, Canvas)
  - Wednesday: 7:00 – 8:00 (Dan, Canvas)
- **Euler Supercomputer consultation (both F2F & Online students)**

Any work day, by appointment (4150ME or online) with Colin, the Sysadmin

- Notes:

- For online students: on the day of the office hours, please email us by 6:30 PM if you plan to utilize the office hours
- Call or email to arrange for meetings outside office hours

# Diversity, at UW-Madison

- Statements below lifted from <https://diversity.wisc.edu>
- Diversity is a source of strength, creativity, and innovation for UW-Madison
  - *We value the contributions of each person and respect the profound ways their identity, culture, background, experience, status, abilities, and opinion enrich the university community*
  - *We commit ourselves to the pursuit of excellence in teaching, research, outreach, and diversity as inextricably linked goals*
  - *The University of Wisconsin-Madison fulfills its public mission by creating a welcoming and inclusive community for people from every background - people who as students, faculty, and staff serve Wisconsin and the world*

# Rules of Academic Conduct

[read the Course Description for a detailed account of what constitutes cheating]

- You are encouraged to discuss assignments with other class students
  - Post and read posts on Forum
- Getting **verbal** advice and suggestions from anybody when all parties are away from a computer is fine
- copy/paste of non-trivial code is not acceptable
  - Non-trivial = more than a line or so
  - Includes reading someone else's code and then going off to write your own
- Use of third-party libraries that directly implement the solution of a HW/Project is not acceptable unless explicitly asked to do so
- We will use Stanford's [Moss](#) to detect code plagiarism

# Cheating, consequences

- First time
  - Everybody involved gets the assignment score to be zero
  - I will email the advisor[s] of the parties who stood to benefit
- Second time
  - The Dean of your college will be brought into the loop
- Historical data
  - People cut corners more often early on in the semester, when ME759 things are more stressful/chaotic
  - There are about three cases of cheating that we typically come across each semester

# Textbook/Pointers to Info

- No textbook is required, but there are some recommended books/docs:
  - R. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, Prentice Hall, 3rd Edition, 2015
  - NVIDIA, [\*GPU Programming Guide\*](#), version 10.0
  - David B. Kirk and Wen-mei W. Hwu: *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, 2016
  - Jason Sanders and Edward Kandrot: *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley, 2010
  - Peter Pacheco: *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011
  - T. Mattson, et al.: *Patterns for Parallel Programming*, Addison Wesley, 2005

# Another source of info: ME459

- ME459: “Computing Concepts for Applications in Engineering”
  - The less mature brother of ME759
- ME459: learning about basics, getting started in computing in science and engineering
- ME759: using parallel processing, in the pursuit of speed
- There is a 2.5 weeks worth of content overlap between ME459 & ME759
- A lot of reading assigned from the ME459 slides

# ME459: quick overview of topics covered

- Shell programming (bash, sh)
- The Linux command line
- Remote access (of Euler) via ssh
- Scheduling for execution on Euler
- Version control with git
- Elements of C programming
- Building/linking/debugging/profiling
- Build management with CMake
- Instruction Level Parallelism issues
- The memory hierarch
- Caches
- Virtual memory
- Finite precision arithmetic
- Vectorization
- Elements of code optimization
- Elements of operating systems

There is an expectation that you are familiar with the concepts above or are ready to go through some of the ME459 slides and assigned readings

# ME759: Course Related Information

- Course offered on an accelerated track
  - Three lectures per week, each 75 minutes long
- Why accelerated track?
  - Gives us one month to work on a meaningful Final Project (more on this later)
- Last lecture: April 10 or so
  - 28 lectures: Just like a regular semester yet compressed in 2.5 months
- After April 10:
  - Office hours will run as before
  - Homework will continue to be assigned

# Schedule, looking ahead

- No class on the following dates (Dan out of town):
  - February 14, March 30, April 1, April 3
- Won't affect the class – we'll still conclude after 28 lectures, on April 10

# Final grade breakdown

- Homework 45%
- Final Project 25%
- Final Exam 25%
- Course Participation 5%
  

---

- Total 100%

## NOTE:

- Questions related to homework/exam scores must be raised within seven days after receiving the grade

# Assignments

- There will be 12 assignments
  - Assigned on Th, due next Th (one week turnaround)
  - Homework due at 9 PM on Th
- Class demanding because of the time-consuming assignments, particularly early on
  - First three or four assignments: start early, give yourself plenty of time
    - It'll be challenging to set up a process to work on your assignment
    - Use Piazza to ask questions about how to get going on Euler, about the work process

# Assignments

- On the due date, your **git repo** will be pulled at 9 pm to get the material graded by the TAs
- Anything that goes into your repo after 9 pm will not be picked up
  - Another way of saying that “no late work will matter”
- Two assignments with lowest scores will be dropped when computing your final HW score

# Assignments: Nuts and Bolts

- Each of you will have own **git private** repo. Everything will be under git-enabled version control
  - Setting up your git repo: part of the first assignment, instructions provided to you
- TAs and instructor have access to your private git repo
- More instruction provided in each assignment

# Assignments: Nuts and Bolts

- Grader will run a script at 9:0 PM that will pull your assignment solution.
  - Files that are not pushed prior to 9:00 PM are not going to be picked up by the scripted `git pull` operation
- Grader will check solutions on Linux, running on the Euler cluster

# Re-emphasizing, the “late homework” issue

- There are more than 100 students in this class
- We will not offer exceptions for late work. The only way possible to keep a level playing field
- The mechanism in place for accommodating this: you have to assignments that you don't do
- If you must change due date of homework/project: contact us at least 3 weeks in advance
  - Examples, what would fly: attending a conference, deadline for research project, etc.
  - In most cases, homework deadline will be moved early for you (and you'll get the assignment early too)

# Hitting the Ground Running

- First assignment goes out tomorrow (emailed to you)
- Available on the class website
- Due in one week
  - Th, January 30, at 9 PM
- Assignment is a C programming warm up + learning the Euler ropes

# Exams-related issues

- No midterm exam
- Final exam will be comprehensive – April 15, at 7 pm
- There will be a review session for the Final Exam,
  - April 14, at 7 pm
  - Organized online, through Canvas
  - Review session will be recorded

# 759 Final Project

- Final Project Proposal due on March 27 (9 PM, in Canvas)
  - You'll receive feedback by April 3
- Final Project due at the time when the Final Exam starts
  - Exact date TBA

# Final Project

- Final Project (accounts for 25% of final grade)
- Your work on Final Project to start no later than April 4:
  - It is an individual project or outcome of a 2- or 3-student team
  - You choose a problem that suites your research or interests
  - I encourage you to tackle a meaningful problem
    - Attempt to solve a useful problem rather than a problem that you are confident that you can solve
    - Projects that are not successful are ok, provided you aim high enough and demonstrate good work

# Example, Previous Final Project: GPU Scheduling on a Cluster

- 2011 Final Project
- Used today at the Large Hadron Collider (LHC) – CERN
  - One petabyte of data processed every day
    - The equivalent of around 210,000 DVDs
  - The center hosts 11,000 servers with 100,000 processor cores
  - Some 6000 changes in the database are performed every second
  - The Grid runs more than two million jobs per day
- For some jobs, LHC used the Condor Scheduler
  - Started at UW-Madison, Professor Miron Livny
  - Two Condor students took 759 back in the day...

# Final Project: Default Options

- In case you don't have any research topic that you could use as a vehicle for this project I'll provide at least two default choices
- Project 1: work with a granular dynamics code
  - Profile existing code
  - Improve performance of the implementation via parallel computing
- Project 2: work on fluid dynamics code
  - Profile existing code
  - Improve performance of the implementation via parallel computing
- These two projects are ongoing work in my lab

# Class Participation

- Accounts for 5% of final grade
  - Participate on Piazza
  - Get involved in what goes on during lecture (if you choose to show up)
- Forum: a quick way to get your questions answered by TA/instructor/other ME759 colleagues
- **Piazza forum** was critical in the past – very useful resource

# Examples, Piazza related

- Post on Piazza your approach to working on your Windows box and interacting with Euler, which runs Linux
  - Remote connection tools/approaches
  - Automatic authentication for login
  - File access (mounting Euler as a drive in Windows Explorer)
  - Shell customization
- How to clone the git repo
- How to use a debugger other than gdb
- Etc.

# Class Participation, Examples

- To all you good citizens out there: please answer questions on Piazza
- Ideally, you'd beat the TAs and instructor to suggesting a fix or answering a question
  - Don't worry if you only get it 75% right – no problem...

# Scores and Grades

<u>Score</u>	<u>Grade</u>
93-100	A
86-91	AB
78-85	B
70-77	BC
60-69	C
50-59	D

- Grading will not be done on a curve
- Final score will be rounded to the nearest integer prior to having a letter assigned
  - Example:
    - 85.59 becomes AB
    - 85.27 becomes B

# Staying in Touch...

- Email me only if you have a personal problem
  - Examples:
    - Good: Schedule a one-on-one meeting outside office hours
    - Bad: Asking me clarifications on Problem 2 of the current assignment (this needs to be on the Forum)
    - Bad: telling me that you can't compile your code (this should also go to the Forum)
- Any course-related question should be posted on the Forum
  - We continuously monitor the Forum
  - If you can answer a Forum post, please do so (counts towards your 5% class participation and helps me as well)
  - Keeps all of us on the same page
- The forum is **\*\*very\*\* useful**

# Prerequisites

- This is a high-level graduate class in a fluid topic
- Familiarity with C is expected
  - You can probably be fine if you are a friend of Java
- You are expected to be at ease with handling ME459 concepts
- Modest programming skills and a perseverant spirit are necessary
  - Understanding pointers
  - Being able to wrestle with a compile error on your own
  - You have heard of a debugger
  - You have heard of a profiler

# We'll speak C in this class. And some entry level C++

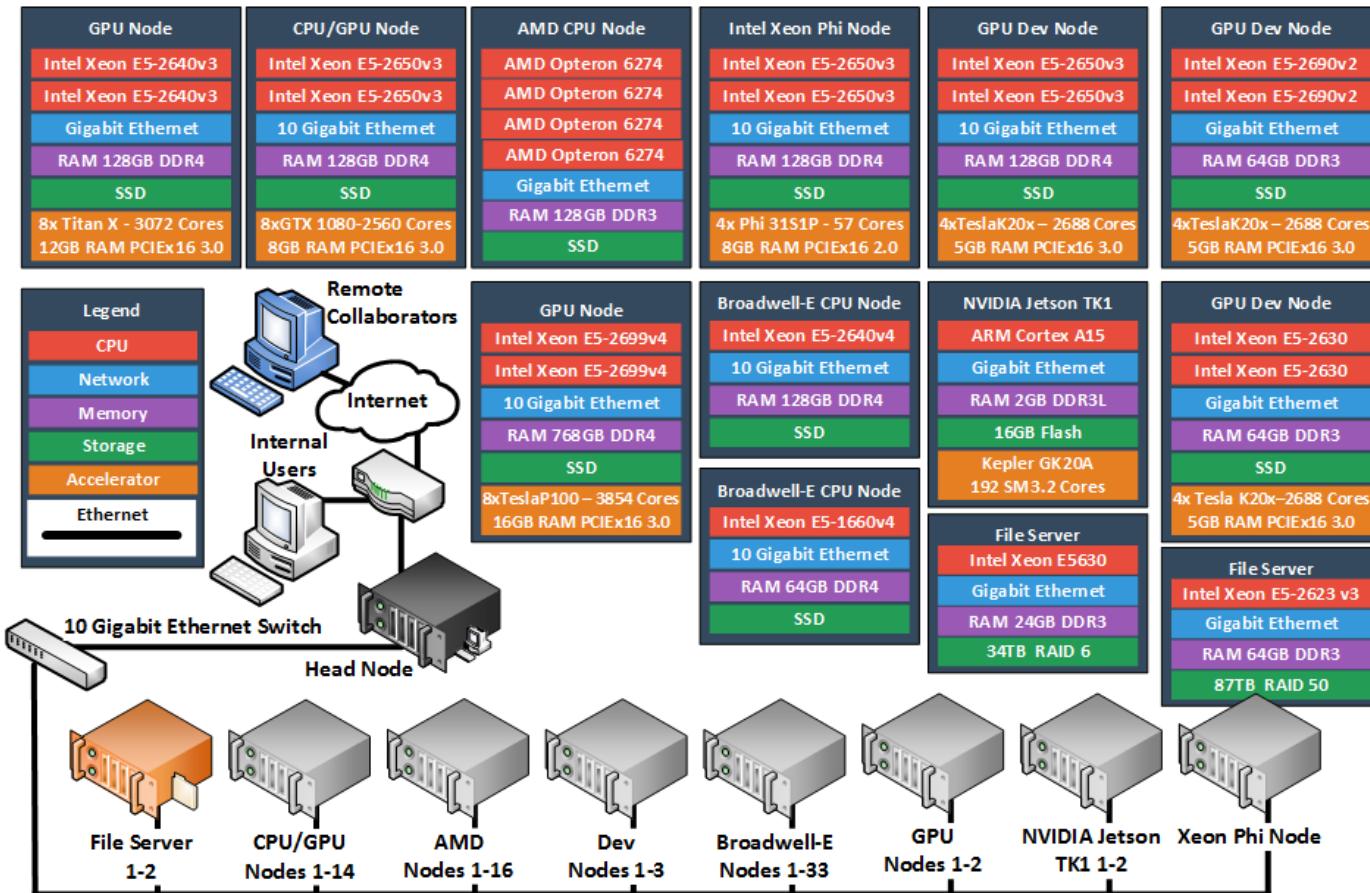
- ME759: One-semester journey in the pursuit of speed
- C programming is what's being used for low level stuff. Because it's fast
  - The Linux kernel is written in C. And the Windows kernel
- A low level language like C is not friendly but the sky is the limit: you can do as you wish
  - We'll use entry level C++ for some features that are clunky in C
- CUDA, OpenMP, MPI: they cater to the C programmer (and FORTRAN too)

# Hardware Aspects

- The course designed to leverage a dedicated CPU/GPU cluster
  - Called Euler
- Each student receives an individual account that will be used for
  - GPU computing
  - OpenMP multi-core computing
  - MPI-enabled parallel computing
  - Version control (through `git`)
- Advice: if possible, do all the programming on a [local](#) or [CAE](#) machine. Move to Euler for “production” runs
  - Do the homework on your favorite laptop or desktop
    - Caveat: make sure your code runs on Euler since there’s where the TA will run your code

# The Euler Cluster [pic is obsolete, but gives you an idea]

<http://wacc.wisc.edu/infrastructure/>



# Example, an Euler node

- IBM POWER 9 architecture node – one year old
  - \$54,000
- 128 logical CPU cores via 4-way SMT (Symmetric Multi-Threading)
- Four NVIDIA Volta V100 GPUs connected to host using NVLink 2.0 interconnect
  - Each GPU has 32GB of memory
- Host-to-Device (CPU↔GPU) data transfers over the NVLink interconnect allows programs to move data at nearly 25x the speed of the PCIe 3.0 bus on Intel x86 systems
  - CPU-to-GPU Bandwidth: approx. 300 GB/s
- About 30,000 billion double precision operations per second

# Supercomputer [mis]use

- Euler: over 200 accounts each year
- Crashing Euler is not good
  - Stuff is due: assignments, project deadlines, research, papers, conference presentations, etc.
- Gold rule: do not run your code on the Euler \*head\* node
  - “head node” – where you land when you ssh into Euler
  - Head node used to schedule how the entire supercomputer works
  - Through Slurm scheduling you will get your job scheduled for execution
    - Execution will take place on a Euler node
      - The said Euler node is chosen by the scheduler, in a process transparent to you

# Supercomputer [mis]use

- Your account will be temporarily blocked if you run code on the head node
  - You'll need to email Dan and Euler sysadmin to unblock your account
- Pay attention, particularly right before the homework is due
  - That's the worst time to get your account frozen

# Software Issues

- Operating system of choice: Linux
  - Euler runs CentOS Linux release 8
- Miscellaneous libraries/releases:
  - CUDA: 10.2
  - OpenMPI: 3.1
  - OpenMP: 4.5
- Compilers and such
  - GCC 8.2.1 for most of your homework
  - clang 9.0.0 and PGI 19.10 for some OpenMP/OpenACC jobs

# Software Issues

- For build management – we'll rely on makefiles generated with CMake
  - Scripts will be available to you in order to facilitate compile/link/debug/profile process
  - CMake: See assigned reading in syllabus
- Debugging and profiling tools:
  - gdb: debugger under Linux
  - cuda-gdb: debugger for CUDA applications running on the GPU
  - NVIDIA Profiler: Nsight
- Most of these tools are embedded in Visual Studio and Eclipse
  - OK to work under Windows, yet make sure your code compiles/runs on Euler before submitting

# Course Emphasis

- There are multiple choices when it comes to implementing parallelism
  - PThreads, Intel's TBB, OpenMP, MPI, Ct, Cilk, CUDA, etc.
- Course focuses on parallelism enabled by
  - CUDA running on Graphics Processing Unit (GPU) cards, for fine-grain parallelism
  - OpenMP standard, aimed both at fine and coarse-grain parallelism
  - Message Passing Interface (MPI) standard, aimed at coarse grain parallelism

# Course Objectives

- Get familiar with today's software and hardware that can speed up your code
  - Mostly done through parallel computing, at multiple levels
- Recognize applications/problems that can draw on advanced computing
- Gain basic skills that will help you map these applications onto a parallel computing hardware/software stack
  - Write code, build, link, run, debug, profile
- Introduce basic software design patterns for parallel computing
- Expand your mindset when it comes to writing software for scientific computing

# Course Objectives [Cntd.]

- What I'll try to accomplish
  - Provide enough information for you to start writing software that can leverage parallel computing to hopefully reduce the amount of time required by your simulations to complete
- I will not attempt to...
  - Propose new parallel computing languages or language features
  - Establish how compilers should support advanced computing
  - Etc.
- To summarize,
  - We are interested in this stuff since we are consumers of parallel computing

# How we'll go about it...

- Before talking about how to write software to run on a piece of hardware, we'll learn something about that hardware asset
- “learn something” – pick up those things we need to know as *\*programmers\**
  - Level of detail provided not enough to be useful to people whose interests are in hardware
- Focus not on low level hardware detail
  - Dedicated courses already available
    - E C E 353: Introduction to Microprocessor Systems
    - E C E 354: Machine Organization and Programming
    - E C E 552: Introduction to Computer Architecture
    - E C E 752: Advanced Computer Architecture I

# “High Performance Computing for Applications in Engineering.” Why This Title?

- Computer Science: ISA, Limits to Instruction Level Parallelism and Multithreading, Speculative Execution, Pipelining, Memory Hierarchy, Memory Models, Cache Coherence, etc.
  - Long story short: how should a processor be built?
- Electrical Engineering: how will we build the processor that the CS colleagues have in mind?
  - Lots of microarchitecture issues
- Science & Engineering: we’re faced with solving large problems
  - This class: Use parallel computing to solve these large problems

# The Big Picture

- “Big Data”: everybody’s talking about it
- Two parts to “Big Data”
  - Producing it (computer simulation)
  - Processing it (data analysis)
- “In pursuit of speed” – this is the mantra of this course, that’s all ME759 attempts to offer

# Overview of Material Covered

- Warm up: Basic concepts related to sequential computing (review, some ME459 concepts)
- Overview of parallel computation paradigms and supporting hardware/software
- GPU computing and the CUDA programming model
- OpenMP programming
- MPI programming
- Heterogeneous parallel computing with CUDA and/or OpenMP and/or MPI

# At the Beginning of the Road...

- This field (advanced computing) is very much in flux, high rate of change
- There will be issues that I don't know and/or don't understand
  - I always end up learning something new, in most cases with your help
- There might be questions that you ask for which I don't have an answer
  - I'll follow up on these and get back with you (on the Forum)
- Most of my statements should be qualified by the preamble “*On most systems...*”, or “*By and large...*”
  - You can always find exceptions, relatively few true blanket statements that can be made

# The ME759, how I see it

- Class is time consuming because of the assignments
- The Piazza forum is a place where you ME759 learn
- There will be a lot of “googling” for the class
  - Dealing w/ situations when you’ll have to address pesky hurdles on your own

# What ME759 offers

- Access to top of the line hardware
- Exposure to new programming techniques for speeding up computing in your work/research
  - In some cases, this might be a game changer
  - Winning ticket: top notch hardware + new programming approaches
- Be bold (the opportunity presents itself w/ the Final Project)

# The Price of reaching 1 Gflop/second

- 1961:
  - Combine 17 million IBM-1620 computers
  - At \$64K apiece, when adjusted for inflation, this would cost \$8.3 trillion
- 2000:
  - About \$1,000
  - Kentucky Linux Athlon Testbed, a 64+2 node Beowulf cluster built by the University of Kentucky COE in 2000
- 2015:
  - 8 cents
  - Celeron G1830 & Radeon R9 295X2 System
- 2018: 6 cents
  - NVIDIA GTX 1080 Ti @ \$699 apiece

# GPU Acceleration: Pascal P100

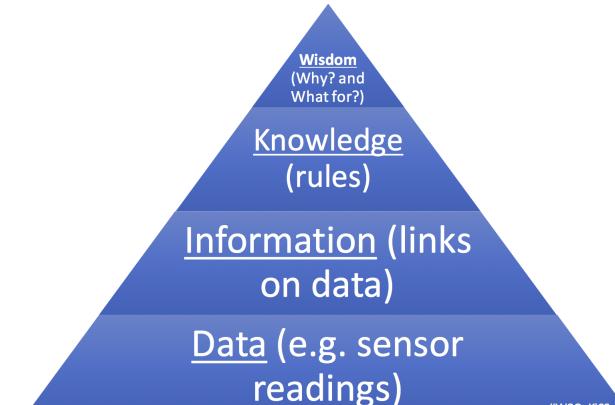
- Pascal P100: **4,670,000,000,000** double precision FMA operations per second
- Very high bandwidth: 720 GB/s
- Amazing at data processing (data parallelism)

# GPU Acceleration: Volta V100

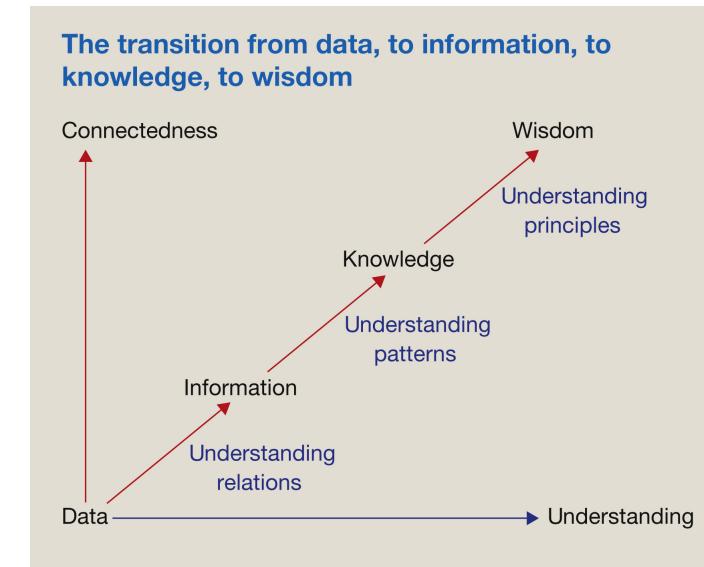
- Volta V100: 7,800,000,000,000 double precision FMA operations per second
- Very high bandwidth: 900 GB/s
- Amazing at data processing (data parallelism)

# Computing a means to a goal: from information to wisdom

- Lots of data
  - Sensing (ubiquitous sensing → lots of data)
  - Data collection (customers, industrial processes, stock market, sports, etc.)
  - Generated through simulation
- To be useful, data requires processing
  - This is where ME759 is meant to help



[<https://www.w3.org/community/kiss/files/2017/02/DIKW.png>]→



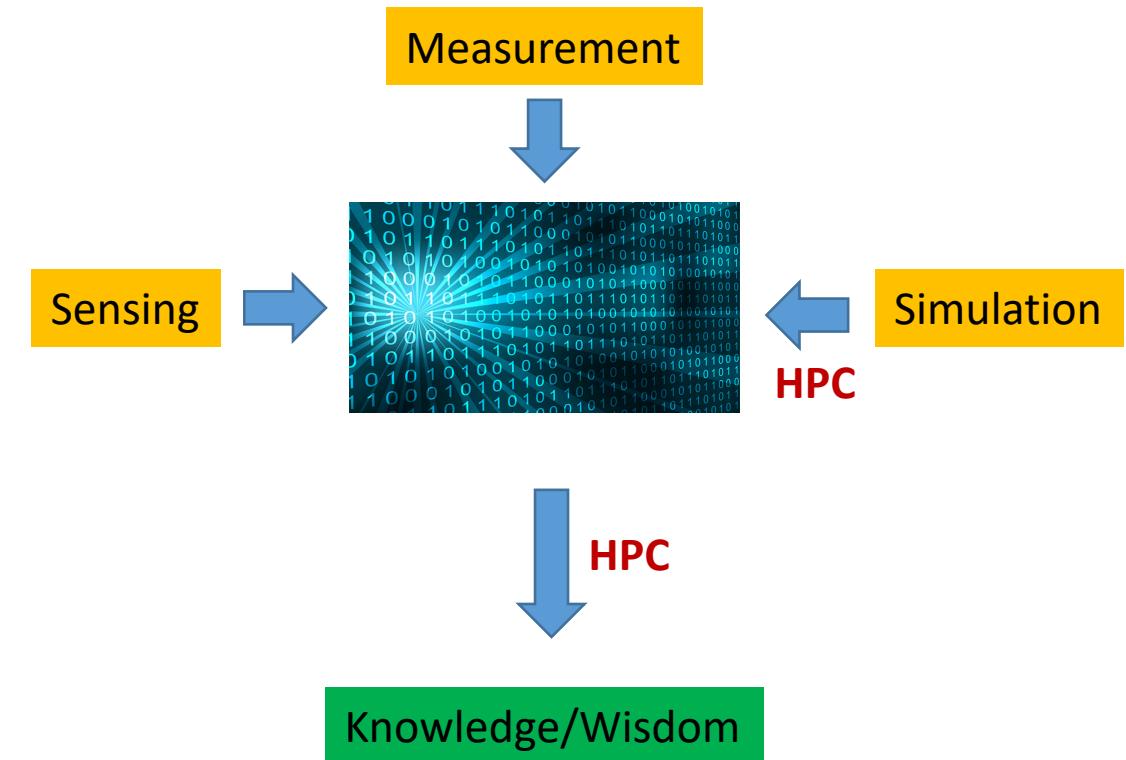
[<https://www.sciencedirect.com/science/article/pii/S1472029916301813>]→

# High Performance Computing (HPC), and where useful

- People from Oakridge National Lab, Argonne National Lab, Los Alamos National Lab:

- HPC: The activity of using, for instance, 10,000 nodes to run one very large application
  - This application can't be run on one machine: not enough memory space, for instance

- People from ME759 and many other folks
  - HPC: The ability to tap into the resources of commodity hardware to extract speed for data generation and/or processing
    - Unlikely: You might have 10,000 nodes to use
    - Likely: you have one GPU, or four multi-core CPUs on one motherboard, or perhaps 10 nodes in a cluster



# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

## Lecture 02

01/24/2020

# Quote of the day

“Sometimes I'll start a sentence and I don't even know where it's going. I just hope I find it along the way.”

Michael Scott, Regional Manager, Dunder Mifflin Paper Company, Inc.

## Second quote of the day

“Would I rather be feared or loved? Easy. Both. I want people to be afraid of how much they love me.”

Michael Scott, Regional Manager, Dunder Mifflin Paper Company, Inc.

# Before we get started...

- Last time: Overview of the course
  - ME759:
    - We'll learn things about hardware
    - We'll learn things about software environments that help us tap into that hardware
    - Use knowledge about software and hardware to run our codes faster
    - The first five-six assignments are hard, stay focused up front – things will get steady state in the second part of the semester
- Today
  - A first step towards starting to consider the hardware that our code will run on
  - It's not only the software that's important. For speed, we must keep in mind the hardware that our software will run on
  - The bottom line: it takes to tango – the software half & the hardware half
- Other tidbits:
  - Important reading assigned today, consult the syllabus:
    - Euler use policy
    - Slurm use and batch scripting for Slurm with sbatch
    - The “module” utility
    - Most importantly, read the material on C programming. And then read it again. Make sure you get “pointers”
  - First assignment went out yesterday, due on Th at 9 pm

# The Hardware-Software Interplay

# This segment's purpose

- Understand the basics of how the hardware works for the purpose of writing better software
- Immediate focus: discuss how a sequence of **instructions** processes a bunch of **data**

# From C Code to Machine Instructions (1/4)

- Your program is the result of a compile step in which a source file is turned into an executable
  - Your [lines of code](#), converted into [machine instructions](#)
- What's an "executable"?
  - Sequence of machine instructions that can be processed, one by one, by the processor
- Execution concludes; i.e., your program done, when last instruction was taken care of

```
#include <stdio.h>
/* Simplest C Program */
int main(int argc, char **argv) {
    printf("Hello World!\n");
    return 0;
}
```



```
euler ~/ME759> gcc -o helloWorld helloWorld.cpp
euler ~/ME759> ls
helloWorld  helloWorld.cxx
euler ~/ME759> ./helloWorld
Hello World!
euler ~/ME759>
```

# From C Code to Machine Instructions (2/4)

- Punch line: There is a difference between a line of code and a machine instruction

- Example:

- Line of C code: `a[4] = delta + a[3]; //line of C code`

- MIPS assembly code generated by the compiler (three instructions):

```
lw $t0, 12($s2) # reg $t0 gets value stored 12 bytes from address in $s2  
add $t0, $s4, $t0 # reg $t0 gets the sum of values stored in reg $s4 and reg $t0  
sw $t0, 16($s2) # store what's in $t0 at mem location 16 bytes from address in $s2
```

- Set of three corresponding MIPS instructions produced by the compiler:

```
100011100100100000000000000000001100  
00000010100010000100000000100000  
1010111001001000000000000000010000
```

# From C Code to Machine Instructions (3/4)

- **C code** – what you write to implement an algorithm
- **Intermediate Representation** – compiler converts your code to a specialized representation that it can optimize
- **Assembly code** – another intermediate step for compiler, humans can still read it
  - Spend 5 mins here: <https://godbolt.org>
- **Machine code/Instructions** – what the assembly code gets translated into and what the control unit (CU) digests
  - Machine code: what you see in an editor (`notepad`, `vim`, etc.) if you open an executable file (basically gibberish)
  - There is a **direct** correspondence between a line of assembly code and instructions
    - (this is often one-to-one; particularly true for certain processors)

# From C Code to Machine Instructions (4/4)

- Observations:
  - The compiler goes from C code to machine instructions without producing anything else (unless you ask for it)
    - That is, no assembly is saved for you to peek at (again, unless you ask the compiler for it)
  - Back in the day, people programmed computers via assembly code
  - Today writing assembly code done only for critical parts of a program by people who want to highly optimize the execution and choose to overrule the compiler

# More on the “assembly instructions” side of things

- We'll not cover assembly in ME759
- If you really are into computing though, you should learn how to read “assembly code”
- Why?
  - In critical cases you might want to check that the compiler does what you think it should do
- Examples: you optimize the code with `-O3` and want to confirm that
  - A for-loop got unrolled to reduce the number of jump conditions and improve changes for pipelining
  - A function got inlined to avoid a pair of jumps that ruin pipelining

Example: the same C code  
leads to different assembly code (and different  
set of machine instructions, not shown here)

```
int main(){
    const double fctr = 3.14/180.0;
    double a = 60.0;
    double b = 120.0;
    double c;
    c = fctr*(a + b);
    return 0;
}
```

C code

x86 ISA

```
call    __main
        fild   LC0
        fstpl -40(%ebp)
        fild   LC1
        fstpl -32(%ebp)
        fild   LC2
        fstpl -24(%ebp)
        fild   -32(%ebp)
        faddl -24(%ebp)
        fild   LC0
        fmulp %st,
        %st(1)      fstpl -16(%ebp)
                    movl  $0, %eax
                    addl  $36, %esp
                    popl  %ecx
                    popl  %ebp
                    leal   -4(%ecx),
        %esp
        ret
LC0:   .long 387883269
        .long 1066524452
        .align 8
LC1:   .long 0
        .long 1078853632
        .align 8
LC2:
```

Legend:  
**fild** load floating point  
**fmlulp** multiply & pop register  
**stack**  
**movl** move from one register to another

Intermediate  
Representation +  
Optimization

```
main:
0, gp= 8           .frame    $fp,48,$31    # vars= 32, regs= 1/0, args=
                  .mask     0x40000000,-4
                  .fmask    0x00000000,0
                  .set      noreorder
                  .set      nomacro
                  addiu   $sp,$sp,-48
                  sw      $fp,44($sp)
                  move   $fp,$sp
                  lui     $2,%hi($LC0)
                  lwc1
...
mul.d  $f0,$f2,$f0
swc1   $f0,32($fp)
swc1   $f1,36($fp)
move   $2,$0
move   $sp,$fp
lw     $fp,44($sp)
addiu $sp,$sp,48
j      $31 MIPS ISA
$LC0:
...
$LC1:
...
$LC2:
...
.ident "GCC: (Gentoo 4.6.3 p1.6, pie-0.5.2)
```

Legend:  
**sw** store word  
**addiu** add immediate unsigned  
**mul.d** multiply (double)

# fldr, sw, addiu, lw, mul.d, ... - What is this???

- There is a limited number of stipulations that the CU understands and thus can take care of
- This set of commands/stipulations makes up the **Instruction Set Architecture** (ISA)

# Instruction Set Architecture (ISA)

- The same line of C code can lead to a different set of instructions on two different computers
- This is so because two CPUs might implement two different ISAs
- ISA: defines a “vocabulary” subsequently used by the compiler to specify the actions of a processor

# ISA: Two more important schools of thought

- One school of thought: promotes the **RISC** paradigm
- Second school of thought: promotes the **CISC** paradigm
- RISC or CISC? Which one is better?
  - Topic of perpetual debate, like Bud or Miller Draft

# The RISC ISA flavor

- RISC Architecture – Reduced Instruction Set Computing Architecture
  - An instruction coded into a **fixed** set of bits (used to be 32, now it's 64)
  - Instructions always operate on registers, load / store is separate
  - Promoted by Stanford University and UC Berkeley, in the form of the MIPS and SPARC architectures
    - Nowadays, Arm's RISC-based CPUs power mobile devices everywhere
      - Implementations by NVIDIA, Samsung, Qualcomm, TI, and many more
      - Somewhere between 8 and 10 billion chips based on ARM manufactured annually
    - IBM also created its own RISC variant which is often used in HPC: POWER ISA

# The CISC ISA flavor

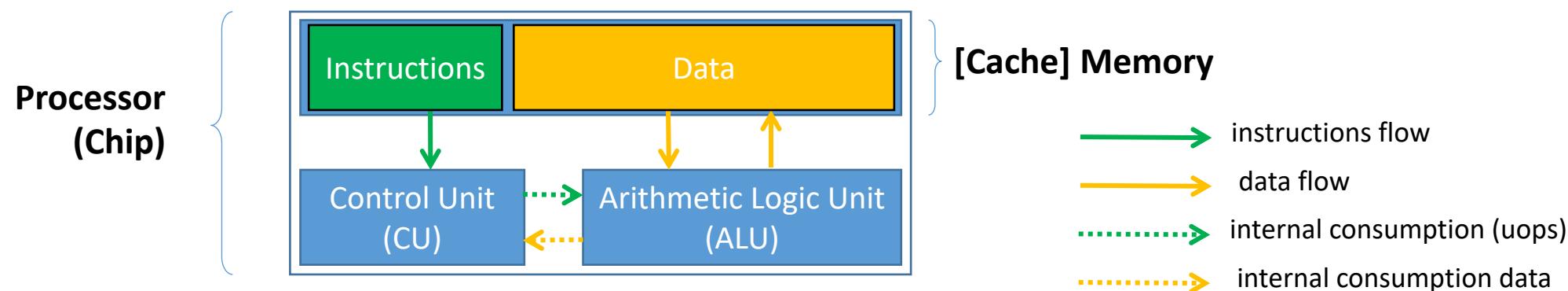
- CISC Architecture – Complex Instruction Set Computing Architecture
  - Instructions have **various lengths**
  - Intel's X86 is the most common example
  - Promoted by Intel and subsequently embraced & augmented by AMD
    - Used in laptops, desktops, workstations, servers

# The main ISA flavors: RISC vs. CISC

- RISC simpler to comprehend, provision for, and work with
- CISC is more expressive
- Decoding CISC instructions is not trivial and eats up power
- A CISC instruction is usually broken down into several micro-operations (uops)
- CISC Architectures: flexibility at the price of complexity and power
  - A more intricate ISA, leads to complex microarchitecture (circuitry, that is) & eats up more power

# Computing requires instructions and data

- In terms of storage and movement, there is no distinction between **data** and **instructions**
  - Instructions are fetched & decoded & executed (more on this later)
  - Data is used to produce results according to rules specified by the instructions
- A piece of data: stored in memory as a string of 0 and 1 bits (8 bits, or 16, or 64, etc. – can even be user defined)
- An instruction: stored in memory as a string of 0 and 1 bits (often 32 or 64 bits – kind of fixed, you don't have a say)



- NOTE: the “data & instructions share same storage and pathways” paradigm formalized by von Neumann in late 1940s
  - The “von Neumann model” (also known as the Princeton model)

# The FDX Cycle

[New Topic: pertains to “what happens with an instruction?” question]

- FDX stands for Fetch-Decode-Execute
- FDX is what keeps the CU and ALU busy
  - FDX is done for instruction after instruction until program completes
- **Fetch:** an instruction is fetched from memory
  - Might look something like this (on 32 bits, MIPS, **lw \$t0, 12(\$s2)**):  
100011100100100000000000000000001100
- **Decode:** this string of 1s and 0s are decoded by the CU
  - Example: here's an "I" (eye) type instruction, made up of four fields

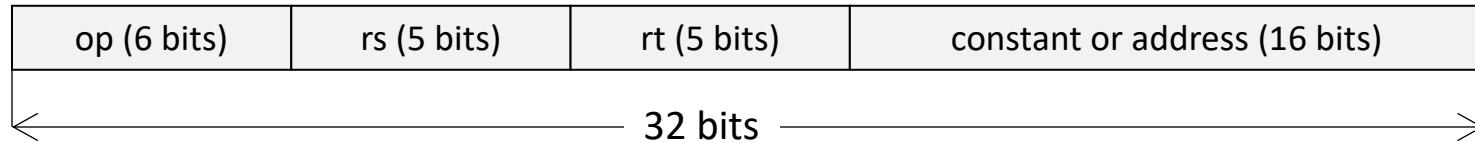
op (6 bits)	rs (5 bits)	rt (5 bits)	constant or address (16 bits)
-------------	-------------	-------------	-------------------------------

A horizontal line with arrows at both ends spans the width of the four fields. Below the line is the text "32 bits".
- **Execute:** once all data (operands) available, instruction is executed
  - The execution can take several cycles of the CPU [more later]

# Decoding: Instructions Types

- The main types of instructions in MIPS ISA:
  - Type I (immediate)
  - Type R (register)
  - Type J (jump)
- Two more flavors for floating point operands
  - FI – like I but for floating point numbers
  - FR – like R but for floating point numbers

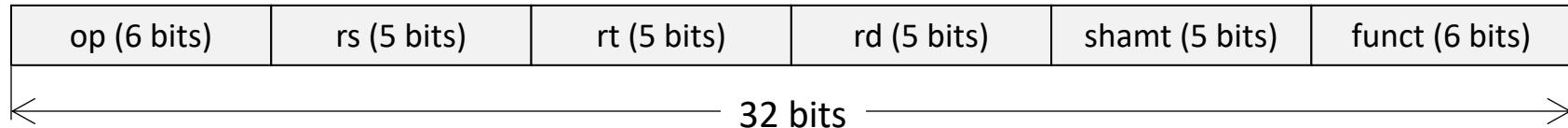
# Type I (MIPS ISA)



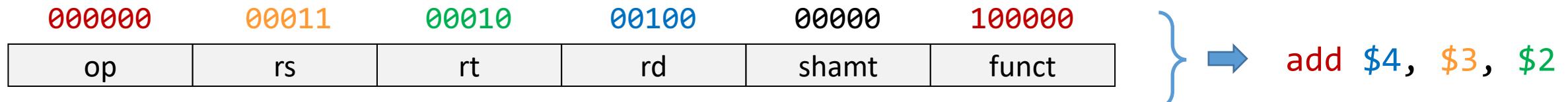
- The first six bits encode the basic operation to be completed (opcode)
- The next group of five bits: in which register the first operand is stored
- The subsequent group of five bits: the target register, rt
- The 16 bit “immediate” value, usually used as the offset value in various instructions
  - “Immediate” means that there is no need to read other registers or jump through other hoops. Immediately good to go
- Example:



# Type R (MIPS ISA)

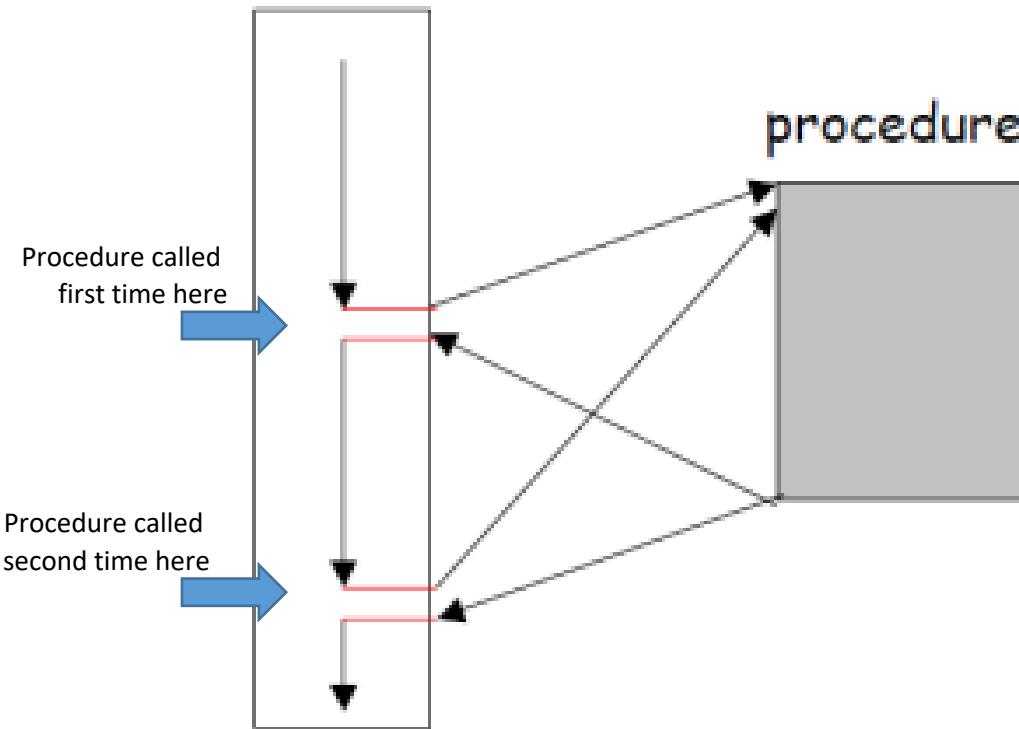


- Type R has the same first three fields op, rs, rt like I-type
- Packs three additional fields:
  - Five bit rd field (register destination)
  - Five bit shamt field (shift amount)
  - Six bit funct field, which is a function code that further qualifies the opcode
- Example:



# Type J (MIPS ISA)

Execution  
instructions



[<http://www.divms.uiowa.edu/~ghosh/1-24-06.pdf>]→



- Type J has an op field and an address field
- Jump instruction has a **word** address, **not an offset**
  - A 26-bit address field allows for jumps to any address between 0 and  $2^{26}-1$
- Longer jumps use the “jump register” instruction:  
`jr $ra # jump to address in register $ra`

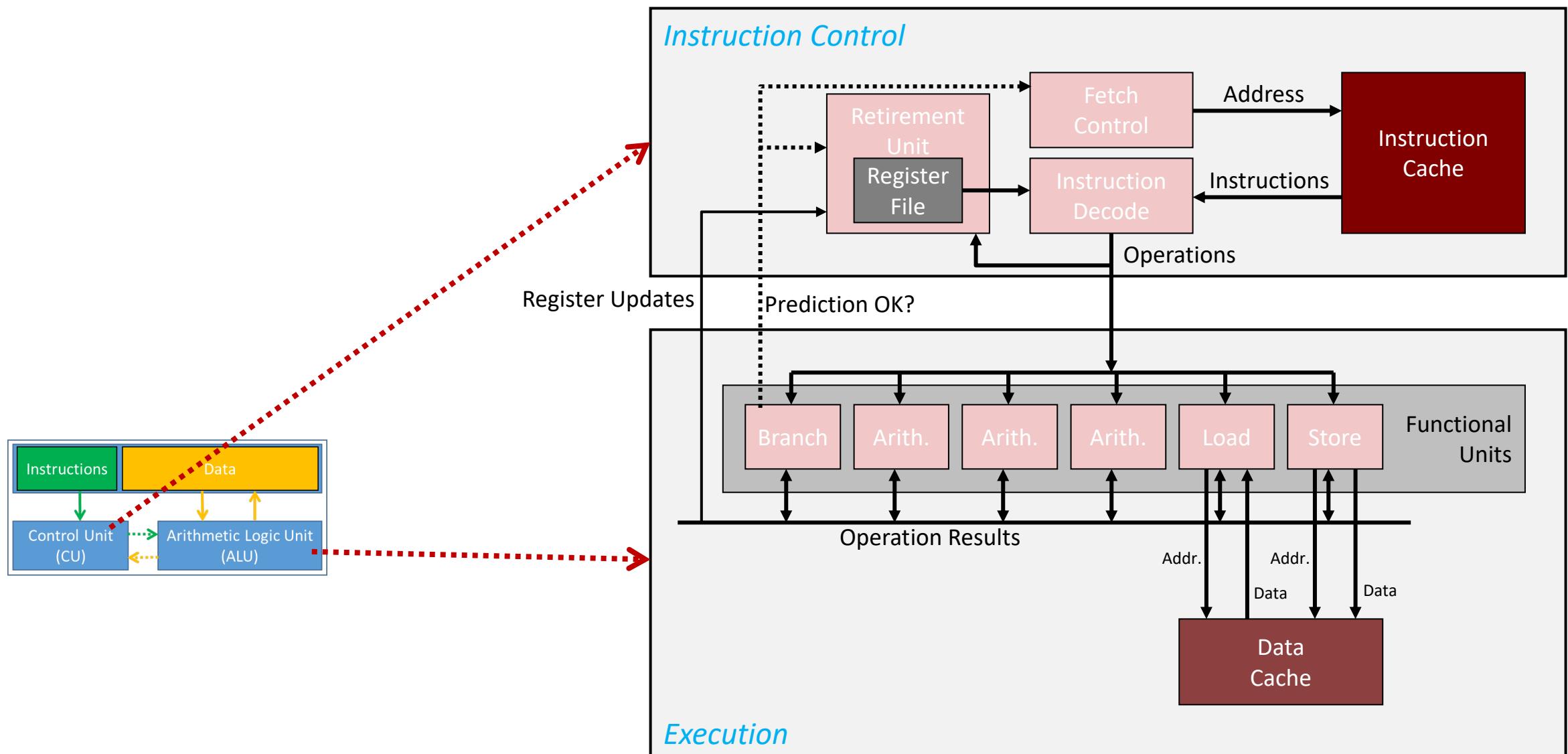
# Picking up the instruction type...

- How does the CU figure out what instruction type it deals with?
  - Is it I, R, or J?
  - This information is baked into the operation code (opcode)

# Instruction Set Architecture vs. Chip Microarchitecture

- ISA – defines a vocabulary
  - Specifies what a processor should be able to do
    - Load, store, jump on less than, etc.
- Microarchitecture – how the silicon is organized to implement the vocabulary embodied by the ISA
- Reflect on this: Intel and AMD both use the x86 ISA. However, they work off different microarchitectures

# Schematic, middle-of-the-road CPU Microarchitecture



# The CPU's Control Unit (CU)

- The CU controls the “datapath” (i.e., the functional units + registers + buses)
- CU has several duties, such as
  - Bringing in the next instruction from memory
  - Decoding the instruction (possibly breaking it up into uops)
  - Engaging in actions to enable “instruction level parallelism” (ILP, more later)
- CU manages/coordinates based on information encoded in the instruction at hand

# The CPU's Arithmetic Logic Unit (ALU)

- Made up of a bunch of so called “execution units” or “functional units”
- In charge of executing arithmetic and load/store operations
- Examples:
  - Arithmetic multiply/add unit (for integers)
  - Arithmetic multiply/add unit (for floats)
  - Special function unit
    - Called upon to compute sin, cos, exp, log, etc.
  - Etc.

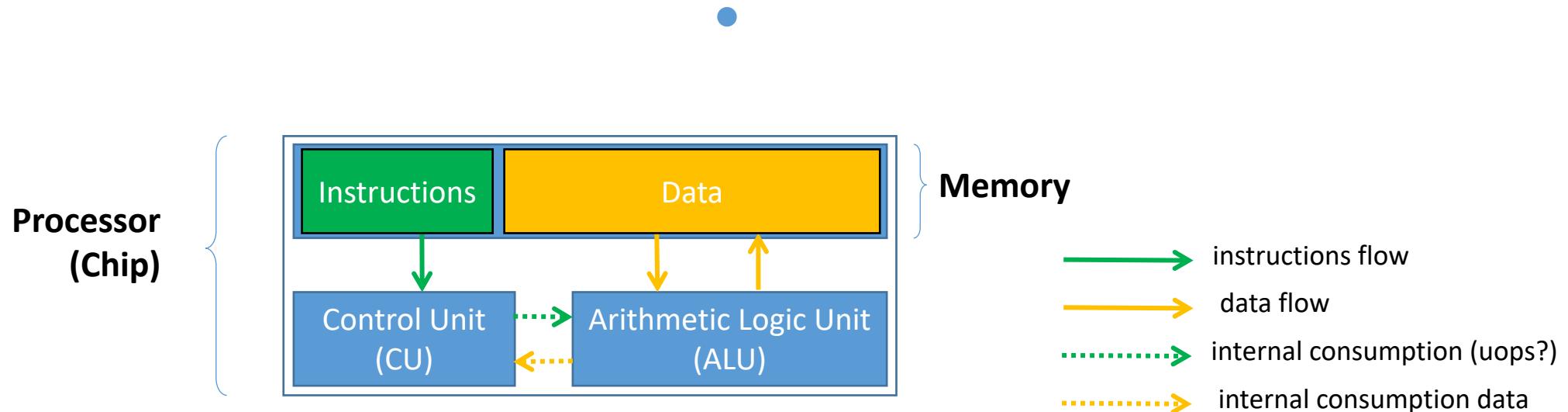
# FDX Cycle – The Execution Part: It All Boils Down to Transistors...

- How does the magic happen?
  - Transistors are organized into complex logical units capable of performing simple operations
  - More transistors increase opportunities for building/implementing in silicon additional functional units that can operate at the same time towards a shared goal

# Number of Transistors, on GPUs [NVIDIA architectures]

- Fermi circ. 2010:
  - 40 nm technology
  - Up to **3 billion** transistors → about 500 scalar processors, 0.5 d.p.Tflops
- Kepler circ. 2012:
  - 28 nm technology
  - Chips w/ **7 billion** transistors → about 2800 scalar processors, 1.5 d.p. Tflops
- Maxwell 2014-2016
  - 28 nm technology
  - Chips w/ **8 billion** transistors → 3072 scalar processors, 6.1 s.p. Tflops
- Pascal 2016-2017
  - 16 nm technology
  - Chips w/ **12 billion** transistors → 3584 scalar processors, 10.6 s.p. Tflops
- Turing 2018 –
  - 12 nm technology
  - Chips w/ **19 billion** transistors → 4608 scalar processors, 16 s.p. Tflops

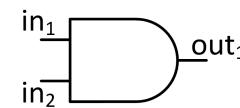
# Transistors: why we love them & want as many as possible



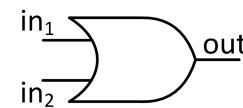
- Focus first on how transistors are used to **perform tasks** (tied to CU and ALU use)
- Later talk about how transistors are used to **store data and instructions**

# Transistors at work: AND, OR, NOT

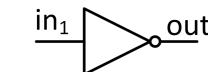
- The NOT logical operation is implemented using **one** transistor
- AND and OR logical ops require **two** transistors



AND



OR



NOT

- Truth tables for AND, OR, and NOT

AND	$in_2=0$	$in_2=1$
$in_1=0$	0	0
$in_1=1$	0	1

OR	$in_2=0$	$in_2=1$
$in_1=0$	0	1
$in_1=1$	1	1

NOT	
$in_1=0$	1
$in_1=1$	0

# Example (1/2)

[cooked up, but makes a point]

- Design a digital logic block that receives three inputs via three bus wires and produces one signal that is 0 (low voltage) as soon as any of the three input signals is low voltage.
  - In other words, it should return 1 if and only if all three inputs are 1
    - A bit-wise “&”

Truth Table

in <sub>1</sub>	in <sub>2</sub>	in <sub>3</sub>	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



Logic Equation:  $out = \overline{\overline{in_1}} + \overline{\overline{in_2}} \cdot \overline{in_3}$

## Legend:

- Overbar  $\overline{\phantom{x}}$  : negation
- Plus  $+$  : logical OR
- Product  $\cdot$  : logical AND

- NOTE: There are several Logic Equations that implement the same Truth Table. There are tradeoffs that one can make (power/space/speed)

## Example (2/2)

[cooked up, but makes a point]

- Easy to figure out the transistor setup once Logic Equation is available

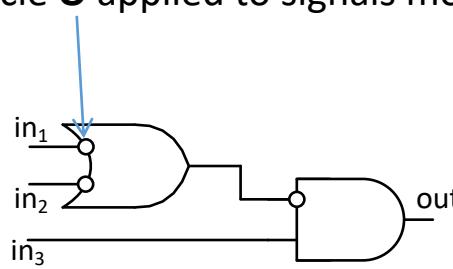
Truth Table

in <sub>1</sub>	in <sub>2</sub>	in <sub>3</sub>	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Logic Equation:

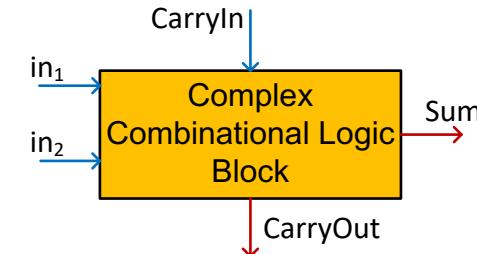
$$out = \overline{\overline{in_1} + \overline{in_2} \cdot in_3}$$

- Solution: digital logic block is a combination of AND, OR, and NOT gates
  - The NOT is represented as a circle O applied to signals moving down the bus



# Example: One Bit Adder (visualize calculating 567+789)

- Implement a digital circuit that produces the CarryOut digit in a one bit summation operation
  - Inputs: three of them ( $in_1$ ,  $in_2$ , CarryIn)
  - Outputs: two of them (Sum, CarryOut)

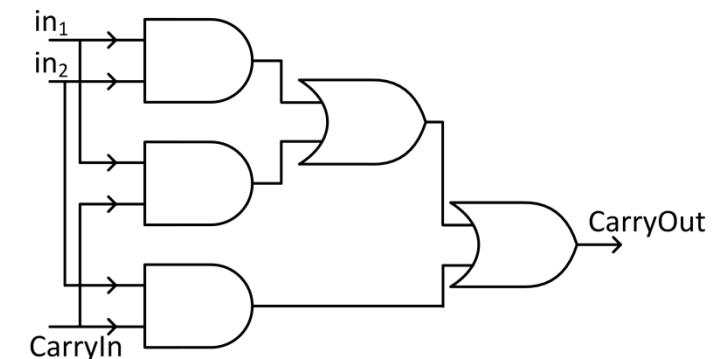


## Truth Table

Inputs			Outputs		Comments
$in_1$	$in_2$	CarryIn	Sum	CarryOut	
0	0	0	0	0	Sum is in base 2
0	0	1	1	0	0+0 is 0; the CarryIn kicks in, makes the sum 1
0	1	0	1	0	
0	1	1	0	1	0+1 is 1, but CarryIn is 1; sum ends up being 0, CarryOut is 1.
1	0	0	1	0	
1	0	1	0	1	1+0 is 1, but CarryIn is 1; sum ends up being 0, CarryOut is 1.
1	1	0	0	1	1+1 is 0, carry 1.
1	1	1	1	1	1+1 is 0 and you CarryOut 1. Yet the CarryIn is 1, so the 0 in the sum becomes 1.

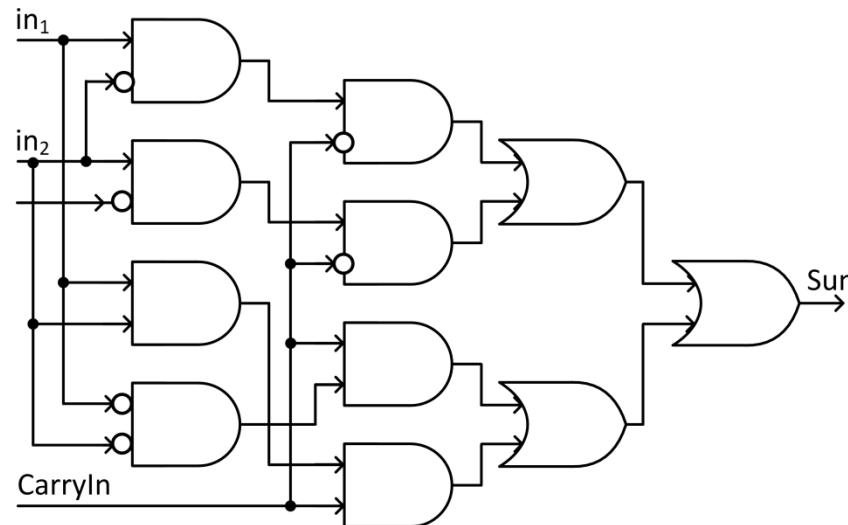
Logic Equation (for CarryOut):

$$\text{CarryOut} = (in_1 \cdot \text{CarryIn}) + (in_2 \cdot \text{CarryIn}) + (in_1 \cdot in_2)$$

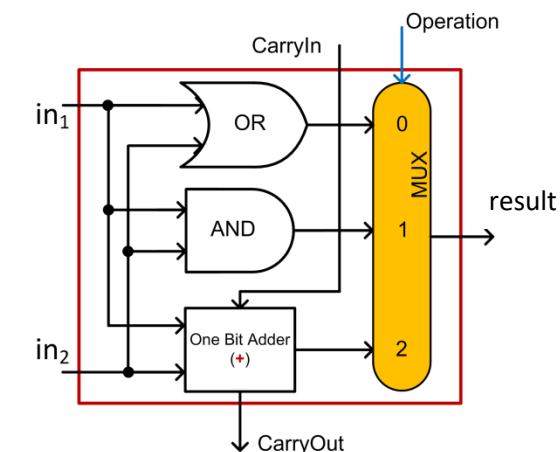


# Integrated Circuits-A One Bit **Combo**: OR, AND, 1 Bit Adder

- 1 Bit Adder, the **Sum** part

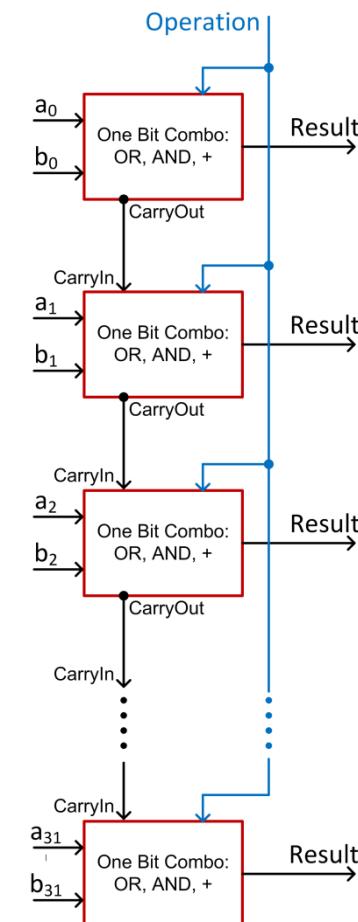


- A One Bit **Combo**: OR, AND, 1 Bit Sum
  - Controlled by the input “Operation”

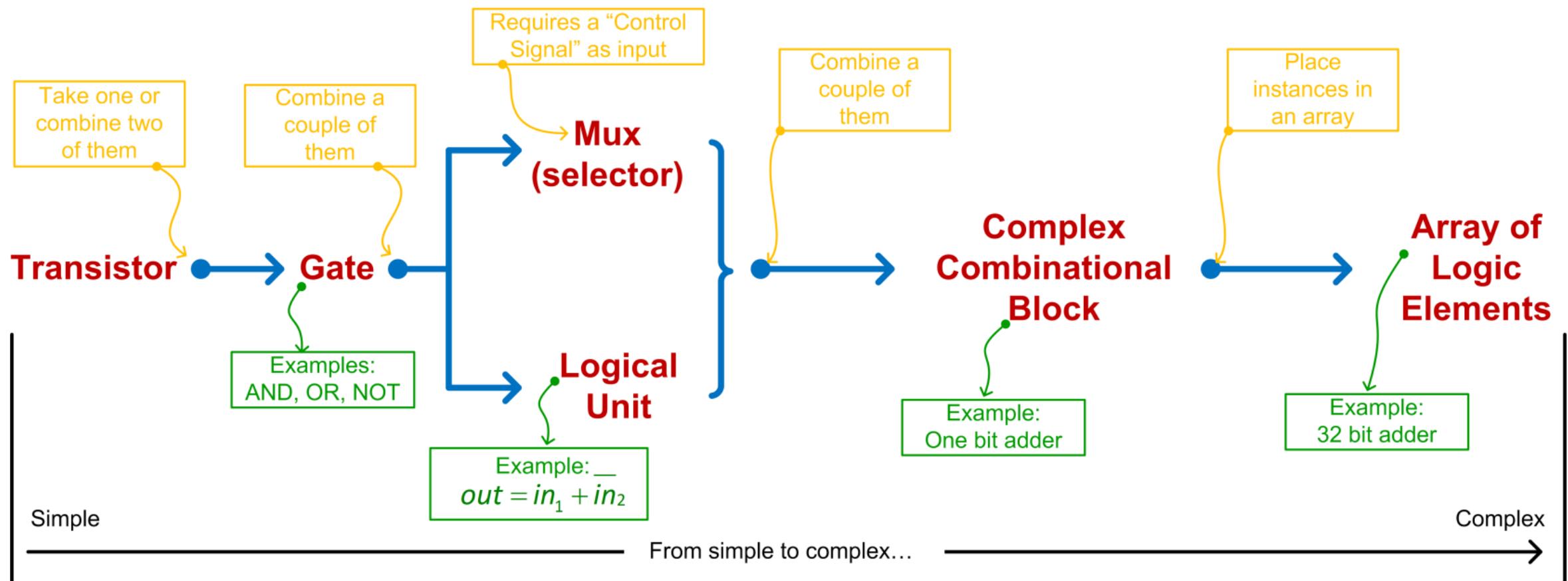


# Integrated Circuits: Ripple Design of 32-Bit Combo

- Combine 32 of the 1 bit combos in an array of logic elements
  - Get one 32 bit unit that can do OR, AND, +
  - This “academic example” design requires about 1152 transistors



# Integrated Circuits: From Transistors to Chip Microarchitecture



# Example: Number of Transistors, on GPUs [NVIDIA architectures]

- Fermi circ. 2010:
  - 40 nm technology
  - Up to 3 billion transistors → about 500 scalar processors, 0.5 d.p.Tflops
- Kepler circ. 2012:
  - 28 nm technology
  - Chips w/ 7 billion transistors → about 2800 scalar processors, 1.5 d.p. Tflops
- Maxwell 2014-2016
  - 28 nm technology
  - Chips w/ 8 billion transistors → 3072 scalar processors, 6.1 s.p. Tflops
- Pascal 2016-2017
  - 16 nm technology
  - Chips w/ 12 billion transistors → 3584 scalar processors, 10.6 s.p. Tflops
- Turing 2018 –
  - 12 nm technology
  - Chips w/ 19 billion transistors → 4608 scalar processors, 16 s.p. Tflops

# [FDEX Cycle: Execution Closing Remarks] It All Boils Down to Transistors...

- For 50 years, every 18 months, number of transistors per unit area doubled (Moore's Law)
  - 2014-2015 technology: feature length was 14 nm (Intel)
  - 2018: 10 nm -12 nm
  - Current technology (2019-2020): 7 nm (top of the line - AMD)
  - Looking ahead (2021): 5 nm
  - Even farther ahead (2023): 3 nm (Samsung?)
- Path forward unclear, what do we use when silicon-metal transistors can't get any smaller?
  - Maybe Carbon Nanotubes?

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

## Lecture 03

01/27/2020

# Quote of the day

“Nothing is particularly hard if you divide it into small jobs.”

-- Henry Ford, American industrialist and a business magnate, founder of the Ford Motor Company [1863 – 1947]

# Pictures of the Day

Workers on the first moving assembly line put together magnetos and flywheels for 1913 Ford autos. Highland Park, Michigan



<http://www.gpschools.org/ci/depts/eng/k5/third/fordpic.htm>

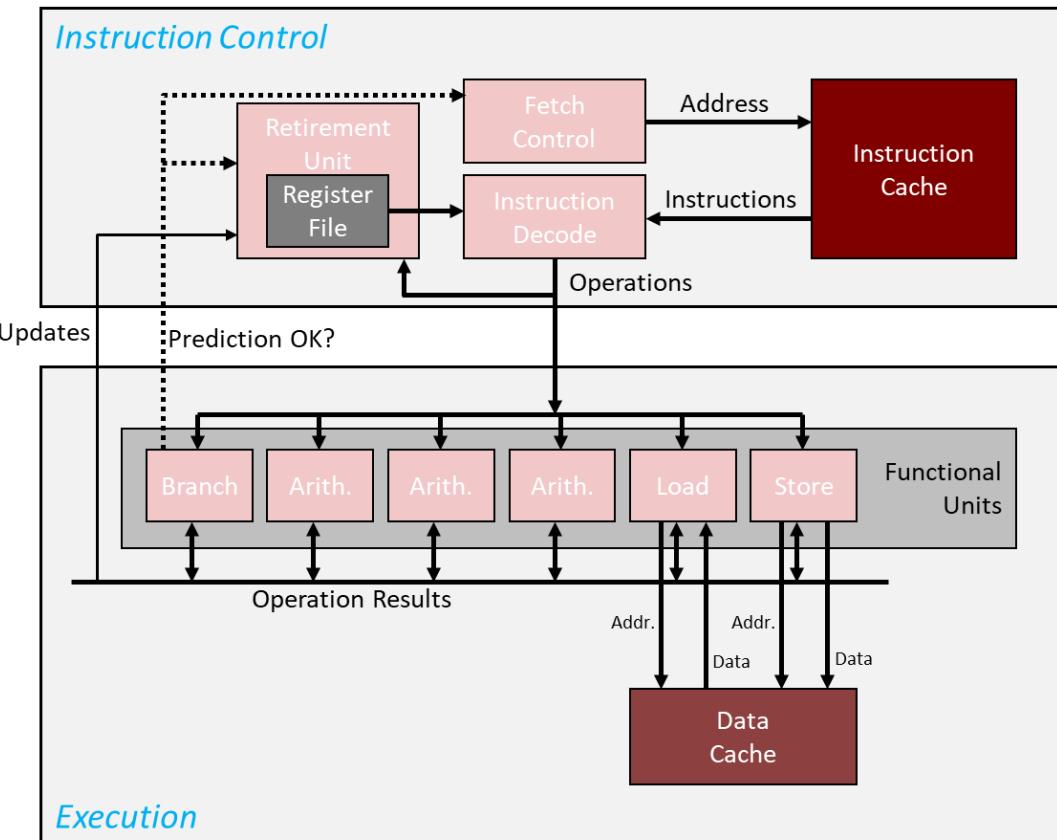


<https://www.rediff.com/business/report/hyundai-uses-580-robots-in-chennai-plant-maruti-5000/20180530.htm>

# Before we get going...

- Last time: the Hardware ↔ Software interplay
  - A line of our C code translated by compiler into several instructions (assembly helps you see these instructions)
  - Instructions get executed on the chip
  - The chip: made up by transistors organized into various pieces of circuitry performing various operations
  - One by one, instructions are fetched-decoded-executed
    - The CU (the thinking) and ALU (the executing) get FDX done
- Today
  - Registers
  - Instruction Level Parallelism
  - Thread Level Parallelism
- Other tidbits:
  - Please take care of the assigned reading
  - First assignment due on Th at 9 pm
  - Use Piazza to get things going (for you or others)

Keep this pic in mind when you visualize how your code gets executed, instruction by instruction



# Registers

# Revisiting an older slide, to get us going

- Line of C code:

```
a[4] = delta + a[3]; //line of C code
```

- MIPS assembly code generated by the compiler (three instructions):

```
lw $t0, 12($s2) # reg $t0 gets value stored 12 bytes from address in $s2  
add $t0, $s4, $t0 # reg $t0 gets the sum of values stored in reg $s4 and reg $t0  
sw $t0, 16($s2) # store what's in $t0 at mem location 16 bytes from address in $s2
```

- Set of three corresponding MIPS instructions produced by the compiler:

```
100011100100100000000000000000001100  
00000010100010000100000000100000  
1010111001001000000000000000010000
```

# Registers, backdrop (1/2)

- Instruction cycle: fetch-decode-execute (FDX)
- CU – responsible for controlling the process that will deliver the request baked into the instruction
- ALU – does the busy work to fulfill the request put forward by the instruction
- The **instruction** that is being executed needs to be **stored somewhere**
- Fulfilling the requests baked into an instruction usually involves input values and generates output values
  - This **data** needs to be **stored somewhere**

# Registers, backdrop (2/2)

- Registers, quick facts:
  - A register is an entity whose role is that of storing information
  - A register is the storage type with shortest latency – it's closest to the CU & ALU
    - Latency – can be of the order of hundreds of picoseconds
- The **number** and **size** of registers used are specific to an ISA
  - The microarchitecture has to implement support per ISA's specifications
  - Example: Back in the day, for the MIPS ISA, there were 32 registers of 32 bits (no more, no less)

# Register Types (1/4)

- Discussion herein covers only several register types typically encountered in a CPU (abbreviation in parenthesis)
  - List not comprehensive, showing only the more important ones
- **Instruction Register** (IR) – holds the instruction that is executed
  - Sometimes known as “current instruction register” CIR
- **Program Counter** (PC) – holds address of the instruction executed next
  - NOTE: unlike IR, PC contains **\*address\*** of the instruction, not actual instruction

## Register Types (2/4)

- **Memory Data Register (MDR)** – holds data read in from memory or, alternatively, produced by the ALU and waiting to be stored in memory
- **Memory Address Register (MAR)** – holds address of RAM memory location where input/output data is supposed to be read in/written out
  - NOTE: unlike MDR, MAR contains **\*address\*** of a mem location, not actual data
- **Return Address (RA)** – the address where upon finishing a sequence of instructions, the execution should jump and commence with the execution of subsequent instruction

# Register Types (3/4)

- Registers on previous two slides are a staple in most ISA
- There are several other registers common to many chip designs
  - These are encountered in different numbers (also depending on ISA)
- Since they come in larger numbers they don't have an acronym
  - Registers for **Subroutine Arguments** (4) – a0 through a3
  - Registers for **Temporary Variables** (10) – t0 through t9
  - Registers for **Saved Temporary Variables** (8) – s0 through s7
    - Saved between function calls

# Register Types (4/4)

- Several other registers are involved in handling function calls
- Summarized below, their meaning only apparent in conjunction with the organization of the virtual memory
  - Stack Pointer (sp) – a register that holds an address that points to the top of the stack
  - Global Pointer (gp) – a register that holds an address that points to the middle of a block of memory in the static data segment
  - Frame Pointer (fp) - a register that holds an address that points to the beginning of the procedure frame (for instance, the previous **sp** before this function changed its value)

# Register, Departing Thoughts

- Registers: a very precious resource
- If they are so good, why can't we have more?
  - Increasing their number is not straightforward
    - Need to change the design of the chip (the microarchitecture) & not enough space (maybe go 3D?)
    - Need to change the very nature of an instruction (and as such ISA gets touched in the process)
- Examples:
  - In 32-bit MIPS ISA, there are 32 registers
  - GP100 GPU (Pascal): 56 SMs, each with 64 SPs. Total register file: 14336K (~3.5 million registers)
    - Seems impressive but it's not:  $56 \text{ SMs} * 2048 \text{ threads/SM} * 32 \text{ registers/thread}$  (guesstimate) = 3.6 million registers needed

# Registers, take home message

- When processing one instruction, the chip needs all instruction-related data in registers
- Fallout, from above statement
  - You might have GBs of data to process. To be processed, this data will have to find its way in registers
    - We'll talk about this data journey into registers later

# Pipelining

“Nothing is particularly hard if you divide it into small jobs.”

Henry Ford

Comic of the day: Too many cooks spoil the broth.





Workers on the first moving assembly line put together magnetos and flywheels for 1913 Ford autos. Highland Park, Michigan

# Pipelining, or the Assembly Line Concept

- Henry Ford: capitalized on, and improved the assembly line idea on an industrial scale and in the process shaped the automotive industry (Ford Model T)
- Vehicle assembly line: a good example of a **pipelined process**
  - The output of one stage (station) becomes the input for the downstream stage
  - Once assembly line primed, a vehicle gets finished/manufactured during each cycle
  - “cycle” is the time it takes from the moment a station gets its input to the moment its output leaves the station



<https://www.rediff.com/business/report/hyundai-uses-580-robots-in-chennai-plant-maruti-5000/20180530.htm>

Historical change of the manufacturing line. Manual assembly line (left); Hyundai assembly line in Chennai, India (right)

# Leaving Car Analogy Behind...

- FDX cycle: Fetch, Decode, Execute - carried out for **each** machine instruction
  - Sometimes called FDXW (W: “Write”)
- A closer look at what gets fetched (instructions and data) and then what happens upon execution leads to a generic five-stage process associated with an instruction
- “generic” above means that, in a first order approximation, these five stages can represent any instruction, although some instructions might not have all five stages:
  - Stage 1: Fetch an instruction
  - Stage 2: Decode the instruction
  - Stage 3: Data access
  - Stage 4: Execute the operation (e.g., might be a request to calculate an address)
  - Stage 5: Write-back into register file

# Rules of thumb: costs to execute an operation > 1 cycle

[refers to older MIPS ISA; cost of X in FD $\Sigma$ ]

- Four cycles for SW
- Five cycles for LW
- Four cycles for add
- Etc.

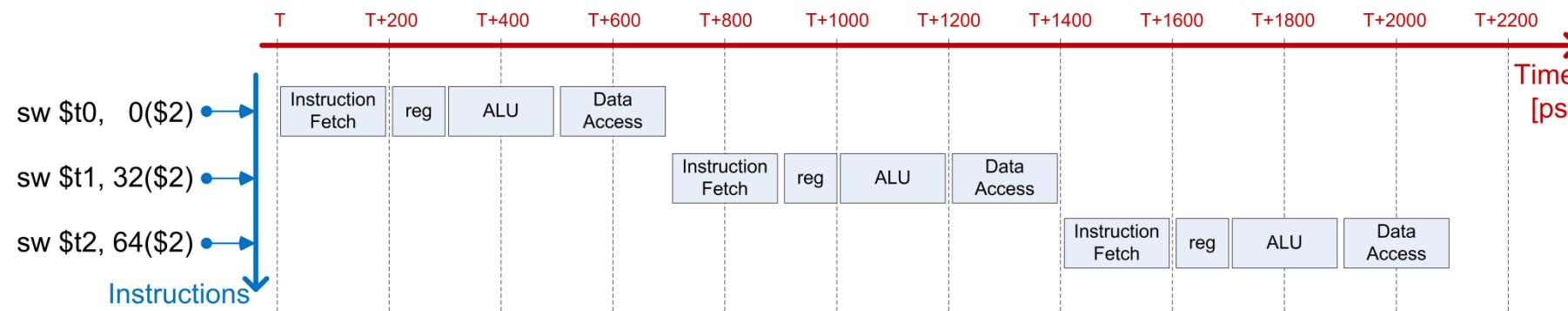
# Pipelining, the basics

- Assume a five-stage pipeline
- Pipelining, cornerstone idea: the following tasks can be worked upon simultaneously when processing five instructions:
  - Instruction 1 is in the 5<sup>th</sup> stage of the FDX cycle
  - Instruction 2 is in the 4<sup>th</sup> stage of the FDX cycle
  - Instruction 3 is in the 3<sup>rd</sup> stage of the FDX cycle
  - Instruction 4 is in the 2<sup>nd</sup> stage of the FDX cycle
  - Instruction 5 is in the 1<sup>st</sup> stage of the FDX cycle

# Example: Streaming for execution 3 SW instructions

```
sw    $t0,  0($s2)  
sw    $t1,  32($s2)  
sw    $t2,  64($s2)
```

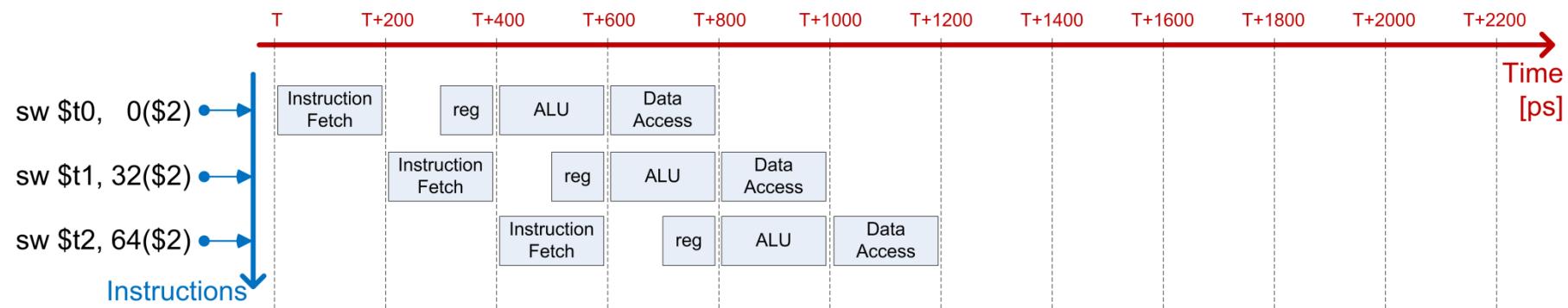
- Case 1: No pipelining – 2100 picoseconds [ps]



# Example: Streaming for execution 3 SW instructions

```
sw    $t0,  0($s2)  
sw    $t1,  32($s2)  
sw    $t2,  64($s2)
```

- Case 2: With pipelining – 1200 picoseconds [ps] (5 GHz clock cycle)



# Pipelining: on its “balanced” attribute

- An ideal situation is when each of pipeline stage takes the same amount of time for completion
  - The pipeline is then **balanced**
- If pipeline not balanced, there will be stations that wait for the laggard station at every single cycle
- Common sense: If there is a station that takes a significantly longer time since it does significantly more than other stations, it should be broken into two and the length of the pipeline increases by one station
  - Leads to deeper pipelines
  - Typical pipeline depth today: 12-15 stages

# Pipelining Benefits: speed up & no code rewrite

- **Benefit 1: faster execution**, owing to simultaneously working on multiple instructions
  - Assume that you have
    1. A very large number of instructions
    2. Balanced stages
    3. A pipeline that is larger than or equal to the number “ $p$ ” of stages associated with the typical ISA instruction
  - If 1 through 3 above hold, in a first order approximation, speed-up you get out of pipelining is approximately “ $p$ ”
- **Benefit 2: no effort** required on the user’s part, no code rewrite needed to reap benefits
  - This kind of parallel processing of stages is transparent to the user
  - Unlike GPU or multicore parallel computing, you don’t have to do anything to benefit from it

# Pipelining, Good to Remember

- The amount of time required to complete one stage of the pipeline: one cycle
- Pipelined processor: one instruction finished in each cycle
- Non-pipelined processor: no overlap when executing instructions
- Important Remark:
  - Pipelining does not decrease the time to complete one instruction but rather it increases the throughput of the processor by overlapping different stages of processing of different instructions

# Pipelining Hazards

- Q: if deep pipelines are good, why not make them deeper?
- A: deep pipelines plagued by **pipelining hazards**
  - **Structural hazards**
  - **Data hazards**
  - **Control hazards**

# Pipeline Structural Hazards: The problem at hand

- A real world assembly line assembles the **same product** for a period of time
- Typically, it can be **quickly reconfigured** to assemble a different product
- The chip processes a broad spectrum of instructions that come one after another
  - Example: A J-type instruction coming after a R-type instruction, which comes after three I-Type instructions
  - If they were the same instructions (vehicles), designing a pipeline (assembly line) is straightforward
- A structural hazard: the possibility of having a combination of instructions in the pipeline **contending** for the same piece of hardware
  - Not encountered when you assembly the same car model (things are “deterministic” in this case)

21,529,464 VW Beetle vehicles produced (1938 until 2003)



[Alden Jewell: <https://www.flickr.com/photos/autohistorian/32637661426>]→

# Pipeline Structural Hazards: Fixing the problem

- Possible Scenario: you have a six stage pipeline and the instruction in stage 1 and instruction in stage 5 both need to use the same register to store a temporary variable
  - Solution 1: there should be enough registers provisioned so that almost any combination of instructions is ok
  - Solution 2: serialize the access, basically stall the pipeline for a cycle so that there is no contention
  - Solution 3: Out of order execution (OOOE) done statically (at compile time) or perhaps dynamically (at run time)

# Pipeline Structural Hazards: Fixing the problem - comments

- Comment, Solution 1:
  - Adding more registers is a static solution; expensive and consequential (requires a chip design change)
    - Note that “needing one extra register” is just one possible scenarios; there are might be other contention scenarios lurking
- Comment, Solution 2:
  - Stalling the pipeline at run time is a general solution but slows down the execution
    - Done by introducing a “bubble” in the pipeline
- Comment, Solution 3 (OOOE):
  - This is a good compromise: it’s general and when you can re-order instructions there is no slowdown

# Pipeline Data Hazards (1/2)

[very common]

- Example, to get the discussion going – Consider assembly below, with a five-stage pipeline chip:

```
add $t0, $t2, $t4      # $t0 = $t2 + $t4
addi $t3, $t0, 16       # $t3 = $t0 + 16 ("add immediate")
```

- The first instruction is processed in five stages
- Its output (value stored in register  $\$t0$ ) is needed in the very next instruction
- Data hazard: unavailability of  $\$t0$  to the second instruction, which references this register
- Resolution (less than ideal)
  - Pipeline stalls to wait for the first instruction to fully complete

# Pipeline Data Hazards (2/2)

```
add $t0, $t2, $t4    # $t0 = $t2 + $t4  
addi $t3, $t0, 16    # $t3 = $t0 + 16 ("add immediate")
```

- Alternative [the good] Resolution: use “forwarding” or “bypassing”
- Key observation: the value that will eventually be placed in  $\$t0$  is available after stage 3 of the pipeline (where the ALU actually computes this value)
- Provide the means for that value in the ALU to be made available to other stages of the pipeline right away
  - Avoids stalling - don't have to wait several cycles before the value made its way into  $\$t0$
  - This process is called “intermediate result forwarding”
- Supporting forwarding does not guarantee resolution of all scenarios
  - Occasionally the pipeline ends up stalled for a couple of cycles
- OOOE helps (insert unrelated instruction, wedged between two instructions that depend on each other)

# Pipeline Control Hazards (1/2)

- What happens when there is an “if” statement in a piece of C code?
- A corresponding machine instruction decides the program flow
  - Specifically, should the “if” branch be taken or not?
- Processing this very instruction to figure out next instruction (branch or no-branch) will take a number of cycles
- Should the pipeline stall while this instruction is fully processed and the branching decision becomes clear?
  - If yes: strategy always works, but is slow
  - If no: rely on branch prediction and proceed fast but cautiously

# Pipeline Control Hazards (2/2): Branch Prediction

- **Static Branch Prediction** (1<sup>st</sup> strategy out of two):
  - Always predict that the branch will not be taken and schedule accordingly
    - That is, always schedule the “then” branch, never the “else” branch
  - There are other heuristics for proceeding: for instance, for a do-while construct it makes sense to always be jumping back at the beginning of the loop
    - Similar heuristics can be produced in other scenarios (a “for” loop, for instance)
- **Dynamic Branch Prediction** (2<sup>nd</sup> strategy out of two):
  - At a branching point, the branch/no-branch decision can change during the life of a program based on recent history
  - In some cases branch prediction accuracy hits 90%
- Note: when your prediction is wrong you have to discard the [micro]instruction[s] executed speculatively and take the correct execution path

# [New topic] Pipelining vs. Multiple-Issue (1/2)

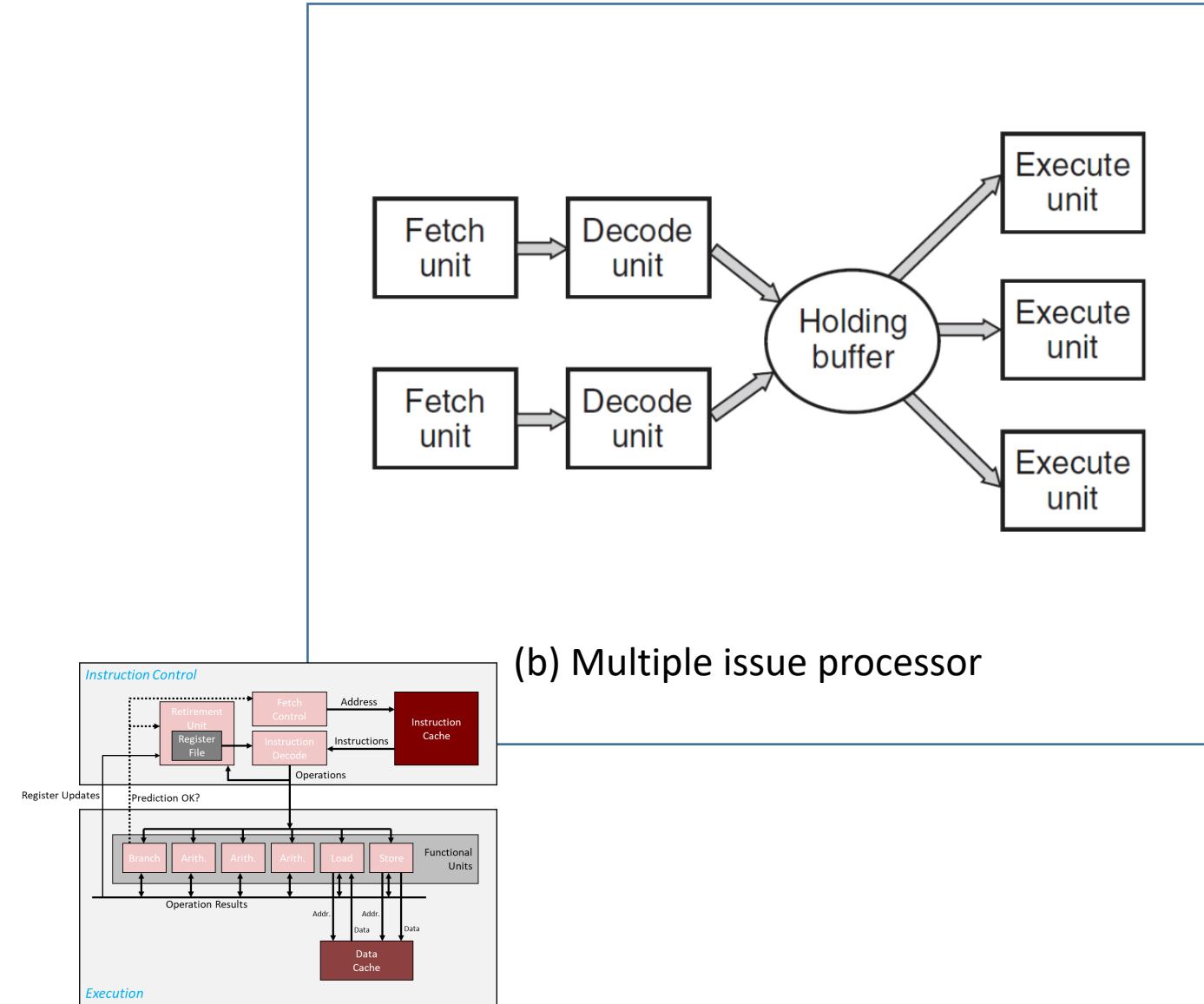
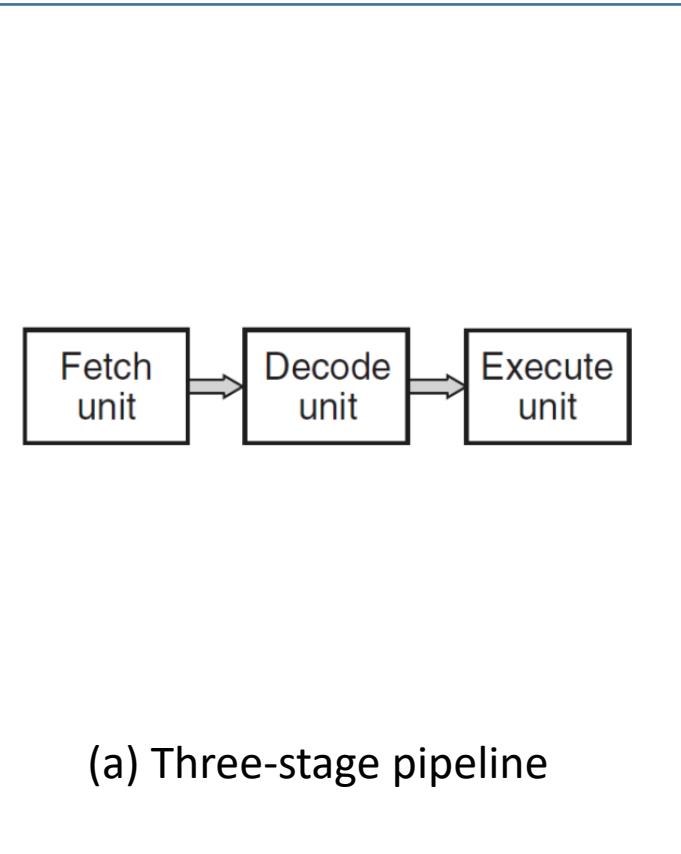
- Multiple-issue: what is that?
  - In sequential computing, a multiple-issue processor core has the hardware chops to issue more than one instruction per cycle
- Pipelining different than “Multiple-Issue,” which is another way of speeding up execution
- Example: the C code below leads to a set of instructions that can be launched for execution at the same time
  - IMPORTANT: No cross-dependency in updating a and c

```
int a, b;
float c, d;
//some code setting up a, b, c, d
a += b;
c += d;
```

# Pipelining vs. Multiple-Issue (2/2)

- On average, more than one instruction is processed by the **same core** in the **same clock cycle**
- Multiple-Issue can be done statically or dynamically
  - **Static** multiple-issue:
    - Predefined, doesn't change at run time
    - Who uses it: Intel, in AVX512 (but not only)
  - **Dynamic** multiple-issue:
    - Determined at run time, the chip has dedicated hardware resources that can identify and execute additional work
    - Who uses it: Intel, AMD, IBM, NVIDIA, etc.

# Pipelining vs. Multiple Issue



# Attributes of Dynamic Multiple-Issue

- The data dependencies between instruction being processed takes place at run time
- Checking for dependencies is complex, requires high cost in time and energy
- Checks in place to make sure result is ok
  - “ok”: no difference if the instructions are executed “multiple-issue” or “single issue”
- NOTE: a chip that can do multiple-issue is also called a **superscalar** architecture

# The “Instruction-Level Parallelism” (ILP) concept

- Pipelining, OOOE, and multiple-issue are presentations of what's called Instruction-Level Parallelism (ILP)
- ILP pursues various strategies to execute simultaneously multiple instructions
- Important: these instructions belong to the same program or process; i.e., belong to **\*one\*** thread
  - “process” – how the operating systems calls your program

# ILP: various angles of attack (quick summary of ILP's bag of tricks)

- **Instruction pipelining**: execution of multiple instructions can be partially overlapped; each instruction divided into sub-steps (termed: micro-operations)
- **Superscalar execution**: multiple execution units are used to execute multiple instructions in parallel
- **Out-of-order execution**: instructions execute in any order but without violating data dependencies
- **Register renaming**: a technique used to avoid data hazards and thus lead to unnecessary serialization of program instructions caused by the reuse of registers
- **Speculative execution**: allows the execution of complete instructions or parts of instructions before being sure whether this execution is required
- **Branch prediction**: used to avoid delays (termed: stalls). Used in combination with speculative execution.

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

## Lecture 04

01/29/2020

# Quote of the day

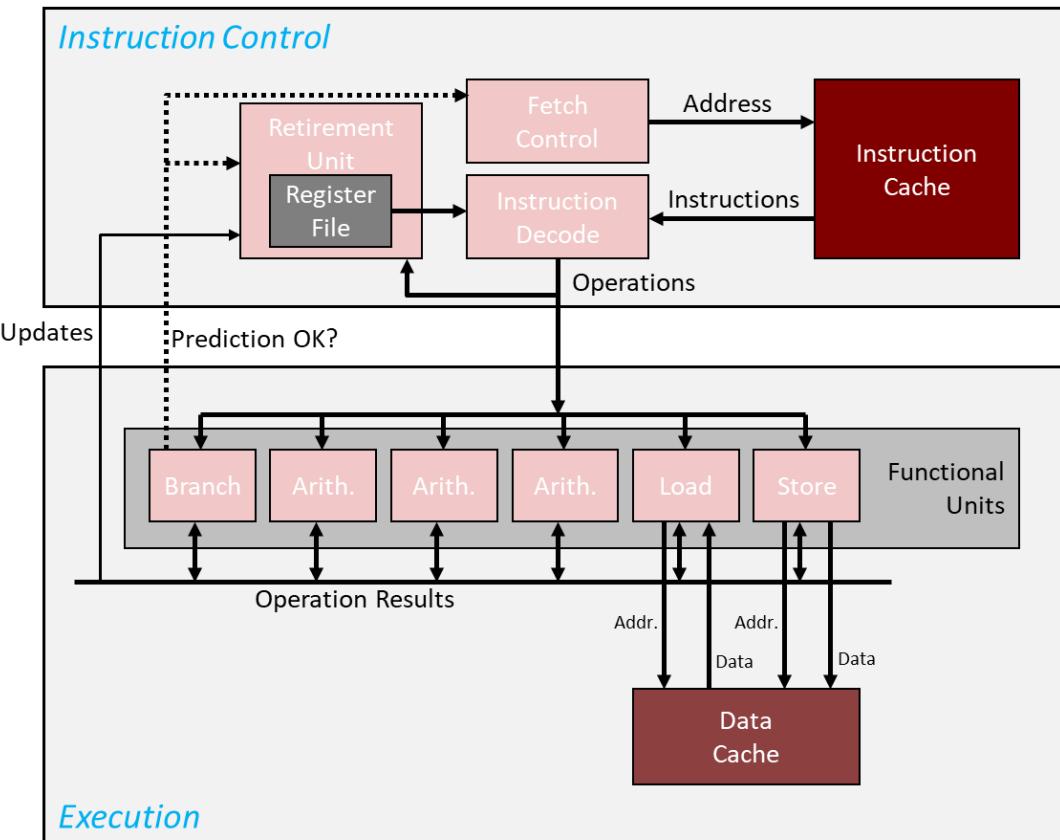
“The real question is not whether machines think, but whether men do.”

-- Burrhus Frederic Skinner, Professor of Psychology at Harvard University [1904 – 1990]

# Before we get going...

- Last time: the Hardware ↔ Software interplay
  - Registers
  - ILP tricks to get the code running fast
    - Pipelining is the poster child
    - But other good ones too: OOOE, speculative execution, etc.
- Today
  - Thread Level Parallelism (TLP)
  - Measuring execution performance, quick thoughts
  - The memory hierarchy
- Other tidbits:
  - NOTE: Office hour on Th is from 10:00 - 11:30 am
  - Office hours for online students: email before 6:30 pm
  - Get an account on Euler; you need it for HW01
  - Please take care of the assigned reading
  - First assignment due on Th at 9 pm
  - Use Piazza to get things going (for you or others)

Keep this pic in mind when you visualize how your code gets executed, instruction by instruction



# Thread Level Parallelism (TLP) [happens on a single core]

- ILP not the only “parallel” show in town
- Another one: chip executes simultaneously instructions from **different processes** or **different threads**
  - Called Thread-Level Parallelism (**TLP**)
- Important to keep in mind: we are talking about parallelism on one hardware core, not multicore

# ILP vs. TLP: how are they different

- Recall concept of “Program Counter” (PC): a register that holds address of next instruction
- If all instructions executed by the chip in parallel are associated with one PC, we have ILP
- TLP: the stream of instructions processed by the chip associated with more than one PC

# Process vs. thread – getting a bit into the weeds [1/2]

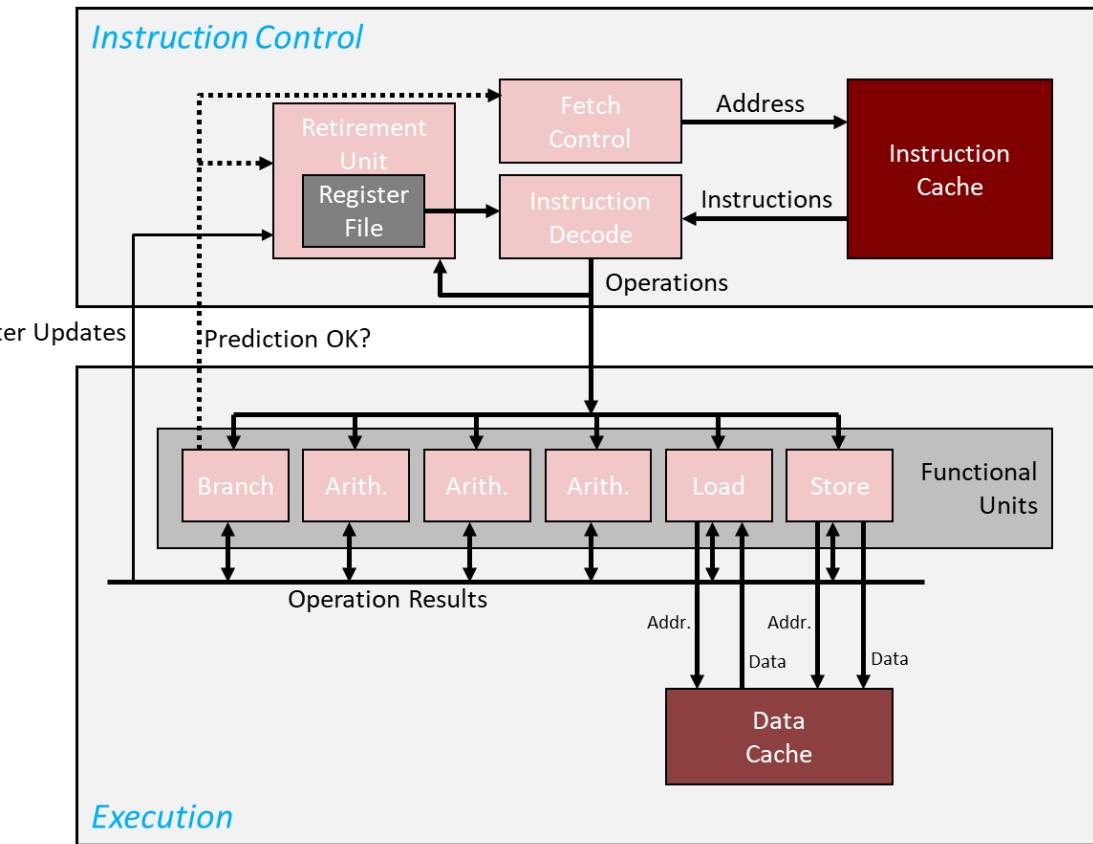
- TLP: chip executes simultaneously instructions from **different processes** or **different threads**
- Examples:
  - TLP is what happens when two **processes** such as Matlab and Microsoft Word run on the chip
    - Because two different processes, there are two different PCs
  - TLP: two (or even more) OpenMP or POSIX **threads** simultaneously run on the chip
    - Because two different threads, there are two different PCs

# Process vs. thread – getting a bit into the weeds [2/2]

- How similar: Both processes and threads are independent sequences of execution
- How different:
  - Threads (of the same process) run in a **shared memory space**
    - This is how shared-memory parallel computing takes place. Like OpenMP
  - Processes run in **separate memory spaces**
    - This is how distributed-memory parallel computing takes place. Like MPI\*
- What is a memory space?
  - To be continued, in one week (after discussing the concept of Virtual Memory)

# Why did folks come up with this TLP idea?

- Over time, transistor budget ballooned
  - Owing to Moore's Law
- As such, many Functional Units available
- TLP allows us to put them to good use



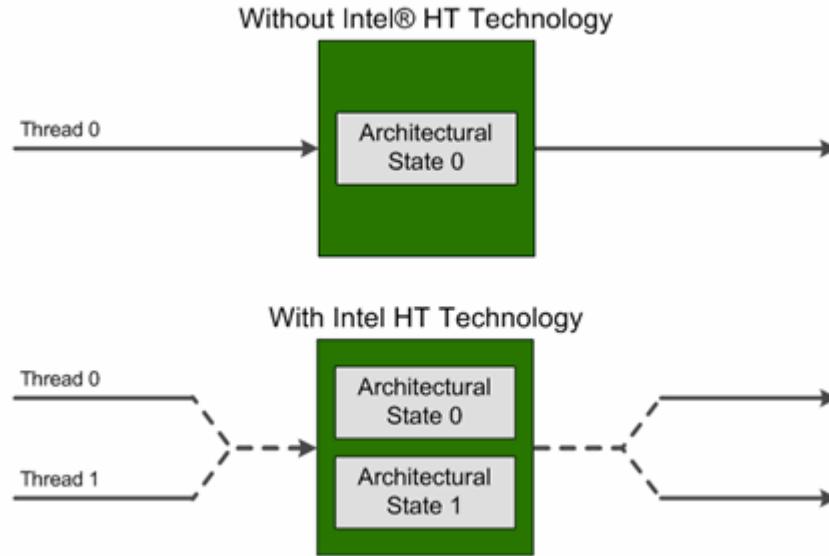
# An example of TLP: Intel's HTT feature

- Typical Intel core supports the “Hyper-Threading Technology” (HTT)
- HTT: two threads/processes, each seeing its own stream of machine instructions, active at any given time
  - “active”: also called “in flight”
- The scheduler tries to issue instructions from both processes, at the same time
- The beefing up of the CU/ALU hardware required to support HTT is actually pretty minor
  - Extra hardware required to save execution state of a thread when idle and not running (for lack of data or available functional units)

# Intel's HTT Feature

- In general, TLP requires the participation of the Operating System (OS)
- The OS sees one physical chip as two virtual chips; OS “talks” to both of them
- Is HHT helpful?
  - Yes: when running two modestly demanding processes
    - Example: Running MS-Word and Spotify at the same time
      - Execution threads are independent, when one execution thread stalls the other can be picked up and its PC advanced
  - No: high performance computing, when one stream of instruction saturates the memory bandwidth
    - Nothing left for anybody else to step in and nibble on excess-hardware leftover
- NOTE: as of recently, the trend has been for TLP to be scaled back owing to security issues

# HTT, Intel's perspective



- **Highlights:**

- The processor core maintains two architectural states, each of which can support its own execution thread
- Many of the internal microarchitectural hardware resources are shared between the two threads

# HTT: The Bottom Line

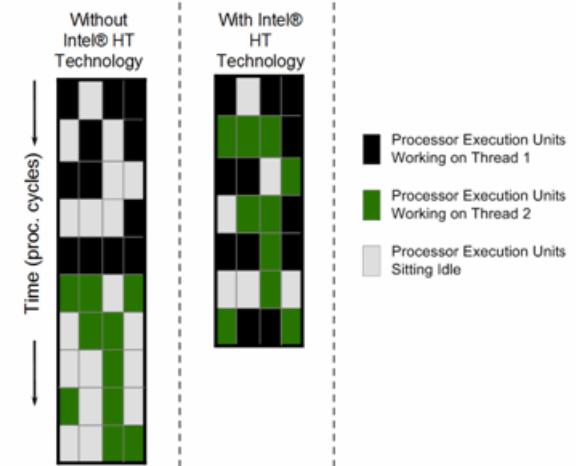
- Essentially, one physical core shows up as two virtual cores

- Why is it good?

- More execution units within the ALU are used at any given time
    - Higher utilization rate of the hardware assets
  - When one execution thread stalls (has a cache miss, branch mispredict, pipeline bubble, etc.), the other execution thread continues processing instructions at nearly the same rate as a single thread running on the core

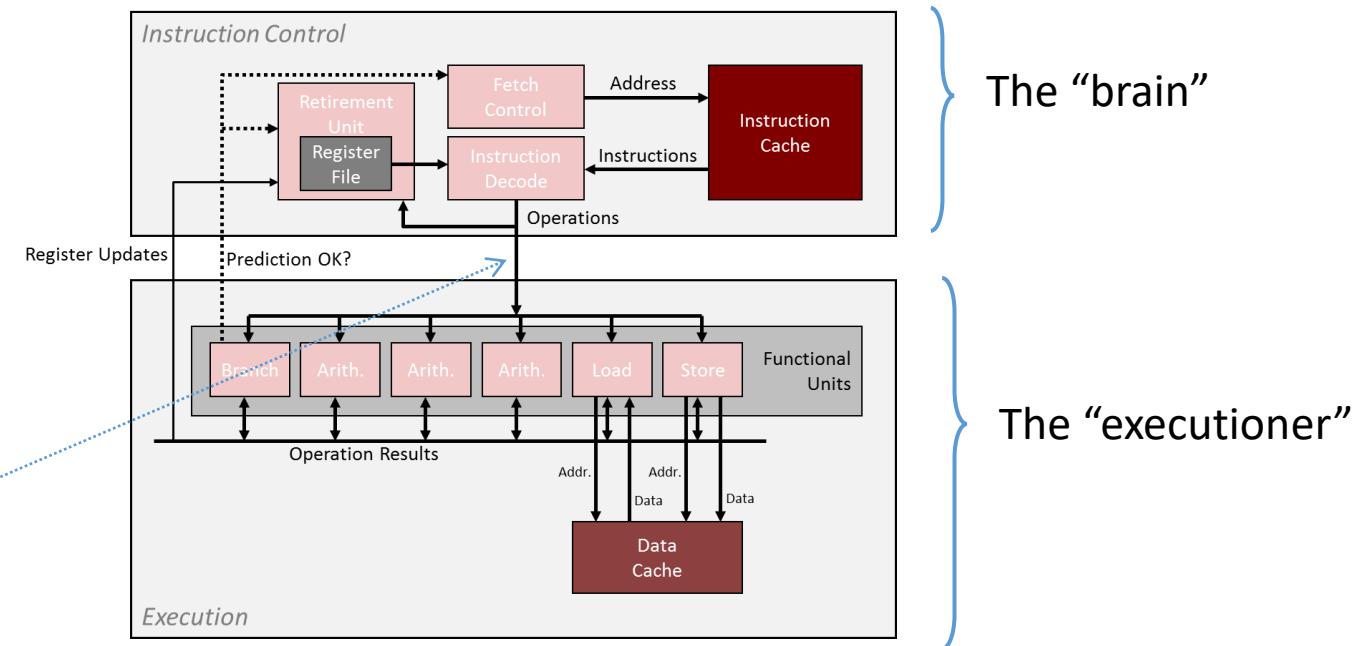
- NOTE: HTT is one manifestation of what TLP

- IBM, AMD (Zen micro-architecture) do it too



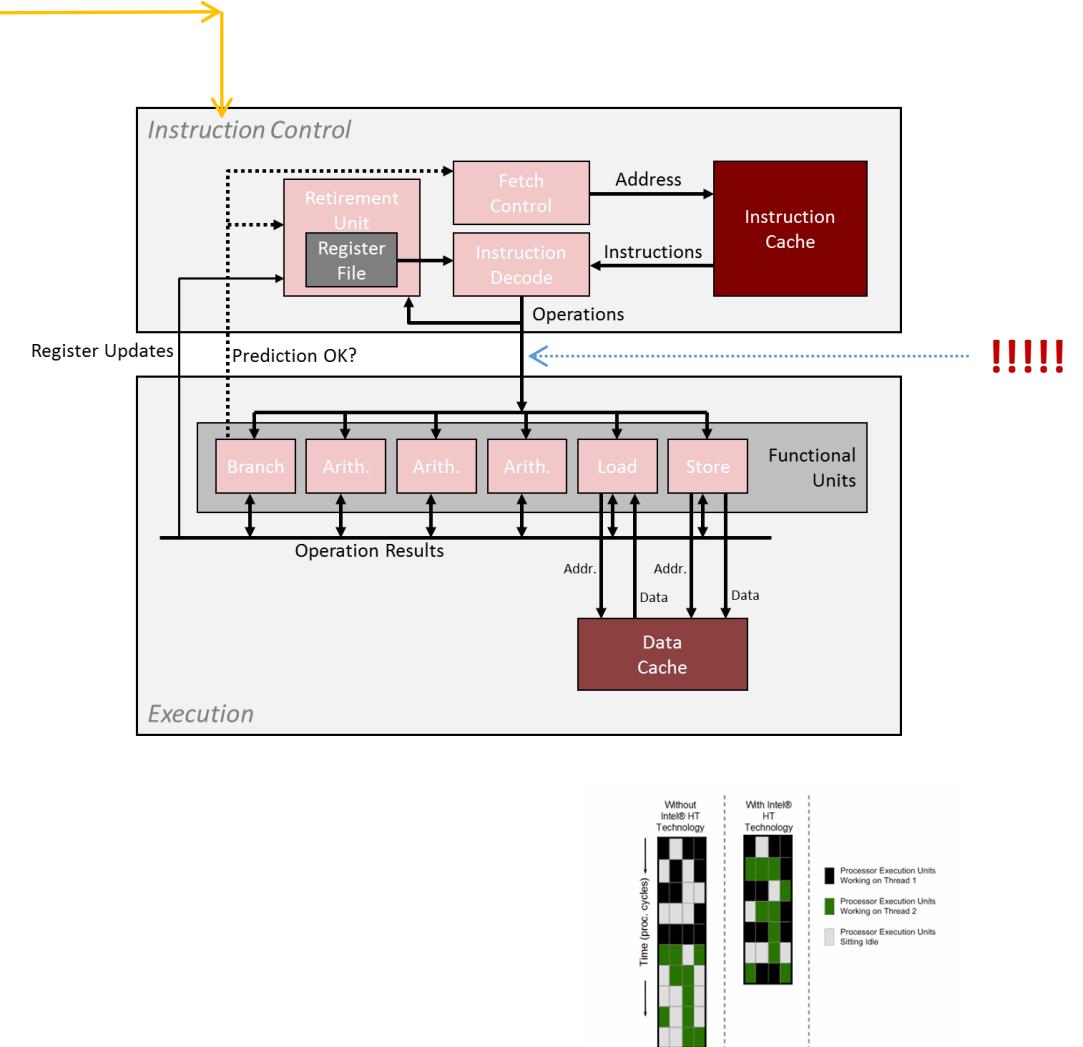
# TLP: further discussion

- Multi-threading, a possible taxonomy  
(Hennessey & Paterson)
  - Coarse grain multi-threading
  - Fine grain multi-threading
  - Simultaneous multi-threading
- This taxonomy: It all goes back to what goes through this pipe at each clock cycle



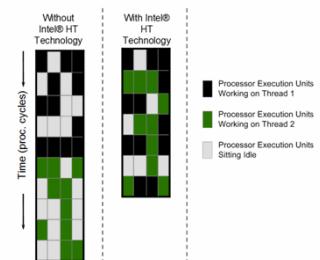
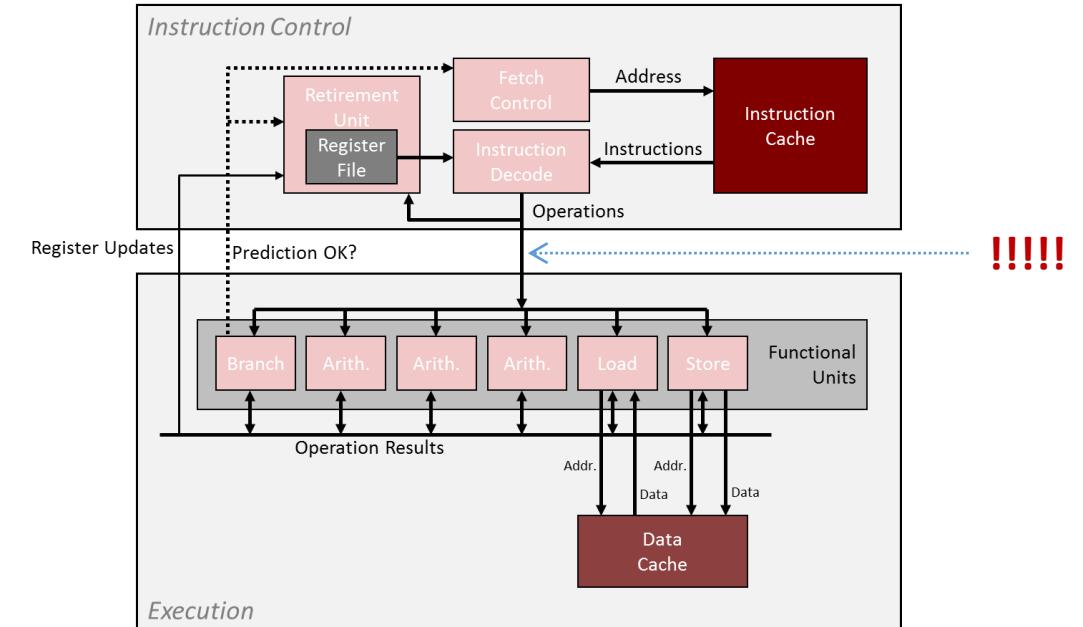
# Coarse multi-threading

- The **Instruction Control** issues instructions from one thread for many cycles
  - Bursts of tens or hundreds of instructions
- **Instruction Control** switches to issuing from another thread only when the first thread runs into a long-latency operation
  - Examples: cache miss, interrupt, etc.
  - “long-latency”: Hundreds of cycles or so



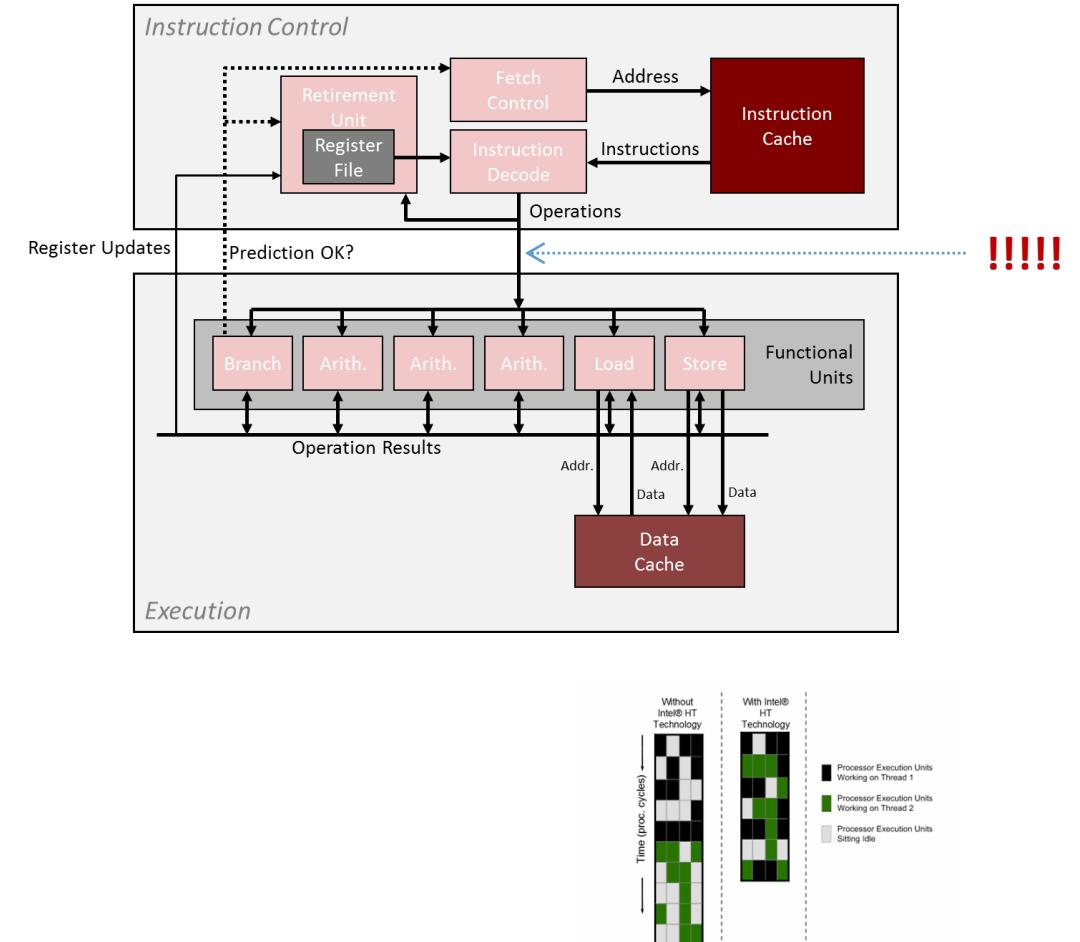
# Fine multi-threading

- Instruction Control can issue instructions from a different thread each cycle
  - “**Barrel-gun style**” processing: thread from where next instruction is picked for execution can change at every cycle
  - GPU computing similar to this
- More nimble/sophisticated compared to coarse multi-threading



# Simultaneous multi-threading

- Simultaneous multi-threading is like fine multi-threading with one caveat:
  - Instruction Control can issue instructions from **different threads** in the same cycle
- NOTE:
  - Intel's HTT is a “simultaneous multi-threading” technology

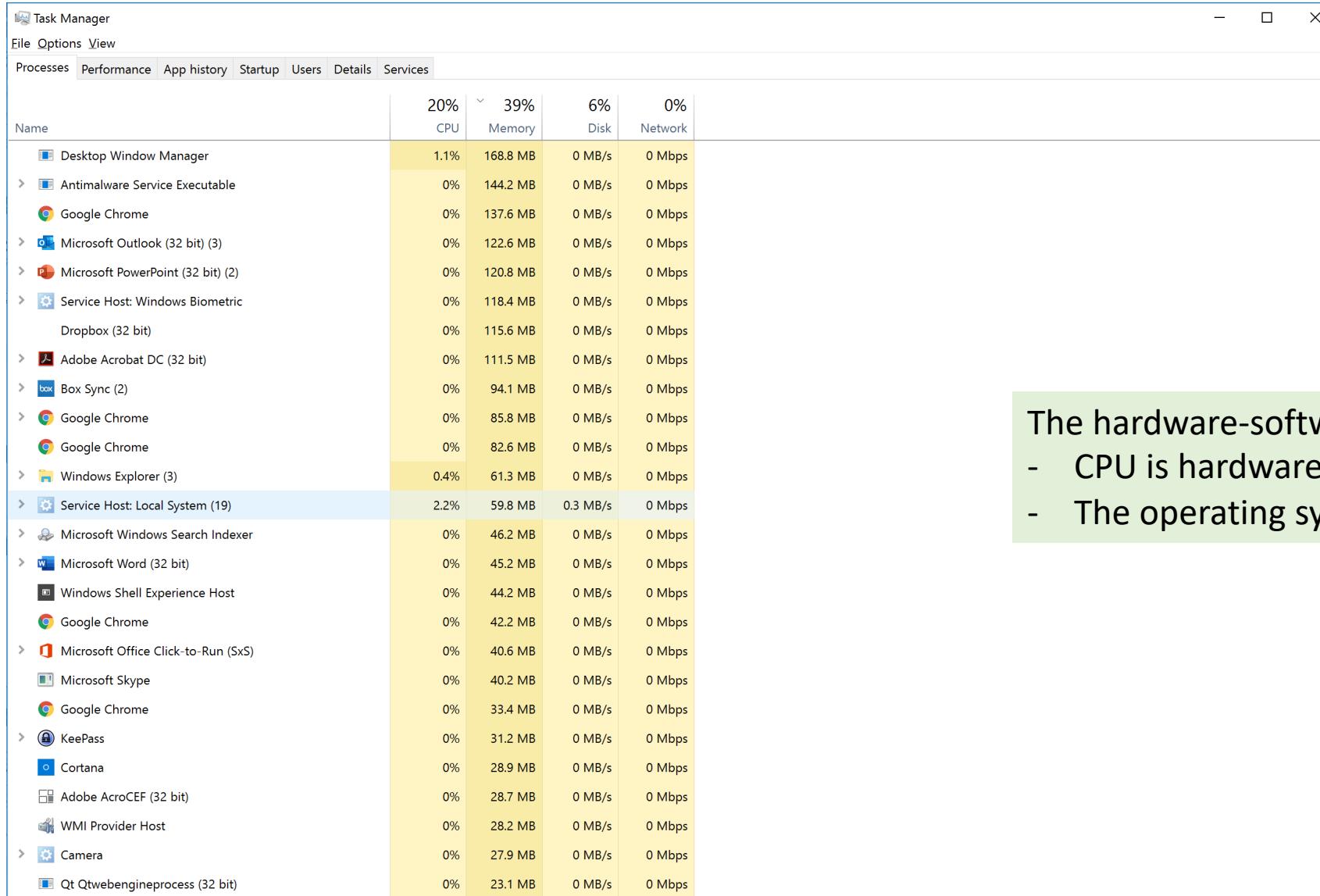


# Superscalar processor or TLP processor. Wrapping it up

- Superscalar vs. TLP
  - Superscalar: instructions associated with one PC
  - TLP: instructions associated with two PCs
- Superscalar: one PC on the chip
  - HW has the means to execute more than one instruction per cycle
  - Super**scalar**: it's more than a "scalar"; i.e., one thread
- TLP: Processor handles instructions from different threads/processes
- **NOTE:** we are not talking about multiple-cores; we are still talking about one CPU core here

# Measuring Speed of Execution

# Backdrop: CPU/OS juggle many programs at the same time



The screenshot shows the Windows Task Manager with the "Processes" tab selected. The table lists various programs and services along with their CPU usage, memory usage, disk activity, and network activity. The columns are labeled: Name, CPU, Memory, Disk, and Network.

Name	20% CPU	39% Memory	6% Disk	0% Network
Desktop Window Manager	1.1%	168.8 MB	0 MB/s	0 Mbps
Antimalware Service Executable	0%	144.2 MB	0 MB/s	0 Mbps
Google Chrome	0%	137.6 MB	0 MB/s	0 Mbps
Microsoft Outlook (32 bit) (3)	0%	122.6 MB	0 MB/s	0 Mbps
Microsoft PowerPoint (32 bit) (2)	0%	120.8 MB	0 MB/s	0 Mbps
Service Host: Windows Biometric	0%	118.4 MB	0 MB/s	0 Mbps
Dropbox (32 bit)	0%	115.6 MB	0 MB/s	0 Mbps
Adobe Acrobat DC (32 bit)	0%	111.5 MB	0 MB/s	0 Mbps
Box Sync (2)	0%	94.1 MB	0 MB/s	0 Mbps
Google Chrome	0%	85.8 MB	0 MB/s	0 Mbps
Google Chrome	0%	82.6 MB	0 MB/s	0 Mbps
Windows Explorer (3)	0.4%	61.3 MB	0 MB/s	0 Mbps
Service Host: Local System (19)	2.2%	59.8 MB	0.3 MB/s	0 Mbps
Microsoft Windows Search Indexer	0%	46.2 MB	0 MB/s	0 Mbps
Microsoft Word (32 bit)	0%	45.2 MB	0 MB/s	0 Mbps
Windows Shell Experience Host	0%	44.2 MB	0 MB/s	0 Mbps
Google Chrome	0%	42.2 MB	0 MB/s	0 Mbps
Microsoft Office Click-to-Run (SxS)	0%	40.6 MB	0 MB/s	0 Mbps
Microsoft Skype	0%	40.2 MB	0 MB/s	0 Mbps
Google Chrome	0%	33.4 MB	0 MB/s	0 Mbps
KeePass	0%	31.2 MB	0 MB/s	0 Mbps
Cortana	0%	28.9 MB	0 MB/s	0 Mbps
Adobe AcroCEF (32 bit)	0%	28.7 MB	0 MB/s	0 Mbps
WMI Provider Host	0%	28.2 MB	0 MB/s	0 Mbps
Camera	0%	27.9 MB	0 MB/s	0 Mbps
Qt Qtwebengineprocess (32 bit)	0%	23.1 MB	0 MB/s	0 Mbps

The hardware-software interplay:  
- CPU is hardware  
- The operating system (OS) is software

# Nomenclature (1/3)

- **Wall Clock Time**
  - Amount of time from the beginning of a program to the end of the program
  - Includes; i.e., factors in, all the housekeeping (running other programs, OS tasks, etc.) that CPU does while running our program
  - Perhaps most meaningful indicator of performance since this is what you see
- **CPU Execution Time**
  - Counts only the amount of time that is effectively dedicated to *\*your\** program
  - Many other programs running on your machine at the same time (time slicing)
  - Requires a profiling tool to assess (like `gprof`, for instance)
- On a dedicated machine; i.e., a quiet machine, Wall Clock Time and CPU Execution Time should be identical

# Nomenclature (2/3)

- Qualifying CPU Execution Time further:
  - **User time** – the time spent processing instructions compiled out of code generated by the user or in libraries that are directly called by user code
  - **System time** – time spent in support of the user's program but in instructions that were not generated out of code written by the user
    - OS support: open file for writing/reading, throw an exception, etc.
  - The line between the *user time* and *system time* is somewhat blurred, sometimes hard to delineate and differentiate
  - “system vs. user”: We bother to delineate since we can only cut into the “user time”; you can't rewrite system code

# Nomenclature (3/3)

- Clock cycle, cycle, tick – the length of the period for the processor clock
- Clock cycle: sets the pace at which the execution proceeds
  - In a pipeline process, in a perfect scenario, each clock cycle sees the completion of one instruction
- A constant value dictated by the frequency at which processor operates
  - Examples:
    - A 1 GHz processor has clock cycle of 1 nanosecond
    - A 2 GHz processor has clock cycle of 0.5 nanoseconds (500 picoseconds)

# The CPU Performance Equation

- Three ingredients control the amount of time for execution of a program:
  - Number of instructions that your program has (Instruction Count)
  - Average number of “clock-cycles per instructions” (CPI)
  - Clock Cycle Time
- The CPU Performance Equation reads:  $CPU\ Execution\ Time = \text{Instruction\ Count} \times CPI \times \text{Clock\ Cycle\ Time}$
- Alternatively, using the clock rate

$$CPU\ Execution\ Time = \frac{\text{Instruction\ Count} \times CPI}{\text{Clock\ Rate}}$$

# CPU Performance: How can we improve it?

$$CPU\ Execution\ Time = \text{Instruction Count} \times CPI \times Clock\ Cycle\ Time$$

- To improve performance the product of three factors should be reduced
- For a long time, we've piggy backed on "let's increase the frequency"; i.e., reduce clock cycle time
  - We eventually hit a wall this way (the "Power Wall")
- As repeatedly demonstrated in practice, reducing the Instruction Count (IC) often times leads to an increase in CPI. And the other way around.

# Side Trip: Revisiting, RISC or CISC?

- Ongoing argument – whether RISC or CISC is the better ISA
- The former is simple and therefore can be optimized easily. Yet it requires a large number of instructions to accomplish something in your C code
  - High IC but low CPI
- The latter is complex but instructions are expressive. Leads to few but expensive instructions to accomplish something in your C code
  - Low IC but high CPI
- Specific example: ARM vs. x86

# SPEC CPU Benchmarks

- Benchmarks help us understand what processor does well what
- Idea: gather a collection of programs that use a good mix of instructions and flex the muscles of a chip
- The “collection of programs”:
  - Should be non-trivial and widely used
  - Should be as objective as possible: not favor a chip manufacturer at the expense of another one
  - Example: a compiler is used extensively, so it makes sense to have it included in the benchmark
- Two common benchmarks:
  - Tied to programs dominated by floating point operations: CFP2006
  - Tied to programs dominated by integer arithmetic: CINT2006

# SPEC CPU Benchmark: Example, highlights AMD performance

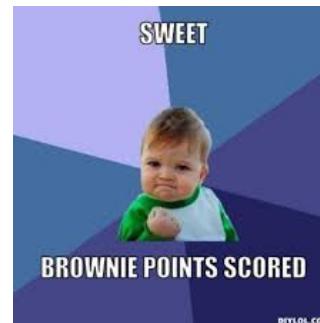
CINT2006 Programs		AMD Opteron X4 – 2356 (Barcelona)			
Description	Name	Instruction count [ $\times 10^9$ ]	CPI	Clock Cycle Time [seconds $\times 10^{-9}$ ]	CPU Exec. Time [seconds]
Interpreted string processing	perl	2118	0.75	0.4	637
Block-sorting compression	bzip2	2389	0.85	0.4	817
GNU C compiler	gcc	1050	1.72	0.4	724
Combinational optimization	mcf	336	10.00	0.4	1,345
Go game (AI)	go	1658	1.09	0.4	721
Search gene sequence	hmmer	2783	0.80	0.4	890
Chess game (AI)	sjeng	2176	0.96	0.4	837
Quantum computer simulation	libquantum	1623	1.61	0.4	1,047
Video compression	h264avc	3102	0.80	0.4	993
Discrete event simulation library	omnitpp	587	2.94	0.4	690
Games/path finding	aster	1082	1.79	0.4	773
XML parsing	xatancbmk	1058	2.70	0.4	1,143

# SPEC CPU Benchmark: Example, wrap up

- There are programs for which the CPI is less than 1.
  - Suggests that multiple issue is at play
- Why are there programs with CPI of 10?
  - The pipeline stalls a lot, most likely due to repeated cache misses and system memory transactions

# Important, common sense observation

- You get the brownie points for writing code whose instructions fly through the chip



- Nobody particularly impressed by instructions that take a long time to execute

# Big New Topic: Memory Aspects

# Transistors: why we love them and want to have lots of them

- Transistors used to make the CU and ALU work
- Transistors also used for memory, to store data and instructions (focus on this aspect next)

# SRAM: significant consumer of transistors

- SRAM – Static Random Access Memory
  - Integrated circuit whose elements combine to make up memory arrays
  - An “element”: a special circuit, called **flip-flop**
  - One flip-flop requires **four to six transistors**
  - Each of these elements stores **one bit** of information
  - Very **short access time**:  $\approx O(0)$  ns (order of magnitude(0):  $10^0 = 1$ )
  - Uniform access time of any element in the array (yet it's different to write than to read)
  - “Static” refers to the fact that once set, the element stores the value set as long as the element is powered
  - **Bulky**, since a storing element is “fat”; problematic to store a lot per unit area (compared to DRAM)
  - **Expensive**, since it requires four to six transistors and special layout and support requirements

# SRAM: It's Expensive

- Eats up relatively big chunk of the transistor budget
- Back of the envelop calculation:
  - Decent processor: 4 billion transistors
  - Assume 32 MB of L3 cache
    - 1 B = 8 bits
  - Assume 4 transistors per bit
    - Budget of transistors:  $32 \times 10^6 \times 8 \times 4 \approx 10^9$
    - 25% of all transistors used for L3 cache :-(

# DRAM

- DRAM type memory: the information stored as a charge in a **capacitor**
  - No charge: 0 signal
  - Some charge: 1 signal
- The good: **cheap**, requires only one capacitor and one transistor
- The bad: capacitors leak, so the charge or lack of charge should be reinforced every so often, dynamically
  - D in DRAM comes for the “dynamically” attribute
- Moral of the story: State of the capacitor should be refreshed every **millisecond** or so
  - Refreshing requires a small delay in memory accesses
    - Is this delay incurred often? (first order approximation answer)
      - Given frequency at which memory is accessed, refreshing every millisecond means issues might appear once every million cycles
      - Moral of the story: refresh is a **rare event** (in the life of the execution cycle)

# SRAM vs. DRAM: wrap-up

- SRAM access time: of the order of nanoseconds or tens of nanoseconds
  - Expensive but fast
  - Transistor hog
  - Needs no refresh
- DRAM access time: of the order of hundreds of nanoseconds
  - Less expensive but slow
  - Higher capacity per unit area
  - Needs refresh every 10-100 ms
  - Sensitive to disturbances
- Limit case: a 100X speedup if you can work off the SRAM

	Transistors per bit	Access Time	Persistent?	Sensitive?	Price	Applications
SRAM	6	1X	Yes	No	100X	Cache memories
DRAM	1	100X	No	Yes	1X	Main Memory

[Rauber&Runger]→

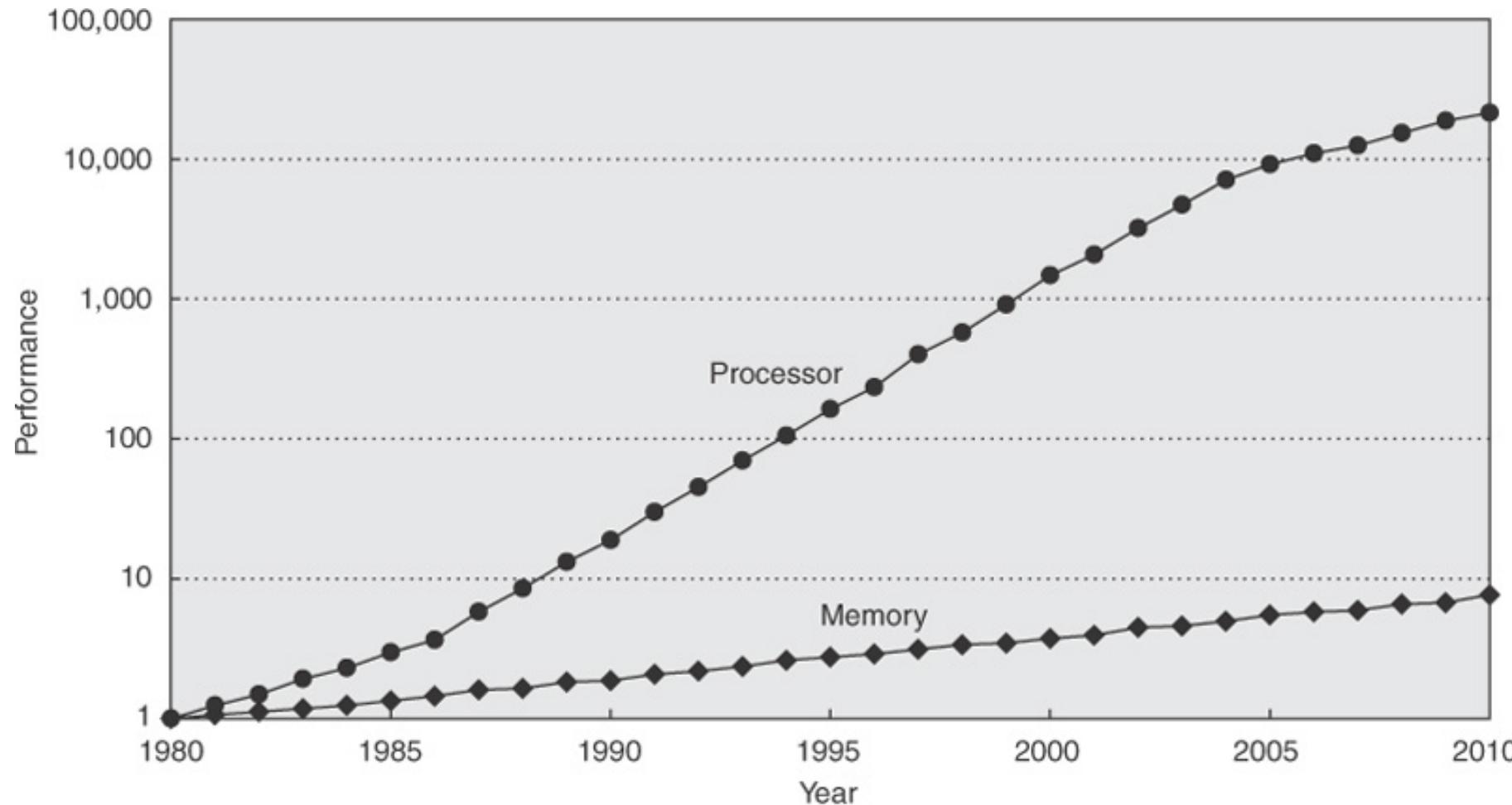
# Feature Comparison Between Memory Types

	SRAM	DRAM	Flash
Speed	Very fast	Fast	Very slow
Density	Low	High	Very high
Power	Low	High	Very low
Refresh	No	Yes	No
Retention	Volatile	Volatile	Non-volatile
Mechanism	Bi-stable Latch	Capacitor	Fowler-Nordheim tunneling

# New Topic: The Memory Hierarchy

- Common Sense Observations
  - Observation 1: over time, speed of data processing increased faster than speed at which we could bring over data to be processed
  - Observation 2: the bigger the memory used to store data, the longer it takes to find/access an entry therein
    - Compromise between size and speed of access

# Backdrop: Widening of the Processor-DRAM Speed Gap



# Consequences of common sense observations

- Consequences of Observations 1 & 2: a **hierarchy of memories** has emerged
  - It took some time to shape up, happened through a continuous improvement process
  - Managing the hierarchy not trivial → a dedicated hardware asset called MMU (memory management unit) handling this task
    - MMU's purview: cache control, virtual memory management, memory protection, bus arbitration, etc.

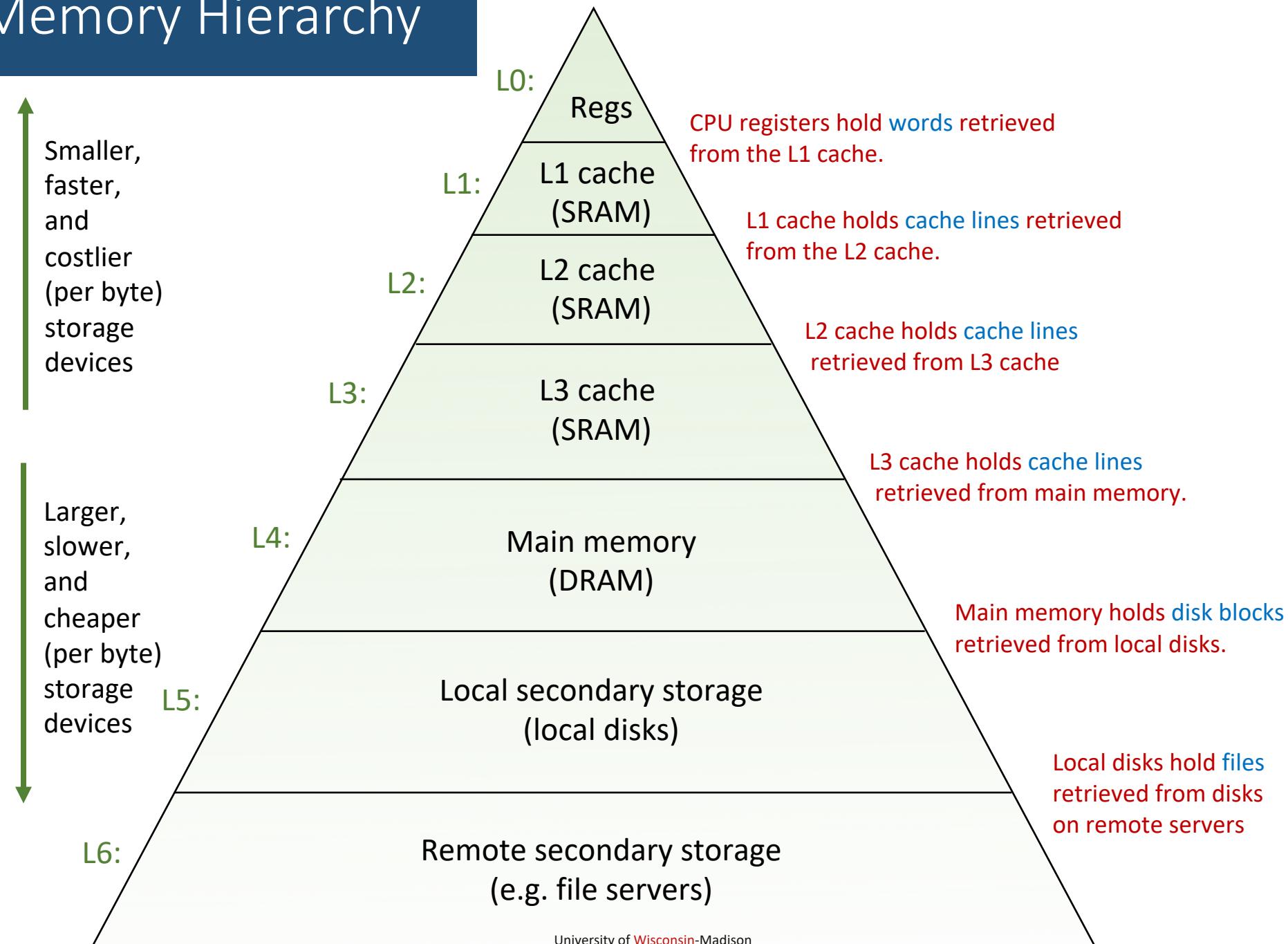
# The memory hierarchy, how it shapes up

- Since SRAM is expensive and bulkier, can't have too much
  - Plagued by space & cost constraints
- Compromise:
  - Have some SRAM on-chip, making up what is called the “cache”
  - Have a lot of inexpensive DRAM off-chip, making up the “main memory”
    - This is what people have in mind when they ask you “How much memory does your system have”?
- In our code, we hope to have low “average memory access time” by hitting the cache repeatedly instead of taking costly trips to main memory

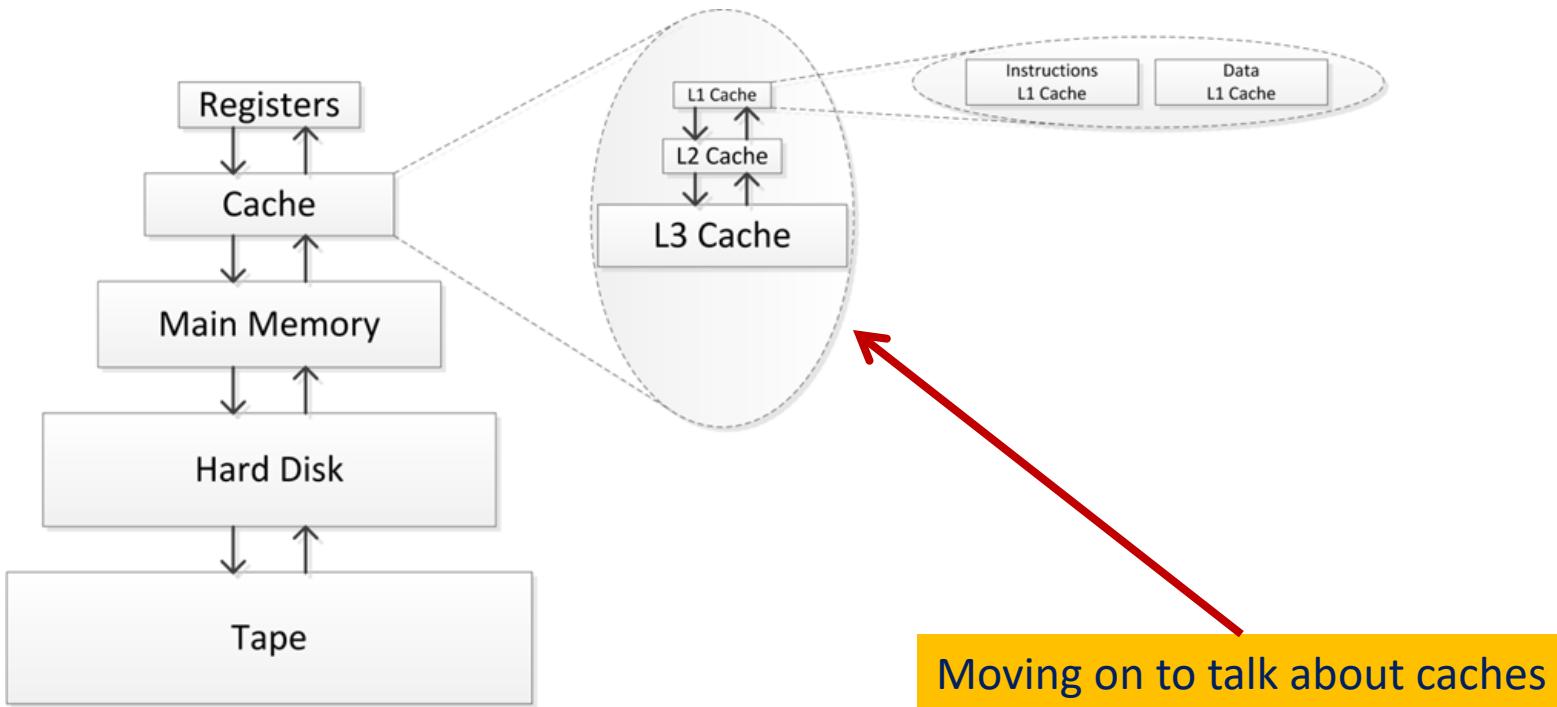
# There's even a cache hierarchy...

- Simplest memory hierarchy:
  - Main Memory + One Cache (typically called L1 cache)
- Today's memory architectures typically have deeper hierarchy: L1+L2+L3
  - L1 faster and smaller than L2
  - L2 faster and smaller than L3
- All caches are typically on the chip and are SRAM

# Typical Memory Hierarchy



# Memory Hierarchy



# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 05

01/31/2020

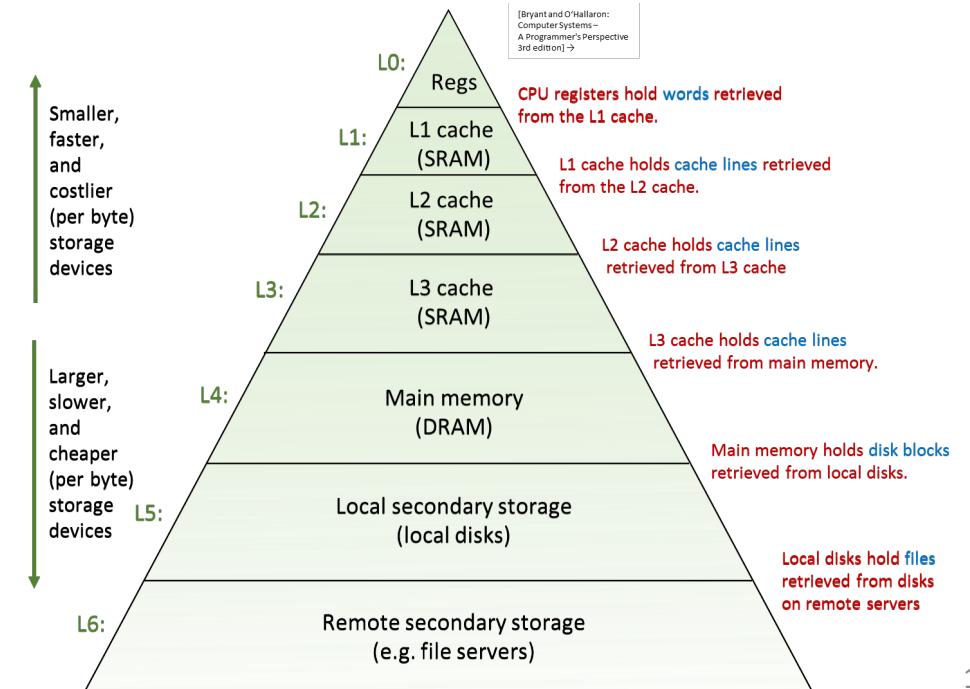
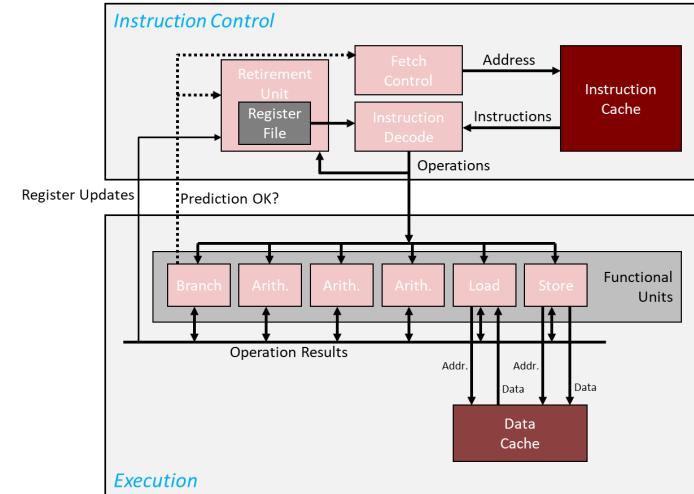
# Quote of the day

“English? Who needs that? I'm never going to England.”

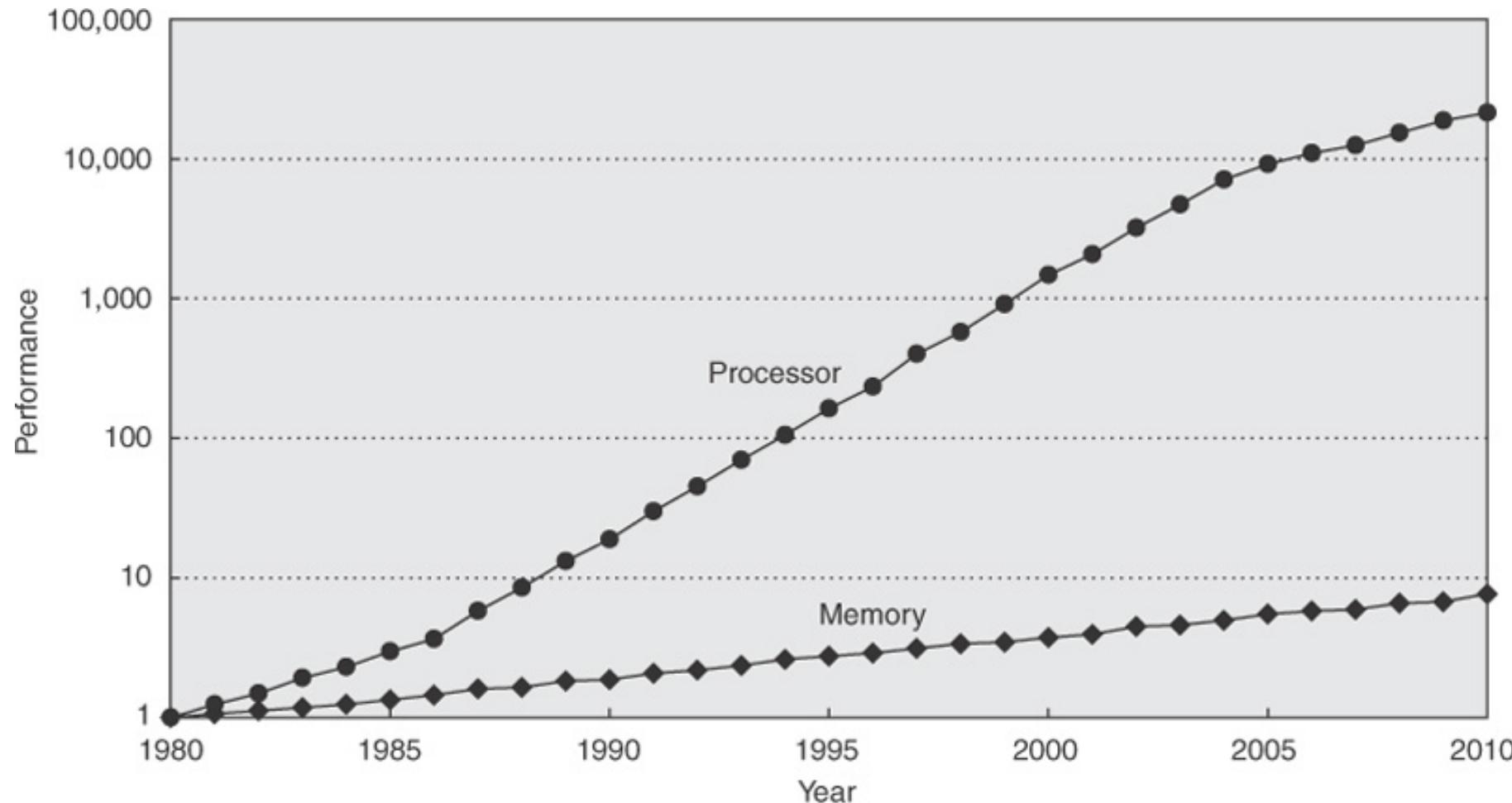
-- Homer Simpson, Safety Inspector, anticipating Brexit [1989 - ]

# Before we get going...

- Last time: the Hardware ↔ Software interplay
  - Thread Level Parallelism (TLP)
  - Measuring execution performance, quick thoughts
  - The memory hierarchy
- Today
  - The Cache
  - The virtual memory
- Other tidbits:
  - Please take care of the assigned reading
  - Another assignment went out, due Th at 9 pm
  - Use Piazza to get things going (for you or others)

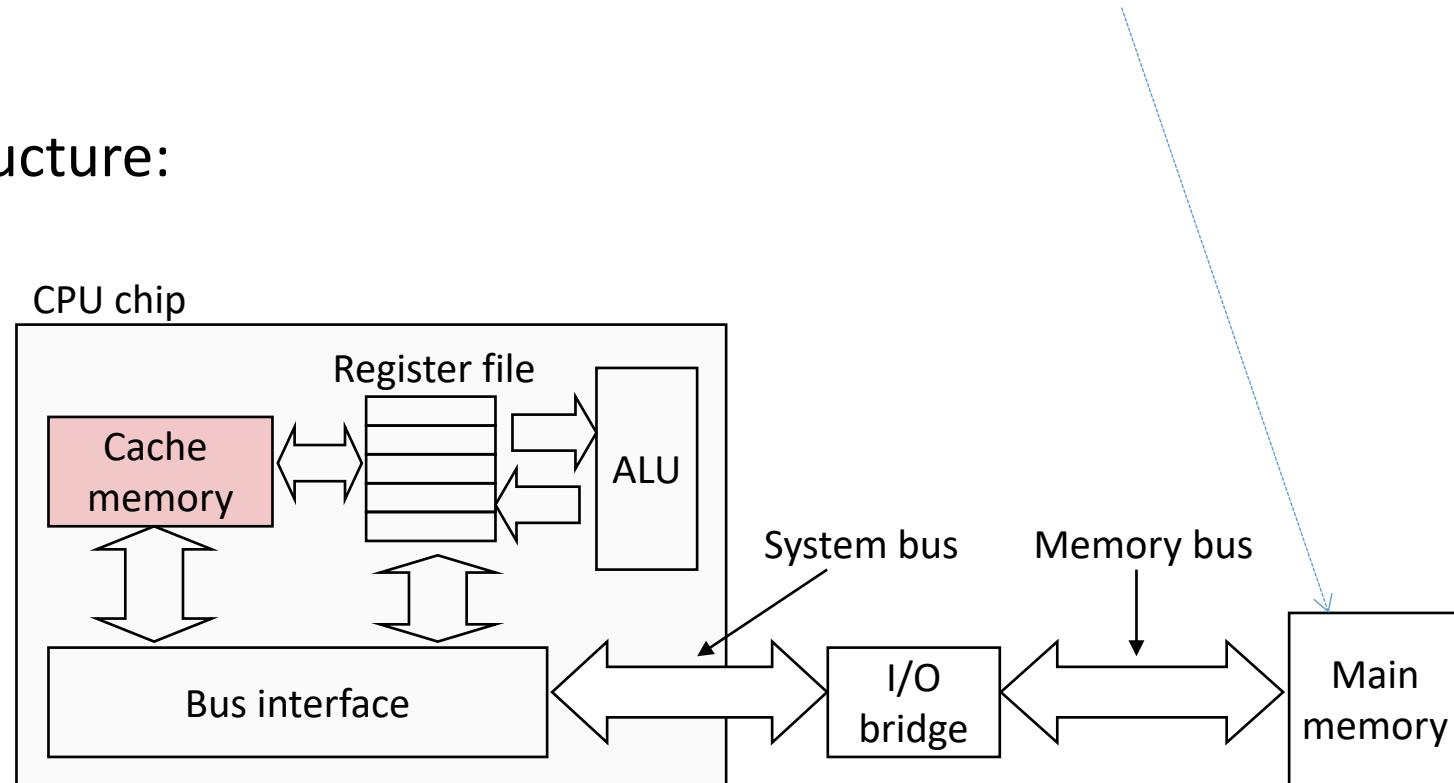


# Memory Speed: Widening of the Processor-DRAM Speed Gap



# Cache Memories

- Cache memories are small, fast, on-chip SRAM-based memories
  - Purpose: to store close to the CU/ALU frequently accessed blocks of main, or system, memory
- Typical system structure:

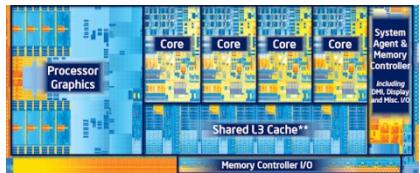


# What it Really Looks Like

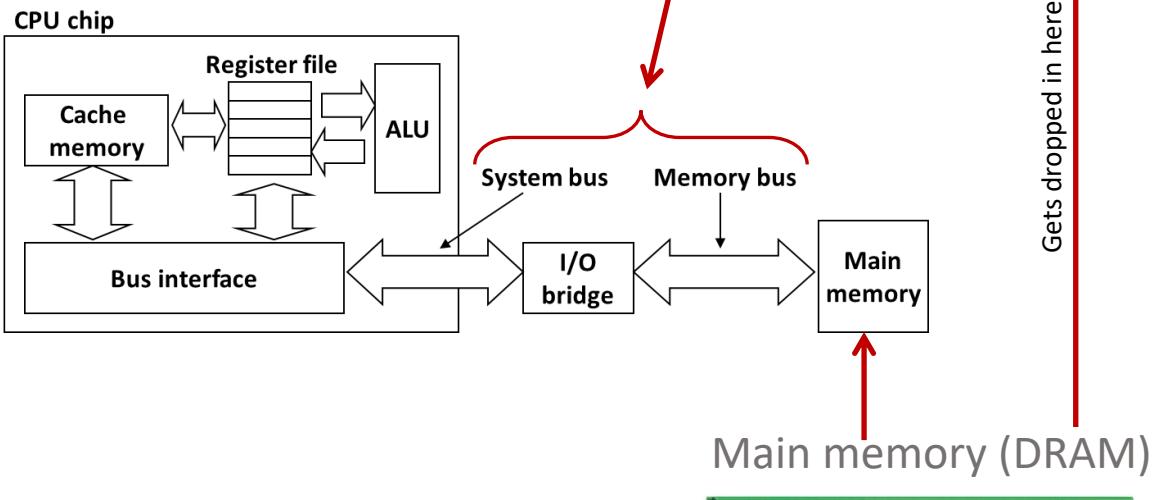
CPU (Intel Core i7)



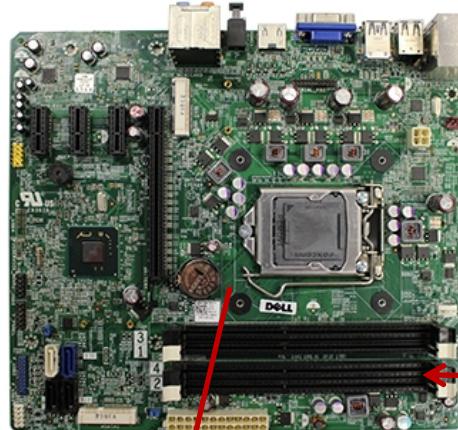
Source: PC Magazine



Source: techreport.com



Motherboard *Source: Dell*



Desktop PC

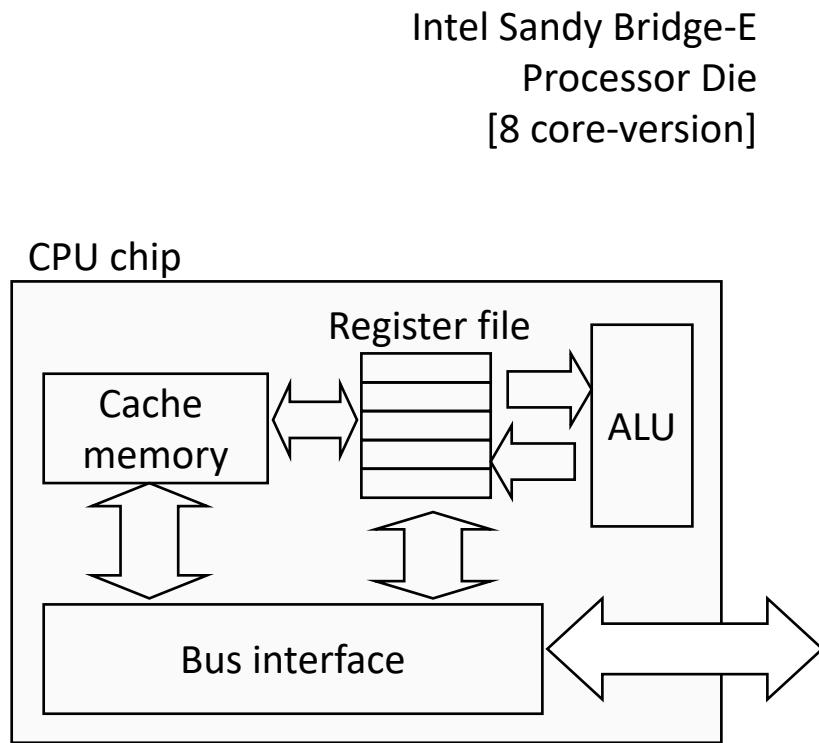


Source: Dell

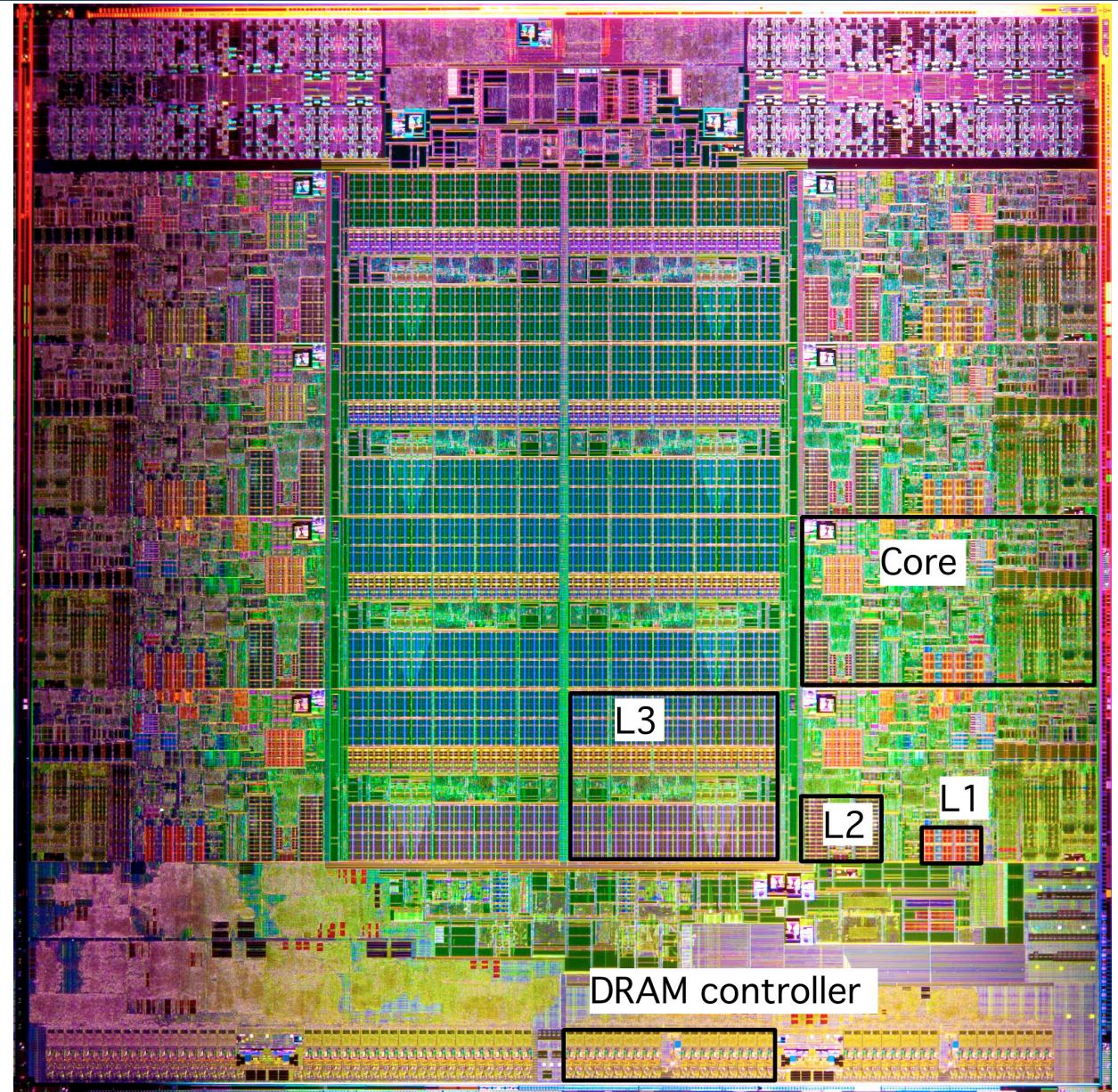


Source: Dell

# What it Really Looks Like [contd.]

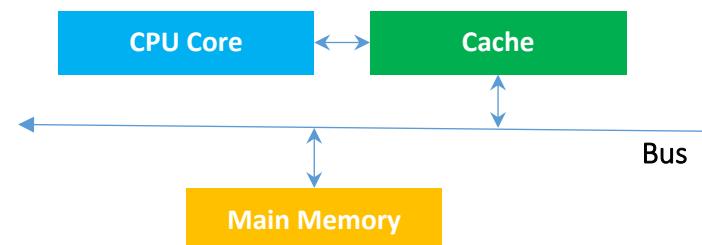


L1: 32KB Instruction + 32KB Data  
L2: 256KB  
L3: 5–20MB



# How Cache Comes into Play

- All\* read/write operations go through cache
- Caches used to hold both data and instructions
  - Data and instructions caches are in general different
    - Intel: L1 DC (data cache) and L1 IC (instruction cache) have been distinct since of 1993

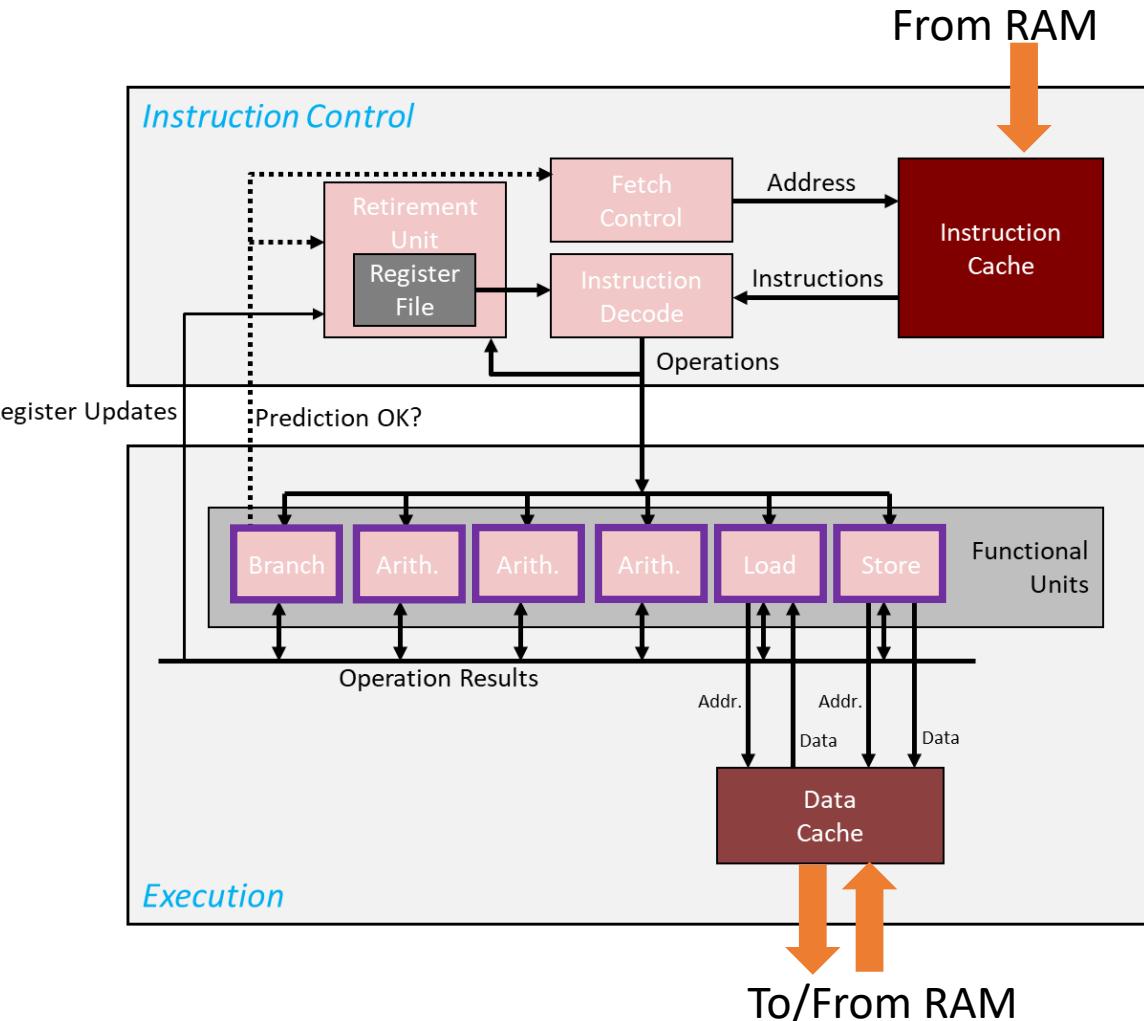


# Caches: Come In Two Flavors

- Data caches feed processor with data manipulated during execution
  - If processor would rely on data provided by main memory the execution would be pitifully slow
    - “Processor Clock” much faster than the “Memory Clock”
      - Caches alleviate the memory pressure – processors are data hungry
- Instruction caches: used to store instructions
  - Simpler to deal with compared to the data caches
    - By and large, instruction use is much more predictable than data use

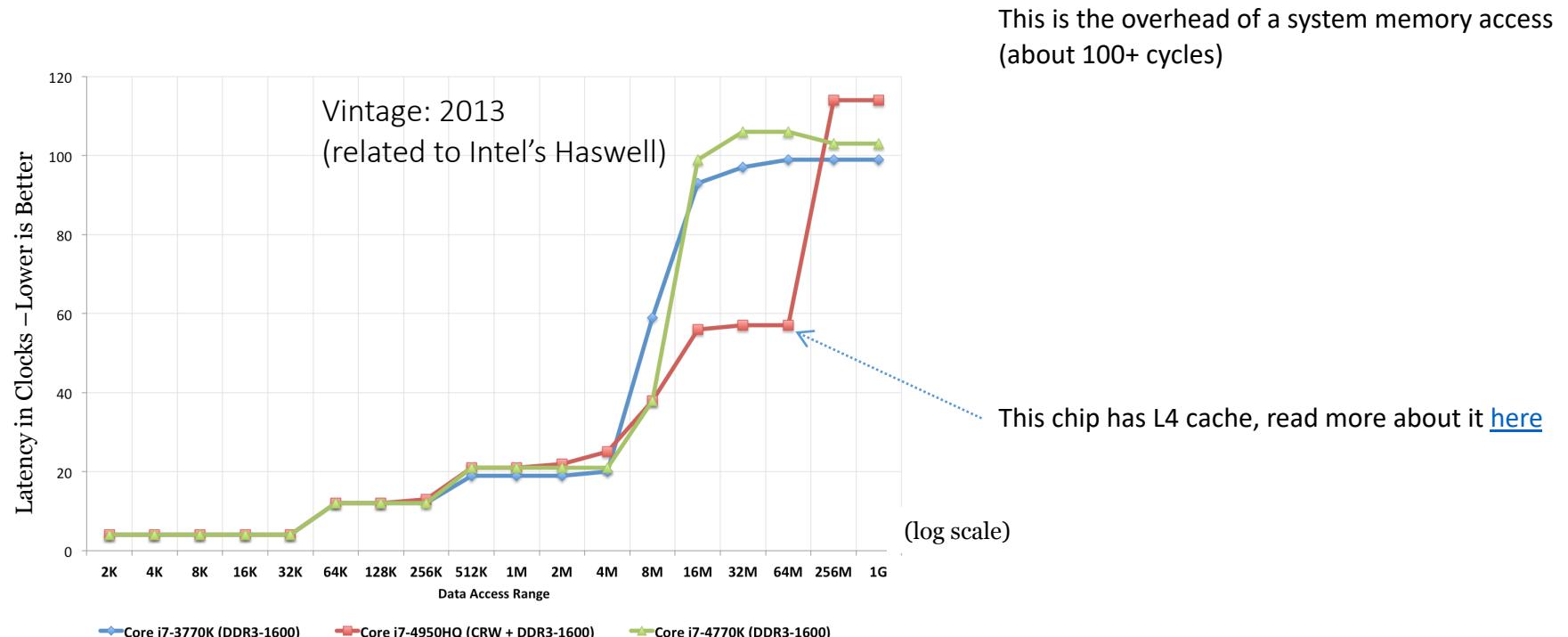
# Processor Clock vs. Memory Clock

- Processor Clock: (look for the box  )
  - Dictates/defines the frequency at which **Functional Units** operate
- Memory Clock: 
  - Dictates/defines the frequency used to move data into/out from main memory into cache



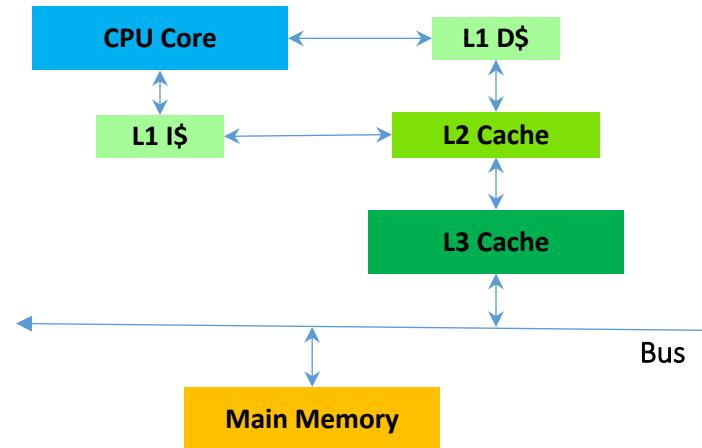
# Caches: How large? Also, how fast?

- Ratio between cache size and main memory size
  - Reasonable to expect something like [1:1000](#)
    - 2013: 4 MB of L3 cache, versus 4 GB of main memory
    - 2017: 55 MB of L3 cache, versus 64 GB of main memory

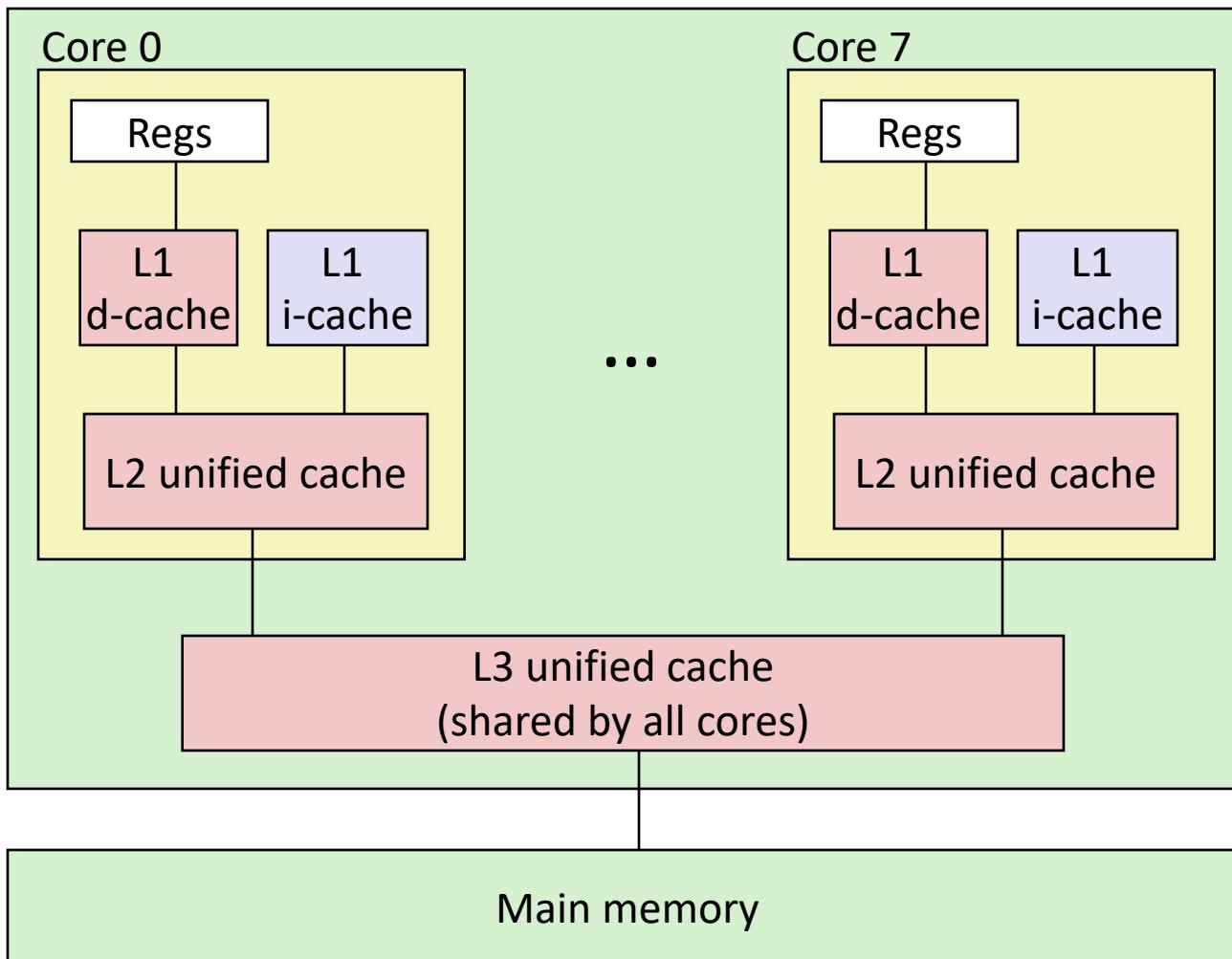


# Multiple Levels of Cache

- Commodity chips today have 3 cache levels
  - (some chips have a fourth cache level, see previous slide)



## Processor package



L1 i-cache and d-cache:  
32 KB, 8-way assoc.

Access: 4 cycles

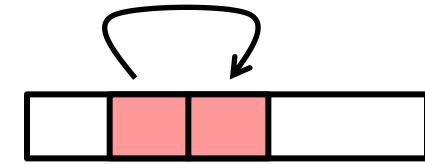
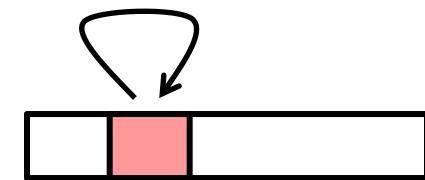
L2 unified cache:  
256 KB, 8-way assoc.  
Access: 10 cycles

L3 unified cache:  
8 MB, 16-way assoc.  
Access: 40-75 cycles

Block size: 64 bytes for all  
caches.

# Why Caches Work: the “Principle of Locality”

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently
- **Temporal locality:**
  - “Recently referenced items are likely to be referenced again in the near future”
- **Spatial locality:**
  - “Items with nearby addresses tend to come into use together”



# Spatial and Temporal Locality

- *Temporal Locality* for memory access by a program
  - Idea: If you access a variable at some time, then you'll probably keep accessing the same variable for a while
    - Example: a for loop with some variables inside the loop → you keep accessing those variables as long as you loop
- *Spatial Locality* for memory access by a program
  - A memory access pattern characterized by bursts of repeated requests for data that is physically located within the same memory region
  - “Bursts” because these accesses should happen in a sufficiently short interval of time (lest the cache line gets evicted)

# Example, regarding concept of “locality”

```
// some simple code snippet
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data references
  - Reference array `a` elements in succession (stride-1 reference pattern) → **spatial locality**
  - Reference variable `sum` each iteration → **temporal locality**
- Instruction references
  - Reference instructions in sequence → **spatial locality**
  - Cycle through loop repeatedly → **temporal locality**

# Example: Adding the Entries in a 2D Matrix

- Example used over several slides to show how “locality” and caches impact performance
- Very related to your second assignment

# Example [motivation]: Adding the Entries in a 2D Matrix

- Two equivalent ways to add the entries in a matrix

$$\begin{pmatrix} 2 & -3 & 0 \\ -1 & 1 & 7 \end{pmatrix}$$

- Row-wise:

$$sumElements = \underbrace{(2 - 3 + 0)}_{\text{First row}} + \underbrace{(-1 + 1 + 7)}_{\text{Second row}} = 6$$

- Column-wise:

$$sumElements = \underbrace{(2 - 1)}_{\text{First column}} + \underbrace{(-3 + 1)}_{\text{Second column}} + \underbrace{(0 + 7)}_{\text{Third column}} = 6$$

- Example used over several slides to show how “locality” and caches impact performance

# Squashed format vs. 2D format

- Storing a matrix: **squashed format** OR **2D format**
- Example: 
$$\begin{pmatrix} 2 & -3 & 0 \\ -1 & 1 & 7 \end{pmatrix}$$
- **Squashed format** – using one long contiguous array of 6 entries. Like in your homework
- **The 2D format** – using two arrays:
  - Each has 3 entries
  - Each of the two arrays stored somewhere in the memory (they might be far from each other)

# First way to represent 2D matrix: the “squashed,” 1D approach

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

struct squashedMatrix {
    /// 2D matrix stored row-wise in a one dimensional array
    unsigned int height; // number of rows in matrix
    unsigned int width; // number of columns in matrix
    double *pMatVals;
};

int main() {
    struct squashedMatrix mat1D;

    mat1D.height = 1000; // perhaps you set this based on user input
    mat1D.width = 2000; // perhaps you set this based on user input
    mat1D.pMatVals = (double *)malloc(mat1D.height * mat1D.width * sizeof(double));

    unsigned int i, j;
    for (i = 0; i < mat1D.height; i++)
        for (j = 0; j < mat1D.width; j++)
            mat1D.pMatVals[i * mat1D.width + j] = 1. / (i + j + 1.);

    // Compute sum of elements
    double sum = 0.;
    for (i = 0; i < mat1D.height; i++)
        for (j = 0; j < mat1D.width; j++)
            sum += mat1D.pMatVals[i * mat1D.width + j];

    printf("The sum of the elements is: %f\n", sum);

    // Done with the matrix; free the mem
    free(mat1D.pMatVals);

    return 0;
}
```

- One long array stores all entries in the matrix
  - E.g., a  $2 \times 3$  matrix will have an array of size 6
- 2D matrix stored in 1D array, row-wise fashion
  - All the entries in the first row are stored first, followed by the entries in the second row, which are followed by the matrix in the third row, etc.
- Note that we use malloc once
  - We'll have to use free once when we're done
- Windows executable called “matrix1D.exe”

# A second way to represent the 2D matrix: the 2D approach

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

struct Matrix2D {
    /// 2D matrix stored row-wise in a one dimensional array
    unsigned int height; // number of rows in matrix
    unsigned int width; // number of columns in matrix
    double **pRows;
};

int main() {
    struct Matrix2D mat2D;
    mat2D.height = 1000; // perhaps you set this based on user input
    mat2D.width = 2000; // perhaps you set this based on user input

    mat2D.pRows = (double **)malloc(mat2D.height * sizeof(double *));

    unsigned int i;
    for (i = 0; i < mat2D.height; i++)
        mat2D.pRows[i] = (double *)malloc(mat2D.width * sizeof(double));

    // Get meaningful values in this matrix
    unsigned int j;
    for (i = 0; i < mat2D.height; i++)
        for (j = 0; j < mat2D.width; j++)
            mat2D.pRows[i][j] = 1. / (i + j + 1.);

    // Compute sum of elements
    double sum = 0;
    for (i = 0; i < mat2D.height; i++)
        for (j = 0; j < mat2D.width; j++)
            sum += mat2D.pRows[i][j];

    printf("The sum of the elements is: %f\n", sum);

    // Don't data anymore, free mem; note that free is used twice
    for (i = 0; i < mat2D.height; i++)
        free(mat2D.pRows[i]);
    free(mat2D.pRows);

    return 0;
}
```

- First we allocate room to store a number of pointers to double
  - There are “height” of these “double \*” pointers
  - These pointers stored in the array pRows
- For each matrix row (and there are “height” of them), we allocate enough room to store “width” doubles
  - These are the values that show up in a row of the matrix
- Note that I use malloc multiple times
  - We’ll have to use free multiple times
- Windows executable called “matrix2D.exe”

# The squashed approach, revisited: locality vs. non-locality

```
#include <Windows.h> // need this for timing
#include <stdio.h>
#include <stdlib.h>

struct squashedMatrix {
    // 2D matrix stored row-wise in a one dimensional array
    unsigned int height; // number of rows in matrix
    unsigned int width; // number of columns in matrix
    double *pMatVals;
};

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Pass 1 for row-wise, pass 2 for column-wise sum.\n");
        return 1;
    }
    int option = atoi(argv[1]);

    struct squashedMatrix mat1D;

    mat1D.height = 2000; // perhaps you set this based on user input
    mat1D.width = 2000; // perhaps you set this based on user input
    mat1D.pMatVals = (double *)malloc(mat1D.height * mat1D.width * sizeof(double));

    unsigned int i, j;
    for (i = 0; i < mat1D.height; i++)
        for (j = 0; j < mat1D.width; j++)
            mat1D.pMatVals[i * mat1D.width + j] = 1. / (i + j + 1.);
```

```
// Compute sum of elements
double sum = 0.;
__int64 ctr1 = 0, ctr2 = 0, freq = 0; // need this for timing
if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0) {
    if (option == 1) {
        for (i = 0; i < mat1D.height; i++)
            for (j = 0; j < mat1D.width; j++)
                sum += mat1D.pMatVals[i * mat1D.width + j];
    } else if (option == 2) {
        for (j = 0; j < mat1D.width; j++) {
            for (i = 0; i < mat1D.height; i++)
                sum += mat1D.pMatVals[i * mat1D.width + j];
        }
    } else {
        printf("Bad input option.\n");
        return 1;
    }
    QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
}

printf("The sum of the elements is: %f\n", sum);
printf("Time spent in microseconds: %f\n", (ctr2 - ctr1) * 1.0 / freq);

// Done with the matrix; free the mem
free(mat1D.pMatVals);

return 0;
}
```

# Results, release build: 6 to 7 times faster if accessed w/ locality

```
TRUMAN Release> ./matrixRep1D.exe
Pass 1 for row-wise, pass 2 for column-wise sum.
TRUMAN Release>
TRUMAN Release>
TRUMAN Release> ./matrixRep1D.exe 1
The sum of the elements is: 2772.088785
Time spent in microseconds: 0.003622
TRUMAN Release>
TRUMAN Release>
TRUMAN Release> ./matrixRep1D.exe 2
The sum of the elements is: 2772.088785
Time spent in microseconds: 0.023060
TRUMAN Release>
```

OS Name	Microsoft Windows 10 Enterprise 2016 LTSB
System Type	x64-based PC
Processor	Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz, 3401 Mhz, 6 Core(s), 12 Logical Processor(s)
Installed Physical Memory (RAM)	32.0 GB
Compiler	Visual Studio 2015 (Enterprise)

# The 2D approach, revisited: locality vs. non-locality

```
#include <Windows.h> // need this for timing
#include <stdio.h>
#include <stdlib.h>

struct Matrix2D {
    /// 2D matrix stored row-wise in a one dimensional array
    unsigned int height; // number of rows in matrix
    unsigned int width; // number of columns in matrix
    double **pRows;
};

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Pass 1 for row-wise, pass 2 for column-wise sum.\n");
        return 1;
    }
    int option = atoi(argv[1]);

    struct Matrix2D mat2D;
    mat2D.height = 2000; // perhaps you set this based on user input
    mat2D.width = 2000; // perhaps you set this based on user input

    mat2D.pRows = (double **)malloc(mat2D.height * sizeof(double *));
    unsigned int i;

    for (i = 0; i < mat2D.height; i++)
        mat2D.pRows[i] = (double *)malloc(mat2D.width * sizeof(double));

    // Get meaningful values in this matrix
    unsigned int j;
    for (i = 0; i < mat2D.height; i++)
        for (j = 0; j < mat2D.width; j++)
            mat2D.pRows[i][j] = 1. / (i + j + 1.);
```

```
// Compute sum of elements
double sum = 0;
__int64 ctr1 = 0, ctr2 = 0, freq = 0; // need this for timing
if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0) {
    if (option == 1) {
        for (i = 0; i < mat2D.height; i++)
            for (j = 0; j < mat2D.width; j++)
                sum += mat2D.pRows[i][j];
    } else if (option == 2) {
        for (j = 0; j < mat2D.width; j++)
            for (i = 0; i < mat2D.height; i++)
                sum += mat2D.pRows[i][j];
    } else {
        printf("Bad input option.\n");
        return 1;
    }
    QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
}

printf("The sum of the elements is: %f\n", sum);
printf("Time spent in microseconds: %f\n", (ctr2 - ctr1) * 1.0 / freq);

// Don't data anymore, free mem; note that free is used twice
for (i = 0; i < mat2D.height; i++)
    free(mat2D.pRows[i]);
free(mat2D.pRows);

return 0;
```

# Results, release build: $\approx$ 5 times faster if accessed w/ locality

```
TRUMAN Release> ./matrixRep2D.exe
Pass 1 for row-wise, pass 2 for column-wise sum.
TRUMAN Release>
TRUMAN Release>
TRUMAN Release> ./matrixRep2D.exe 1
The sum of the elements is: 2772.088785
Time spent in microseconds: 0.004075
TRUMAN Release>
TRUMAN Release>
TRUMAN Release> ./matrixRep2D.exe 2
The sum of the elements is: 2772.088785
Time spent in microseconds: 0.020078
TRUMAN Release>
```

OS Name	Microsoft Windows 10 Enterprise 2016 LTSB
System Type	x64-based PC
Processor	Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz, 3401 Mhz, 6 Core(s), 12 Logical Processor(s)
Installed Physical Memory (RAM)	32.0 GB
Compiler	Visual Studio 2015 (Enterprise)

# The two scenarios, side by side: squashed and proper 2D

```
TRUMAN Release> ./matrixRep1D.exe
Pass 1 for row-wise, pass 2 for column-wise sum.
TRUMAN Release>
TRUMAN Release>
TRUMAN Release> ./matrixRep1D.exe 1
The sum of the elements is: 2772.088785
Time spent in microseconds: 0.003622
TRUMAN Release>
TRUMAN Release>
TRUMAN Release> ./matrixRep1D.exe 2
The sum of the elements is: 2772.088785
Time spent in microseconds: 0.023060
TRUMAN Release>
```

```
TRUMAN Release> ./matrixRep2D.exe
Pass 1 for row-wise, pass 2 for column-wise sum.
TRUMAN Release>
TRUMAN Release>
TRUMAN Release> ./matrixRep2D.exe 1
The sum of the elements is: 2772.088785
Time spent in microseconds: 0.004075
TRUMAN Release>
TRUMAN Release>
TRUMAN Release> ./matrixRep2D.exe 2
The sum of the elements is: 2772.088785
Time spent in microseconds: 0.020078
TRUMAN Release>
```

OS Name	Microsoft Windows 10 Enterprise 2016 LTSB
System Type	x64-based PC
Processor	Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz, 3401 Mhz, 6 Core(s), 12 Logical Processor(s)
Installed Physical Memory (RAM)	32.0 GB
Compiler	Visual Studio 2015 (Enterprise)

# Final Exam type question: Qualitative Estimates of Locality

- Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer
- **Question:** Below, do we have good locality with respect to array a?

```
int sum_array_rows(int** a)
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

# Final Exam type question: Qualitative Estimates of Locality

- **Question:** Does this function have good locality with respect to array a?

```
int sum_array_cols(int** a)
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

# Taking it to the next level: 3D Matrices – good locality

OS Name	Microsoft Windows 10 Enterprise 2016 LTSB
System Type	x64-based PC
Processor	Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz, 3401 Mhz, 6 Core(s), 12 Logical Processor(s)
Installed Physical Memory (RAM)	32.0 GB
Compiler	Visual Studio 2015 (Enterprise)

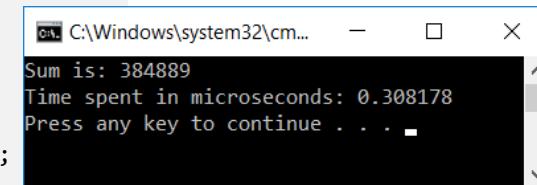
```
#include<iostream>
#include<Windows.h> // need this for timing

int main() {
    const int M = 700;
    __int64 ctr1 = 0, ctr2 = 0, freq = 0;
    double*** bPPP;
    bPPP = (double***)malloc(M * sizeof(double**));
    for (int i = 0; i < M; i++) {
        bPPP[i] = (double**)malloc(M * sizeof(double*));
        for (int j = 0; j < M; j++) {
            bPPP[i][j] = (double*)malloc(M * sizeof(double));
            for (int k = 0; k < M; k++)
                bPPP[i][j][k] = 1. / (i + j + k + 1.);           // storing some smallish number
        }
    }

    double sum = 0.;
    if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0) {
        // sum up all entries in 3D array
        for (int i = 0; i < M; i++)
            for (int j = 0; j < M; j++)
                for (int k = 0; k < M; k++)
                    sum += bPPP[i][j][k];
    }

    QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
}

std::cout << "Sum is: " << sum << std::endl;
std::cout << "Time spent in microseconds: " << ((ctr2 - ctr1) * 1.0 / freq) << std::endl;
return 0;
}
```



# Taking it to the next level: 3D Matrices – bad locality

OS Name	Microsoft Windows 10 Enterprise 2016 LTSB
System Type	x64-based PC
Processor	Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz, 3401 Mhz, 6 Core(s), 12 Logical Processor(s)
Installed Physical Memory (RAM)	32.0 GB
Compiler	Visual Studio 2015 (Enterprise)

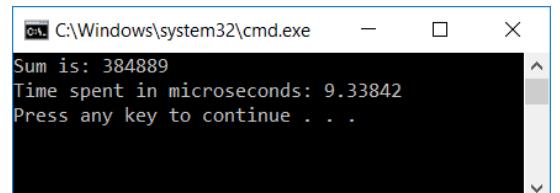
```
#include<iostream>
#include<Windows.h> // need this for timing

int main() {
    const int M = 700;
    __int64 ctr1 = 0, ctr2 = 0, freq = 0;
    double*** bPPP;
    bPPP = (double***)malloc(M * sizeof(double**));
    for (int i = 0; i < M; i++) {
        bPPP[i] = (double**)malloc(M * sizeof(double*));
        for (int j = 0; j < M; j++) {
            bPPP[i][j] = (double*)malloc(M * sizeof(double));
            for (int k = 0; k < M; k++)
                bPPP[i][j][k] = 1. / (i + j + k + 1.);           // storing some smallish number
        }
    }

    double sum = 0.;
    if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0) {
        // sum up all entries in 3D array
        for (int k = 0; k < M; k++)
            for (int j = 0; j < M; j++)
                for (int i = 0; i < M; i++)
                    sum += bPPP[i][j][k];

        QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);
        QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    }

    std::cout << "Sum is: " << sum << std::endl;
    std::cout << "Time spent in microseconds: " << ((ctr2 - ctr1) * 1.0 / freq) << std::endl;
    return 0;
}
```



# Running the code in Debug vs. Release mode: big difference?

OS Name	Microsoft Windows 10 Enterprise 2016 LTSB
System Type	x64-based PC
Processor	Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz, 3401 Mhz, 6 Core(s), 12 Logical Processor(s)
Installed Physical Memory (RAM)	32.0 GB
Compiler	Visual Studio 2015 (Enterprise)

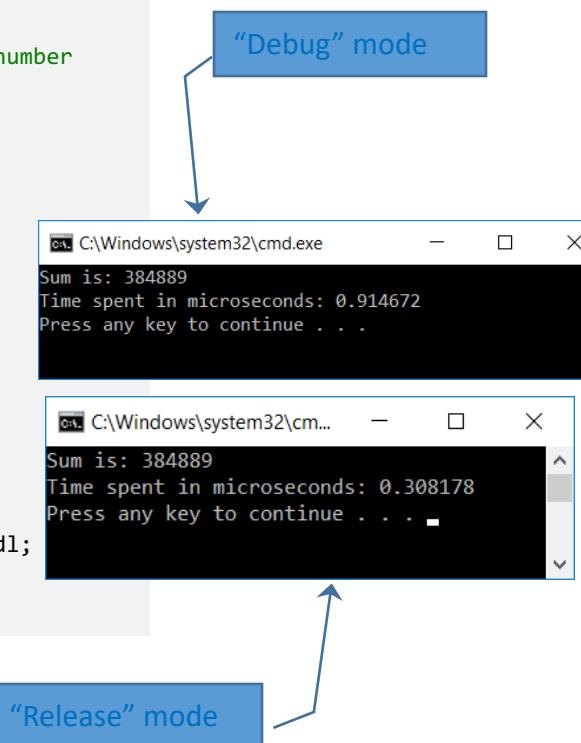
```
#include<iostream>
#include<Windows.h> // need this for timing

int main() {
    const int M = 700;
    __int64 ctr1 = 0, ctr2 = 0, freq = 0;           // need this for timing
    double*** bPPP;                                // use this to store values
    bPPP = (double***)malloc(M * sizeof(double**));
    for (int i = 0; i < M; i++) {
        bPPP[i] = (double**)malloc(M * sizeof(double*));
        for (int j = 0; j < M; j++) {
            bPPP[i][j] = (double*)malloc(M * sizeof(double));
            for (int k = 0; k < M; k++)
                bPPP[i][j][k] = 1. / (i + j + k + 1.);      // storing some smallish number
        }
    }

    double sum = 0.;
    if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0) {
        // sum up all entries in 3D array
        for (int i = 0; i < M; i++)
            for (int j = 0; j < M; j++)
                for (int k = 0; k < M; k++)
                    sum += bPPP[i][j][k];

        QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);
        QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    }

    std::cout << "Sum is: " << sum << std::endl;
    std::cout << "Time spent in microseconds: " << ((ctr2 - ctr1) * 1.0 / freq) << std::endl;
    return 0;
}
```



# Three observations related to our 2D and 3D examples

- **Observation 1:** Final value for `sum` - stays the same (at least as far as we can tell)
  - We'll talk more about this when we do parallel computing w/ CUDA and then OpenMP
- **Observation 2:** Difference between Debug vs Release builds
  - 3X to 4X speedup normal to see when going from debug to release (see also results on next slide)
- **Observation 3:** Amount of time to get the result quite different
  - Release mode build: the undesirable solution for 3D matrix is 30 times slower
    - This is the case you see in practice – you build in release

# Observations on “Observation 1”

- The fact that the results are the same is remarkable

The image shows two separate command-line windows from Windows. Both windows are titled 'C:\Windows\system32\cmd...' and have a standard window title bar with minimize, maximize, and close buttons. The top window displays the following text:  
Time spent in microseconds: 9.31855  
Sum is: 384889.16037645395  
Press any key to continue . . .  
The bottom window displays:  
Time spent in microseconds: 0.30175  
Sum is: 384889.16037645395  
Press any key to continue . . .

- Why remarkable?
  - Finite precision arithmetic: addition loses its commutative and associative attributes

$$A_1 + \dots + A_{10} \approx A_{10} + \dots + A_1$$

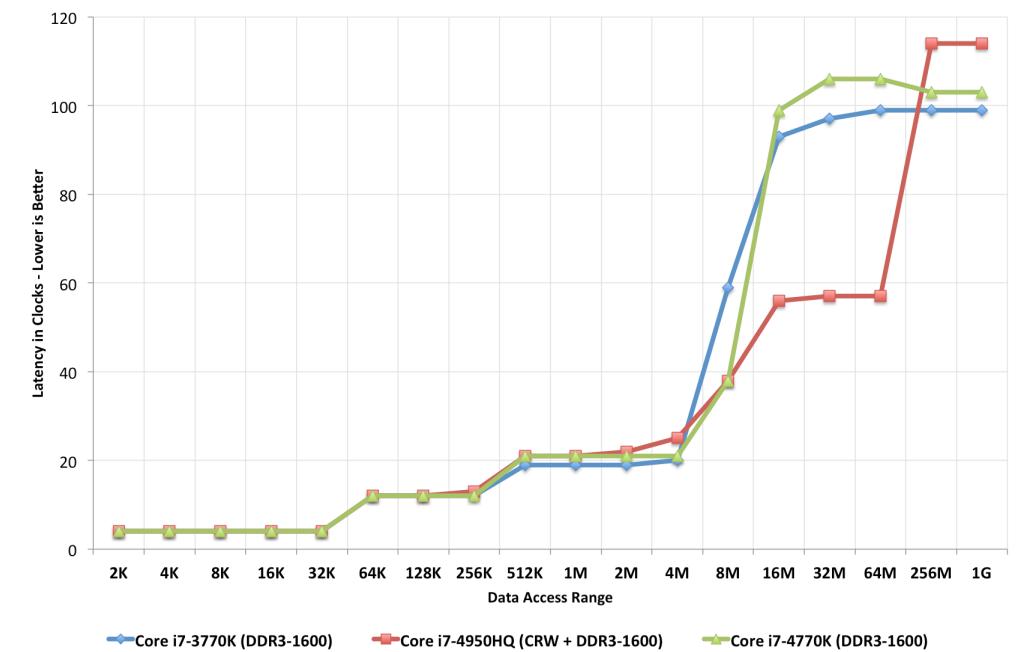
Because of finite precision

$$((\dots ((A_1 + A_2) + A_3) + A_4) + \dots + A_{10}) \approx (A_1 + (A_2 + (\dots + (A_9 + A_{10})) \dots ))$$

Because of finite precision

# Observations on “Observation 3” (most important)

- First time in 759 discussing how to improve performance
  - This type of gain; i.e., 30X speed up, not encountered every day...
- Q: What mechanism comes into play to yield a 30X speed gain?
- A: The cache
  - L1 cache: 4 cycles
  - System memory access: 120 cycles



# Indirection and how it's hurting... An example.

[one slide side trip]

- Indirection: given an index  $i$ , you don't need  $a[i]$ 
  - Rather, you need  $a[\text{indir}[i]]$  (this is one level of indirection; can go higher yet...)
- What's hurting here?
  - First, when  $\text{stride}$  is large, you index into an array  $\text{indir}$  of non-negative integers but keep missing
    - No locality...
  - Second, you will subsequently jump in the array  $a$  all over the place
    - Very likely you'll just keep missing
- Unfortunately, indirection in computing really common

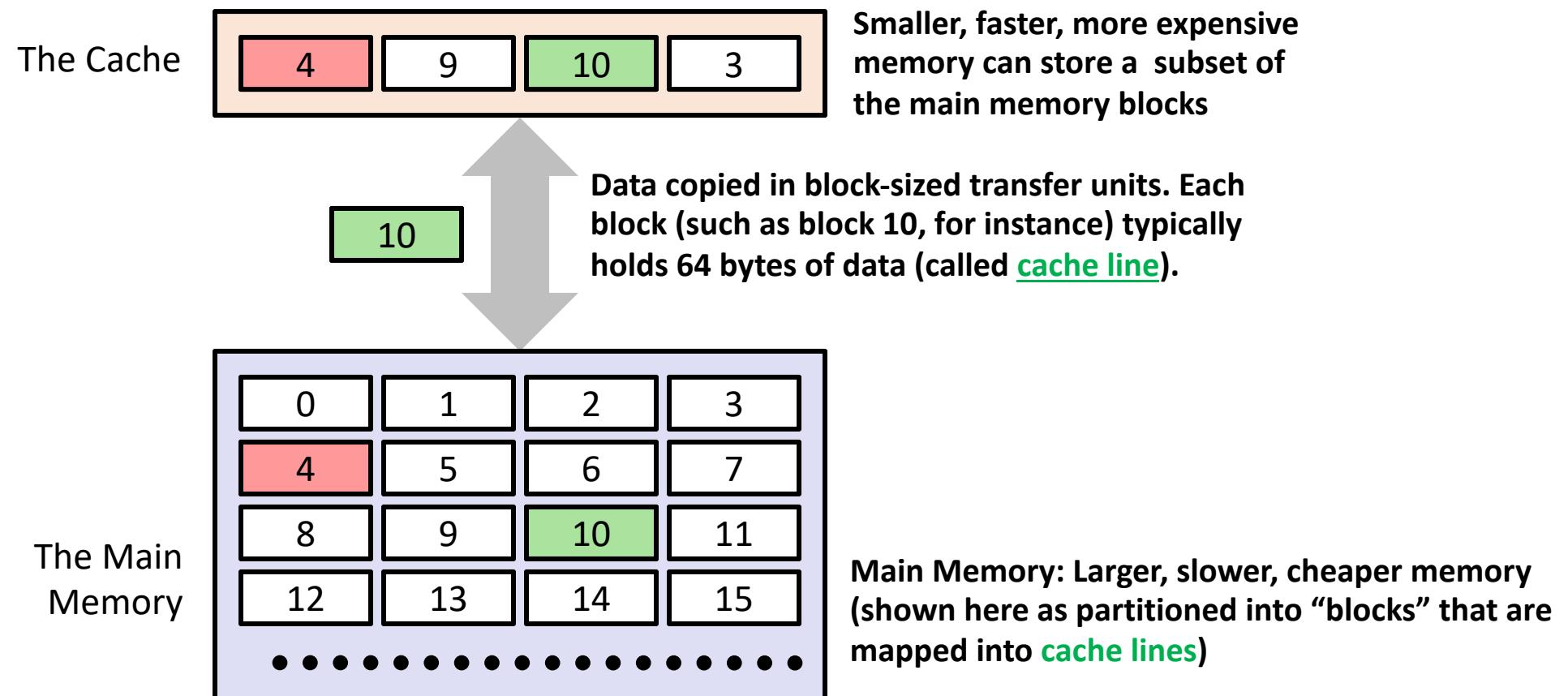
```
void loop(int* a, unsigned int* indir, int N, int stride)
{
    int sum = 0;
    for (int i = 0; i < N; i+=stride)
        sum += a[indir[i]];
}
```

# The Lesson Learned

- Fact: Wide gap between using data and moving data
  - Using data: you do `sin` of that data, `log`, `pow`, add, subtract, etc. (processing the data)
  - Moving data: getting your data from the system memory (64 GB/128/etc. GB that you have on your system)
- Fact: The cache used to close the **big speed gap**
- Trait of well-written programs: they manage to leverage data/instruction locality
  - Locality is what allows caches to come into play
  - Locality necessary, but not sufficient (more later, “false sharing”)

# General Cache Concepts

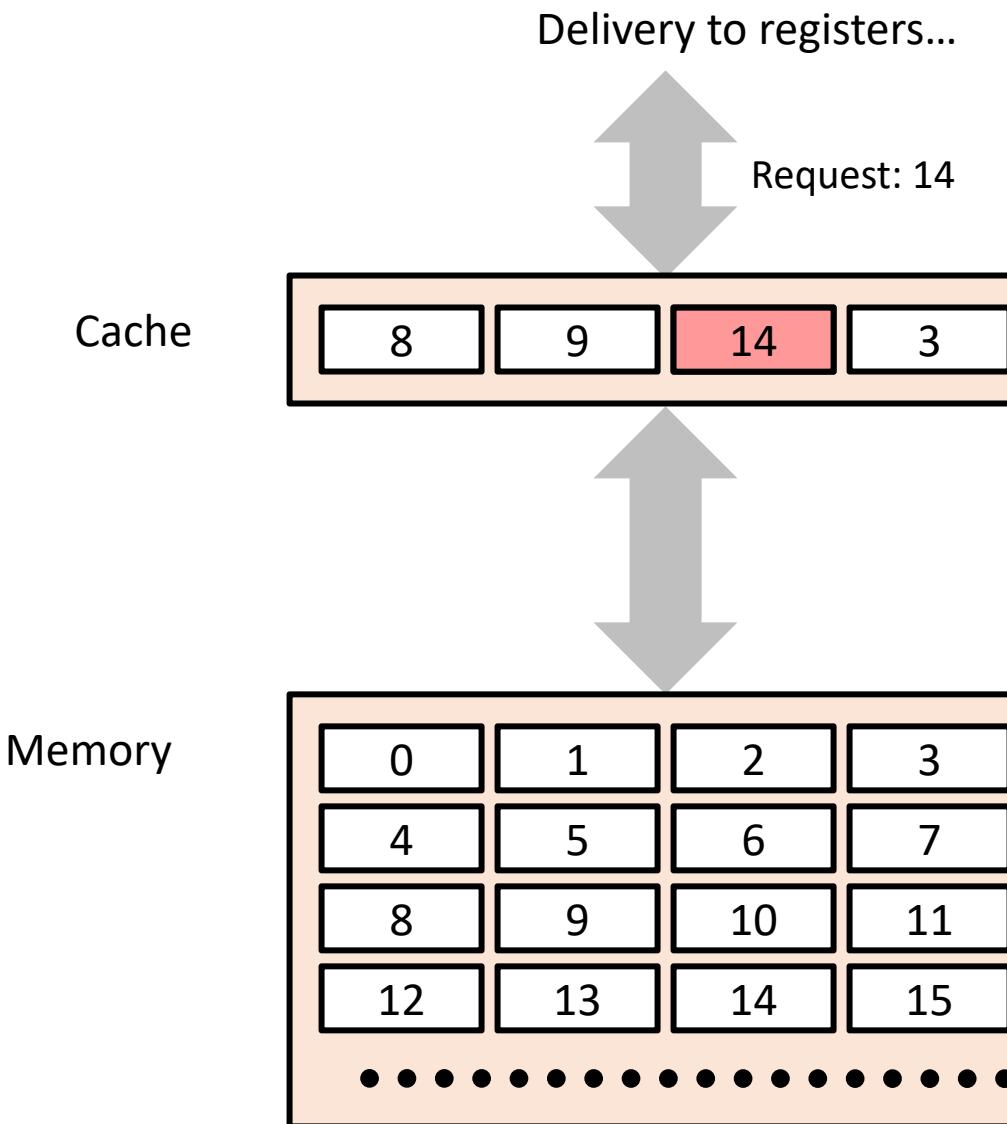
*Everything handled in hardware. Invisible to programmer*



# Cache Line (aka Cache Block)

- When data is brought in cache because a piece of data is needed, the MMU brings more than just the needed piece of data
- Why bring more data than what absolutely necessary?
  - Reason 1: In anticipation – recall principle of spatial locality
    - You'll likely need data that is right next to data that you just called for
  - Reason 2: Because penalty is not high for bringing more: wide buses, little extra cost in bringing more data than exactly what needed
    - For instance, NVIDIA hardware has 32 lanes for bringing data; it's not like if you bring a bit more you have to pay significantly more
- “cache line” - the size of the chunk of data brought from main memory and deposited into cache
  - Many systems have cache line of 64 Bytes (8 doubles OR 16 floats OR 16 ints OR 64 chars, etc.)
  - Cache line size is typically the same for L1, L2 and L3 caches

# General Cache Concepts: Hit

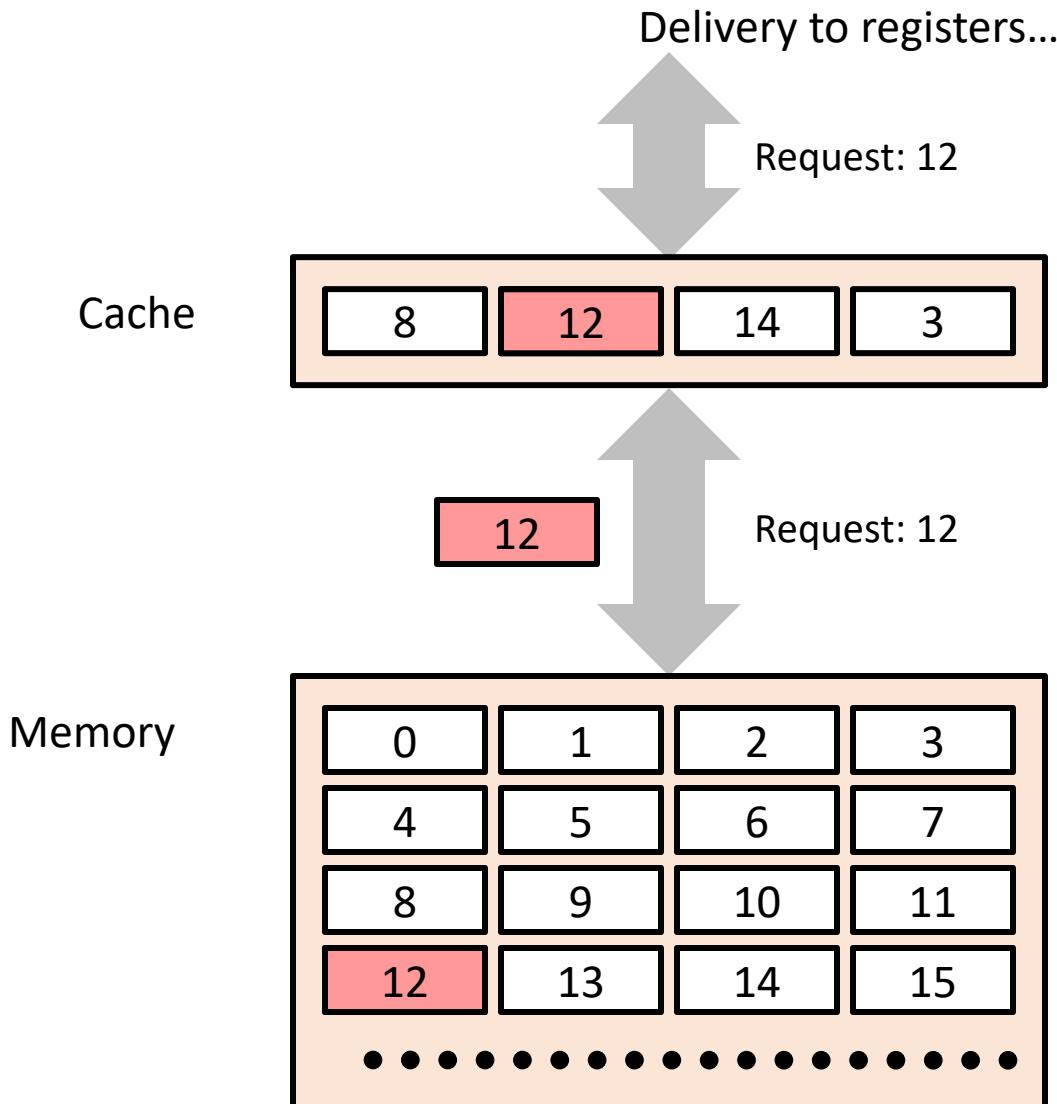


***Data in block b is needed***

***Block b is in cache:  
Hit!***

**Hit Rate**: on average, how many out of 100 mem requests hit the cache?

# General Cache Concepts: Miss



***Data in block b is needed***

***Block b is not in cache:  
Miss!***

***Block b is fetched from  
memory***

***Block b is stored in cache***

- **Placement policy:**  
determines where Block b goes
- **Replacement policy:**  
determines which block gets evicted (victim)

# Cache Miss Classification, by Request Type [1/2]

- Another way to classify cache misses is by the type of request leading to miss
- Cache misses, by request type:
  - Cache **read** miss from an **instruction** cache
  - A cache **read** miss from a **data** cache
  - A cache **write** miss to a **data** cache

# Cache Miss Classification, by Request Type [2/2]

- Cache **read** miss from an **instruction** cache:
  - Generally causes the most delay, because the processor, or at least the thread of execution, has to wait (stall) until the instruction is fetched from main memory
- A cache **read** miss from a **data** cache:
  - Usually causes somewhat less delay
  - OOOE: Instructions not dependent on the cache read (if any) can be issued and executed until the data is fetched from main memory and the stalled instructions can resume execution
  - TLP: Another thread can be picked up for execution
- A cache **write** miss to a **data** cache:
  - Generally causes the least delay – the write can be queued and there are few limitations on the execution of subsequent instructions
  - The processor can continue unless the queue is full and then it has to stall for the write buffer to partially drain

# Reasons for Cache Misses

- **Cold (compulsory) miss**
  - When happens: cache is empty (e.g., beginning of the program, time slicing while having multiple programs in flight)
- **Capacity miss**
  - When happens: Not enough space, working set is too large
    - Example: the active part of main memory overwhelms in size what can be accommodated by the cache
      - You are asking for more than what level  $k$  can accommodate (fit)
- **Conflict miss**
  - When happens: Enough space, but you have bad luck (lots of “conflicts”, as dictated by **placement/replacement** policy in place)
  - Fact: In the memory hierarchy, the cache blocks at level  $k + 1$  map into a small subset of block positions at level  $k$ 
    - **Example:** block  $i$  at level  $k + 1$  must be placed in block  $(i \% 4)$  at level  $k$
  - Conflict misses occur when the level  $k$  cache is large enough, but multiple data objects all map to the same block in the level  $k$  cache.
    - **Example:** Referencing in this order blocks 0, 8, 0, 8, 0, 8, ... would miss every time (since  $0 \% 4 = 8 \% 4$ )

# Placement & Replacement Policies

- The cache is small, while the memory that you try to cache in this cache is large
  - The cache: prime real estate – when somebody comes in, somebody else has to be evicted
- Caching: like bringing folks from a city to live in one apartment building next to some prime attraction
  - “prime attraction”: the registers and the ALU/CU
- **placement policy**: deciding in which “apartment in the building” to drop the cache line
- Sometimes, there are several rooms in an apartment: which one do you assign to the new folks?
  - “**replacement policy**” – if you have *Option A* and *Option B*, which one should you choose: A or B?
    - You’ll kick out tenant from room A, or tenant from room B

# Placement Policies

- Policies that are common:
  - “When you bring a line of cache...”
    - You are free to place wherever (**fully associative** cache)
    - Based on the main memory address of the data, you can put it in one of 4 places (four rooms in an apartment)
      - Or 2 places, 8 places, or k places (for **k-way associative** cache)
    - You can only put it in one place (**direct mapped** cache)

# Placement Policies: further discussion

- Fully associative
  - Pros: you have full control which old cache line you want to kick out (you have a very **flexible replacement policy**)
  - Cons:
    - Higher overhead to find what you're looking for (data can be anywhere in cache)
    - Higher overhead to decide whom to evict (the “victim”)
- Direct mapped
  - Pros: very fast to find if in cache or not
  - Cons: might have to evict a line cache that actually sees a lot of use (no freedom at all in replacement policy)
- K-way associative
  - Decent compromise; i.e., it yields a reasonably decent replacement policy – you have K options to choose from
  - Intel and AMD chips: 2, 4, 8, 16-way associative
    - Depends on cache level and which chip model

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 06

02/03/2020

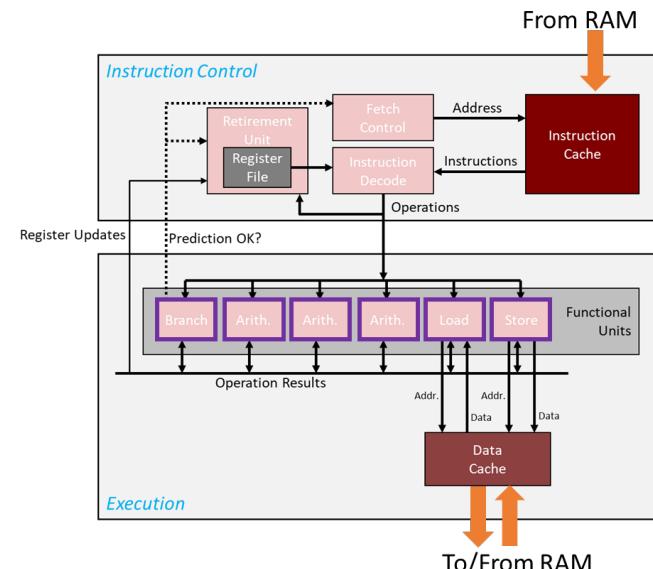
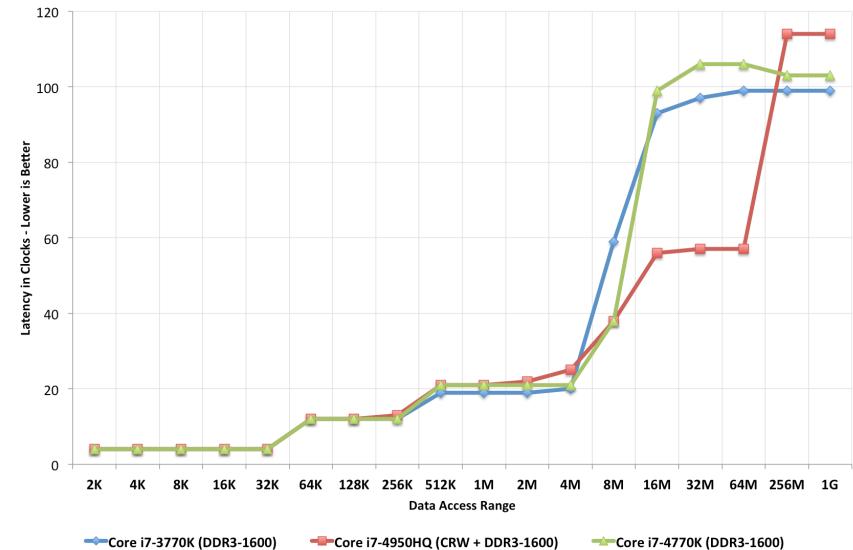
# Quote of the day

“I have three kids and no money... Why can't I have no kids and three money?”

-- Homer Simpson, Safety Inspector [1989 - ]

# Before we get going...

- Last time: Caches
  - Simple example: 30X speedup
  - Typically 3 levels of cache
    - 4 cycles (L1), 12 cycles (L2), 20 cycles (L3)
    - Data from main memory: 100-120 cycles
  - The anatomy of the cache
- Today
  - The cache: wrap up
  - The Virtual Memory
- Other tidbits:
  - Please take care of the assigned reading
  - Assignment due on Th at 9 pm
  - Use Piazza to get things going (for you or others)



# Caches, and the hotel analogy

- Hotel right outside Bigattraction Park
  - Hotel has 64 floors, each floor has 8 rooms. Each room can accommodate 64 folks
  - Bus comes from the city bringing 64 new folks
    - Somebody needs to leave; we get a room for the new folks
- Cache analogy:
  - The city is the large main memory
  - The hotel is the cache
  - The floor is the set
  - The room is the element
  - The 64 folks in the room the analog of the cache line (64 bytes)
- Cache size, in the example above:  $64 \times 8 \times 64 = 32768$  bytes
  - Decent L1 data cache

# Placement & Replacement Policies

- The cache is small, while the memory that you try to cache in this cache is large
  - The cache: prime real estate – when somebody comes in, somebody else has to be evicted
- Caching: like bringing folks from a city to live in one apartment building next to some prime attraction
  - “prime attraction”: the registers and the ALU/CU
- **placement policy**: deciding in which “apartment in the building” to drop the cache line
- Sometimes, there are several rooms in an apartment: which one do you assign to the new folks?
  - “**replacement policy**” – if you have *Option A* and *Option B*, which one should you choose: A or B?
    - You’ll kick out tenant from room A, or tenant from room B

# Placement Policies

- Policies that are common:
  - “When you bring a line of cache...”
    - You are free to place wherever (**fully associative** cache)
    - Based on the main memory address of the data, you can put it in one of 4 places (four rooms in an apartment)
      - Or 2 places, 8 places, or k places (for **k-way associative** cache)
    - You can only put it in one place (**direct mapped** cache)

# Placement Policies: further discussion

- Fully associative
  - Pros: you have full control which old cache line you want to kick out (you have a very **flexible replacement policy**)
  - Cons:
    - Higher overhead to find what you're looking for (data can be anywhere in cache)
    - Higher overhead to decide whom to evict (the “victim”)
- Direct mapped
  - Pros: very fast to find if in cache or not
  - Cons: might have to evict a line cache that actually sees a lot of use (no freedom at all in replacement policy)
- K-way associative
  - Decent compromise; i.e., it yields a reasonably decent replacement policy – you have K options to choose from
  - Intel and AMD chips: 2, 4, 8, 16-way associative
    - Depends on cache level and which chip model

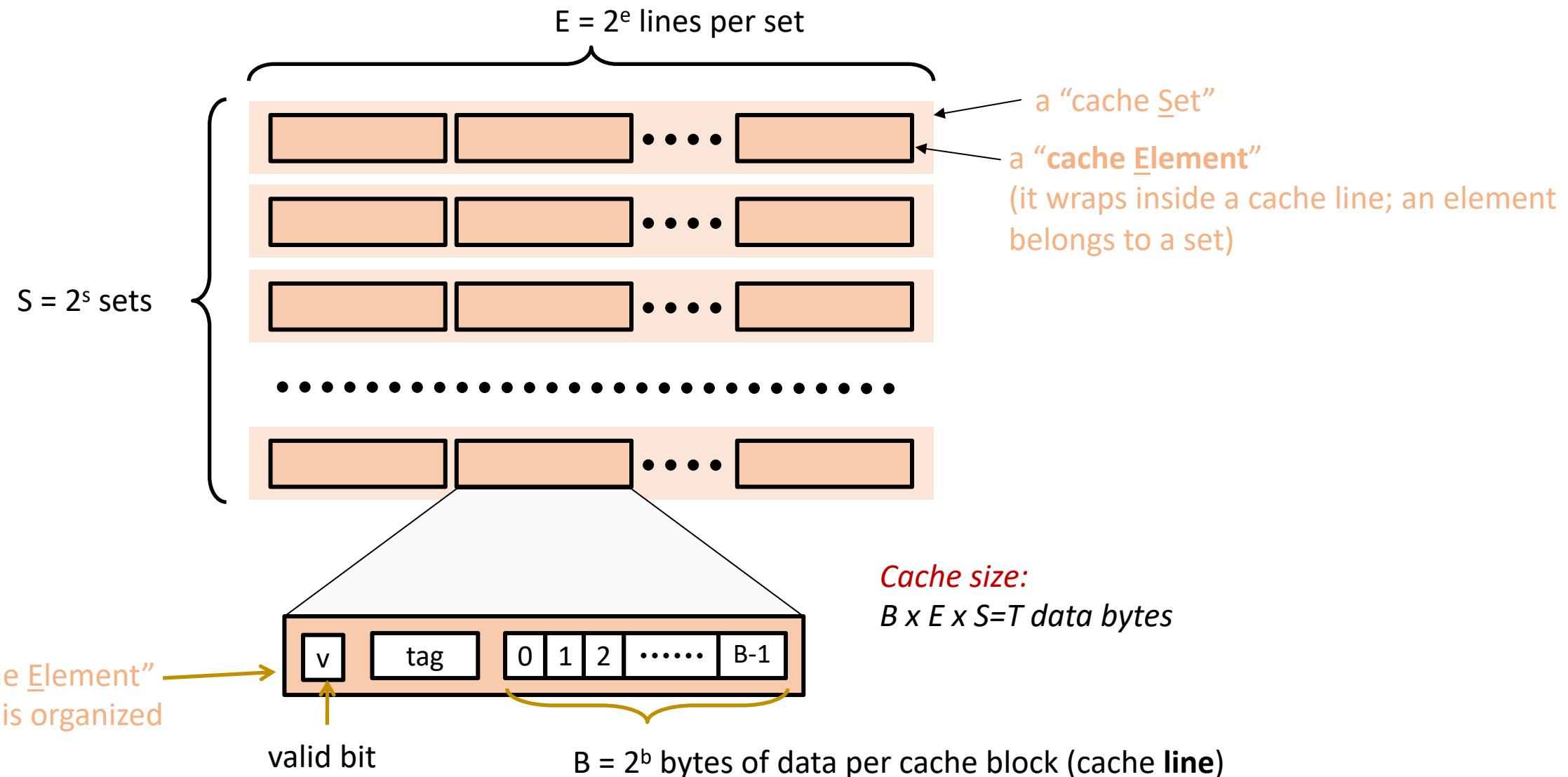
# [New Topic:] General Cache Organization

[Symbols Used and Their Meaning]

- **B**: Number of bytes in a cache line
  - Typically, B is 64
- **E**: The number of cache lines (“Elements”) that combine to make up a set
  - Typically, E is 1, 2, 4, 8, or 16 (other values possible)
- **S**: the number of “Sets” that make up the cache
- **T**: total cache size

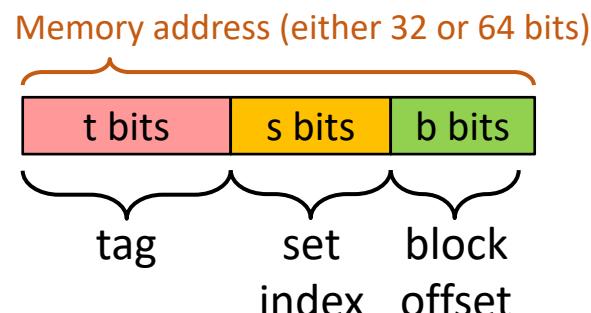
$$B \times E \times S = T$$

# General Cache Organization (B, E, S)



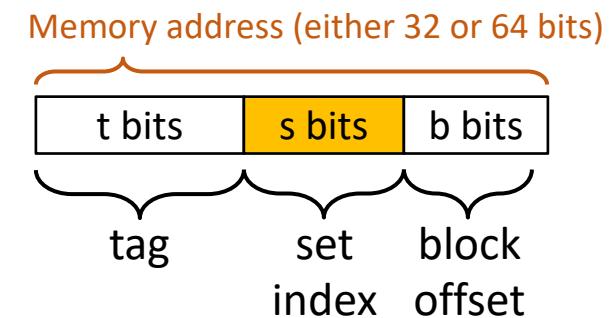
# On finding if data from a main memory address is in cache

- The mission: Read into register a variable at a given memory address
- The important question: Is the value stored at this address by any chance in cache?
- To answer this question, the **addressed is partitioned** in three chunks

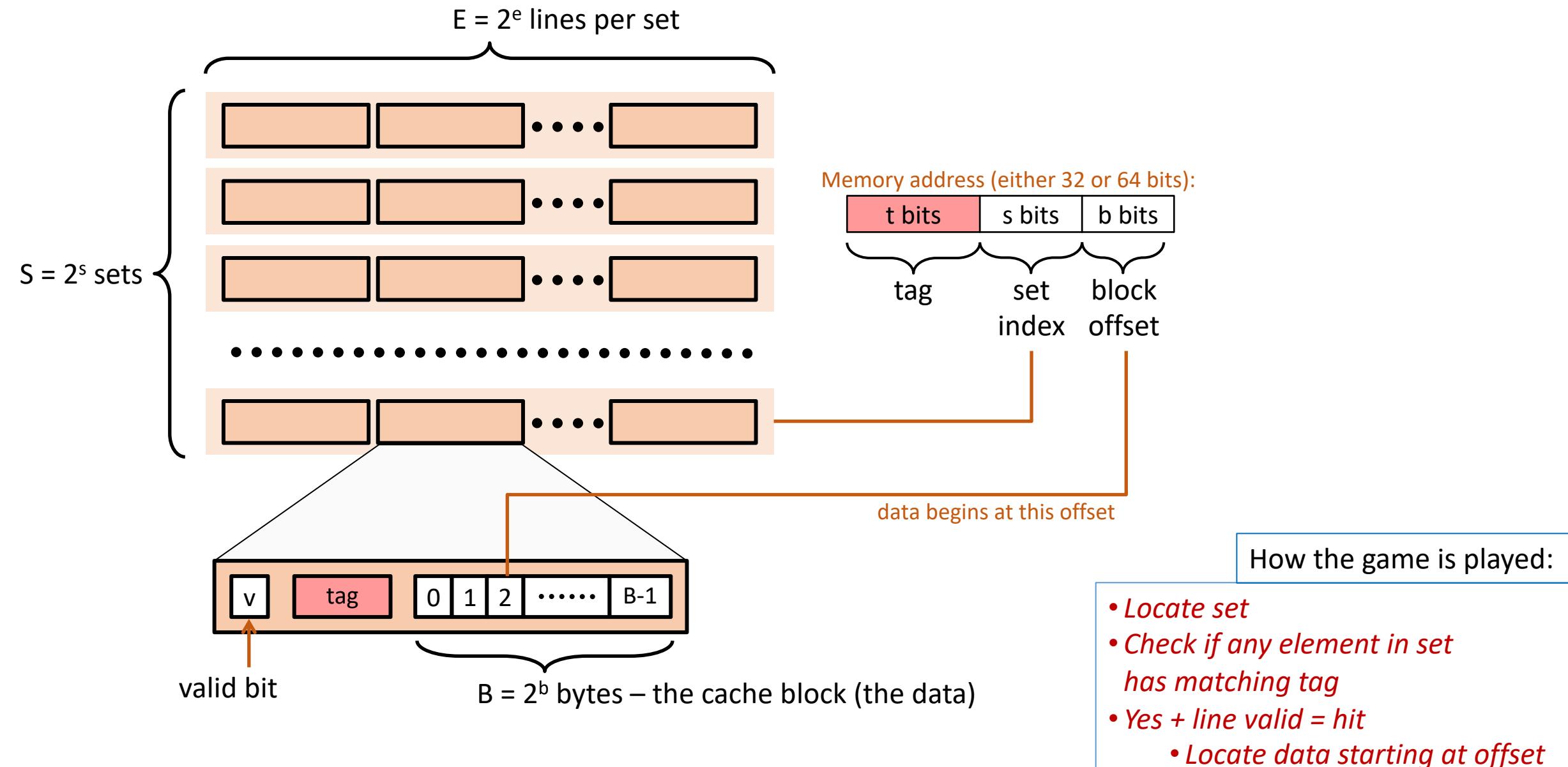


# The key ingredient is the set index

- The set index give me “the floor in the hotel where I should look for Joe who lives at 12 Main St.”
  - The  $s$  bits of the address
  - Hotel must have  $2^s$  floors
- Important remark: there are many addresses in main memory that will have the same set index
  - Translation: folks from many neighborhoods of the town assigned to the same hotel floor



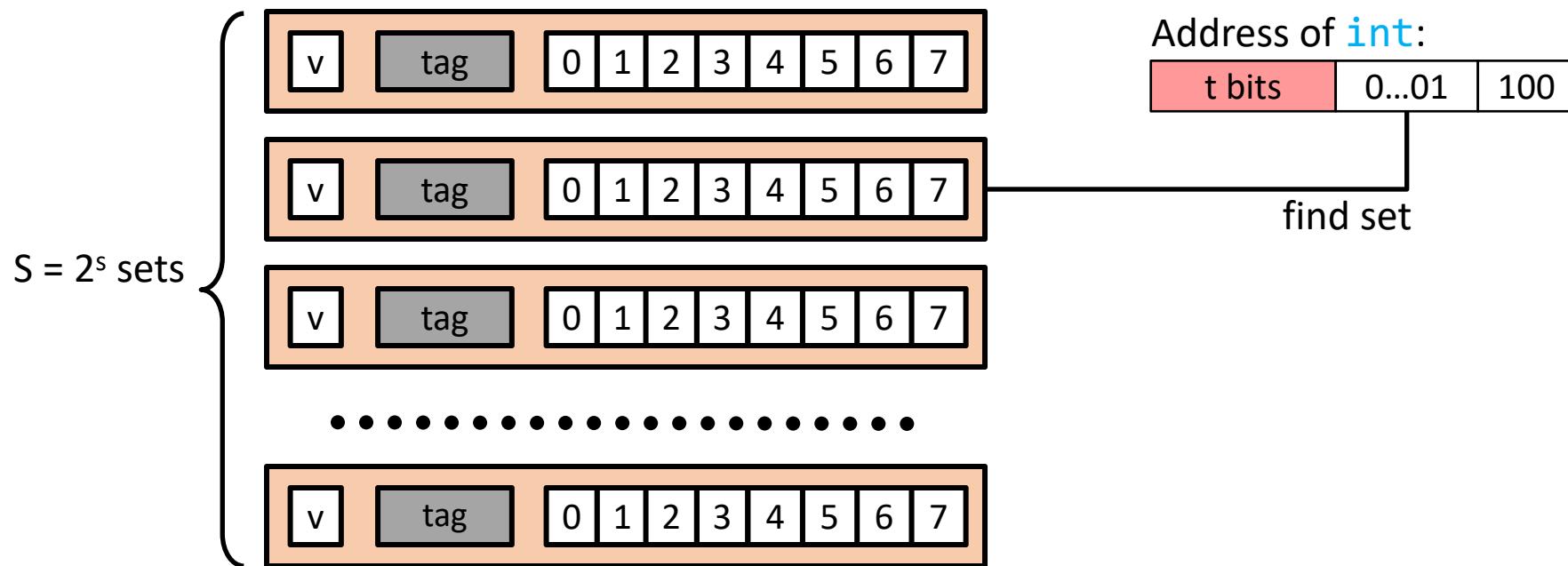
# How Cache Read Takes Place



# Example: Direct Mapped Cache ( $E = 1$ )

Direct mapped: One line per set

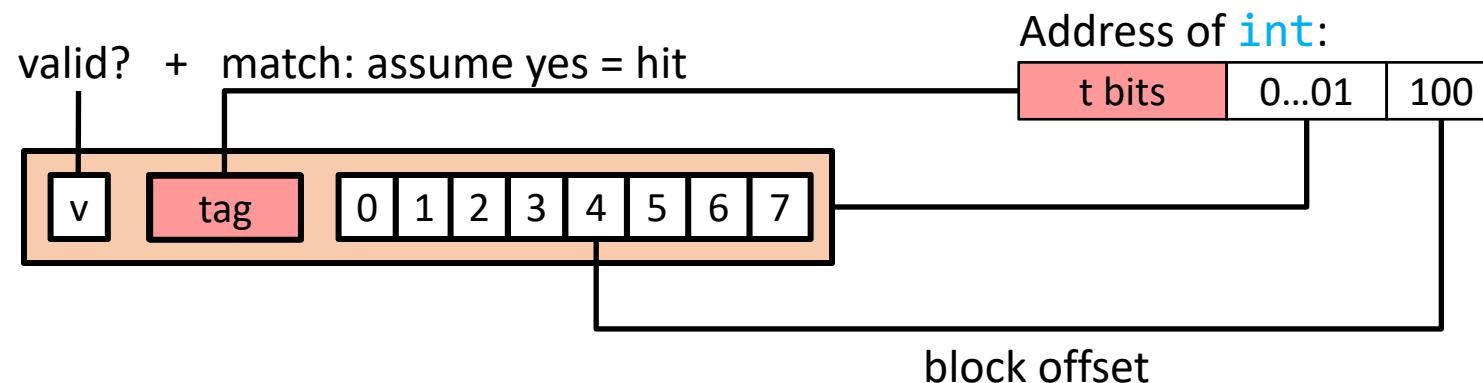
Assume: cache block size 8 bytes (instead of 64)



# Example: Direct Mapped Cache ( $E = 1$ )

Direct mapped: One line per set

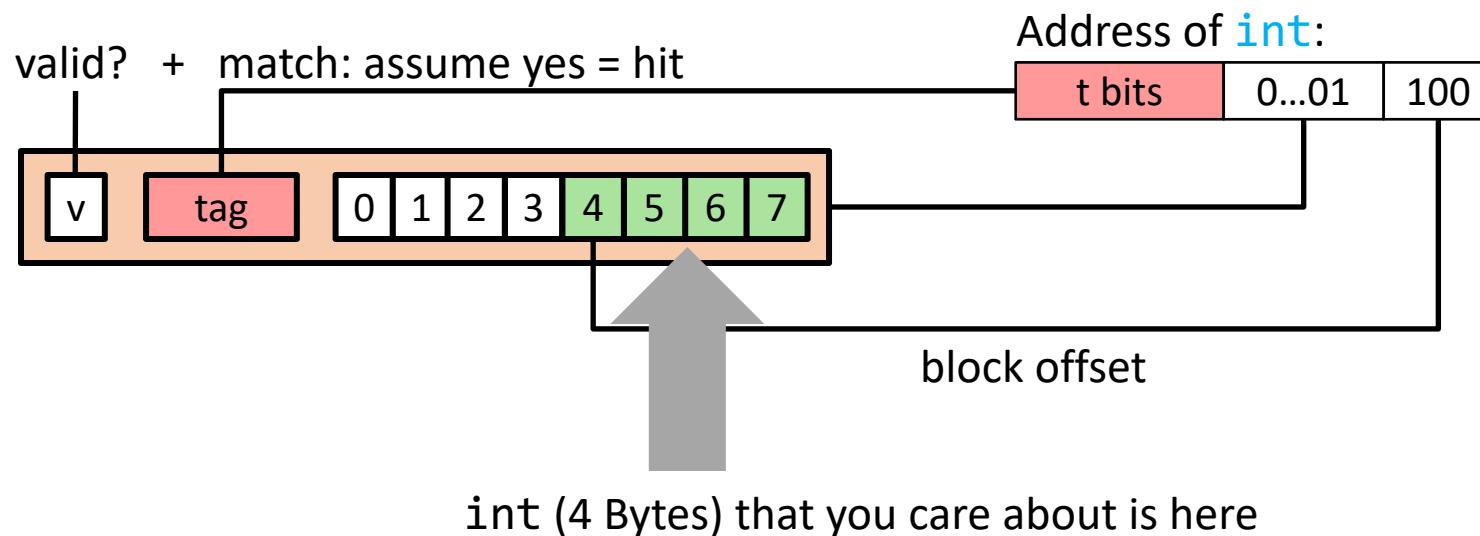
Assume: cache block size 8 bytes (instead of 64)



# Example: Direct Mapped Cache ( $E = 1$ )

Direct mapped: One line per set

Assume: cache block size 8 bytes (instead of 64)

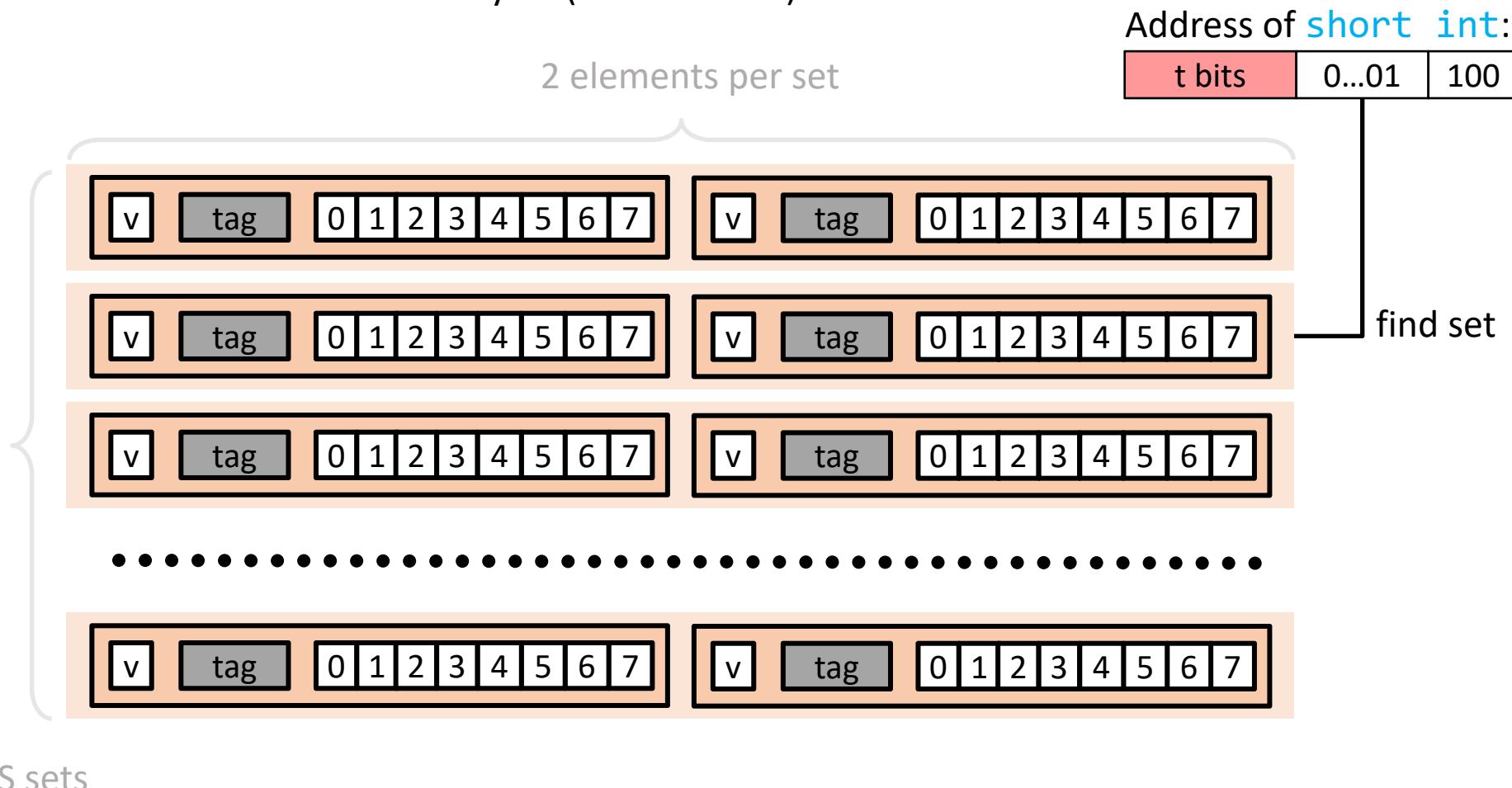


If tag doesn't match: old line is evicted and replaced

# E-way Set Associative Cache (Here: $E = 2$ )

$E = 2$ : Two elements/lines per set

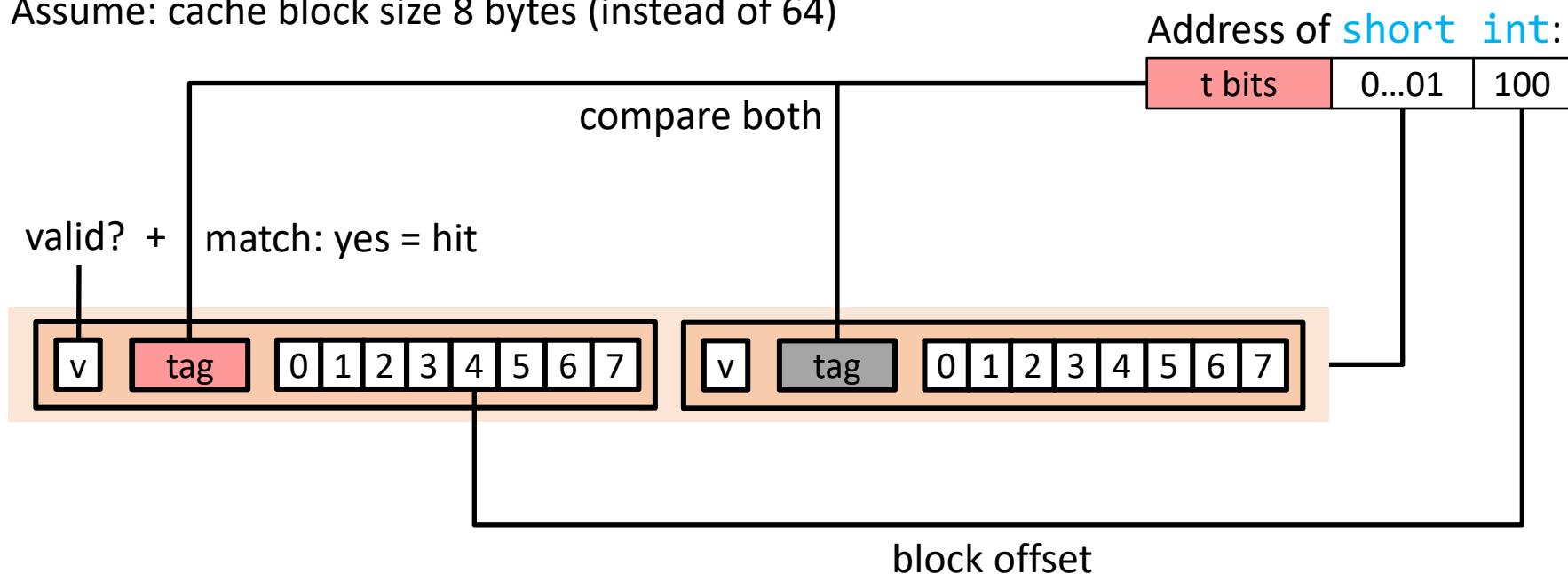
Assume: cache block size 8 bytes (instead of 64)



# E-way Set Associative Cache (Here: E = 2)

E = 2: Two elements/lines per set

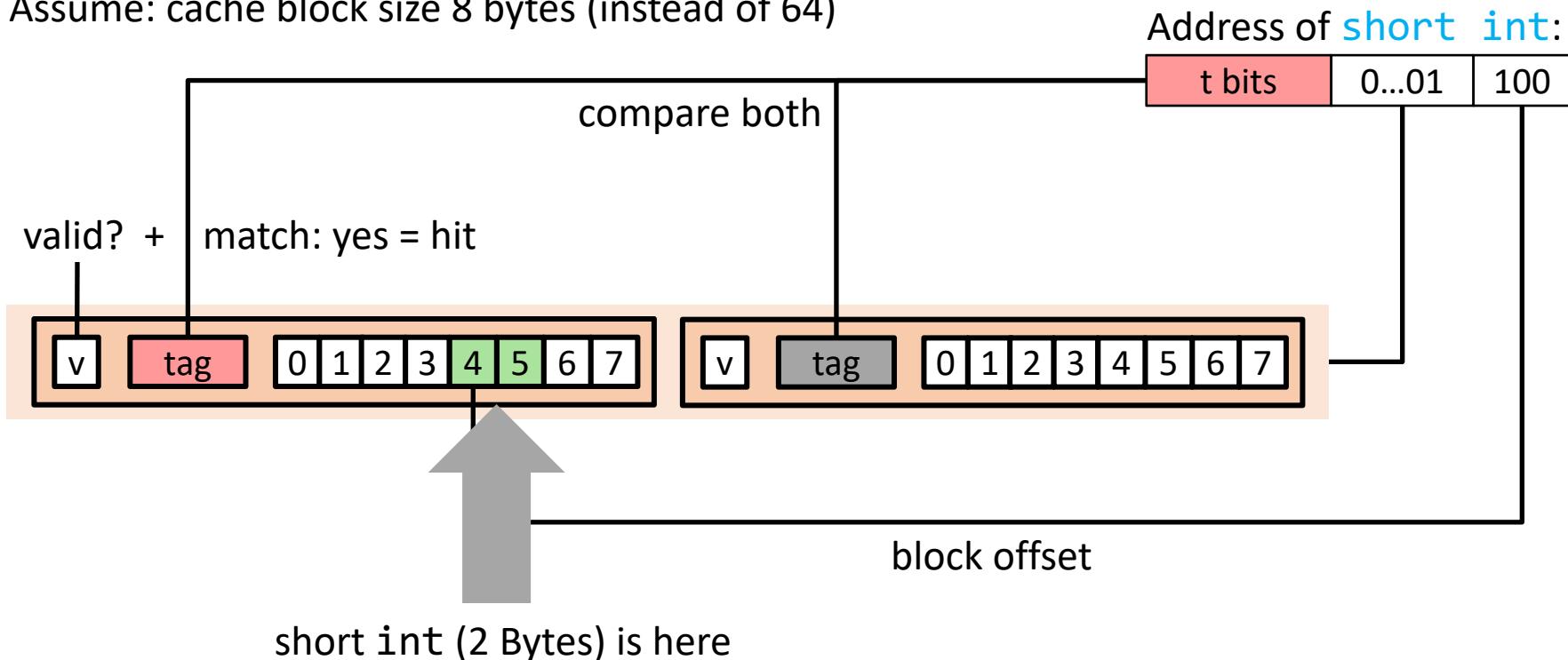
Assume: cache block size 8 bytes (instead of 64)



# E-way Set Associative Cache (Here: E = 2)

E = 2: Two elements/lines per set

Assume: cache block size 8 bytes (instead of 64)

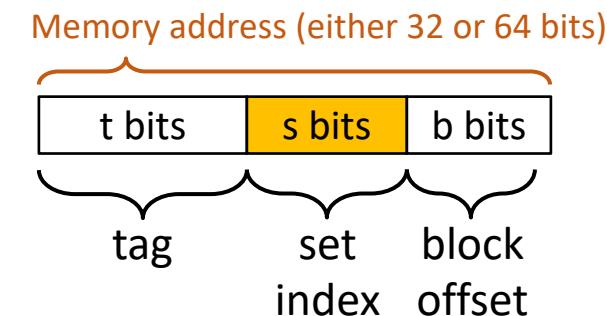


## No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

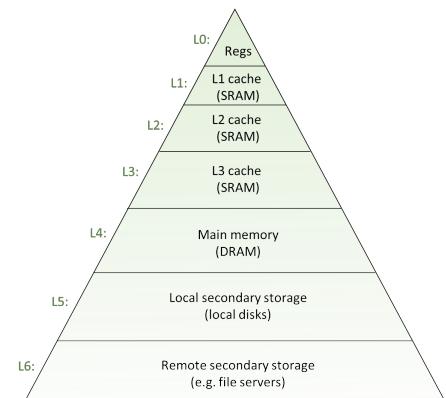
# Quiz: How many neighborhoods sent folks to same hotel floor?

- There are many addresses in main memory that will have the same set index
  - Translation: folks from many neighborhoods of the town assigned to the same hotel floor
- Assume 32 bit OS, and the set index is 6 bits wide (64 sets)
- How many “neighborhoods” of the main memory send folks to the same floor of the hotel?
- How is associativity helping?



# Why these acrobatics?

- Trying to fit a lot of memory (system memory, GBs of it) into little memory (cache, MBs of it)
- Common sense: you have to use tricks to accommodate stuff from a big pot into a small pot
- This dilemma of storing a lot of information in a small space is pervasive
  - Because of the memory hierarchy reality we are operating under
  - Similar tricks for placing data in main memory relative to placing in secondary memory



# Going on a tangent: TLP vs ILP

- Chip can do more than one instruction per cycle
  - ILP flavor: if chip is superscalar, more than one instruction can be wrapped up/clock cycle
  - TLP flavor: two threads/processes run at same time, say A and B
- Cache implications: TLP at a disadvantage
  - On each miss by B, the cache victimizes an approximation of the least-recently-used (LRU) cache block
    - Most of B's misses will victimize blocks last accessed by A rather than last accessed by itself
  - Consequences:
    - The L1 cache turns over sufficiently fast that you can approximately assume that B victimizes all of A's blocks
    - Whether L2 cache is completely flushed depends on B's rate of misses, the L2 size, and how long B runs

# The Caching Landscape in the Memory Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler/CU
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

# Example [using CAE's tux-103]

```
>> less /proc/cpuinfo
```

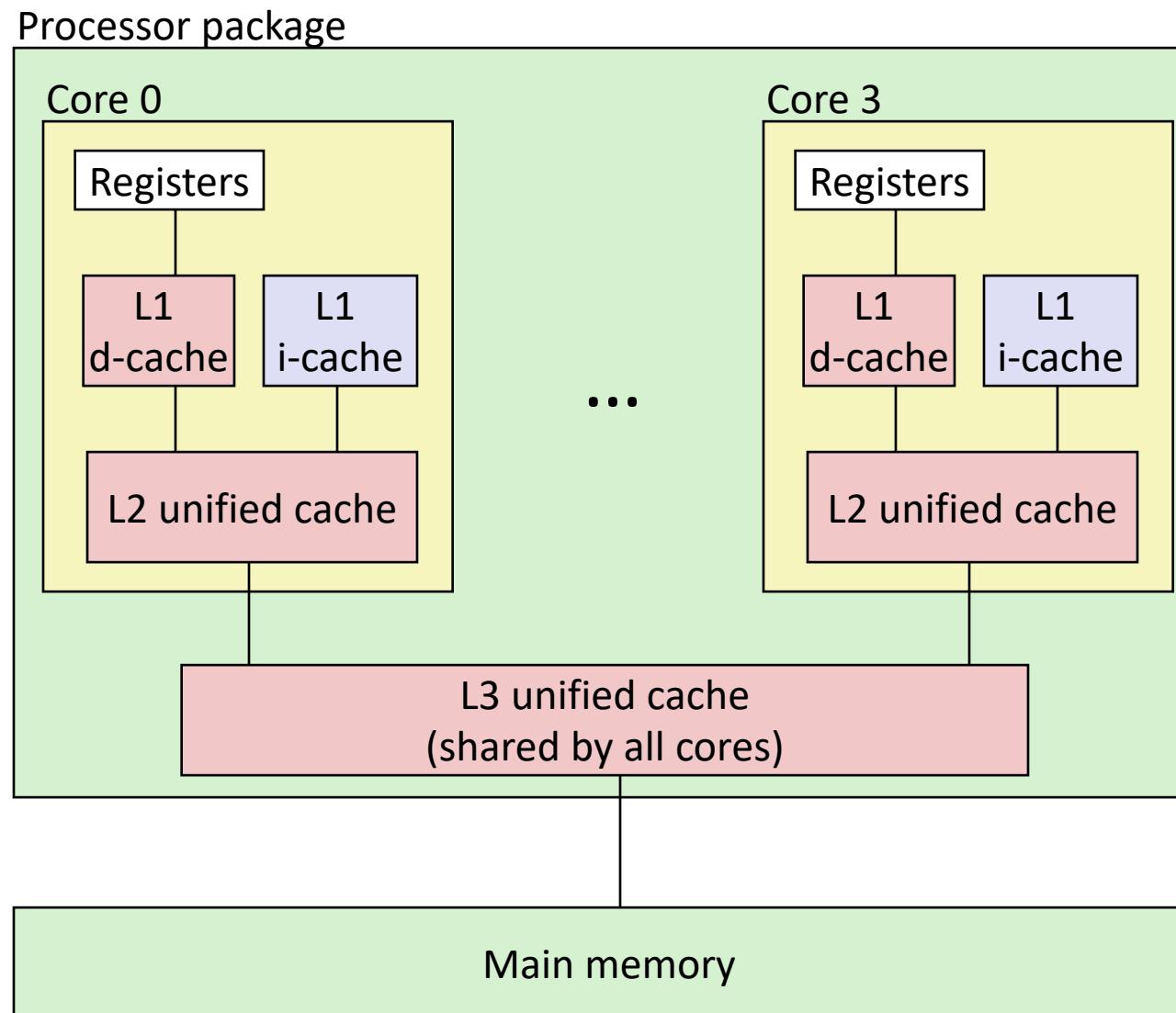
```
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 60
model name    : Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz
stepping       : 3
microcode     : 0x20
cpu MHz       : 3197.500
cache size    : 6144 KB
physical id   : 0
siblings       : 4
core id        : 0
cpu cores     : 4
apicid         : 0
fpu            : yes
fpu_exception  : yes
cache_alignment : 64
address sizes  : 39 bits physical, 48 bits virtual
```

Cache line size: 64 bytes →  $2^6$  Bytes  
8-way associative →  $2^3$   
Amount of memory: 256KB →  $2^{18}$  Bytes  
Number of sets:  $2^{18}/(2^6 \cdot 2^3) = 2^9 = 512$

```
>> grep . /sys/devices/system/cpu/cpu0/cache/index*/*
```

```
/sys/devices/system/cpu/cpu0/cache/index2/coherency_line_size:64
/sys/devices/system/cpu/cpu0/cache/index2/level:2
/sys/devices/system/cpu/cpu0/cache/index2/number_of_sets:512
/sys/devices/system/cpu/cpu0/cache/index2/physical_line_partition:1
/sys/devices/system/cpu/cpu0/cache/index2/size:256K
/sys/devices/system/cpu/cpu0/cache/index2/type: Unified
/sys/devices/system/cpu/cpu0/cache/index2/ways_of_associativity:8
```

# Revisiting an old slide [hopefully it makes better sense now]



L1 i-cache and d-cache:  
32 KB, 8-way,  
Access: 4 cycles

L2 unified cache:  
256 KB, 8-way,  
Access: 10 cycles

L3 unified cache:  
8 MB, 16-way,  
Access: 20-30 cycles

Block size: 64 bytes for all caches.

# We've done memory reads. What about writes?

- Multiple copies of data exist stored in different places and at the same time:
  - L1, L2, L3, Main Memory, Secondary Memory (Disk)
- How should a **write-hit** be handled?
  - **Write-through** (write immediately to main memory as well)
  - **Write-back** (defer write to main memory until replacement of line)
    - Need a dirty bit (indicate whether line different from memory or not)
- How should a **write-miss** be handled?
  - **Write-allocate** (load into cache, update line in cache)
    - Good if more writes to the location follow; i.e., if locality is good
  - **No-write-allocate** (writes straight to memory, does not load into cache)
- Typical combos met in practice:
  - **Write-back + Write-allocate ← more common**
  - Write-through + No-write-allocate

# Cache performance metrics [a nomenclature matter]

- **Miss Rate**

- Defined as  $\text{Miss Rate} = 1 - \text{Hit Rate}$  (note that we defined the Hit Rate in the previous lecture)
- Represents the fraction of memory references that are not found in cache
- Typical numbers (in percentages):
  - As low as 3-10% for L1
  - Can be quite small (e.g., < 1%) for L2, depending on size, nature of problem you run, etc.

- **Hit Time**

- Time to deliver a line in the cache into a register of the processor
  - Includes time to determine whether the line is in the cache
- Typical numbers:
  - 4 clock cycle for L1
  - 10 clock cycles for L2

- **Miss Penalty**

- Additional time required because of a miss
  - Typically 50-200 cycles for main memory (Trend: slightly increasing)

# People Talk Miss Rate, Not Hit Rate

- People talk about miss rate, and not hit rate
- Weird math: Although 97 and 99 close, time-wise, 99% hit rate twice as costly as 97% hit rate
  - Assumptions, for back of the envelope calculation:
    - a) cache hit time of 1 cycle
    - b) miss penalty of 100 cycles
  - 97% vs 99% hit rate: Access cost for 100 mem operations
$$\begin{aligned} \text{97\% hit rate: } & 1 + 1 + \dots + 1 + 100 + 100 + 100 = 397 \text{ cycles} \\ \text{99\% hit rate: } & 1 + 1 + \dots + 1 + 1 + 1 + 100 = 199 \text{ cycles} \end{aligned}$$
$$\frac{397}{199} \approx 2$$
- You'd think that 97% is as good as 99%. It's not...
  - This is why “miss rate” is used instead of “hit rate”: 1% miss rate vs. 3% miss rate

# Writing Cache Friendly Code: Rules of Thumb (more to come)

- Basic facts:
  - You have no *\*direct\** control over what data gets cached
  - However, you are not helpless: ensure your code is “local” (spatially and/or temporally)

when you write code, do not violate locality, if at all possible

# Other performance rule of thumb

- Make the common case go fast
  - Focus on the inner loops of the “heavy” functions
- Try to have cache-friendly code in the inner loops
  - Repeated references to same variables are good (**temporal locality**)
  - “stride of one” reference patterns are good (**spatial locality**)

# Approaching the task of programming

- By and large, people write code by paying attention to
  - Correctness
  - Esthetics of the solution developed
    - Objects, lists, patterns, inheritance, etc. – code is flexible and easy to maintain
  - Portability
- Data Analytics (Big Data): you should think about **performance** when writing/designing software
  - Cache friendly code via “**locality**” is king
  - When you write a line of code, always pause and ask yourself: where is this data that I’m referencing here comes from?
- Departing thoughts:
  - Does “performance” come before “esthetics”? Or “portability”?
  - Good read: “The Mythical Man-Month,” by Frederick Phillips Brooks Jr.

# The “Memory Mountain” Experiment

- Task at hand in this exercise: adding up numbers in an array (just like example of two lectures ago)
  - The numbers must be brought over into the chip to be added
  - Timing how long it takes to finish execution: a good proxy for speed of moving data
    - Why?
    - Because the math is trivial, eats up no time at all:  $\text{sum} = \text{sum} + a[i]$
- Moral of the story: the good cases are the ones with very high “**effective bandwidth**”
- “**effective bandwidth**”: computed by dividing amount of data moved during solving problem to the amount of time required to finish the execution

# Memory Mountain Test Function

```
void test(double* data, int elems, int stride) /* The test function */  
{  
    int i;  
    double result = 0.0;  
    volatile double sink;  
  
    for (i = 0; i < elems; i += stride) {  
        result += data[i];  
    }  
    sink = result; /* So compiler doesn't optimize away the loop */  
}
```

2010 Vintage

```
void test(long *data, long elems, long stride)  
{  
    volatile long accum = 0;  
    long i, sx2 = stride * 2, sx3 = stride * 3, sx4 = stride * 4;  
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;  
    long length = elems;  
    long limit = length - sx4;  
  
    /* Combine 4 elements at a time */  
    for (i = 0; i < limit; i += sx4) {  
        acc0 = acc0 + data[i];  
        acc1 = acc1 + data[i + stride];  
        acc2 = acc2 + data[i + sx2];  
        acc3 = acc3 + data[i + sx3];  
    }  
  
    /* Finish any remaining elements */  
    for (; i < length; i += stride) {  
        acc0 = acc0 + data[i];  
    }  
    accum = ((acc0 + acc1) + (acc2 + acc3));  
}
```

2016 Vintage

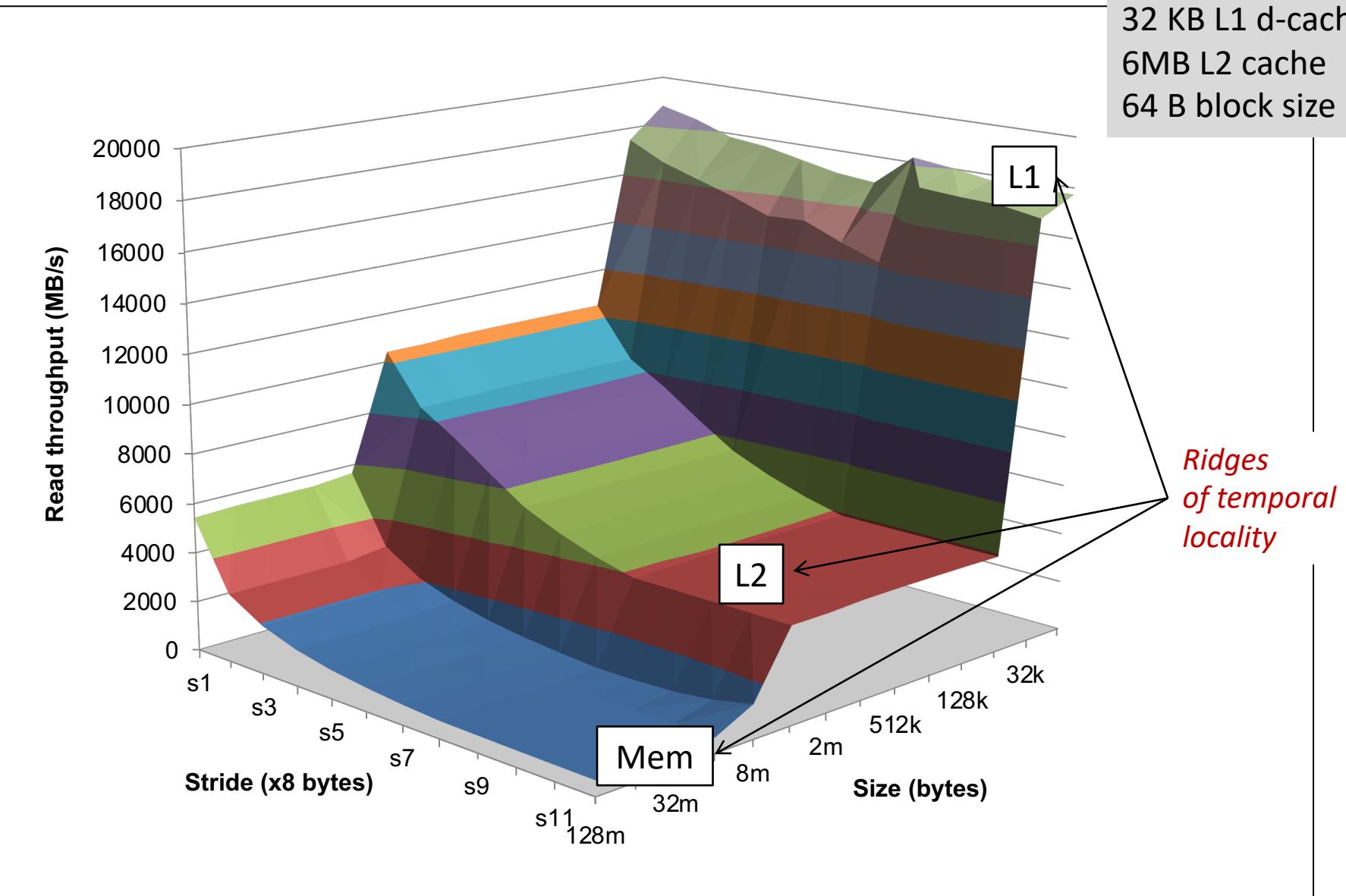
Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

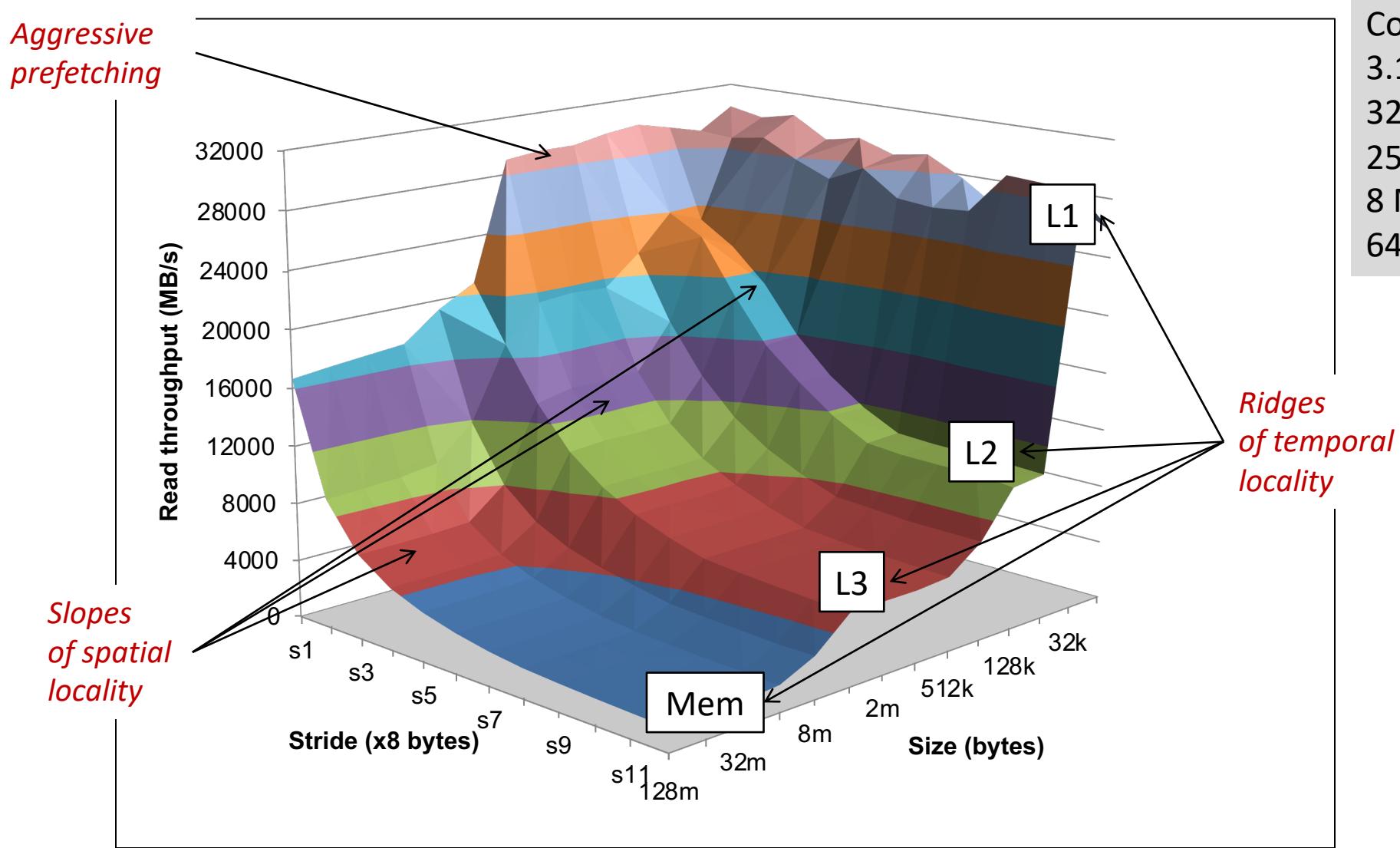
1. Call `test()` once to warm up the caches ( $\leftarrow$ **important!!!**)
2. Call `test()` again and measure the read throughput (MB/s)

NOTE: the “`volatile`” qualifier used to not optimize away the accumulator

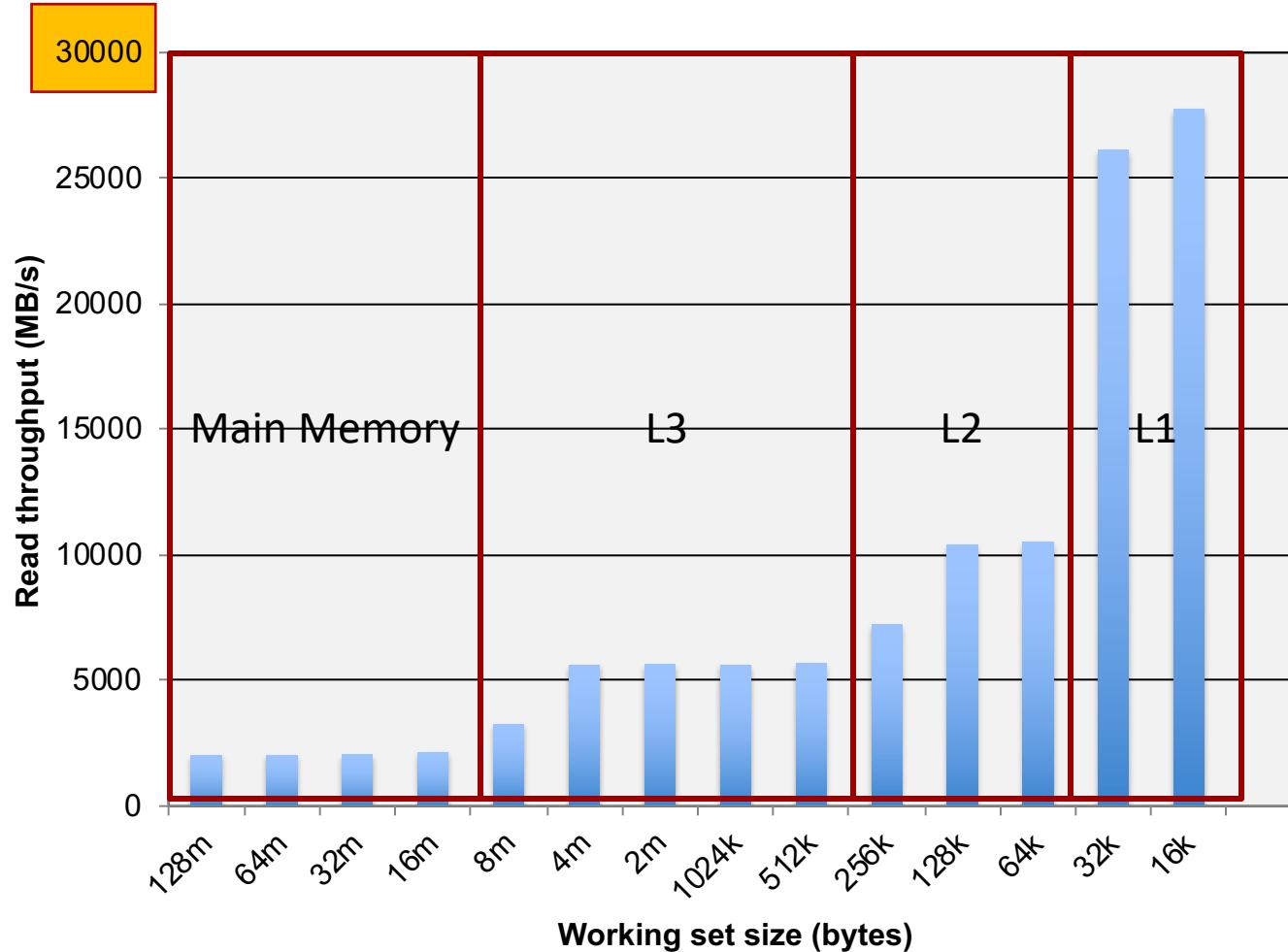
Core 2 Duo  
2.4 GHz  
32 KB L1 d-cache  
6MB L2 cache  
64 B block size



# The Memory Mountain [2014]



# Cache Capacity Effects from Memory Mountain



Core i7 Haswell  
3.1 GHz  
32 KB L1 Data Cache  
256 KB L2 Cache  
8 MB L3 Cache  
64 B block size

Slice through  
memory  
mountain with  
stride=8

# The “Memory Mountain” experiment: closing thoughts

- **Read throughput** (read bandwidth)
  - Number of bytes read from main memory per second (MB/s)
- **Memory mountain:** Measures read-throughput as a function of spatial and temporal locality
  - Compact way to characterize memory system performance

# New Topic: Virtual Memory

# Why talk about “Virtual Memory”?

- Understand how memory transactions take place, and why some incur higher overhead
- Add more credibility to some key points of this course, e.g.,
  - Number crunching is inexpensive
  - Moving information back and forth is expensive
    - Both in terms of time & power

# Why talk about “Virtual Memory”?

- “Virtual Memory” discussion:
  - It will help you understand how unlucky your memory transactions can become in some cases
    - E.g.: page faults, memory thrashing :-(
  - It’ll make you appreciate caches even more
    - They serve **multiple** purposes, e.g., help with: data storage, instruction storage, finding things in main memory (address translation)
  - It helps one understand how unified memory works for GPU computing (discussed later on, after GPU computing segment)

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 07

02/05/2020

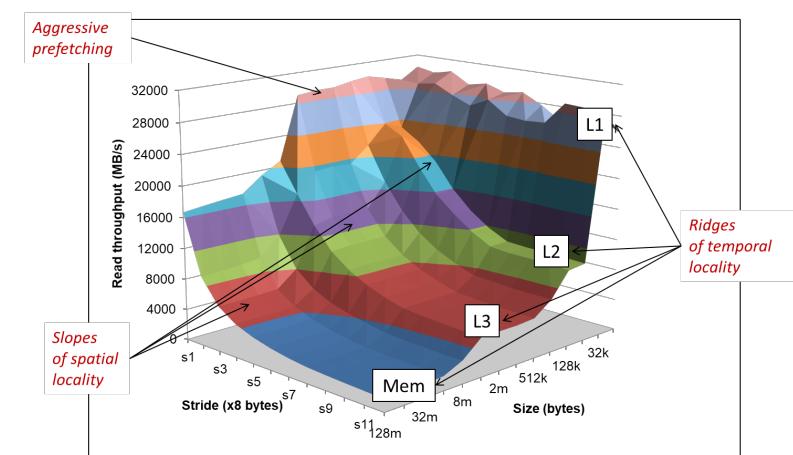
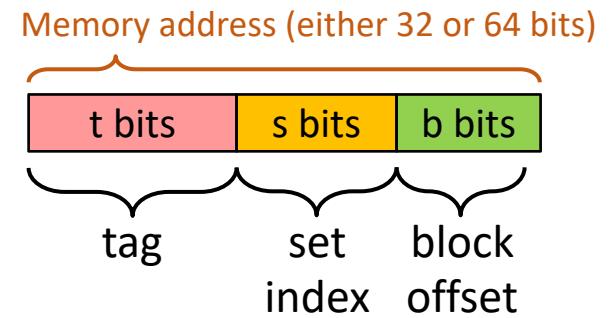
# Quote of the day

“The greatest creative challenge is the struggle to be the architect of your own life.”

-- Wade Davis, Professor of Anthropology, University of British Columbia [1953 - ]

# Before we get going...

- Last time:
  - The mechanics of a cache operation (r/w)
  - The  $B \times E \times S = T$  formula for cache size
  - The memory mountain
- Today
  - The Virtual Memory
  - The 3 walls to sequential computing
- Other tidbits:
  - Assigned reading (syllabus gets updated)
  - Assignment due on Th at 9 pm
  - Use Piazza to get things going (for you or others)



# Motivating Questions/Issues [1/3]

- Question 1: How come you might have only 512MB of main memory on your laptop yet you can C-malloc an array of 1 GB?

# Motivating Questions/Issues [2/3]

- Question 2: How can 20+ processes seemingly run at the same time? How do they each know where they can read/write data from/to?
  - Assume I have 1 GB of RAM, and programA allocates 2 GB of memory and programB allocates 2 GB of memory. How can they run at the same time?
  - Does programA need to know what addresses in memory it can use for data so that it doesn't step on the feet of programB (and perhaps programC, programD, etc.)?

# Motivating Questions/Issues [3/3]

- Question 3: How can you compile a program on a Windows workstation with 16 GB of memory and run it later on a different laptop with 1 GB of memory; i.e., less memory?
  - Are there any problems w/ addresses being referenced by the instructions?

# Preamble, Virtual Memory

- When you compile a program there is no way to know where in the **physical** memory the code will get its data allocated
  - There are other “tenants” that inhabit the physical memory & they are already there before you get there
- Solution: the code compiled under assumption that resulting process has undisputed access to 4 GB of pristine memory (on 32 bit systems)
  - The “pristine memory” is called the **virtual memory space**

# Virtual vs. Physical Memory

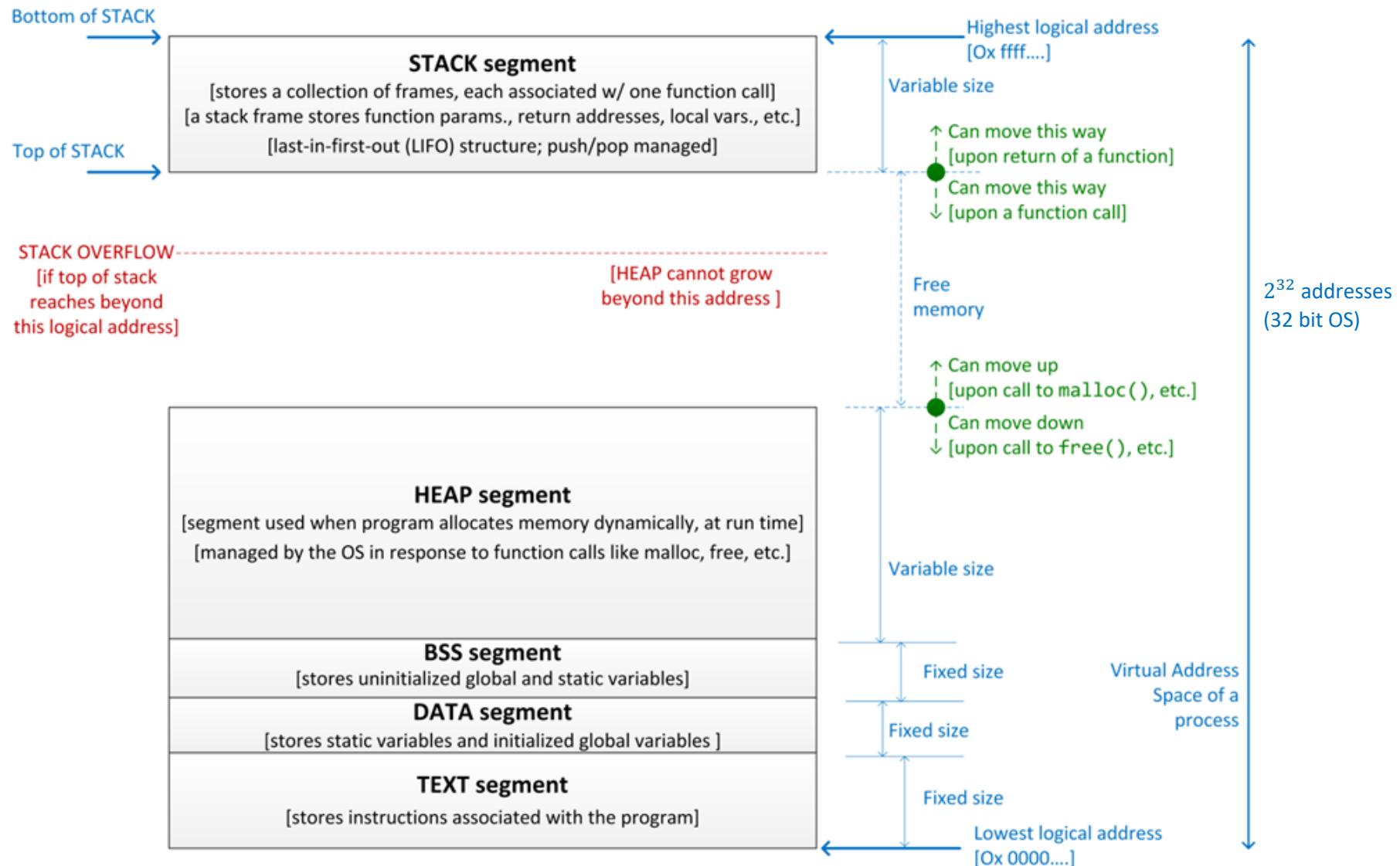
- **Virtual memory**: a dally and quaint fictitious space of  $2^{32}$  addresses (on 32 bit architectures) in which a process sees its **data** being placed, the **instructions** stored, etc.
- **Physical memory**: a busy place that hosts at the same time **data** and **instructions** associated with tens of applications running on the system
- For the virtual↔physical memory trick to work out it take the interplay between the compiler, the operating system (OS), and the execution model embraced by the processor

# Why do we like Virtual Memory?

- The Virtual Memory paradigm allows:
  - Running programs that require more memory than physically available
  - Running multiple programs “simultaneously” (multi-tasking)
  - Handling of memory fragmentation/segmentation insofar as the main memory is concerned
  - Transparent and unitary handling of memory and secondary storage (hard disk / solid state disk)
- What is the hardware component that manages the Virtual Memory show?
  - The Memory Management Unit (MMU)
    - MMU part of the chip these days (back in the day, it used to be an entity by itself, off chip)

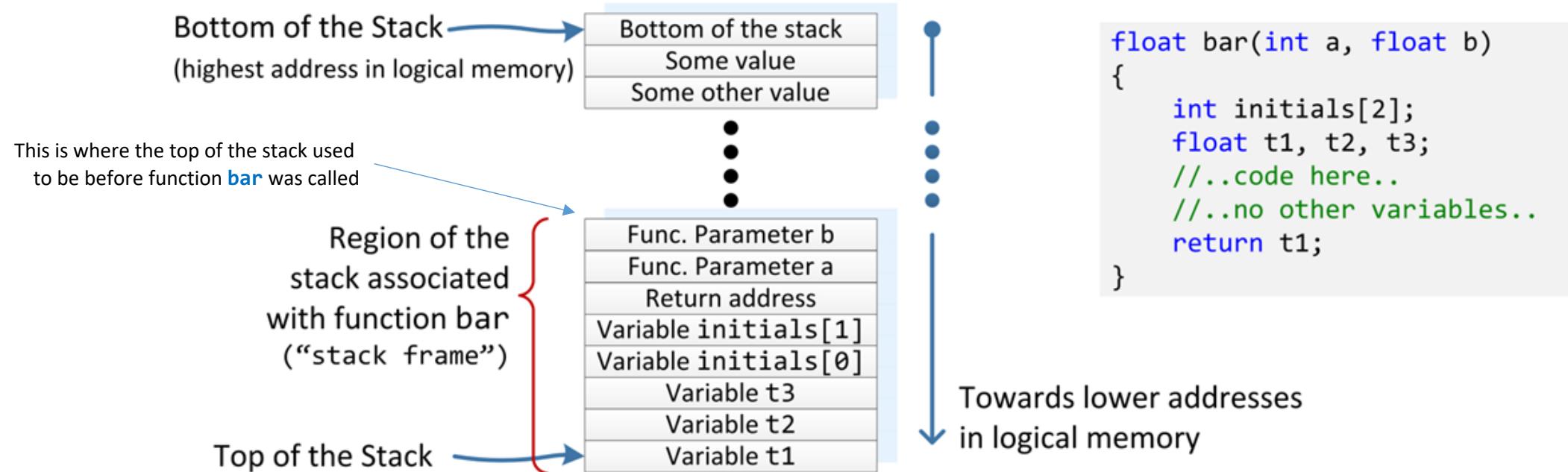
# The Virtual Memory: the anatomy of a living thing

[How perceived by compiler & runtime]



# Anatomy of the Stack

- Function **bar** and associated stack frame
  - Should the function bar be called, this is how the stack should be pushed
  - A “pop” operation is the opposite, it shrinks the stack

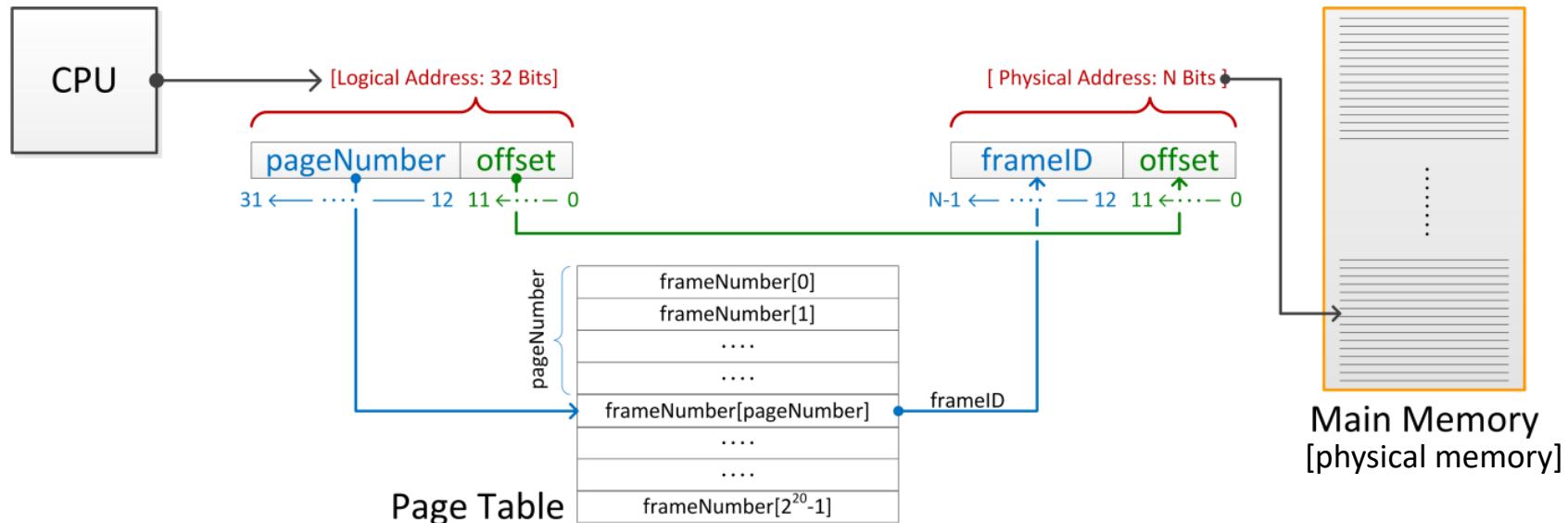


# Virtual Memory: The Page Table

- Virtual memory allows the processor to work in a virtual world in which each process, when run by the processor, seems to have exclusive access to a very large memory space ( $2^{32}$ , addresses, that is)
- This virtual world is connected back to, or mapped back into, the physical memory through a Page Table (PT)
  - This is one level of indirection

# Anatomy of a Virtual Memory Address

- A virtual address split into two parts (another case of address splitting, like for caches, when we split 3 ways):
  - The page number
  - The offset
- Example below assumes a 32-bit OS with virtual pages of  $4096 = 2^{12}$  bytes

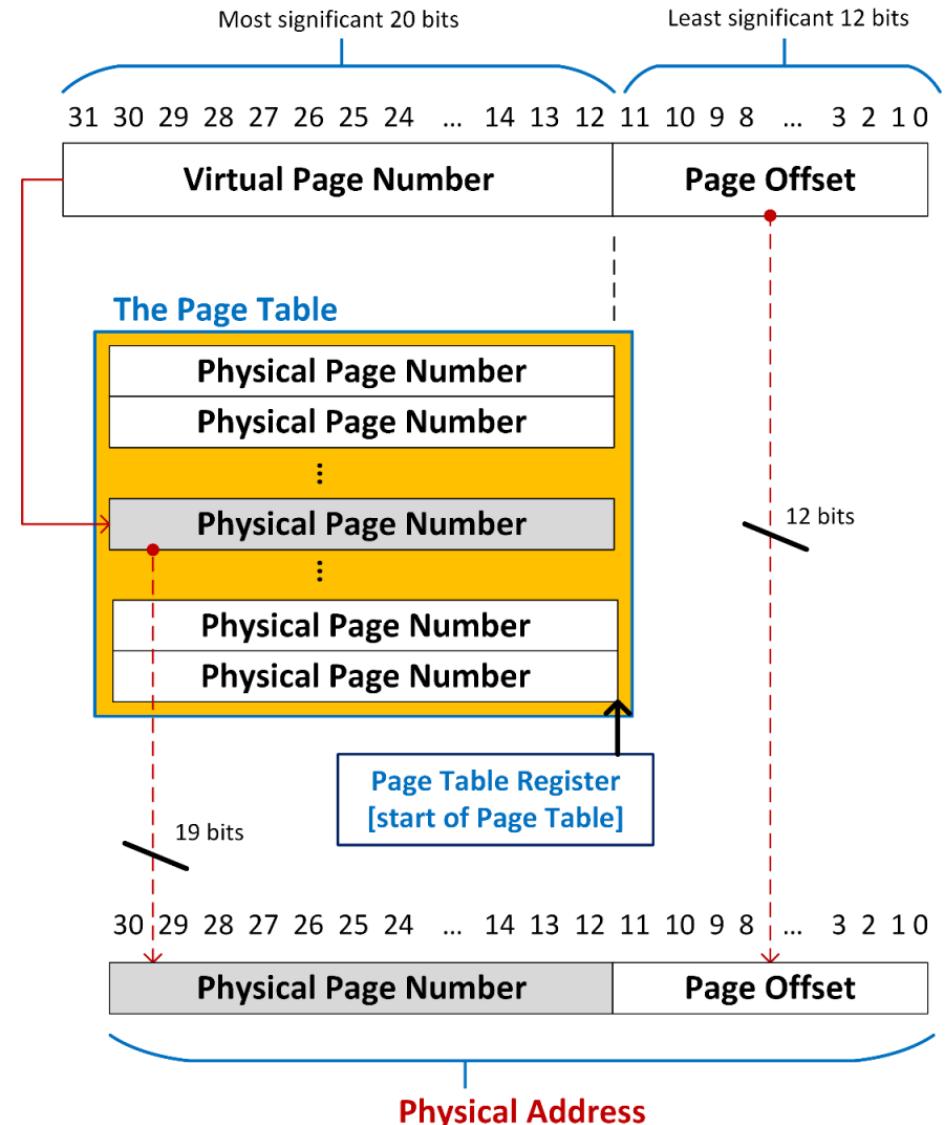


# Anatomy of a Virtual Memory Address

- A **page of virtual memory** corresponds to a **frame of physical memory**
- The size of a page (or frame, for that matter) is typically 4096 bytes
  - Compare to 64 bytes, the size of a cache block (line)
- $2^{12}=4096$ : 12 bits of the address are sufficient to relatively position each byte in a page
  - Bits marked 0 through 11 on the previous slide

# The Translation Process

- Example: imagine that your physical memory is 2 GB
- The physical address has 31 bits:  $2^{31}=2\text{GB}$
- Then the page table converts the bits 12 through 31 of the virtual address into bits 12 through 30 of the physical address



# Comments on the Page Table (PT): Preamble to TLB

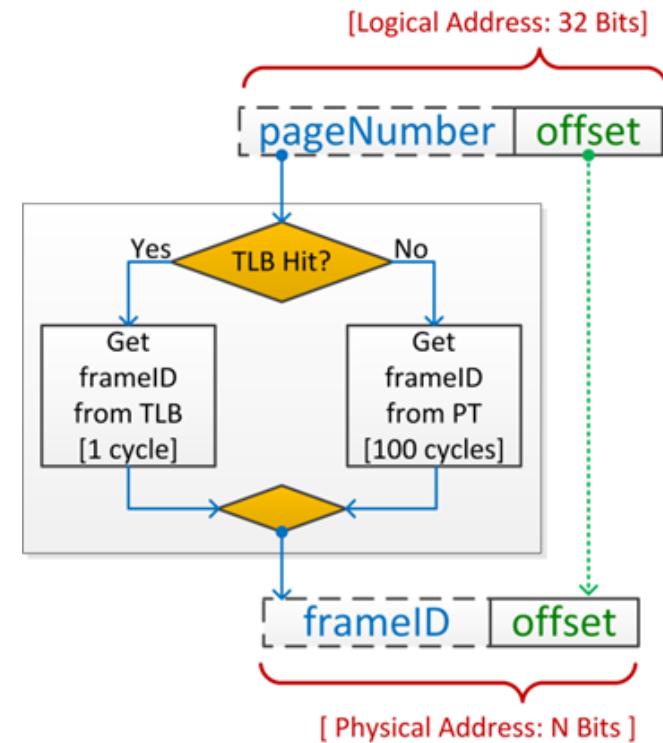
- Page table enables the **translation** of virtual addresses into physical addresses
- Every single process executing on a processor and managed by the OS has its own page table
- Page table is **stored in main memory**
  - For a 32-bit operating system size of a page table can be up to 4 MB in size
    - 4 MB:  $2^{20}$  addresses, each requiring 4(?) bytes to encode
  - Start address of the PT for a process is stored in a dedicated register (PTR)
    - The job of the Operating System to take care of this

# The TLB [Translation Lookaside Buffer] [1/2]

- If you go to Page Table in main memory for **each address translation** this would be very costly
  - One trip to translate address, one trip to actually fetch the data
- There is a “cache” for the **address translation** process: TLB
  - **Translation Lookaside Buffer**: holds the translation of a small collection of virtual page numbers into frame IDs

# Illustration: The role of the TLB

- A TLB is just another cache
  - Implemented in hardware, on-chip
- A TLB miss leads to substantial overhead in the translation of a virtual memory address



# The TLB [Translation Lookaside Buffer] [2/2]

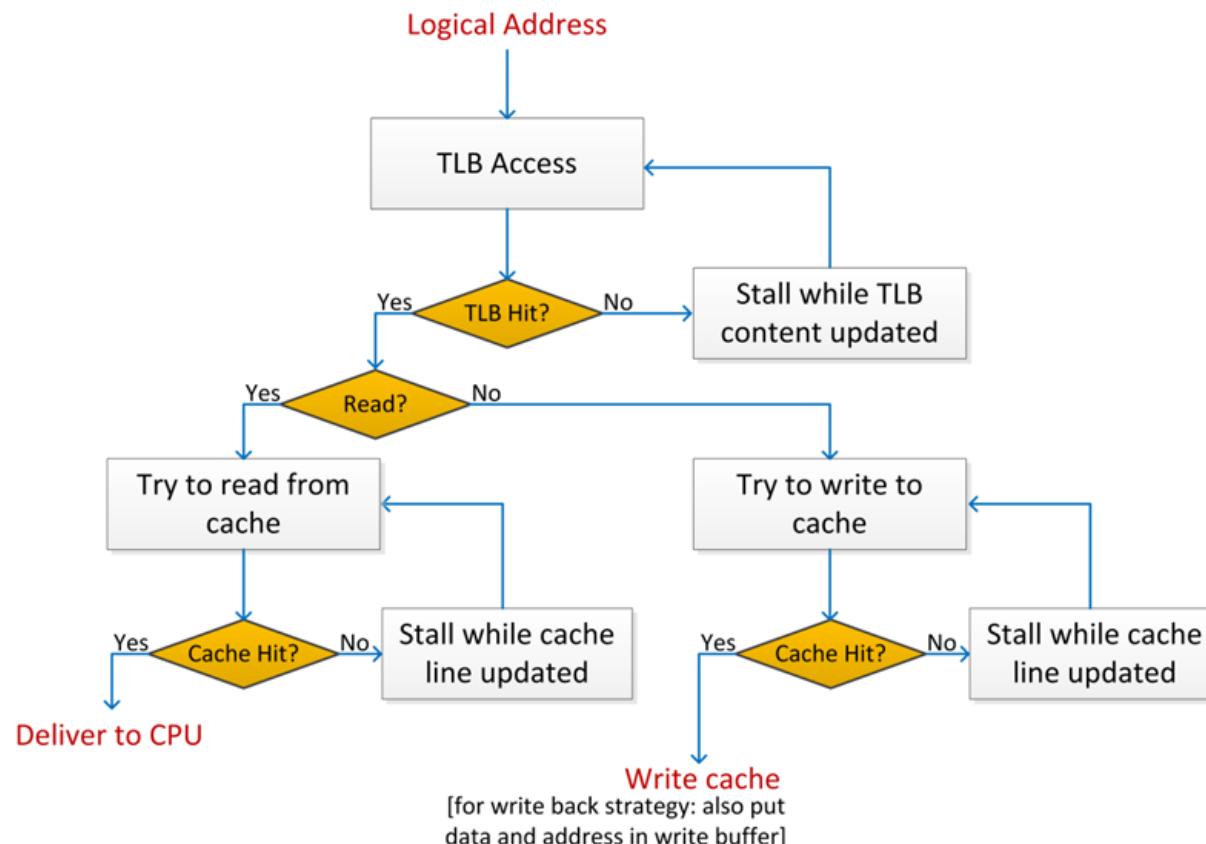
- The Good, the Bad and the Ugly [1966, Rated R, w/ Clint Eastwood, 97% Rotten Tomatoes]
  - The Good: the TLB leads to a hit and allows for quick address translation
  - The Bad: TLB doesn't have the required information cached and a trip to main memory in order
  - The Ugly: the value at address that was just translated is not in cache and trip to main memory in order
  - The Really Ugly: the requested frame not in main memory and a trip to secondary memory in order
    - Called “**page fault**,” you’re hitting significant latency at this point
    - “**memory thrashing**”: repeatedly hitting page fault upon page fault, to the point where time wasted in handling page faults dominates all the rest
- Purpose of this slide: understand what happens when you don’t hit caches and you need to figure out where your data is and then fetch it

# Quick side-trip, speed of memory

- A couple of lectures back, we saw 5X to 30X speedup if we hit the cache (good locality)
  - Scenario: cache miss, you go to main memory
- When handling large data sets, if they don't fit in main memory and spill into secondary memory, lack of locality can lead to 1000X slowdowns
  - Scenario: main memory page fault (even worse if you start getting into memory thrashing)

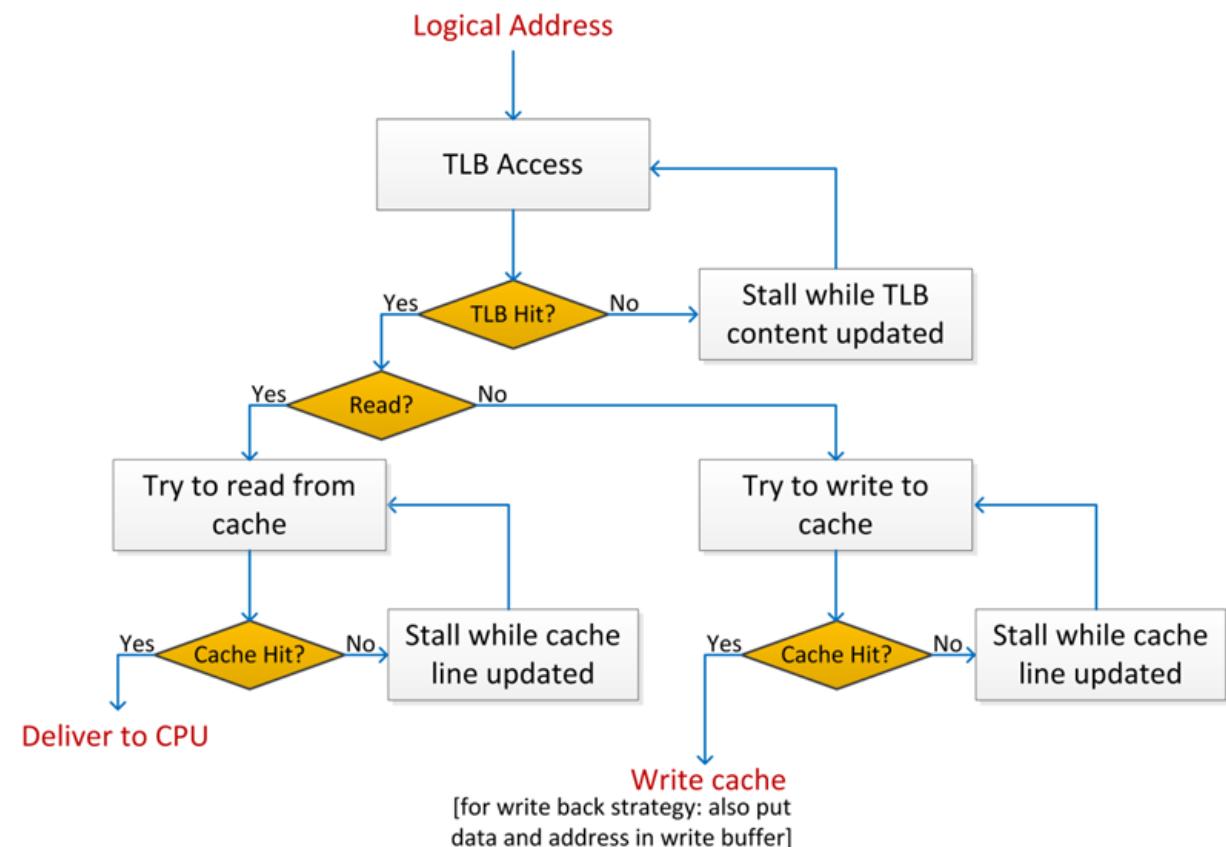
# Memory Access: The Big Picture, Includes Cache

- A simplified version of how a memory request is serviced:



# Quiz: Chop time for Cache read→ Virtual or Physical address?

- To do, for instance, a cache read, you chop the address in three parts
- Based on the picture at the right, what gets chopped in three parts: an address in virtual memory or in physical memory?



# Quiz: How many PTs?

- ILP: you have a superscalar chip and can execute two instructions per clock cycle
  - How many PTs will come into play?
- TLP: your chip has assets to support two processes
  - How many PTs will come into play?

# Memory-related aspects: various other tidbits

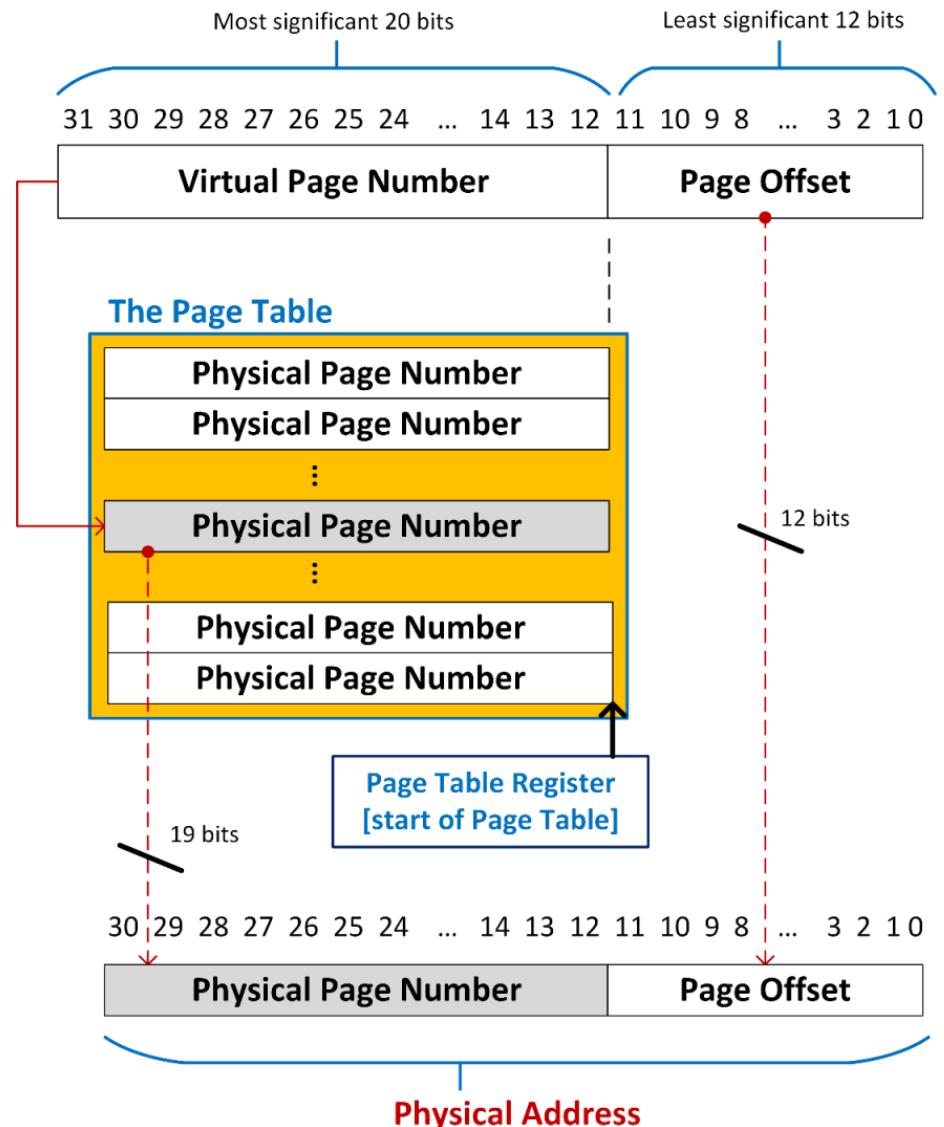
- Going on several tangents
  - Virtual vs. Physical memories: how their size comes into play
  - Pointers in gdb/Visual Studio: pointing into virtual or physical memory?
  - Unit of address resolution
  - 32 to 64 bit transition
  - Frame size

# Virtual vs. Physical memories: how their size comes into play

- Assumption: in this discussion we're dealing with a 32 bit operating system
  - Simpler to talk about it, fewer bits/smaller memories to keep track of. Nothing changes for 64 bit OSs
- Case 1: The physical memory smaller than virtual memory
- Case 2: The physical memory larger than virtual memory

# Case 1: Assume 2GB MB of RAM (my slim laptop)

- 2 GB of RAM: 2X more addresses in virtual memory than you have in physical memory
- Depending how much memory your program uses, multiple virtual pages are going to be mapped into the same physical frame
- **Consequence:** in reality, each entry in the Page Table always has [at least] one additional bit
  - **clean/dirty** bit – indicates that translation to frame should not proceed immediately



# The Clean/Dirty Bit & the concept of Page Fault

- Bit is “clean”: two conditions need to happen for the bit to be clean
  - The frame that the page gets translated into has been loaded in main memory (think onset of execution)
  - Since last translation, that very frame has not been associated with other virtual page
- Bit is “dirty”:
  - We called this before “page fault”, costly scenario to handle
  - The OS will look for the needed frame on disk (secondary memory)
    - The way you can think about it: the RAM (main mem) is the cache of the hard-disk (secondary mem)
    - If bit in Page Table is dirty, the translation refers to address in secondary memory, not in main memory

## Case 2: Assume 16 GB of RAM (beefy workstation)

- $16 \gg 4$  GB: this is good for your code, the likelihood of a memory frame getting evicted is smaller
- PT should be “wide” enough (have enough bits) to recreate addresses to 16 GB of information
  - You need 22 bits for pages of 4096 bytes:
    - 4096: 12 bits
    - 16 GB: need 34 bits to store addresses
    - $34-12=22$  bits needed in each entry of the translation table
  - NOTE: You have the same number of entries in the PT; they’re just wider compared to Case 1.
- Note: clean/dirty bit also present if you have lots of physical memory since other programs run on the machine can render the bit “dirty”
  - Imagine you run your code and Matlab comes in and starts working on 3.4 GB of memory
    - Some of your frames will likely get evicted to disk
      - Recall the concept of memory hierarchy, when things don’t fit, they’re sent down the memory hierarchy

# [Short Topic] gdb/VS Pointers: Pointing In Virtual or Physical Memory?

MemoryIssues (Debugging) - Microsoft Visual Studio

File Edit View Project Build Debug Team Nsight Tools Architecture Test Analyze Window Help Sign in

Process: [11208] pointerVirtMem.exe Lifecycle Events Thread: [17340] Main Thread

Solution Explorer

Search Solution Explorer (Ctrl+;)

Solution 'MemoryIssues' (6 projects)

- ALL\_BUILD
- malware
- matrixRep1D
- matrixRep2D
- pointerVirtMem**
  - References
  - External Dependencies
  - Source Files
    - pointerVirtMem.c
  - CMakeLists.txt
- ZERO\_CHECK

pointerVirtMem.c

```
1 #include <stdio.h>
2
3 int main() {
4     const int arr[3] = {2, -3, 8};
5     const int *p_arr = arr;
6     printf("Address in memory where arr is stored: %p\n", arr);
7     printf("Address pointed to by p_arr: %p\n", p_arr);
8     int b = *(p_arr + 2) + 12;
9     printf("b = %d\n", b);
10    return 0;
11 }
```

Locals

Name	Value	Type
arr	0x0000000dd582ffa18 {2, -3, 8}	const int[3]
b	20	int
p_arr	0x0000000dd582ffa18 {2}	const int *

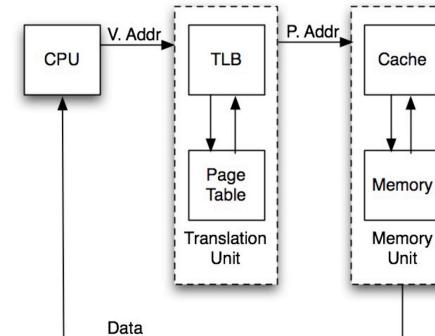
Solution Explorer Class View Autos Locals Threads Modules Watch 1 Find Symbol Results

Compiler Inline Report Compiler Optimization Report Call Stack Breakpoints Exception Settings Output

Ready Ln 9 Col 12 Ch 12 INS ↑ Publish

- Q: Is 0x0000000DD582FFA18 an address in the virtual mem. or in the physical mem.?

- A: This is a virtual memory address
  - You see physical addresses if you are Linus Torvalds and work on kernel development



[<https://inst.eecs.berkeley.edu/~cs61c/sp14/disc/12/d12Sol.pdf>]→

Quick note: addresses are provided in hex representation

Fewer digits to look at (would you prefer to look at 64 binary digits?)

Easy to figure out – each byte filled w/ binary digits translates into two digits out of the 16 hex digits used for the address

# [Short Topic] The Unit of Address Resolution

- How many bits are available for data storage at each address?
- Example:
  - We have  $2^{32}$  addresses that we can access
  - If each address points to a location that stores 8 bits (one byte) then we have 4 GB of addressable memory
  - However, if each address refers to a location that stores 2 bytes, we have 8 GB of addressable memory
- Intel and AMD CPUs: the unit of address resolution is 1 byte (8 bits)
- Consequence: the Intel 32 bit processors “see” a virtual memory space that can be 4 GB big and can reference each byte therein

# [New Short Topic] The 32 to 64 bit migration (1/2)

- If the architecture and OS have 32 bits to represent addresses, it means that  $2^{32}$  addresses can be referenced
- The data sets that we're working with today are really large
  - Having only  $2^{32}$  addresses to work with hardly enough
  - Example: I have 32 GB of RAM and need to work with 8 GB array. How do I reach the end of array? Addresses are not large enough...
- This motivated push towards having addresses represented using 64 bits:
  - Memory space that I can address balloons to  $2^{64}$  bytes

# The 32 to 64 Bit Migration (2/2)

- Note that a 64 bit architecture typically calls for two things:
  - From a **hardware** perspective: size of the registers, translation tables, tags in caches, etc. – they need to change
  - From a **software/OS** perspective, the program “operates” in a huge virtual memory space
    - The new number of addressable bytes: 18,446,744,073,709,551,615
    - In reality, the virtual memory addresses are represented using only about 40 to 50 bits

# Example [using CAE's tux-103]

```
>> less /proc/cpuinfo
```

```
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 60
model name    : Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz
stepping       : 3
microcode     : 0x20
cpu MHz       : 3197.500
cache size    : 6144 KB
physical id   : 0
siblings       : 4
core id        : 0
cpu cores     : 4
apicid         : 0
fpu            : yes
fpu_exception  : yes
cache_alignment: 64
address sizes  : 39 bits physical, 48 bits virtual
```



# [New Short Topic] The size of the frame/page

- Refresher: a memory page, or virtual page, is a fixed-length contiguous block of virtual memory, described by a single entry in the Page Table
- The frame is the fysical (or physical) counterpart of a virtual page
- Typically, size of page is 4 KB but can be of 1024 KB for certain applications
  - Example: working with very large data sets (big data)

# The size of the frame/page

- Large page size, pros
  - Less TLB pressure
  - Why?
    - Fewer pages to blanket the virtual memory means fewer entries in the Page Table, which means
      - Fewer values that compete for storage in the TLB cache. More TLB hits: faster program
      - [not that big of a deal] The size of the Page Table is smaller – less memory needed to store it
- Large page size, cons
  - Might lead to wasteful use of physical memory
    - Example: page size 1024 KB. If you need to allocate 1030 KB of memory, you use two pages/frames
      - If your process doesn't use rest of the second page, then there'll be a sizeable hole in physical memory

# The size of page/frame

Architecture	Smallest page size	Larger page sizes
x86-64	4 KiB	2 MiB, 1 GiB
UltraSPARC Architecture 2007	8 KiB	64 KiB, 512 KiB, 4 MiB, 32 MiB, 256 MiB, 2 GiB, 16 GiB
SPARC v8 with SPARC Reference MMU	4 KiB	256 KiB, 16 MiB
Power Architecture	4 KiB	64 KiB, 16 MiB, 16 GiB
IA-64 (Itanium)	4 KiB	8 KiB, 64 KiB, 256 KiB, 1 MiB, 4 MiB, 16 MiB, 256 MiB
ARMv7	4 KiB	64 KiB, 1 MiB, 16 MiB
32-bit x86	4 KiB	2 MiB, 4 MiB

# Departing thoughts, memory aspects

- Memory transactions:
  - Add overhead, at times rather large, owing to complex process of moving data
    - From **disk** to **main memory** to **cache** to **register**
  - Time and power costly
- Keep in mind the memory hierarchy; understand the size of the memories on your machine
  - The closer to the top of the hierarchy, the better off you are
- If the speed of the code matters, pay attention to how you use memory

# Side-trip: Using C/C++ in this class

- Using C/C++ in this class: good vehicle to explain how memory works
  - Two aspects that bring C close to hardware: pointers & the `malloc` (memory allocation) function call
- The overhead of memory transactions is there in any language (Fortran, Matlab, Python, etc.)
- For your favorite language, learn how indexing into your arrays doesn't violate locality

# Parallel computing: why, and why now?

# Acknowledgements

- Sources for some of the material presented today
  - Hennessy and Patterson (Computer Architecture, 5<sup>th</sup> edition)
  - John Owens, UC-Davis
  - Darío Suárez, Universidad de Zaragoza
  - John Cavazos, University of Delaware
  - Others, as indicated on various slides
- I apologize if I included a slide and didn't give credit where was due
- Any mistakes in these slides belong to me, not the individuals above

# Argument made in this segment of the course

- Sequential computing execution speed pretty much flat for a decade now
- Parallel computing is where speed improvements have come from

# Main causes for sequential computing speed stalling



- Memory Wall
- Instruction Level Parallelism (ILP) Wall
- Power Wall
- Not necessarily walls, but increasingly steep hills to climb

Source: “*The Many-Core Inflection Point for Mass Market Computer Systems*”,  
by John L. Manferdelli, Microsoft Corporation

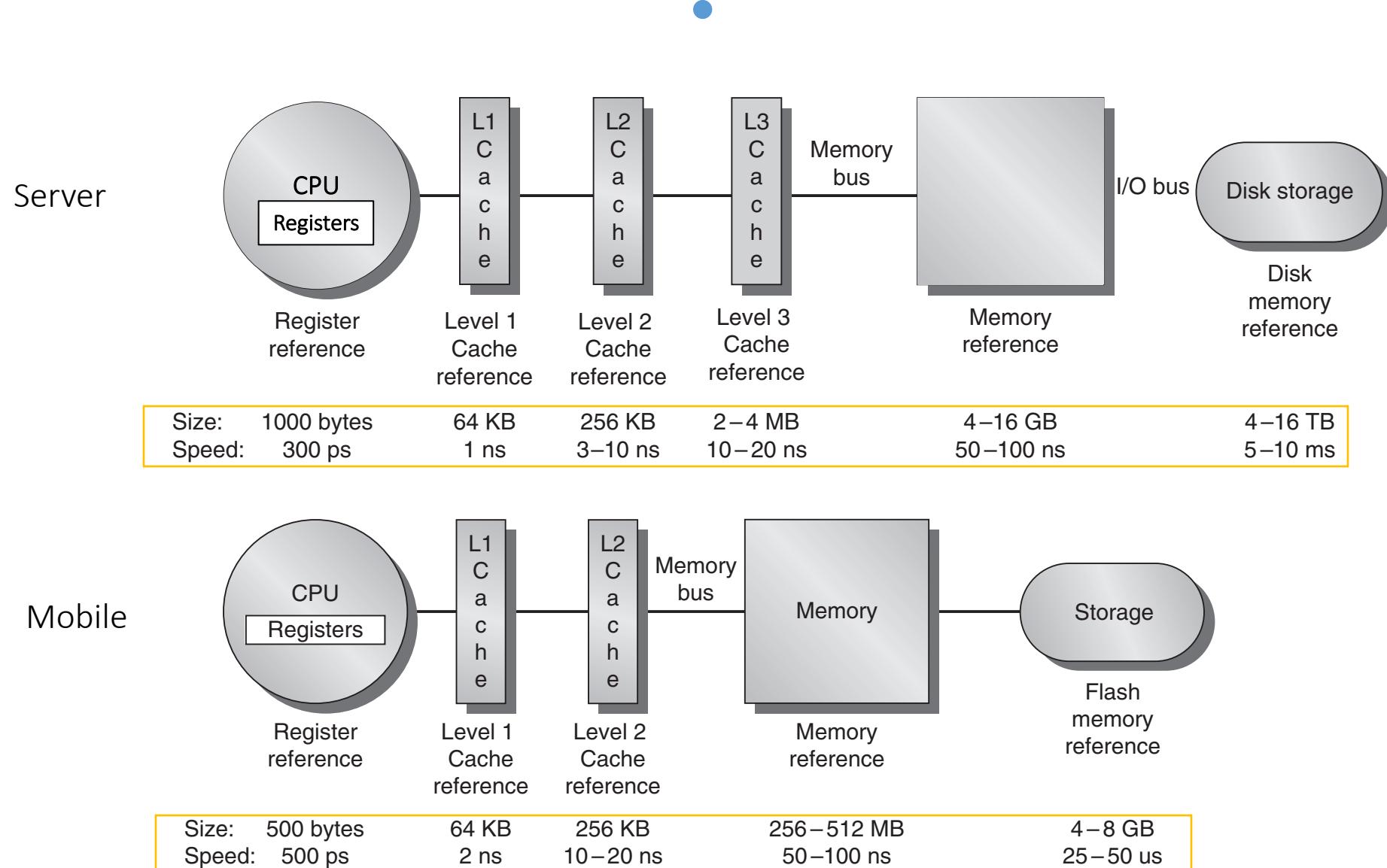
<http://www.ctwatch.org/quarterly/articles/2007/02/the-many-core-inflection-point-for-mass-market-computer-systems/>

# Memory Wall

- Memory Wall: What is it?
  - The growing disparity of speed between CPU and memory outside the chip
- Memory accesses have gradually become a mighty barrier to computer performance improvements
  - The trend: **ever growing caches** to improve **average memory reference time** to fetch or write instructions or data
    - Caches are a transistor black hole
    - Also, as cache gets gradually larger, it also gets gradually slower
- Memory Wall: due to *latency* and limited communication *bandwidth* beyond chip boundaries.

# Memory Latencies

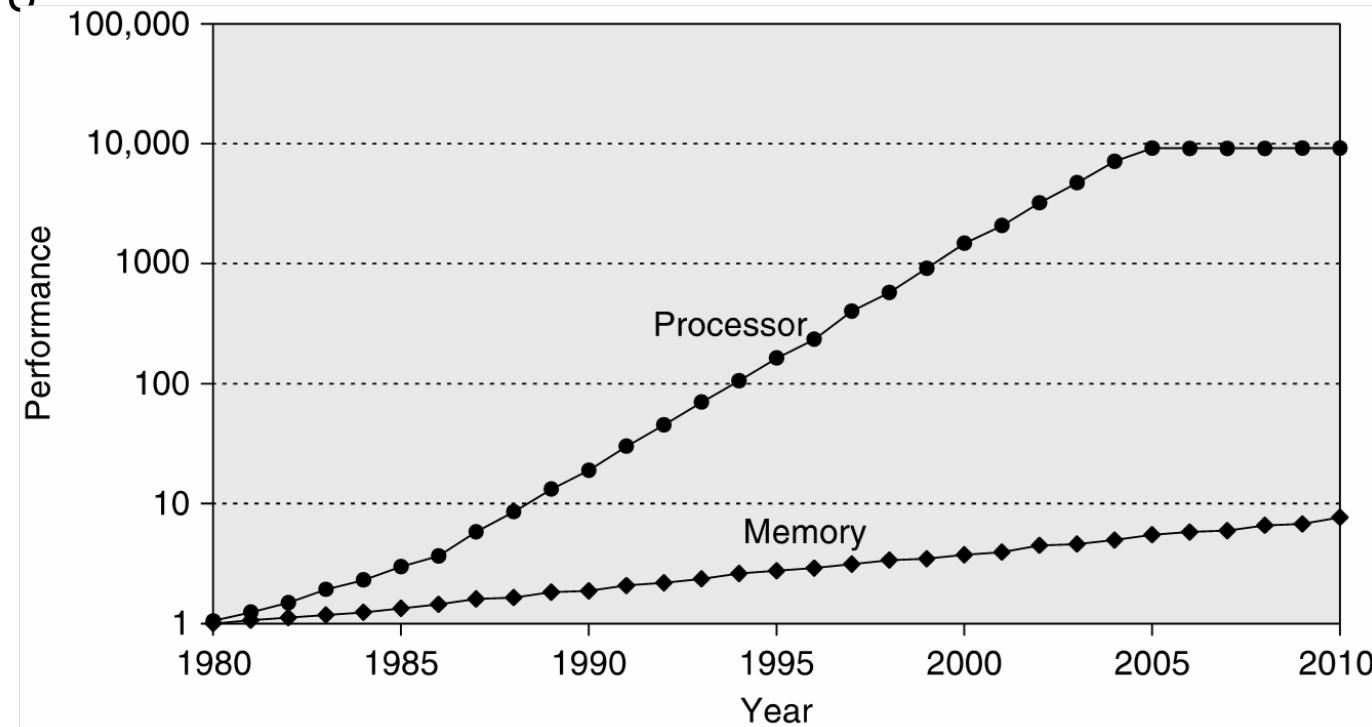
[typical server and mobile computers]



# Memory Speed:

## Widening of the Processor-DRAM Performance Gap

- Plot: \*log\* scale, the increasing gap between CPU and memory
  - The memory baseline: 64 KB DRAM in 1980
- Memory speed increasing rate:  $\approx 1.07/\text{year}$
- Processors improved
  - 1.25/year (1980-1986)
  - 1.52/year (1986-2004)
  - 1.20/year (2004-2010)
- You have the chops to process data, but can't bring data to processor fast enough



# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 08

02/07/2020

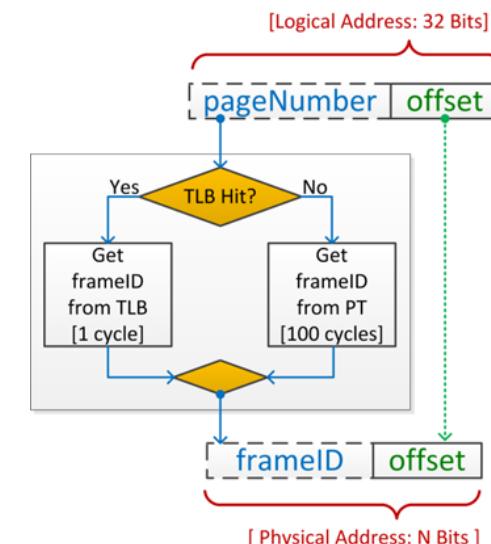
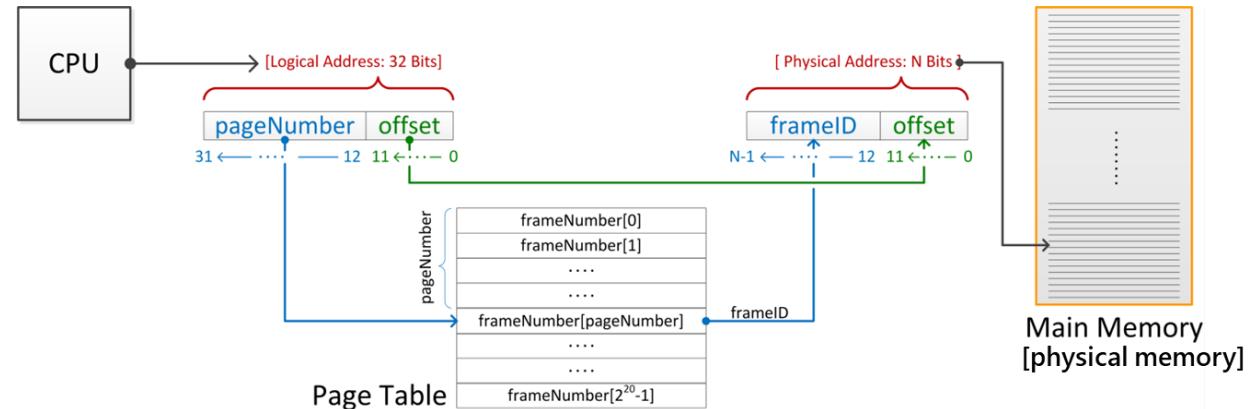
# Quote of the day

“Wikipedia is the best thing ever. Anyone in the world can write anything they want about any subject. So you know you are getting the best possible information.”

-- Michael Scott, Regional Manager, Dunder Mifflin, Inc.

# Before we get going...

- Last time:
  - The Virtual Memory
    - A good thing, for many reasons
  - The Page Table
  - The TLB
- Today
  - The 3 walls to sequential computing
  - Parallel computing, generalities
- Other tidbits:
  - Assigned reading (syllabus gets updated)
  - Assignment due on Th at 9 pm
  - Use Piazza to get things going (for you or others)



# Main causes for sequential computing speed stalling



- Memory Wall
- Instruction Level Parallelism (ILP) Wall
- Power Wall
- Not necessarily walls, but increasingly steep hills to climb

Source: “*The Many-Core Inflection Point for Mass Market Computer Systems*”,  
by John L. Manferdelli, Microsoft Corporation

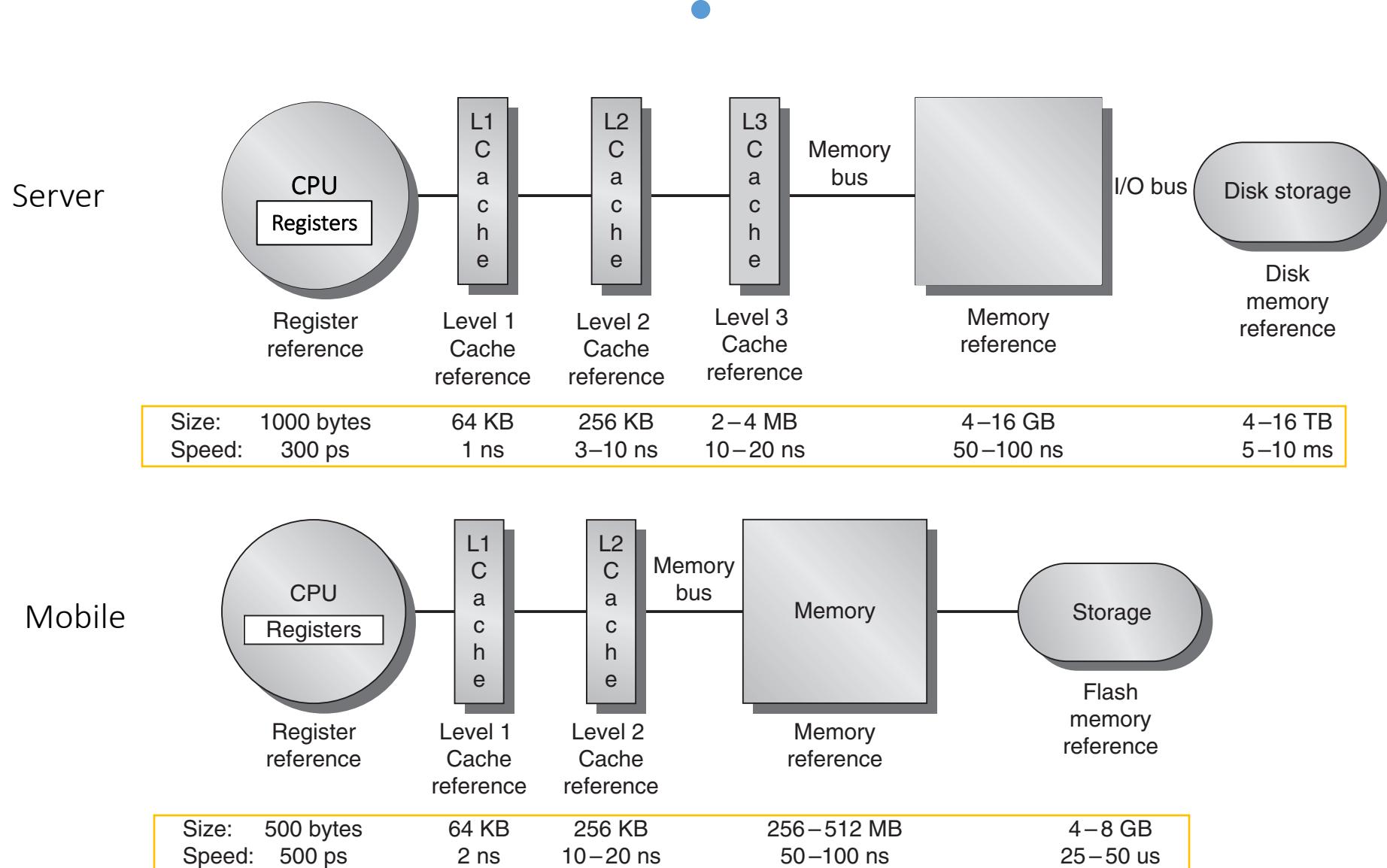
<http://www.ctwatch.org/quarterly/articles/2007/02/the-many-core-inflection-point-for-mass-market-computer-systems/>

# Memory Wall

- Memory Wall: What is it?
  - The growing disparity of speed between CPU and memory outside the chip
- Memory accesses have gradually become a mighty barrier to computer performance improvements
  - The trend: **ever growing caches** to improve **average memory reference time** to fetch or write instructions or data
    - Caches are a transistor black hole
    - Also, as cache gets gradually larger, it also gets gradually slower
- Memory Wall: due to *latency* and limited communication *bandwidth* beyond chip boundaries.

# Memory Latencies

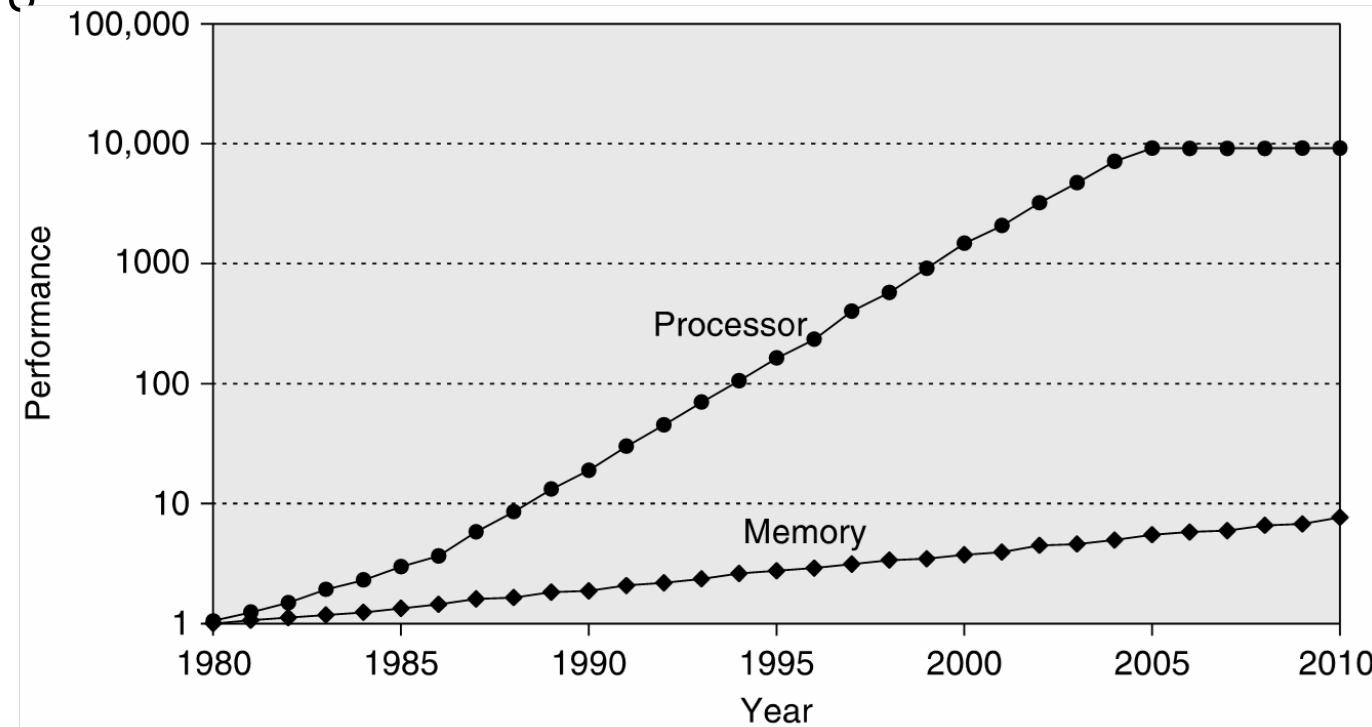
[typical server and mobile computers]



# Memory Speed:

## Widening of the Processor-DRAM Performance Gap

- Plot: \*log\* scale, the increasing gap between CPU and memory
  - The memory baseline: 64 KB DRAM in 1980
- Memory speed increasing rate:  $\approx 1.07/\text{year}$
- Processors improved
  - 1.25/year (1980-1986)
  - 1.52/year (1986-2004)
  - 1.20/year (2004-2010)
- You have the chops to process data, but can't bring data to processor fast enough



# Memory **Latency** vs. Memory **Bandwidth**

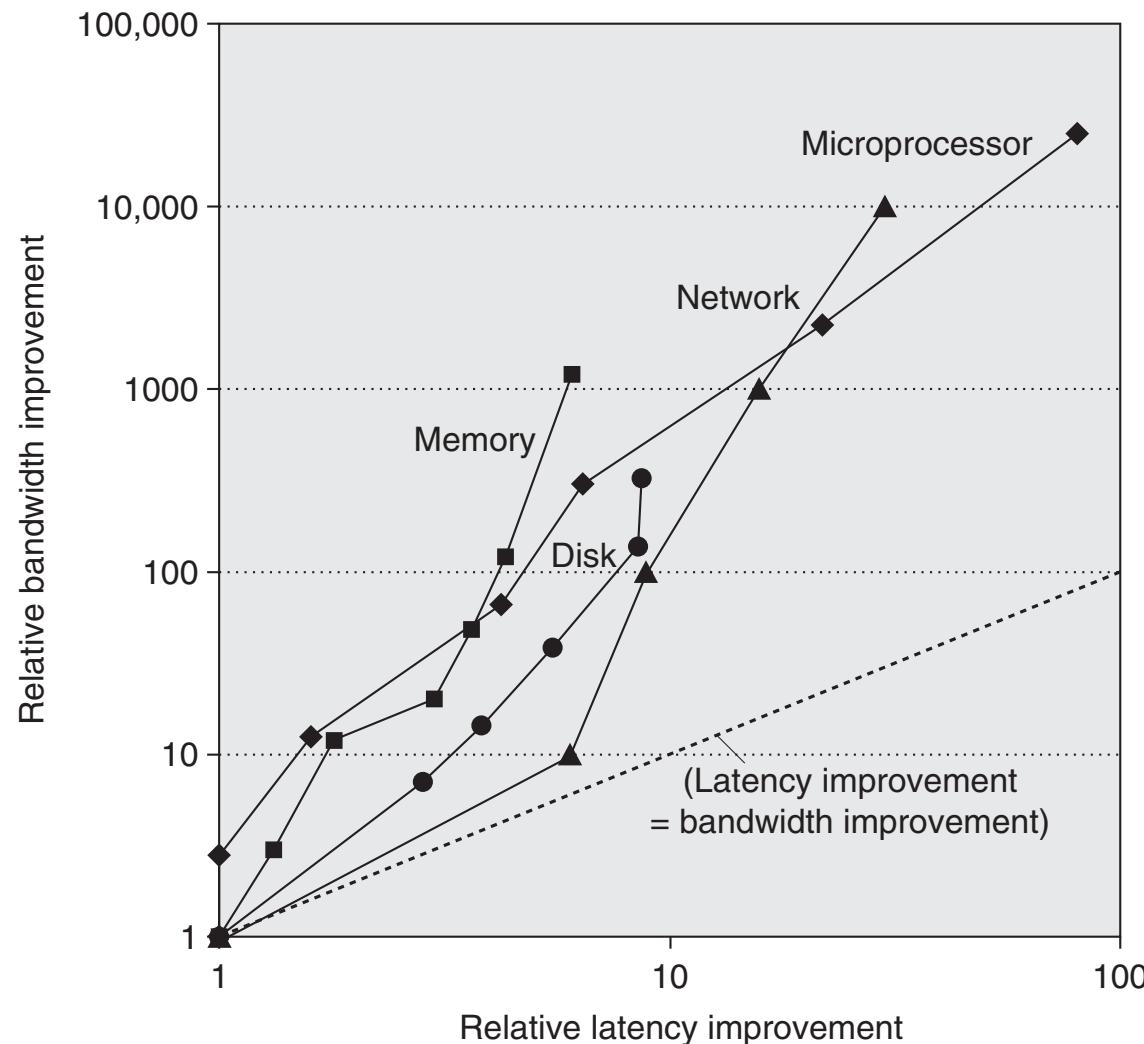
- Latency [in our context]: how much time it takes a packet of data to get from one designated point to another
  - Measured in seconds, also called “lag”
  - The utility “ping” in Linux measures the latency of a network
  - For memory transactions: send 32 bits to destination and back, measure how much time it takes gives you latency
- Bandwidth: how much data can be transferred per second
  - You can talk about bandwidth for memory but also for a network (Ethernet, Infiniband, modem, DSL, etc.)
- Improving Latency and Bandwidth
  - Mostly the job of folks in Electrical Engineering
  - Once in a while, Materials Science colleagues deliver a breakthrough
  - Promising technology: optic networks; layered memory on top of chip; etc.

# Memory Latency vs. Memory Bandwidth



- Memory Access Latency more challenging to improve as opposed to improving Memory Bandwidth
- Improving Bandwidth: add more “pipes”
  - Requires more pins that come out of the chip for DRAM, for instance
  - Adding more pins is not simple – very crowded real estate plus the technology is tricky
- Improving Latency: no easy answer here
- Analogy:
  - If you carry commuters with a train, add more cars to a train to increase bandwidth
  - Improving latency requires the construction of high speed trains (expensive/complex)

# Latency vs. Bandwidth: Recent Evolution of Speed Gains



# Memory Wall, Conclusions

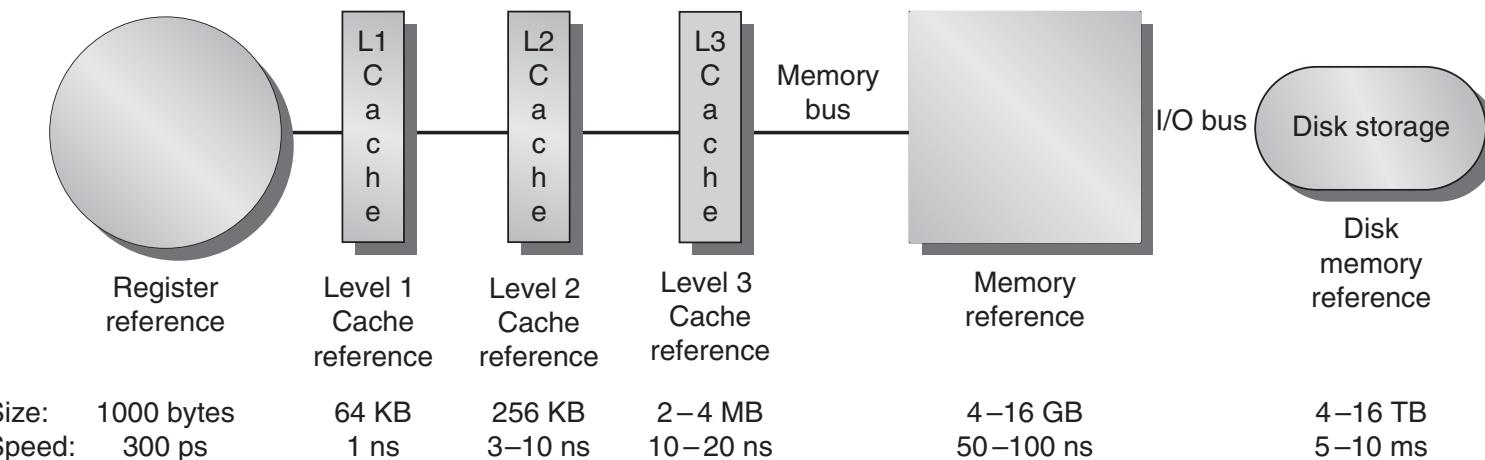
[IMPORTANT SLIDE]

- Take-home message: Long and winding conversations with main memory kills execution speed
- Memory-wise, you may or may not be at top speed for your memory
  - Effective bandwidth==Nominal bandwidth: you move lots of data - ok, fact of life, some applications are just like that
  - Effective bandwidth < Nominal bandwidth: you bring a block of data, use little of it, and ask for data from a different block
    - Examples – sparse matrix operations, combinatorics, etc.

# Memory Wall, Conclusions

[IMPORTANT SLIDE]

- This is an important picture to have in mind when writing software



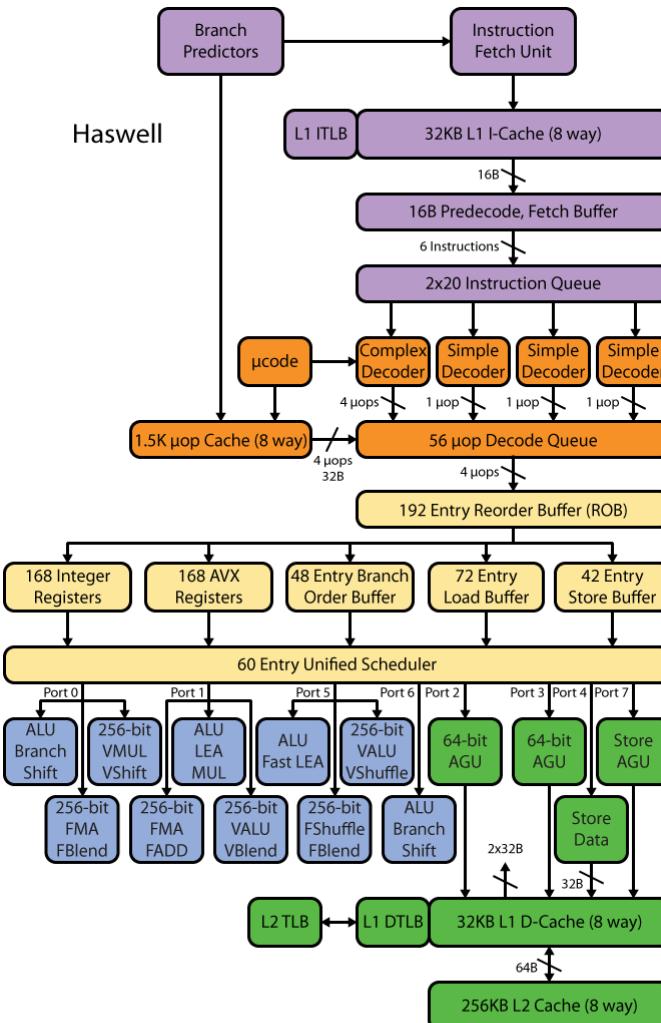
## Memory Access Patterns

To/From Registers	Golden
To/From Cache	Superior
To/From RAM	Trouble
To/From Disk	Sad day...

# The next wall: Instruction Level Parallelism (**ILP**)

- **Instruction pipelining:** the execution of multiple instructions overlapped; each instruction is divided into series of sub-steps (termed: micro-operations)
- **Superscalar execution:** multiple execution units are used to execute multiple instructions in parallel
- **Out-of-order execution:** instructions execute in any order but without violating data dependencies
- **Register renaming:** a technique used to avoid data hazards and thus lead to unnecessary serialization of program instructions
- **Speculative execution:** allows the execution of complete instructions or parts of instructions before being sure whether this execution is required
- **Branch prediction:** used to avoid delays (termed: stalls). Used in combination with speculative execution.

# Intel Haswell example: ILP elicits very complex microarchitecture



# Sample Pipeline Depths

Pipeline Depth	Processors
6	UltraSPARC T1
7	PowerPC G4e
8	UltraSPARC T2/T3, Cortex-A9
10	Athlon, Scorpion
11	Krait
12	Pentium Pro/II/III, Athlon 64/Phenom, Apple A6
13	Denver
14	UltraSPARC III/IV, Core 2, Apple A7/A8
14/19	Core i*2/i*3 Sandy/Ivy Bridge, Core i*4/i*5 Haswell/Broadwell
15	Cortex-A15/A57
16	PowerPC G5, Core i*1 Nehalem
18	Bulldozer/Piledriver, Steamroller
20	Pentium 4
31	Pentium 4E Prescott

# The ILP Wall

- Dedicated hardware speculatively executes future instructions before the results of current instructions are known, while providing hardware safeguards to prevent the errors that might be caused by out of order execution
- Branches must be “guessed” to decide what instructions to execute simultaneously
  - If you guessed wrong, you throw away that part of the result
- Data dependencies may prevent successive instructions from executing in parallel, even if there are no branches
- **Predicting the future** comes at the cost of microarchitecture complexity and power cost

# The ugly head of speculative execution

- Speculative execution difficult to implement safely
- Has become infamous for a recently-revealed series of bugs
  - Meltdown (Intel, IBM, some ARM)
  - Foreshadow (Intel)
  - Spectre (everything before 2019 that uses branch prediction)

# [The last wall] The Power Wall

- Power dissipation related to the clock frequency and feature length
  - Consequence (the wall): clock rates are limited to some threshold value
- Significant increase in clock speed without heroic/expensive cooling not possible. Chips would simply melt
- Clock speed increased by a factor of 4,000 in less than two decades
  - The ability of manufacturers to dissipate heat is limited though...
  - Looking back at the last ten years → the clock rates pretty much flat

# The Power Wall

- Power, and not manufacturing, limits traditional general purpose microarchitecture improvements
  - Point made already in 1999 by F. Pollack, Intel Fellow, see [here](#) .
  - Kind of true, but sadly Intel not pushing the manufacturing frontier anymore
- Leakage power dissipation gets worse as gates get smaller, because gate dielectric thicknesses decrease proportionately

# Dennard's Scaling: Making Moore's Law Useful

- Dennard scaling: dictates how the voltage/current, frequency should change in response to smaller feature size
- Voltage lower and lower, static power losses more prevalent
- The bad news: direct tunneling gate leakage current (thin dielectric)  $\oplus$  transistors too close
- Amount of power dissipated grows to high levels
  - Thermal runaway
  - Moving towards threshold at which computing not reliable
    - What's noise and what's information?

# The “dark silicon”

- Dark Silicon: transistors that manufacturers cannot afford to turn on
- Too many transistors on a chip – cannot afford to power them lest the chip melts

Esmaeilzadeh, H.; Blehm, E.; St.Amant, R.; Sankaralingam, K.; Burger, D., "Dark silicon and the end of multicore scaling," in Comp. Architecture (ISCA), 2011 38th Annual International Symp., pp.365-376, 2011

# One slide detour: “spending energy” rates

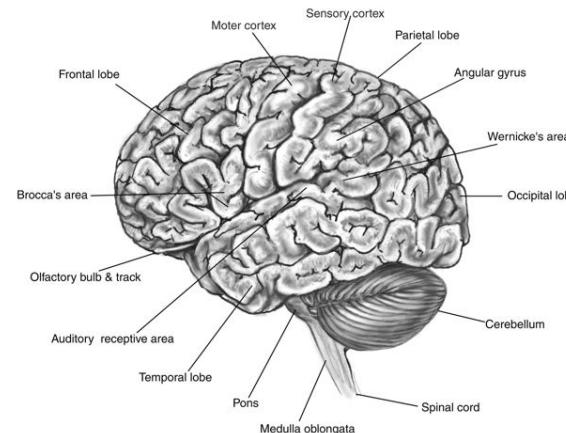
- NVIDIA Tesla K80
  - TDP: 300 Watts



- Intel® Xeon® Processor E7-8890 v3
  - TDP: 165 Watts



- Human Brain
  - 20 Watts
  - Represents 2% of our mass
  - Burns 20% of all energy in the body at rest
  - Our entire body, ball park: 100-120 Watts



# Shift in Perception

Yesterday's school of thought

Increasing clock frequency is primary method of performance improvement

Today's take

Processors parallelism is primary method of performance improvement



Yesterday's school of thought

Less than linear scaling for a multiprocessor is failure

Today's take

Even sub-linear speedups are beneficial as long as you beat the sequential on a watt-to-watt basis

# Conventional Wisdom in Computer Architecture

- Old: Transistors expensive; Power is free
- New: Transistors free; Power management very tricky  
(consequences, “lots of power”: heat to be dissipated & battery management)
- Old: Math is slow; Memory access is fast
- New: Math is fast; Memory slow (200 cycles for DRAM memory access, 1 cycle for pipelined FMA)
- Old: Increasing Instruction Level Parallelism hardware and compilers (Out-of-order, speculation, VLIW, ...)
- New: ILP sees diminishing returns; non-ILP parallelism pursued

Power Wall

Memory Wall

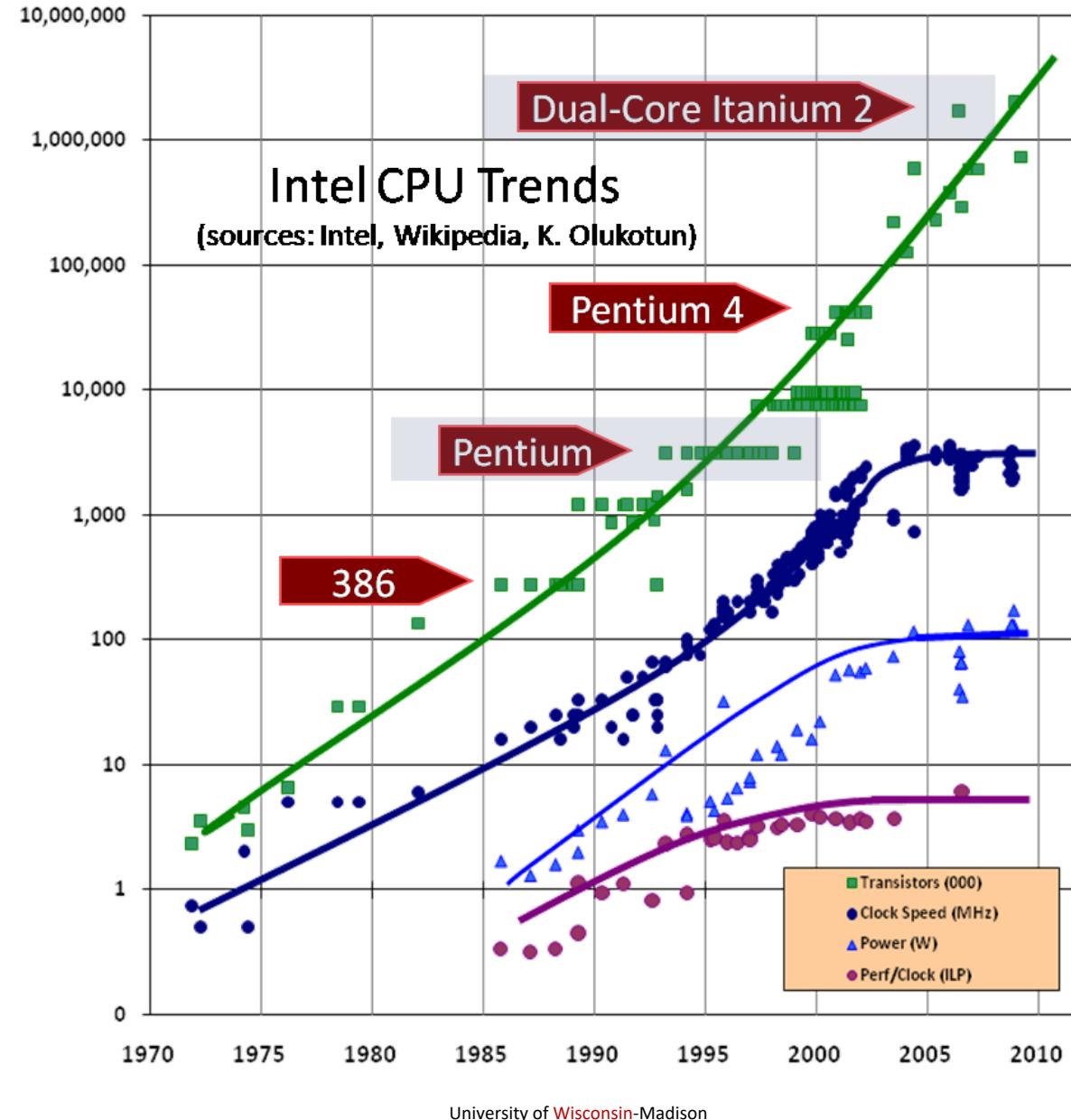
ILP Wall

- Just a bunch of bad news...
- OK, now what?

# The bright spot

- We can produce small[er] transistors and we are increasingly good at packaging them in ICs

# Good news: number of transistors going up



# Moore's Law (1965)

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year [see graph on [next page](#)]. Certainly over the short term this rate can be expected to continue, if not to increase. [Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.](#) That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer."

"Cramming more components onto integrated circuits" by Gordon E. Moore, Electronics, Volume 38, Number 8, April 19, 1965

# Moore's Law

- 1965 paper: Doubling of the number of transistors on integrated circuits every two years
  - Moore himself wrote only about the **density of components (or transistors) at minimum cost**

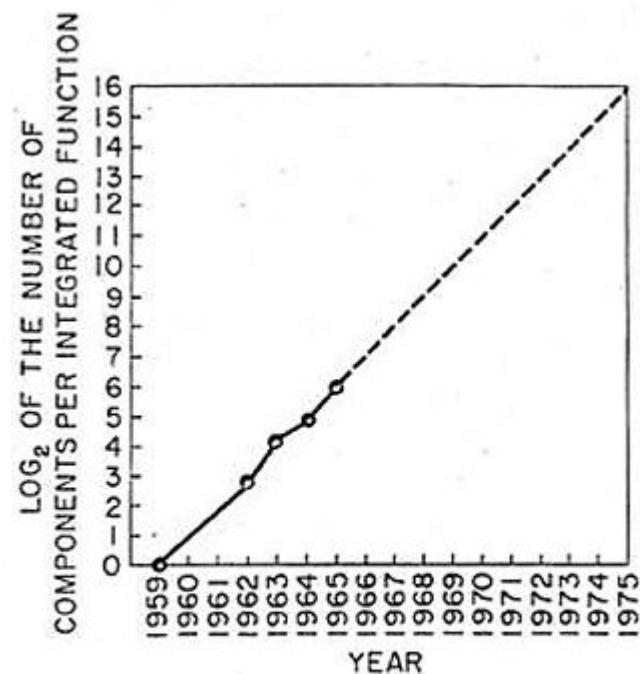
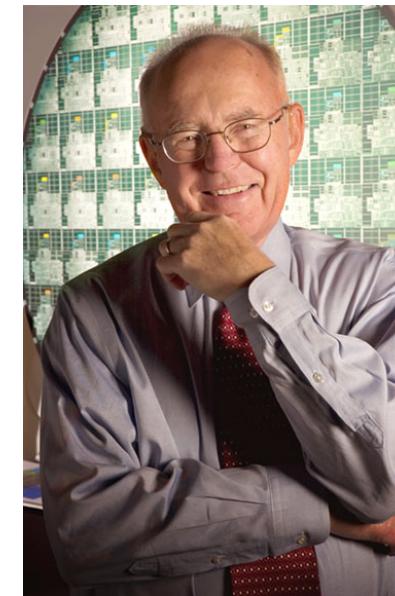


Fig. 2 Number of components per Integrated function for minimum cost per component extrapolated vs time.

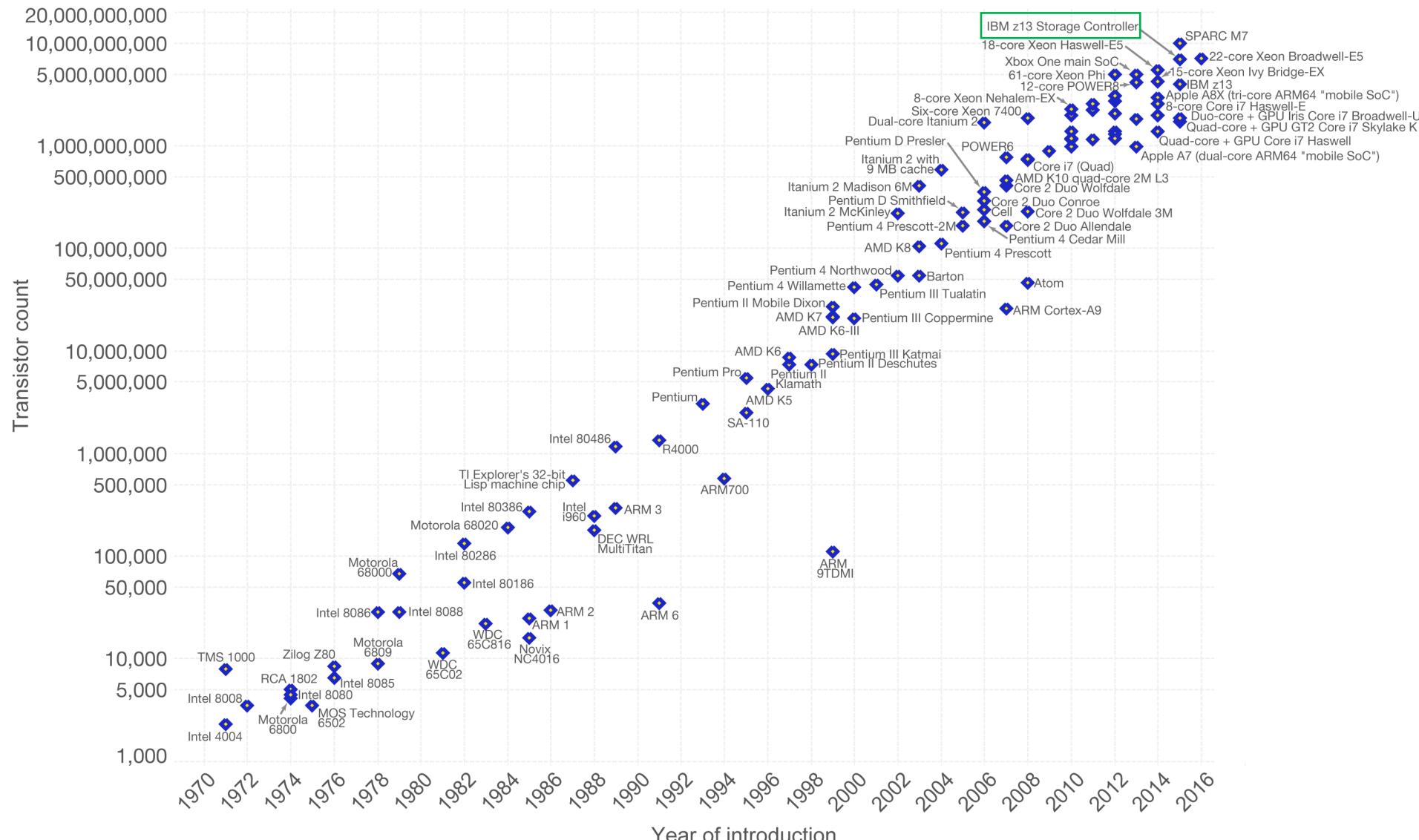


Moore's Law – The number of transistors on integrated circuit chips (1971-2016) 

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.

Our World  
in Data

This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia ([https://en.wikipedia.org/wiki/Transistor\\_count](https://en.wikipedia.org/wiki/Transistor_count))

The data visualization is available at [OurWorldInData.org](http://OurWorldInData.org). There you find more visualizations and research on this topic.

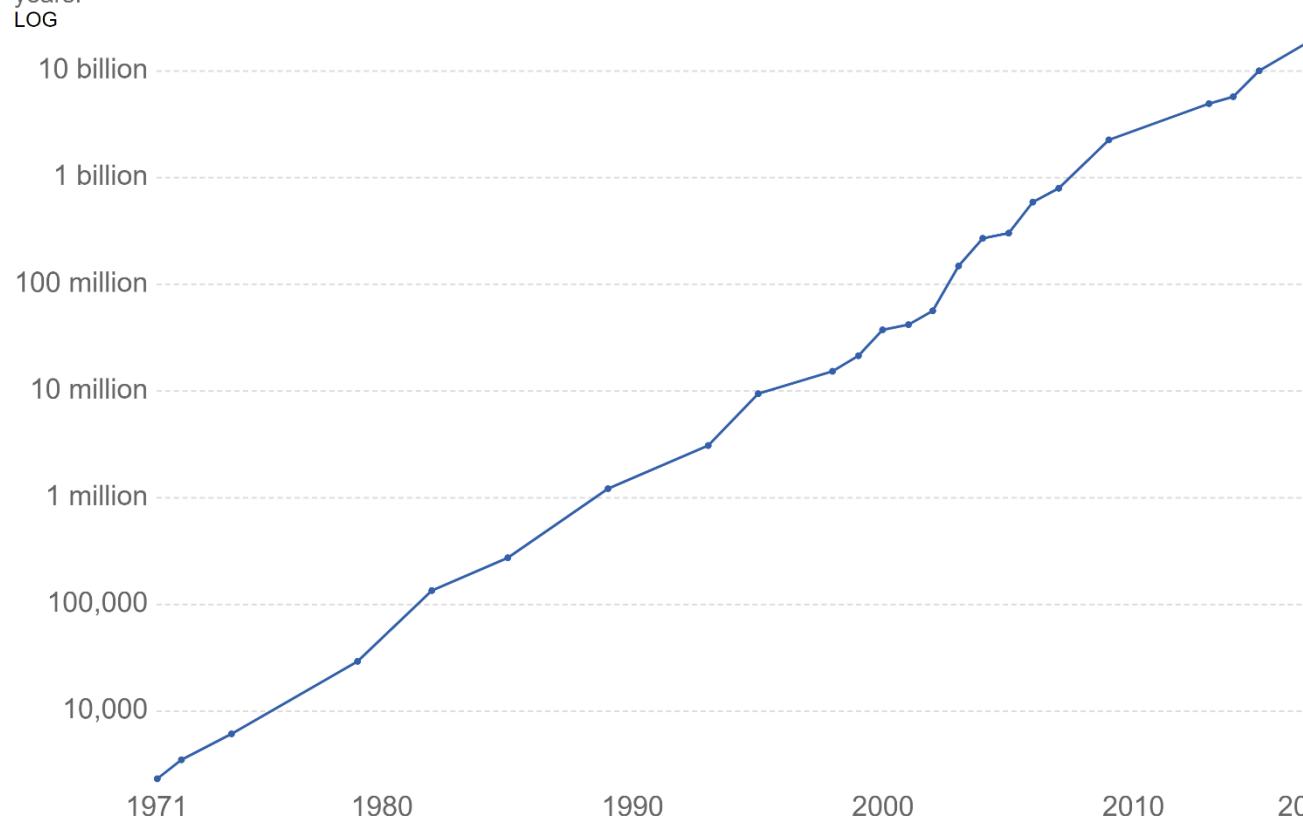
Licensed under CC-BY-SA by the author Max Roser.

# Moving to the processor: transistor counts per microprocessor

## Moore's Law: Transistors per microprocessor

Number of transistors which fit into a microprocessor. This relationship was famously related to Moore's Law, which was the observation that the number of transistors in a dense integrated circuit doubles approximately every two years.

Our World  
in Data



Source: Karl Rupp. 40 Years of Microprocessor Trend Data.

CC BY-SA

# Intel Roadmap [or other folks' roadmap]

- 2013 – 22 nm Tick: Ivy Bridge – Tock: Haswell
- 2015 – 14 nm Tick: Broadwell – Tock: Skylake
- 2016 – 14nm “Refresh” Kaby Lake
- 2017 – 10 nm Cannon Lake (delayed to 2019)
- 2019 – 7 nm [AMD: Radeon Instinct MI60/Zen2]
- 2021 – 5 nm (Samsung & TSMC, in testing)
- 2023 – 3 nm (Samsung?)

# Transistors bonanza: bringing happiness in lots of corners

- Ability to cram more transistors per unit area and still make a profit good in many respects
  - Microprocessors
  - Memory capacity & speed
  - Storage controllers

# Parallel computing: The Ox vs. Chickens Analogy

“If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?” [Seymour Cray→]

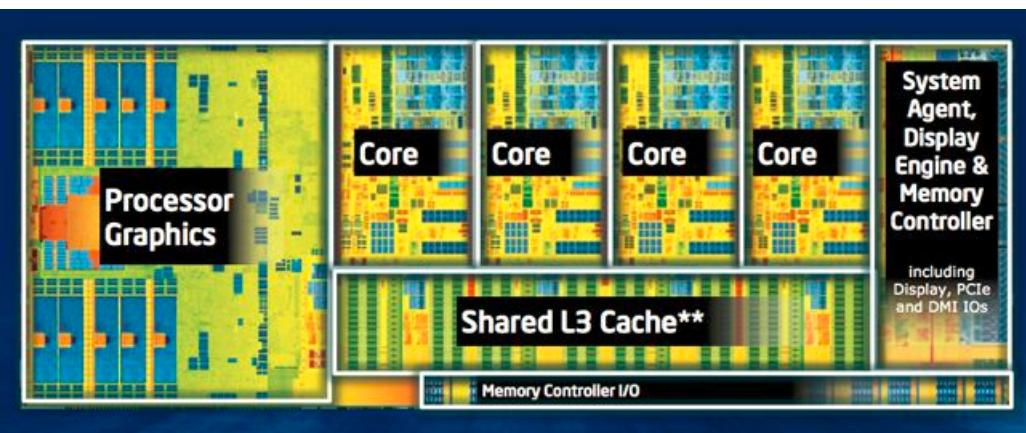
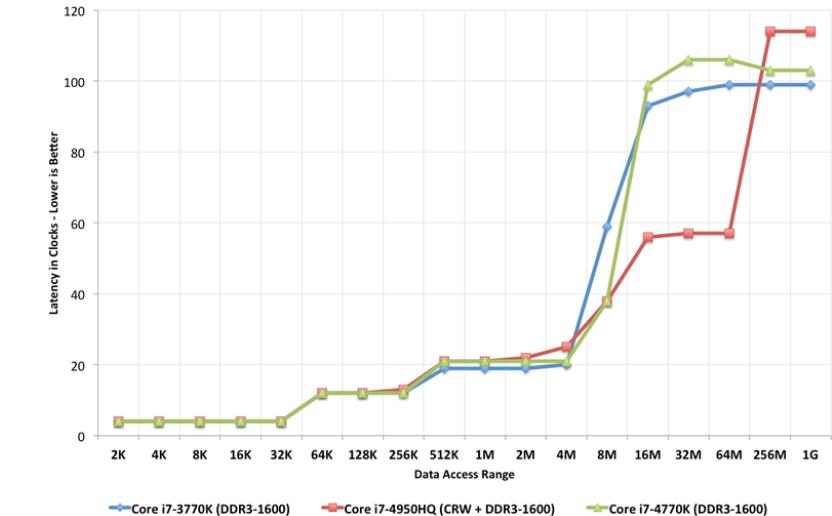
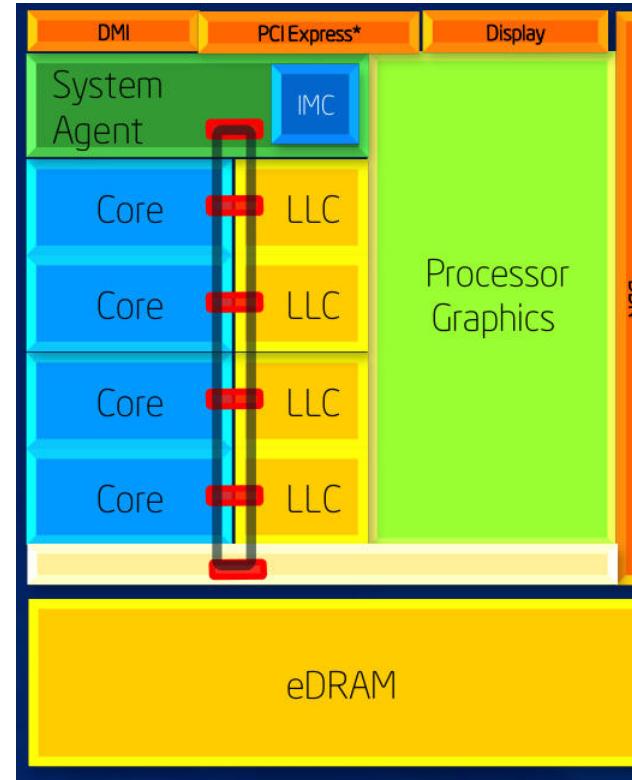
- Ox: one powerful CPU core, running at high frequency, lots of cache to hide memory latency
- Chicken: many meagre processors, running at lower frequency, not much cache or brain (like in CU)
  - Requires other ways to hide memory latency
- Few were eating chicken a decade ago but popular now

# Two examples of Parallel HW: Putting to good use billions of transistors

- Intel Haswell
  - Multicore architecture
  - Four oxen, that is
- NVIDIA Fermi
  - Large number of scalar processors (“shaders”)
  - 512 chicken, that is

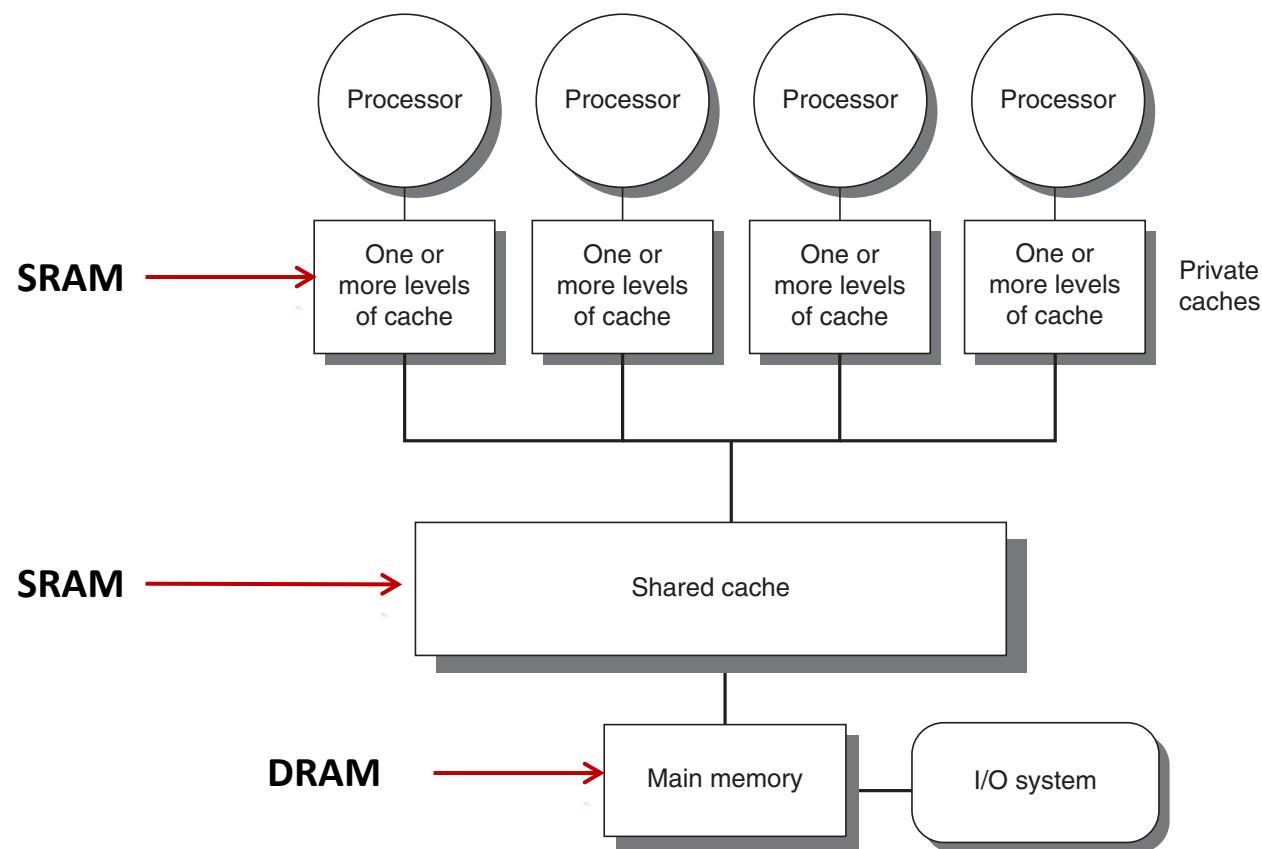
# Intel Haswell

- June 2013
- 22 nm technology
- 1.4 billion transistors
- 4 cores, HTT
- Integrated GPU
- System-on-a-chip (SoC) design



# Haswell example: multi-core architecture (four oxen)

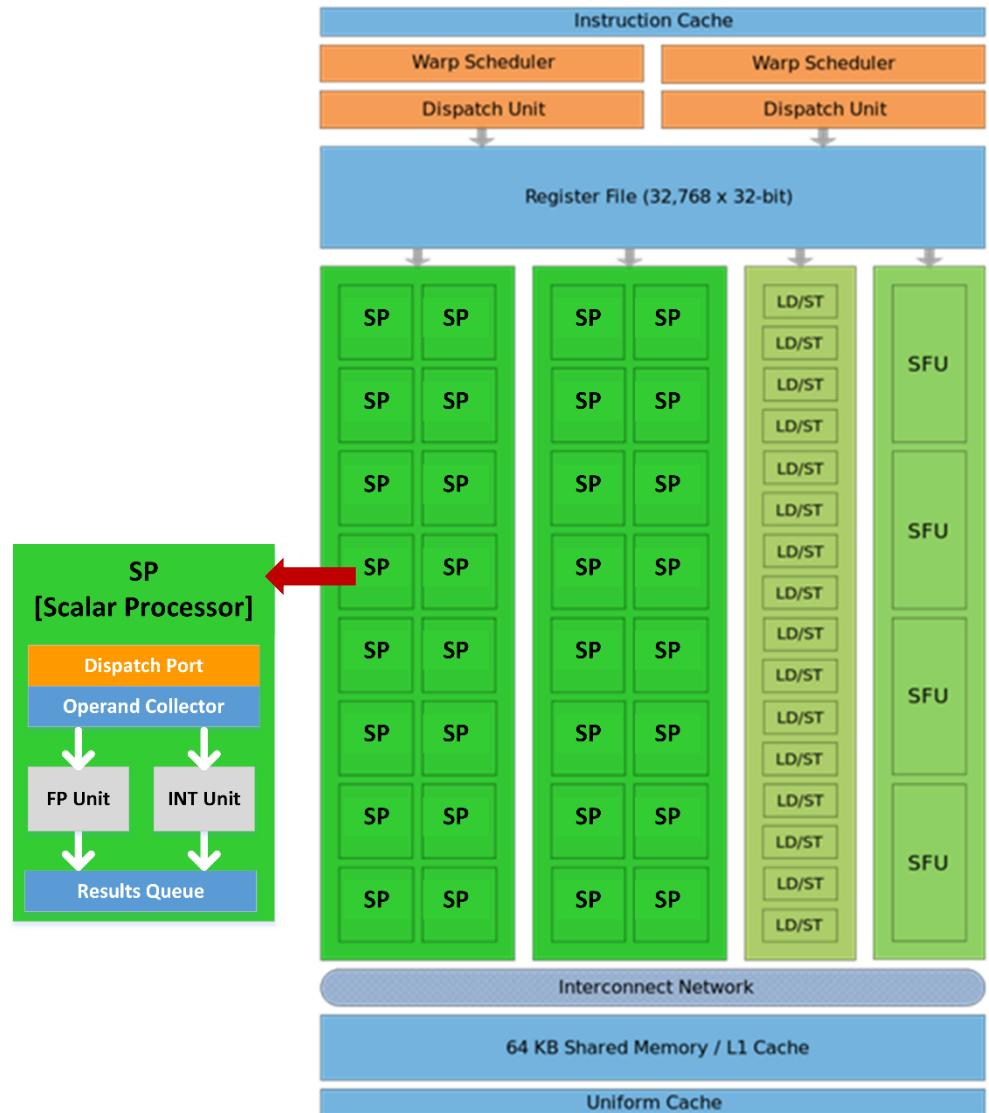
- Shared-Memory Multiprocessor Architecture



# The Fermi Architecture

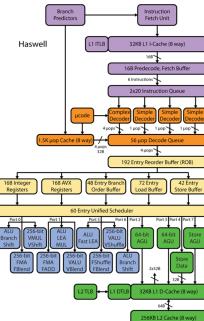
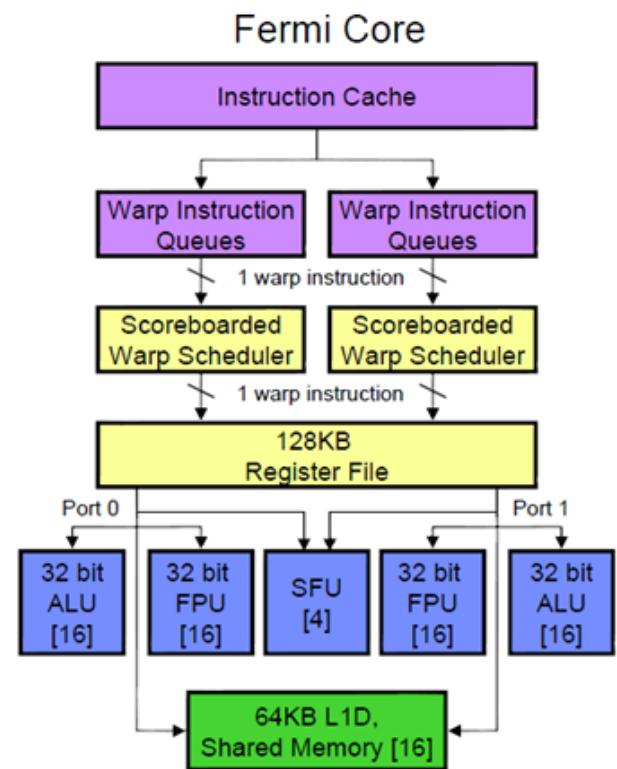
[ Schematic of a NVIDIA Stream Multiprocessor (SM) ]

- Late 2009, early 2010
- 40 nm technology
- Three billion transistors
- 512 Scalar Processors (SPs), or “shaders”
  - This is the [chicken](#)
- L1 cache
- L2 cache
- 6 GB of global memory
- Operates at low clock rate
- High bandwidth (close to 200 GB/s)



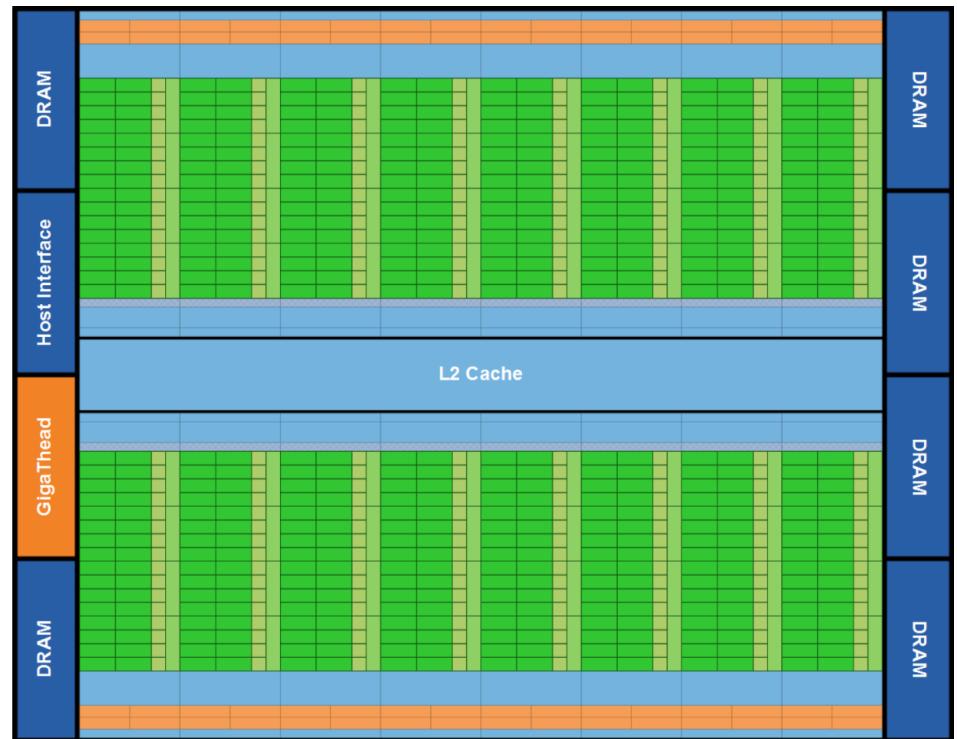
A Fermi SM (Streaming Multiprocessor; one of 16)

Not a whole lot of brain  
here, yet a lot of power



# Fermi: 30,000 feet perspective, the entire chip

- Lots of ALU (green), not much of CU (orange)
- Explains why GPUs are fast for high arithmetic intensity applications
- Arithmetic intensity: high when many operations performed per byte of data



# The CPU and GPU: their true colors

- The GPU is specialized for compute-intensive, highly data parallel computation (owing to its graphics rendering origin)
  - More transistors devoted to **data processing** rather than data caching and control flow
  - Where are GPUs good: **high arithmetic intensity** (the ratio between arithmetic operations and memory operations)



- The video game industry exerts strong economic pressure that pushes things forward in GPU computing

# Beefy CPU vs. beefy GPU comparison, 2019

	<b>GPU – NVIDIA Tesla V100 32GB \$8,995</b>	<b>CPU – Intel® Xeon® Platinum 8180 \$10,009</b>
Processing Cores	5120 CUDA Cores 640 Tensor Cores	28 cores (56 threads, SMT)
Memory	32GB HBM2	- 64 KB L1 (I+D) cache / core - 1 MB L2 cache / core - 38.5 MB L3 cache (1.375 MB / core), average
Clock speed	1.29 GHz	2.5 GHz (3.80 GHz Turbo; Throttles to 2.3 GHz when all FP units are active)
Memory bandwidth	900 GB/s	119.21 GB/s (19.87 GB / channel / s)
Floating Point Throughput	7.8 Double Precision TFLOP/s 15.6 Single Precision TFLOP/s	2.0608 Double Precision TFLOP/s
TDP	300 Watts	205 Watts

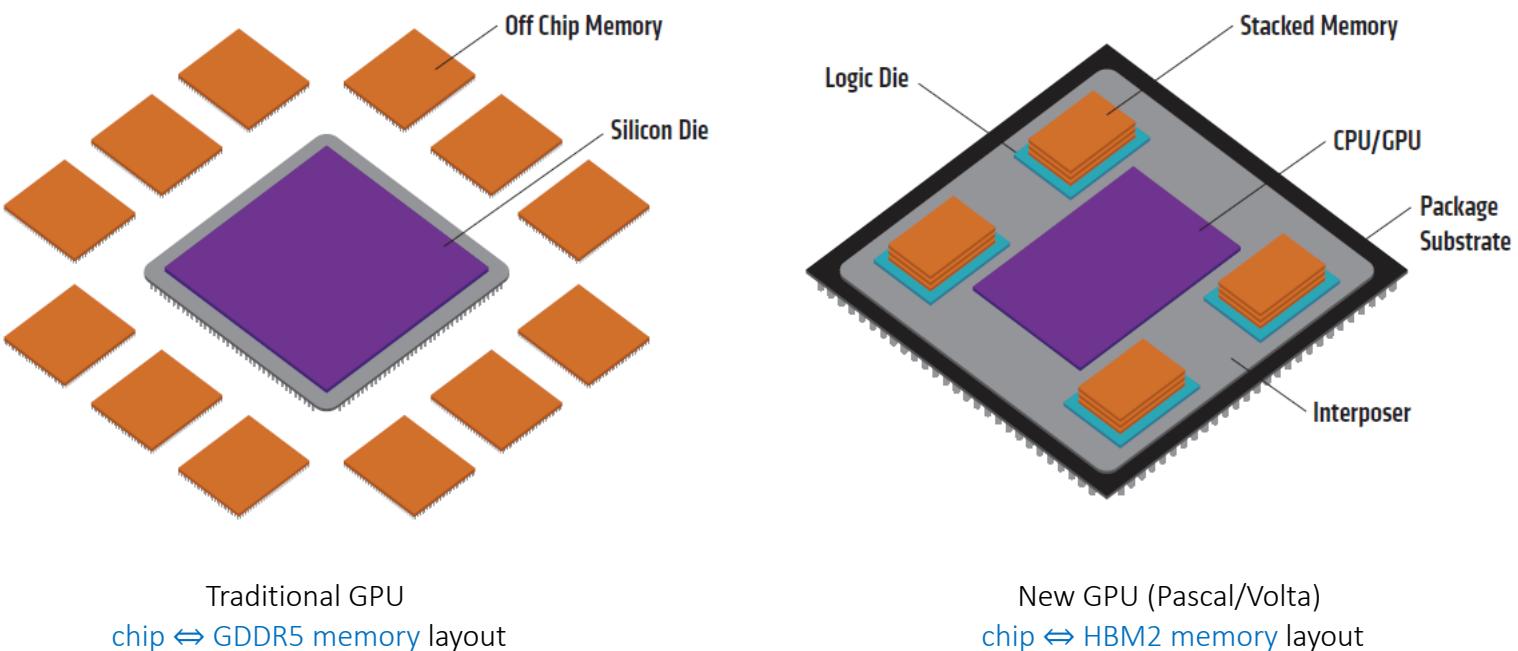
[https://en.wikipedia.org/wiki/Nvidia\\_Tesla](https://en.wikipedia.org/wiki/Nvidia_Tesla)

<https://www.techpowerup.com/gpu-specs/tesla-v100-sxm2-32-gb.c3185>

[https://en.wikichip.org/wiki/intel/xeon\\_platinum/8180](https://en.wikichip.org/wiki/intel/xeon_platinum/8180)

<https://ark.intel.com/products/120496/Intel-Xeon-Platinum-8180-Processor-38-5M-Cache-2-50-GHz->

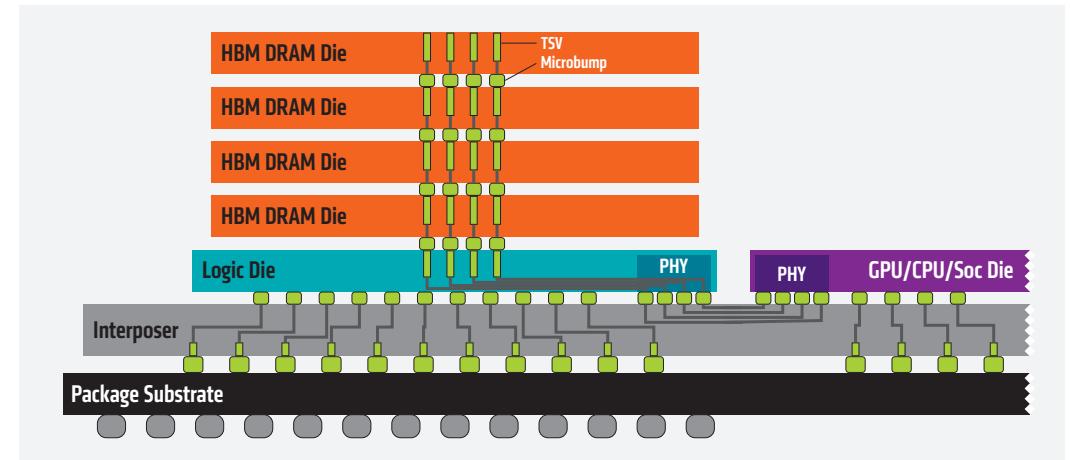
# 3D Stacked Memory



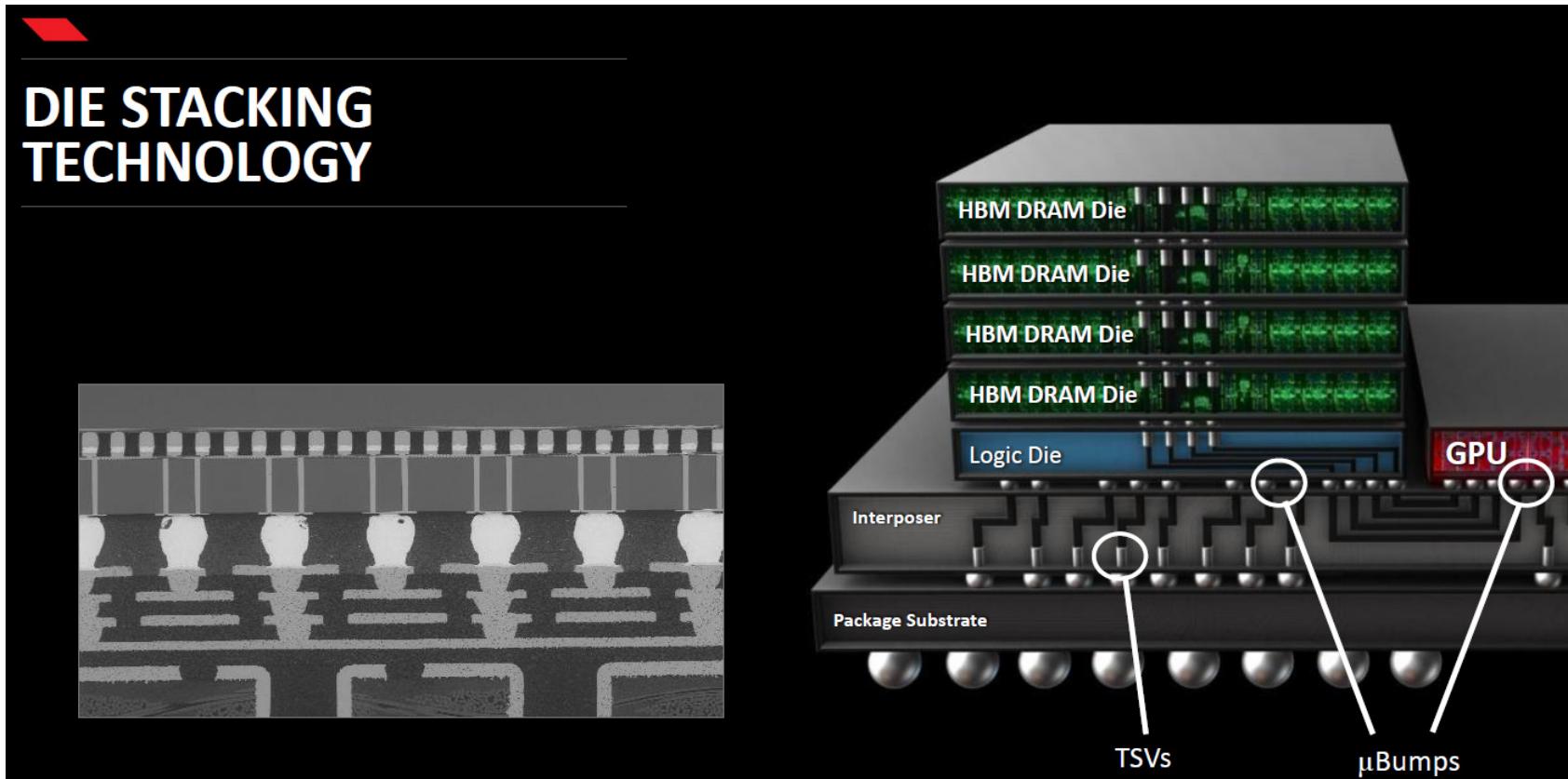
Major step forward in performance.

# 3D Stacked Memory

- SK Hynix's High Bandwidth Memory (HBM)
  - Developed by AMD and SK Hynix
- 1st Generation (HBM1) introduced in AMD Fiji GPUs
  - 1GB & 128GB/s per stack
  - AMD Radeon R9 Fury X: had four stacks → 4GB & 512GB/s
- 2nd Generation (HBM2) used in NVIDIA Pascal/Volta and AMD Arctic Island GPUs
  - 2 GB & 256GB/s bandwidth per stack
  - NVIDIA Volta approximately 900 GB/s memory bandwidth
  - Apparently AMD has 1024 GB/s HBM available

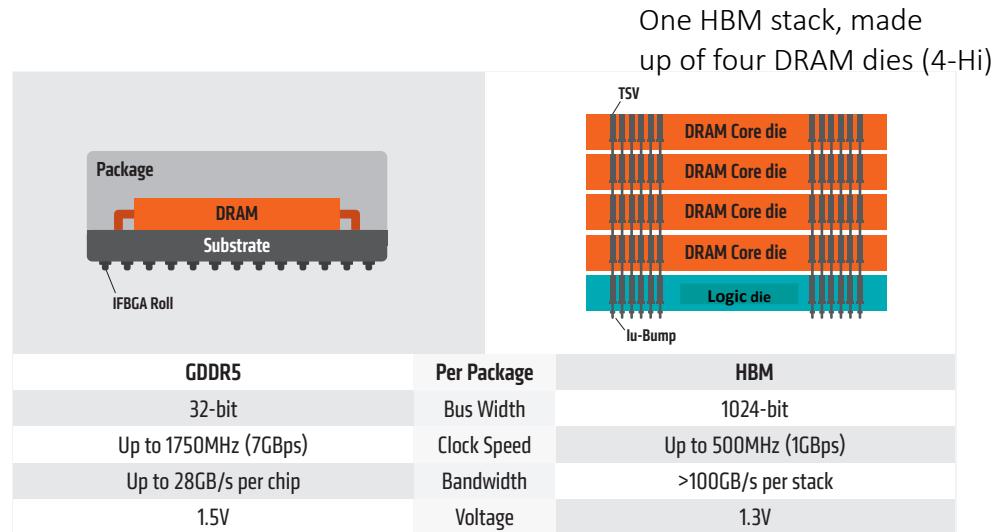
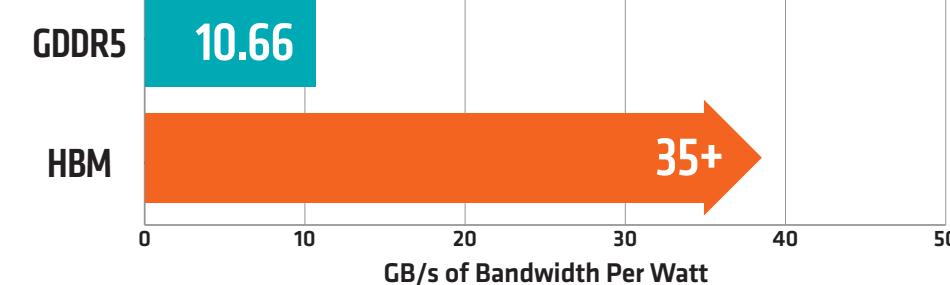
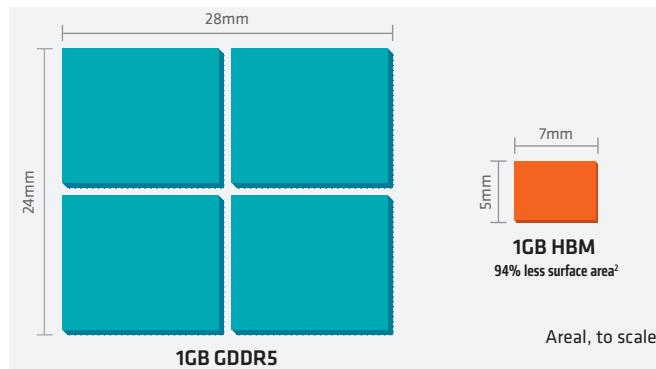


# 3D Stacked Memory



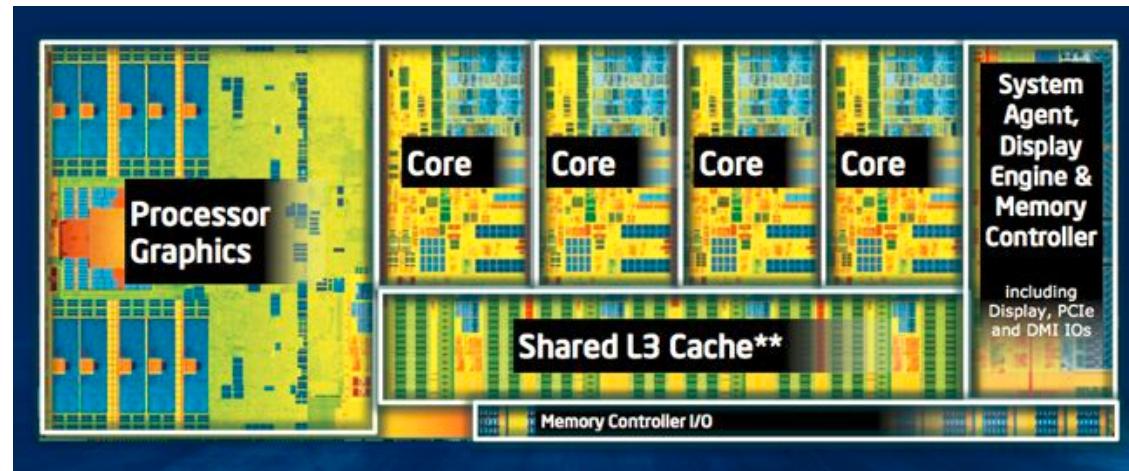
# 3D Stacked Memory

- Shorter distances
  - Smaller latency → nice!
  - Less power required to move data
- Smaller memory footprint
  - 16X smaller footprint

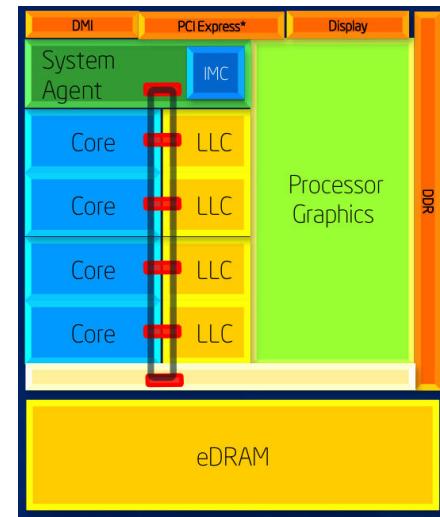


## on-chip vs. on-package vs. off-package

- Where a resource is dictates to a large extent its **communication** latency and/or bandwidth
- Example, **on-chip**: Intel Haswell has the GPU on the same chip with the rest of the CPU
- Example, **on-package**: the 128 MB of eDRAM available on Intel Core i7 4950HQ



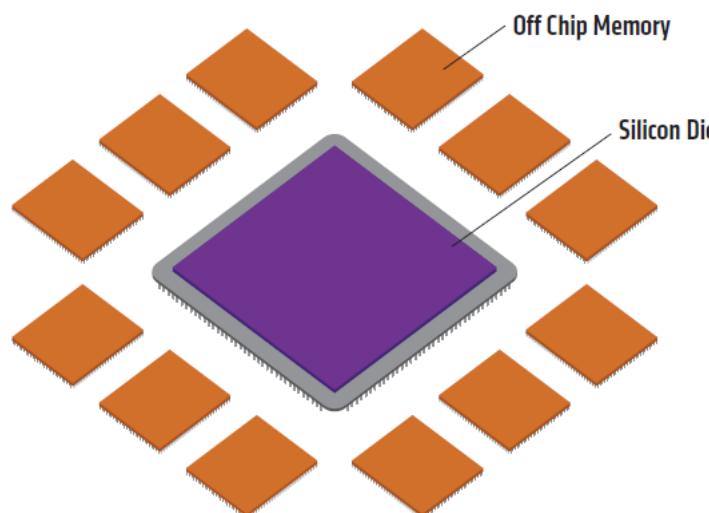
The GPU is on-chip



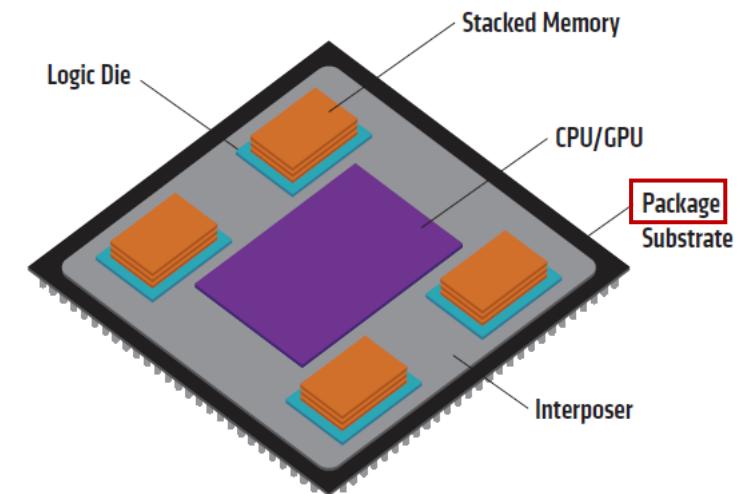
The eDRAM is on-package

## on-chip vs. on-package vs. off-package

- Example, off-package vs. on-package: traditional memory vs. HBM



Off-package Example  
chip  $\Leftrightarrow$  GDDR6 memory

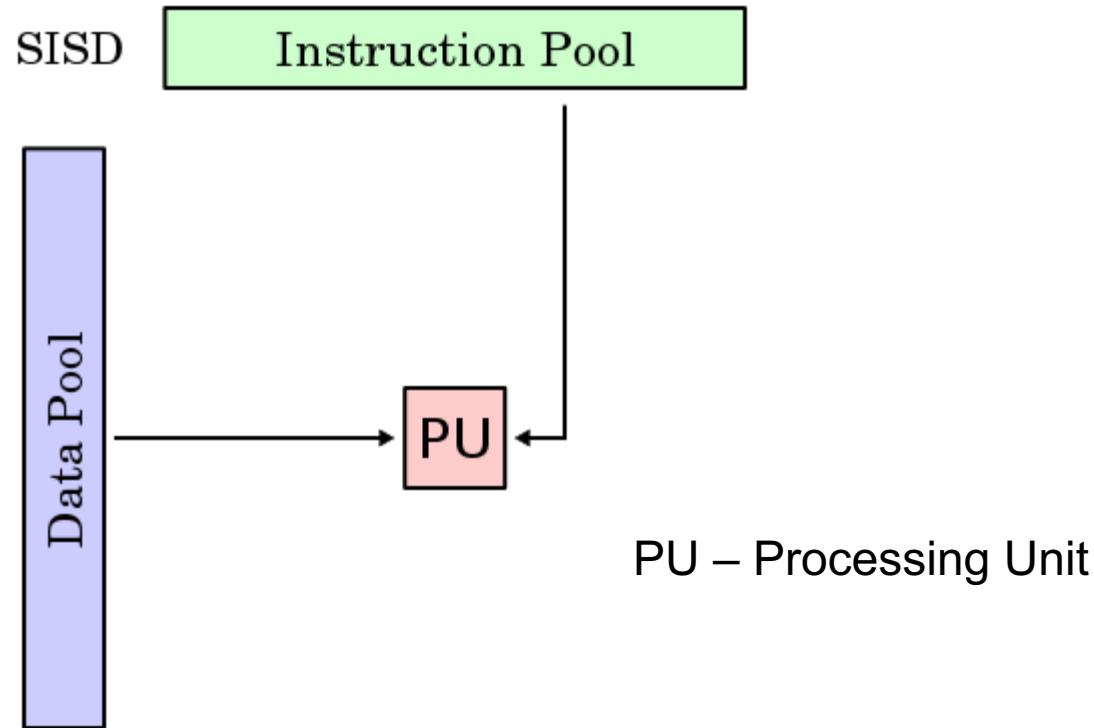


On-package Example  
chip  $\Leftrightarrow$  HBM2 memory

# [New Topic] Flynn's Taxonomy of Architectures

- There are several ways to classify architectures
- Below, classification based on how instructions are executed in relation to data
  - SISD - Single Instruction/Single Data
  - SIMD - Single Instruction/Multiple Data
  - MISD - Multiple Instruction/Single Data
  - MIMD - Multiple Instruction/Multiple Data

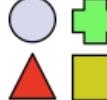
# Single Instruction/Single Data (SISD) Architectures



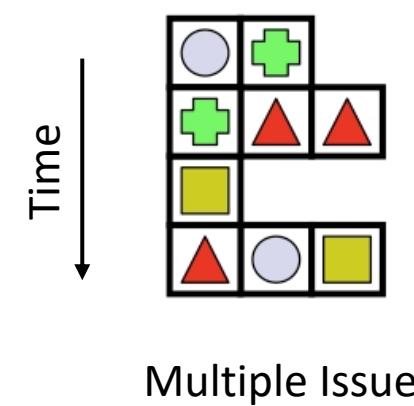
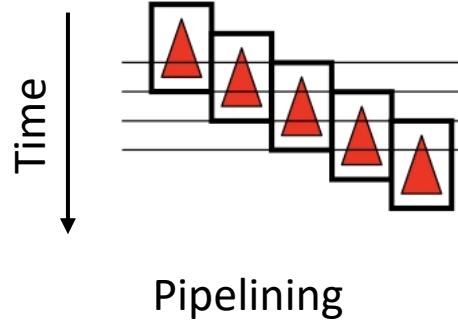
Your desktop, before the spread of dual core CPUs

# Increasing Throughput of SISD

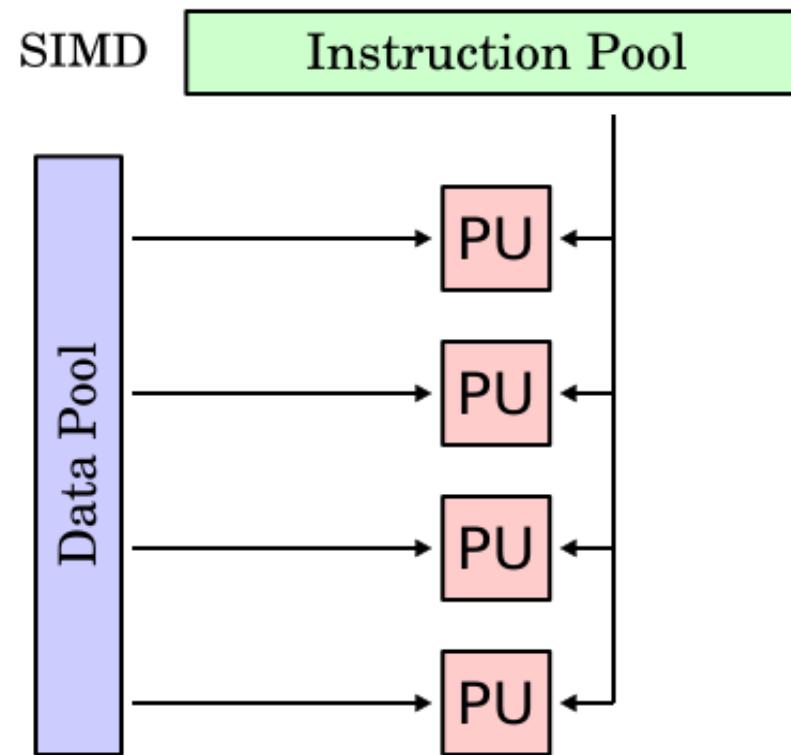
**Instructions:**



Circle (Purple)   Plus (Green)  
Triangle (Red)   Square (Yellow)



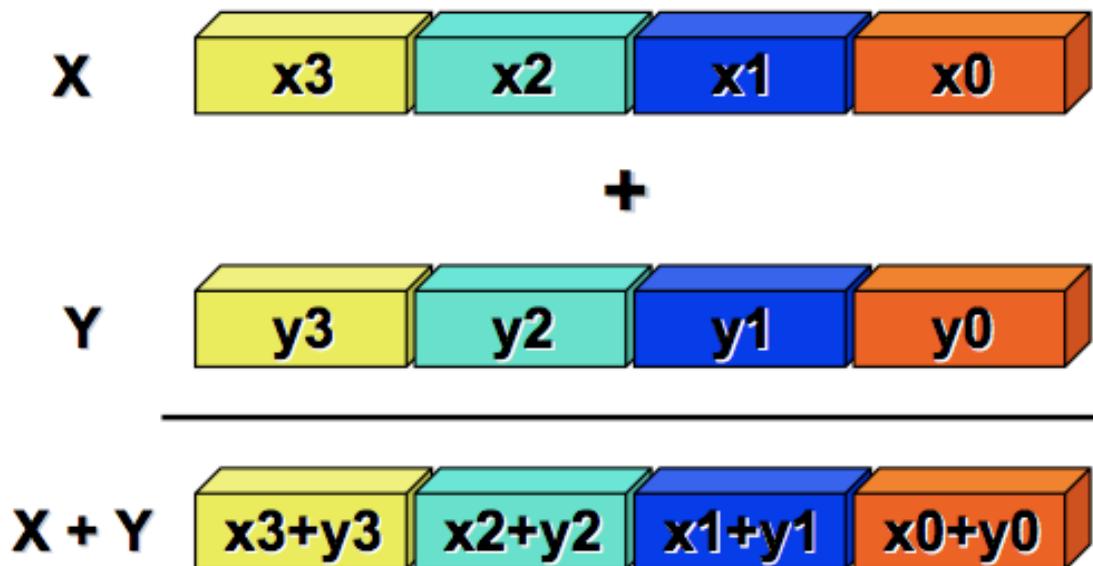
# Single Instruction/Multiple Data (SIMD) Architectures



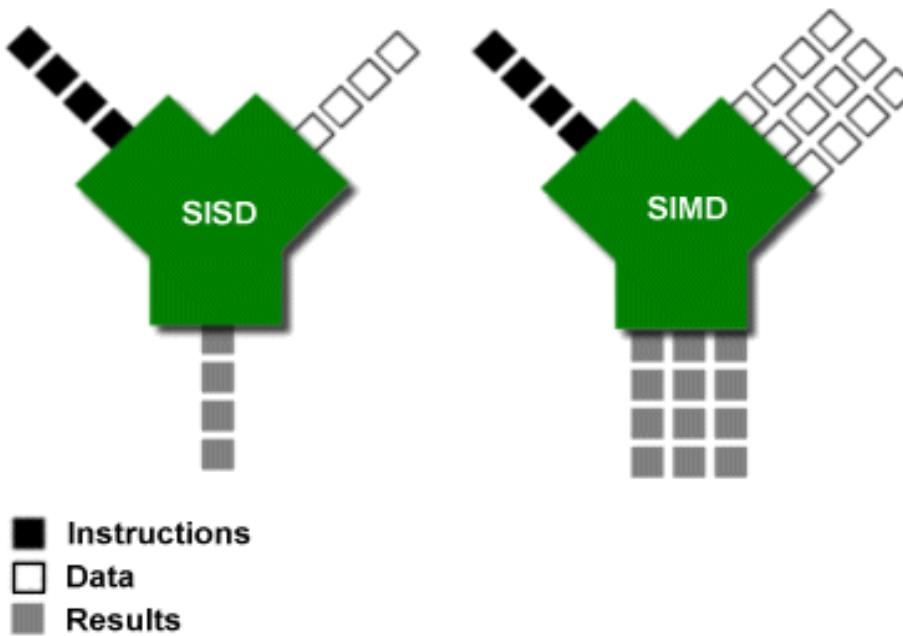
Processors that execute same instruction on multiple pieces of data: GPUs

# Single Instruction/Multiple Data (SIMD)

- Each core runs the same set of instructions on different data
- Examples:
  - Graphics Processing Unit (GPU): processes pixels of an image in parallel
  - CRAY's vector processor



# SISD versus SIMD



Writing a compiler for SIMD architectures is difficult  
(inter-thread communication complicates the picture...)

# SISD, SIMD, and Vector Length

5	+	-1	=	4
6	+	3	=	9
2	+	-2	=	0
-1	+	-1	=	-2
0	+	7	=	7
-2	+	-4	=	-6
9	+	-2	=	7
-4	+	-4	=	-8

A stream of 8 “SISD” instructions



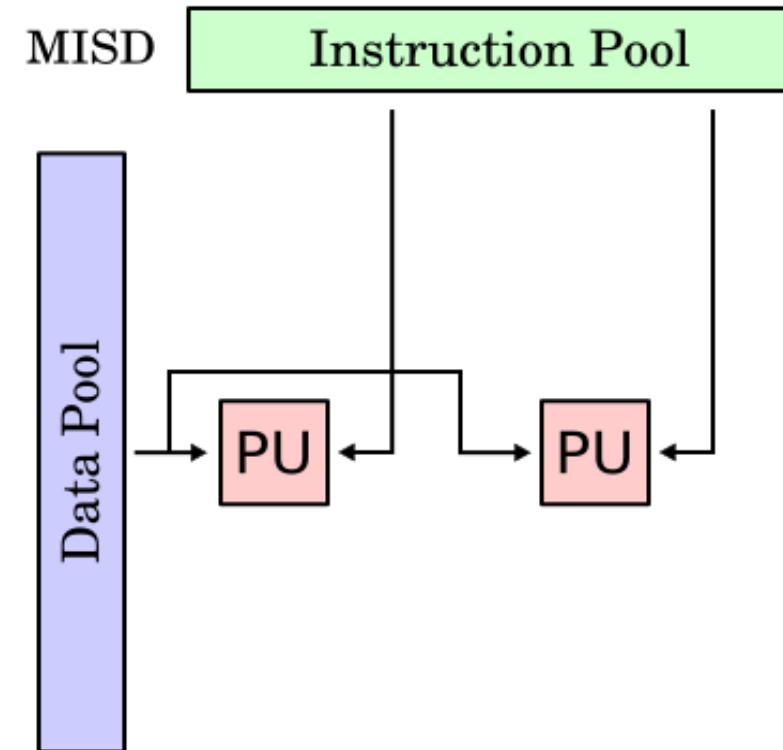
5	-1	4
6	3	9
2	-2	0
-1	-1	-2
0	7	7
-2	-4	-6
9	-2	7
-4	-4	-8

One SIMD instruction

↑  
Vector Length  
↓

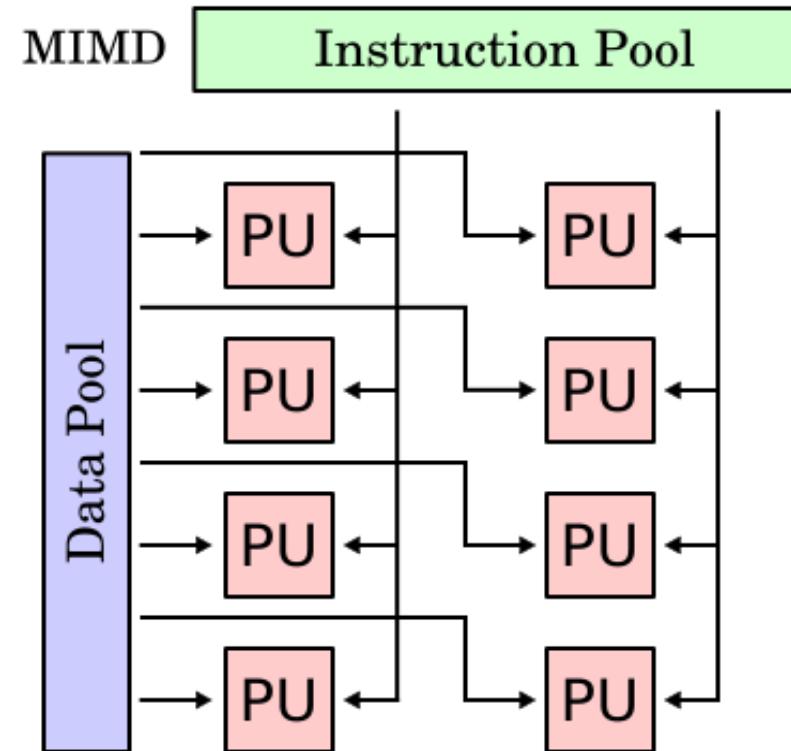
- SIMD: the concept of **vector length**
  - Assume numbers shown above are integers
  - 8 ints  $\times$  4 Bytes each  $\times$  8 Bits/Byte = 256 Bit Vector Length
  - Intel KNL Vector Length: 512 (more later)

# Multiple Instruction/Single Data



Not useful, not aware of any commercial implementation...

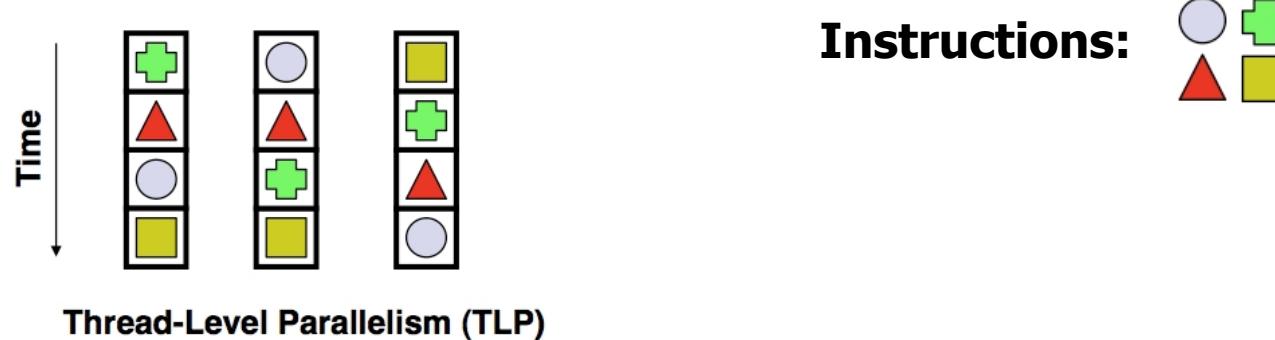
# Multiple Instruction/Multiple Data



Almost all our desktop/laptop chips are MIMD systems

# Multiple Instruction/Multiple Data

- The sky is the limit: each PU is free to do as it pleases
- Can be of either shared memory or distributed memory categories



# [New Topic] Amdahl's Law

"A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude"

"Validity of the single processor approach to achieving large scale computing capabilities," by Gene M. Amdahl, in Proceedings of the "AFIPS Spring Joint Computer Conference," pp. 483, 1967

- Let  $r_s$  capture the amount of time that a program spends in components that can only be run sequentially
- Let  $r_p$  capture the amount of time spent in those parts of the code that can be parallelized.
- Assume that  $r_s$  and  $r_p$  are normalized, so that  $r_s + r_p = 1$
- Let  $n$  be the number of threads used to parallelize the part of the program that can be executed in parallel
- The "best case scenario" speedup  $S$  is

$$S = \frac{T_{old}}{T_{new}} = \frac{r_s + r_p}{r_s + \frac{r_p}{n}} = \frac{1}{r_s + \frac{r_p}{n}}$$

# Amdahl's Law

- Sometimes called the **law of diminishing returns**
- In the context of parallel computing used to illustrate how going parallel with a part of your code is going to lead to overall speedups
- The art is to find for the same problem an algorithm that has a large  $r_p$ 
  - Sometimes requires a completely **different** angle of approach for a solution
- Nomenclature
  - Algorithms for which  $r_p = 1$  are called “embarrassingly parallel”

# Example: Amdahl's Law

- Suppose that a program spends 60% of its time in I/O operations, pre and post-processing
- The rest of 40% is spent on computation, most of which can be parallelized
- Assume that you buy a multicore chip and can throw 6 parallel threads at this problem.
- What is the maximum amount of speedup that you can expect given this investment?
- Asymptotically, what is the maximum speedup that you can ever hope for?

# Trivia

- Gene Amdahl (1922-2015)
- Graduated from UW-Madison (1952)
  - PhD, Mathematics and Physics
- Amdahl's law (1967)

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 09

02/10/2020

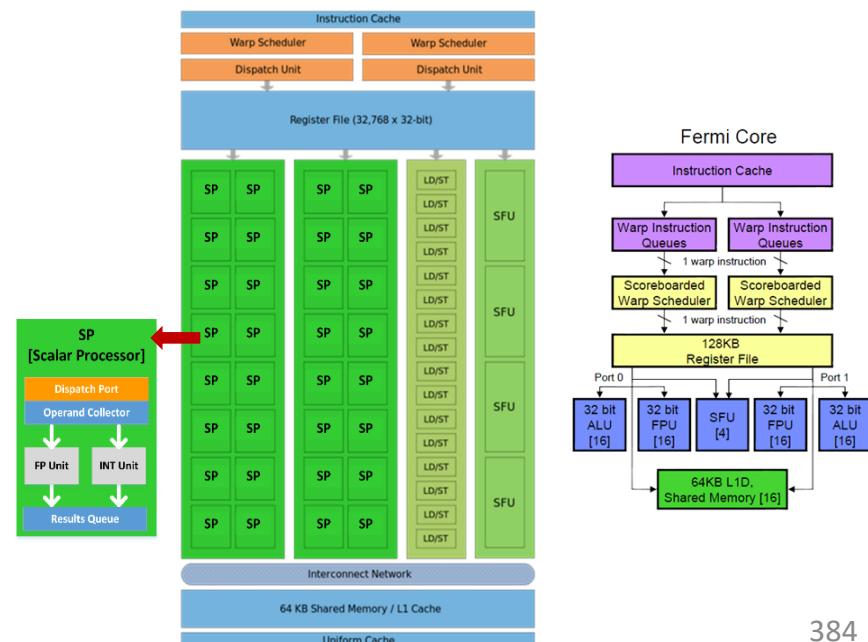
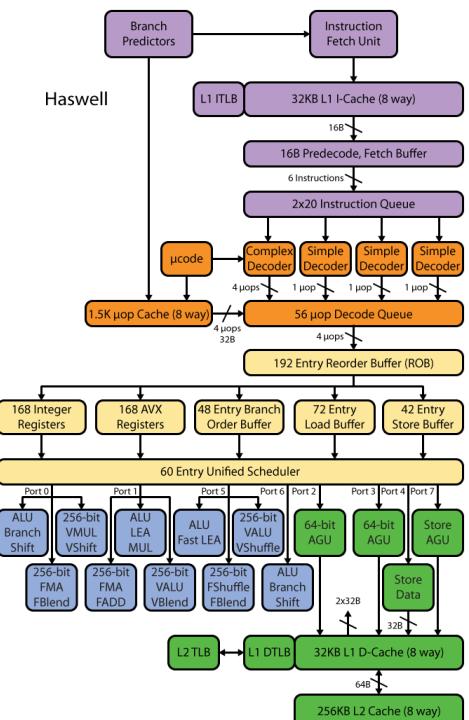
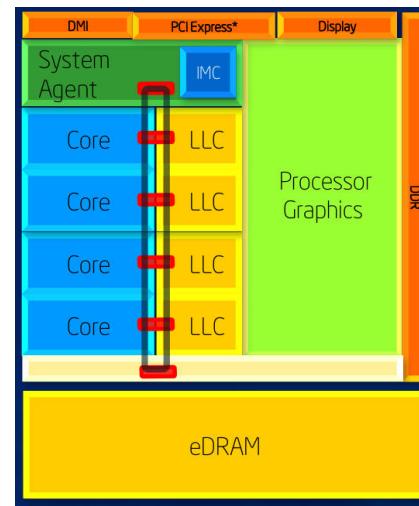
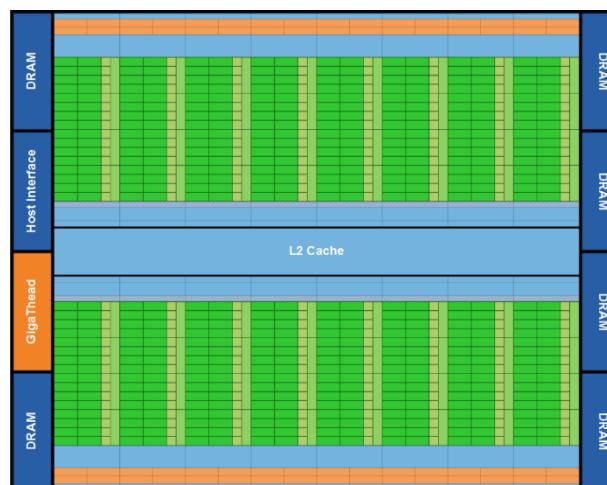
# Quote of the day

“Keep away from people who try to belittle your ambitions. Small people always do that, but the really great make you feel that you, too, can become great.”

-- Mark Twain, American writer, humorist, entrepreneur, publisher, and lecturer [1835 - 1910].

# Before we get going...

- Last time:
  - Sequential computing gains – little hope left owing to:
    - Memory wall
    - Power wall
    - ILP wall
  - Improving the speed of your computation will require parallel computing
    - The “two oxen vs. 512 chicken” analogy
- Today
  - The basics of GPU computing
- Other tidbits:
  - Assigned reading (syllabus gets updated)
  - Assignment due on Th at 9 pm
  - No class on Friday – Dan out of town

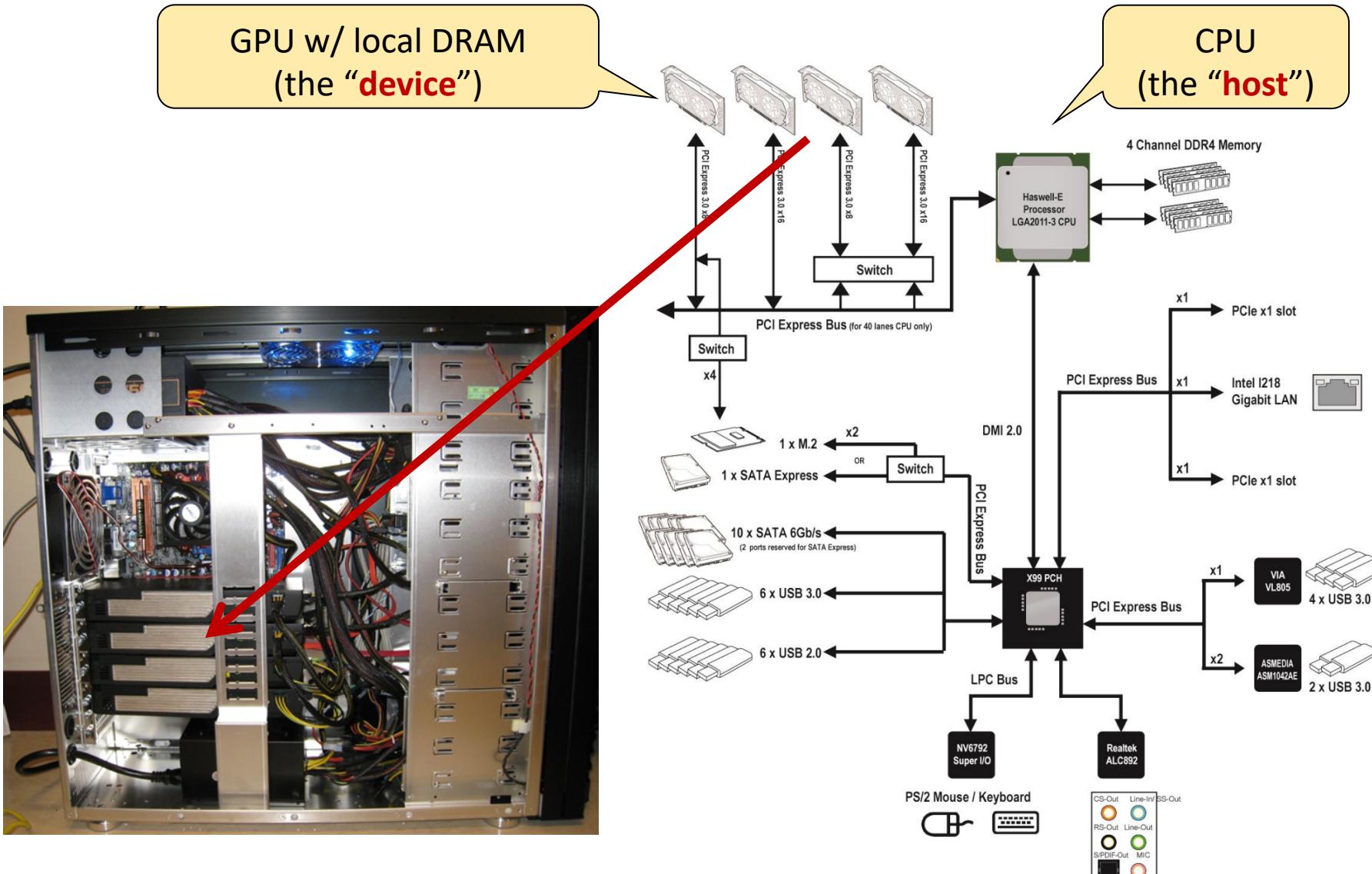


# GPU Computing with CUDA

# The long view

- Goal: Understand how GPUs can be used in Scientific Computing
- Topics
  - Hardware considerations
  - “Hello World” with GPU computing
  - Execution configuration issues
  - The memory model in CUDA
  - Scheduling issues in CUDA
  - Occupancy and optimization issues

# Typical Hardware Architecture

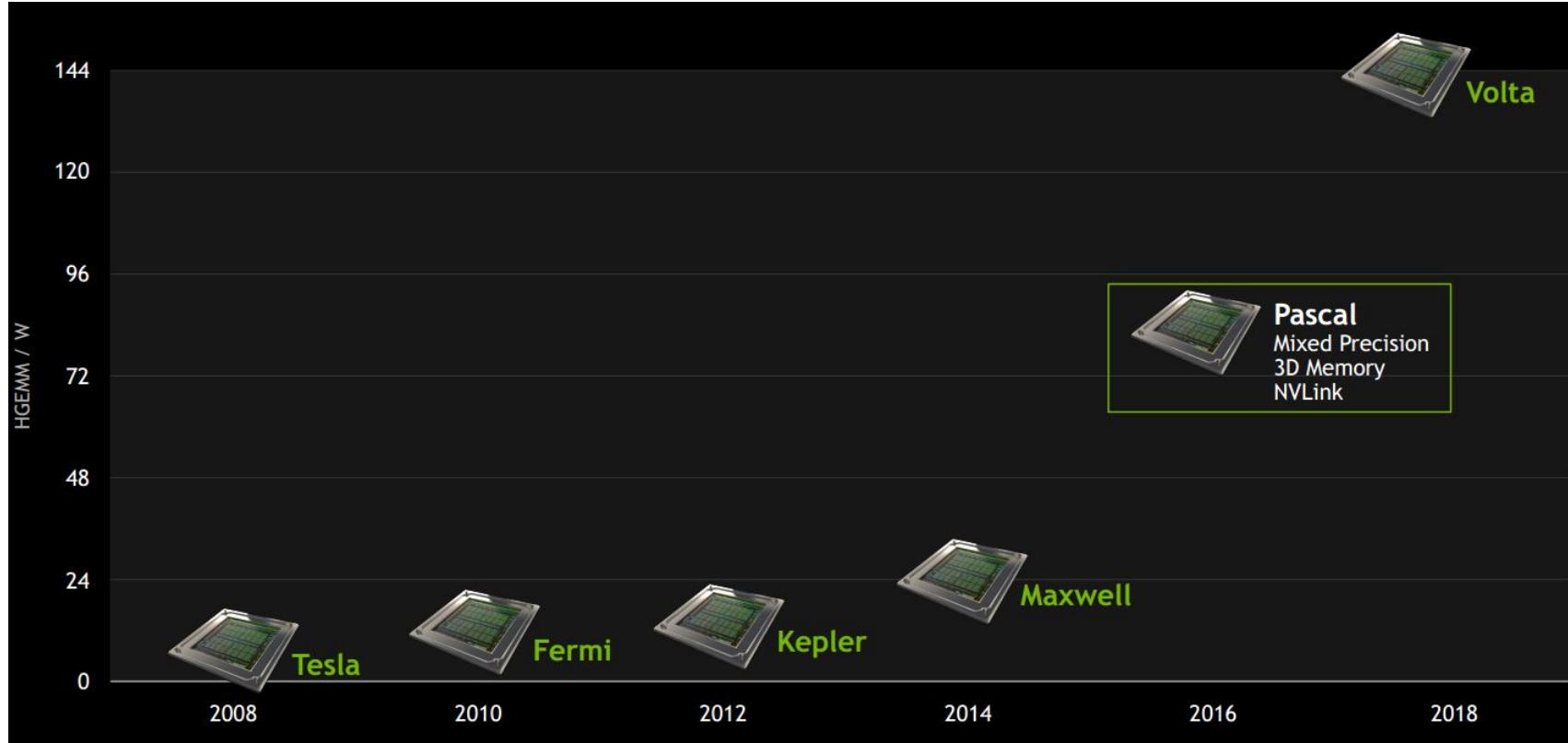


[<http://www.anandtech.com>]→

# Euler Node, 2015 Vintage

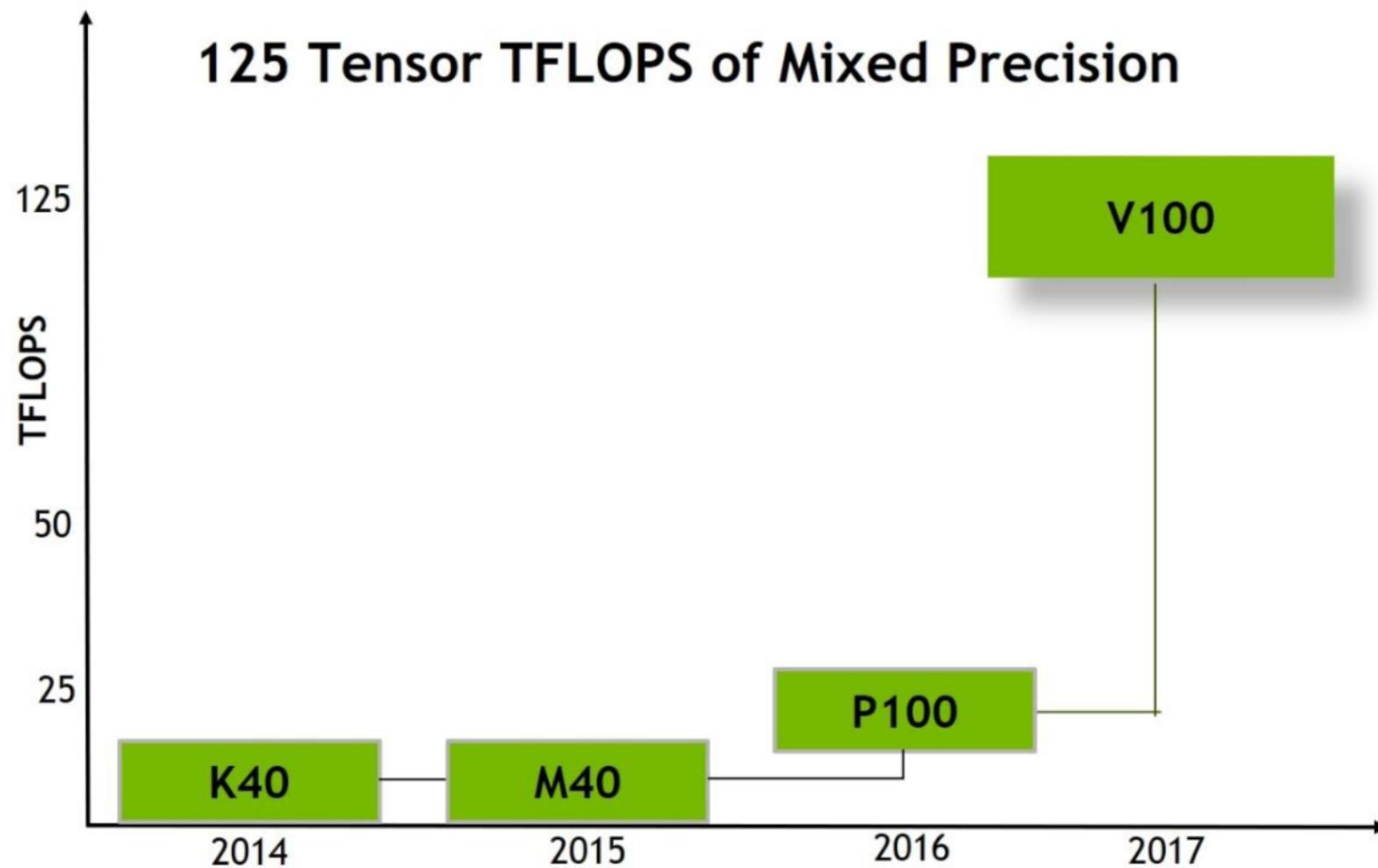


# NVIDIA GPU Roadmap



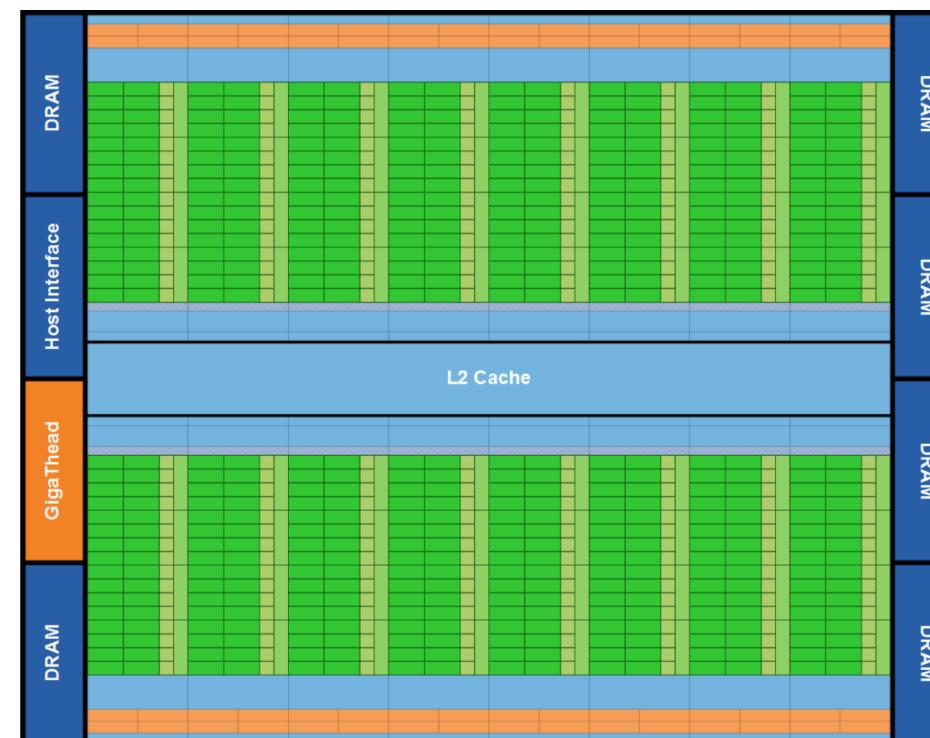
- Latest architecture for scientific computing: Volta

# GPU computing: eager to accommodate Machine Learning needs



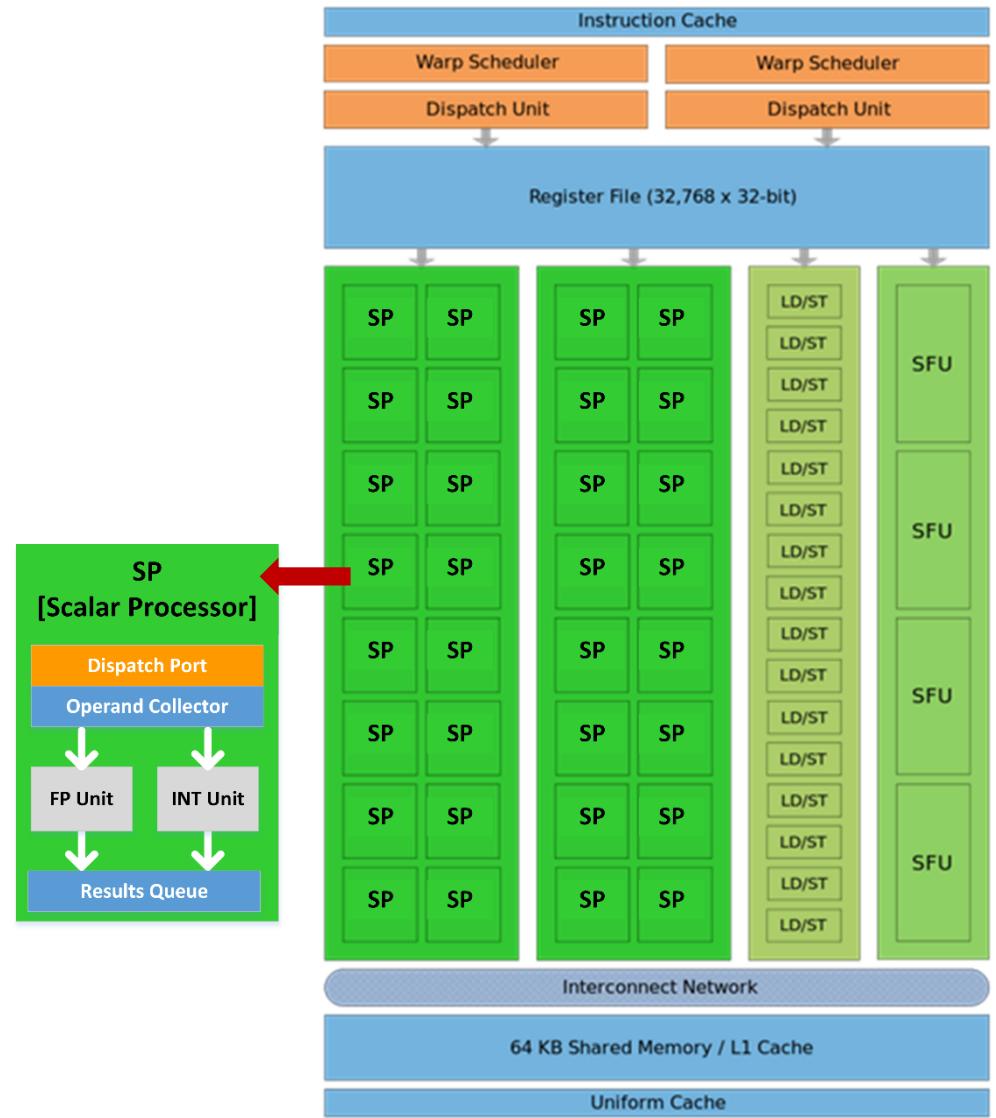
# Fermi: High-level Perspective

- Lots of ALU (green), not much of CU (orange)
- Explains why GPUs are fast for high arithmetic intensity applications
- Arithmetic intensity: high when many operations performed per word of memory access



# The Fermi Architecture

- Late 2009, early 2010
- 40 nm technology
- Three billion transistors
- 512 Scalar Processors (SP), or “shaders”
- L1 cache (64 KB)
- L2 cache (768 KB uniform, shared)
- 6 GB of global memory
- Operates at low clock rate
- Memory bandwidth  $\approx 200\text{GB/s}$



# NVIDIA Volta GV100: microarchitecture schematic and nomenclature

- 21.1 billion transistors
- GPC – 6 of them
  - GPU Processing Cluster
- TPC – 42 of them
  - Texture Processing Cluster
- SM – 84 of them
  - Stream multiprocessor
- A full GV100 has
  - Six GPCs, each of which has
    - Seven TPC, each with
      - Two SMs
- $6 \times 7 \times 2 = 84$



# NVIDIA Volta GV100: microarchitecture schematic and nomenclature

- Each SM has
  - 64 FP32 scalar processors (SPs)
  - 64 INT32 scalar processors (SPs)
  - 32 FP64 scalar processors (SPs)
  - 8 Tensor scalar processors (SPs)
  - Four texture units
  - It's own L0 Instruction Cache
  - A wrap scheduler
  - A Dispatch unit
  - 16,384 registers



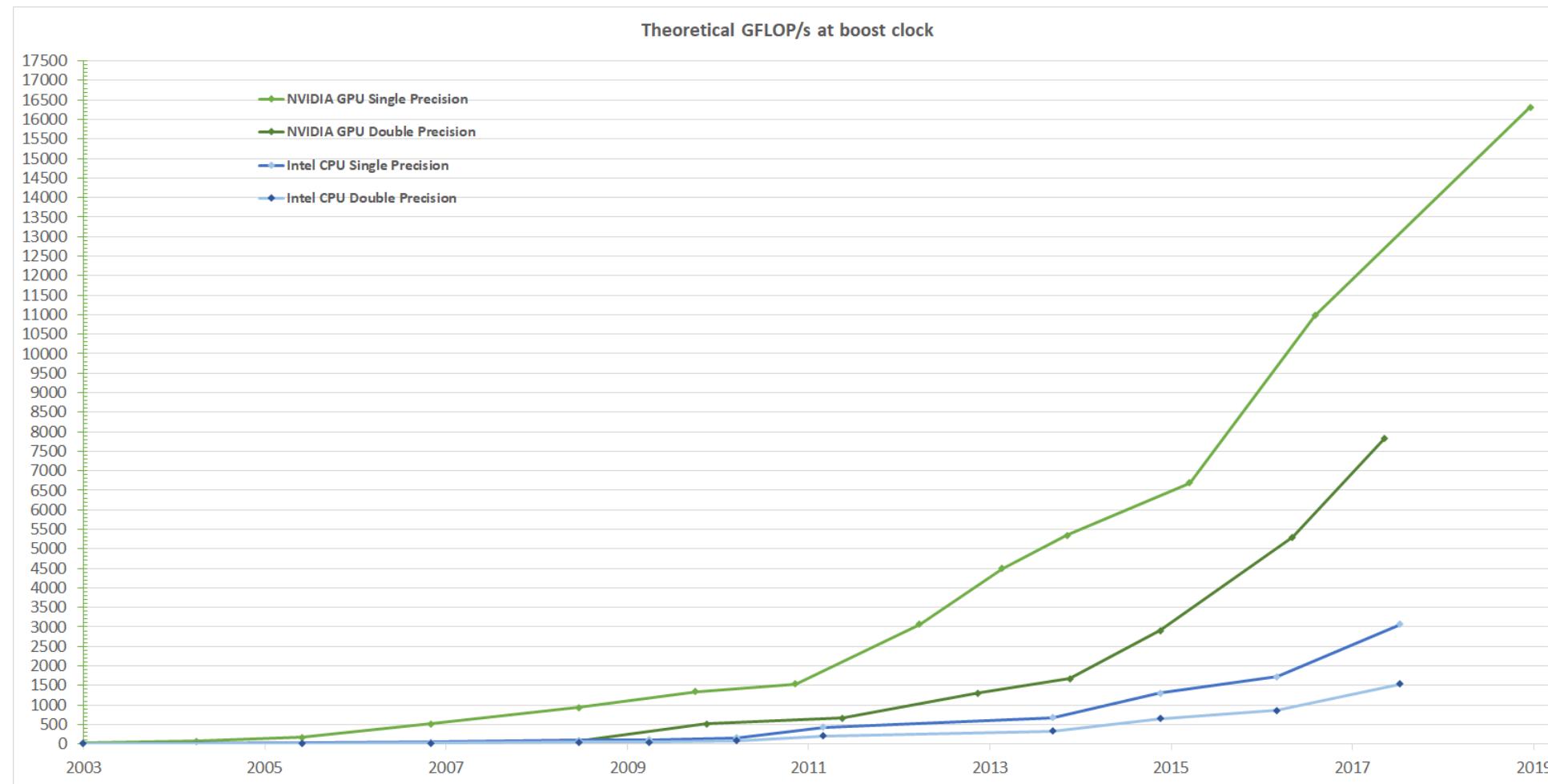
# Performance overview, last four NVIDIA GPU generations

- This is what more transistors buys you →

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS <sup>1</sup>	5	6.8	10.6	15.7
Peak FP64 TFLOPS <sup>1</sup>	1.7	.21	5.3	7.8
Peak Tensor TFLOPS <sup>1</sup>	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm <sup>2</sup>	601 mm <sup>2</sup>	610 mm <sup>2</sup>	815 mm <sup>2</sup>
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

<sup>1</sup> Peak TFLOPS rates are based on GPU Boost Clock

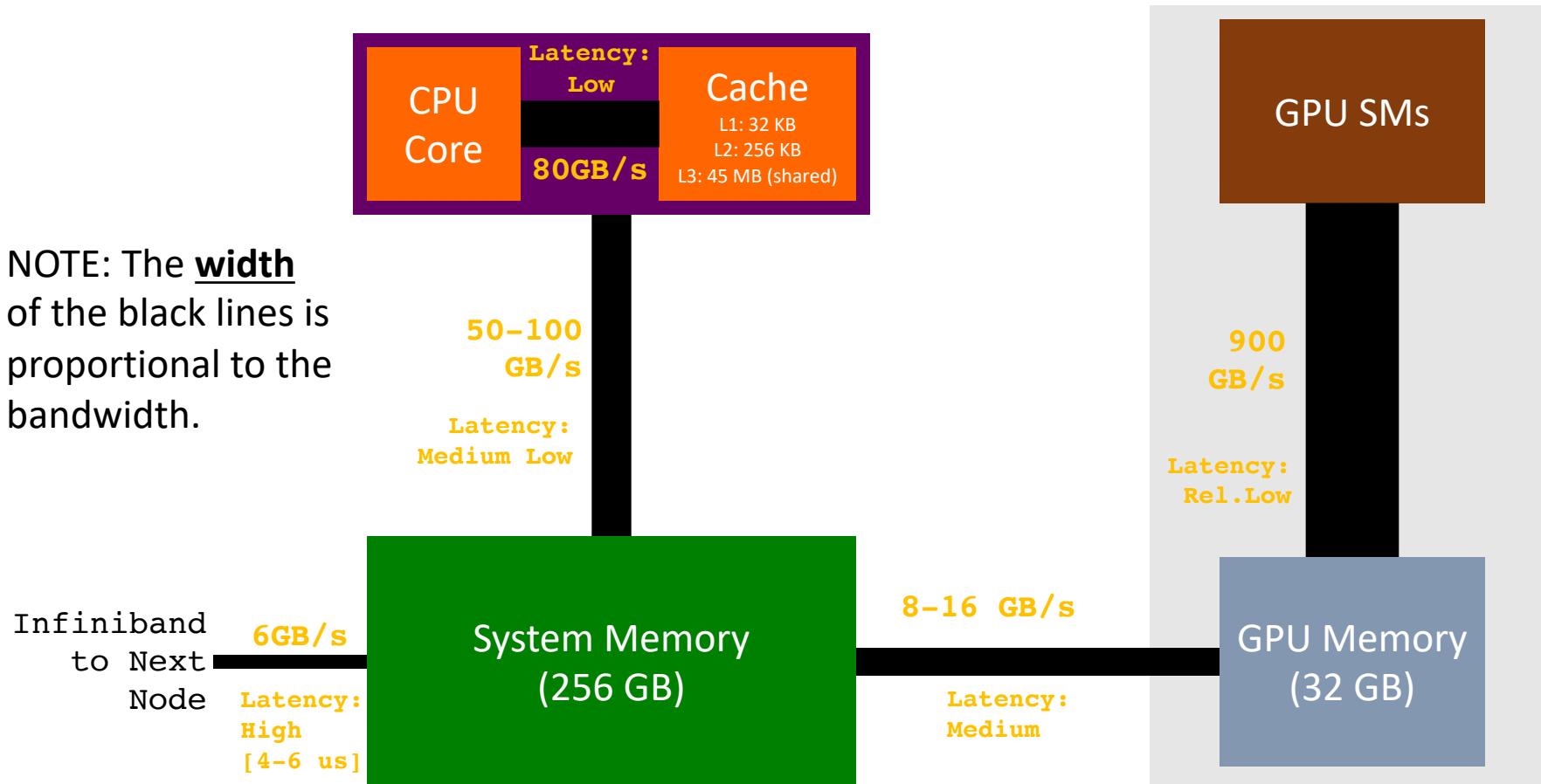
# Theoretical GFLOP/s



# High Speed Requires High Bandwidth

- Assume that you want to add two arrays and reach 1 Tflops: 1E12 ops/second
- You need to fetch/feed 2E12 operands per second...
- Assume each number is stored using 4 bytes (float or integer)
- You need to fetch  $2\text{E}12 * 4$  bytes in a second. This is 8E12 B/s, which is 8 TB/s...
- You haven't taken into account that you probably want to send back the outcome of the operation that you carry out
- The global memory bandwidth on the GPU is in the neighborhood of 1 TB/s ([1000 GB/s](#))
  - About 8 times less than what you need
    - Conclusion: you need data in cache, global memory can't sustain 1 Tflops

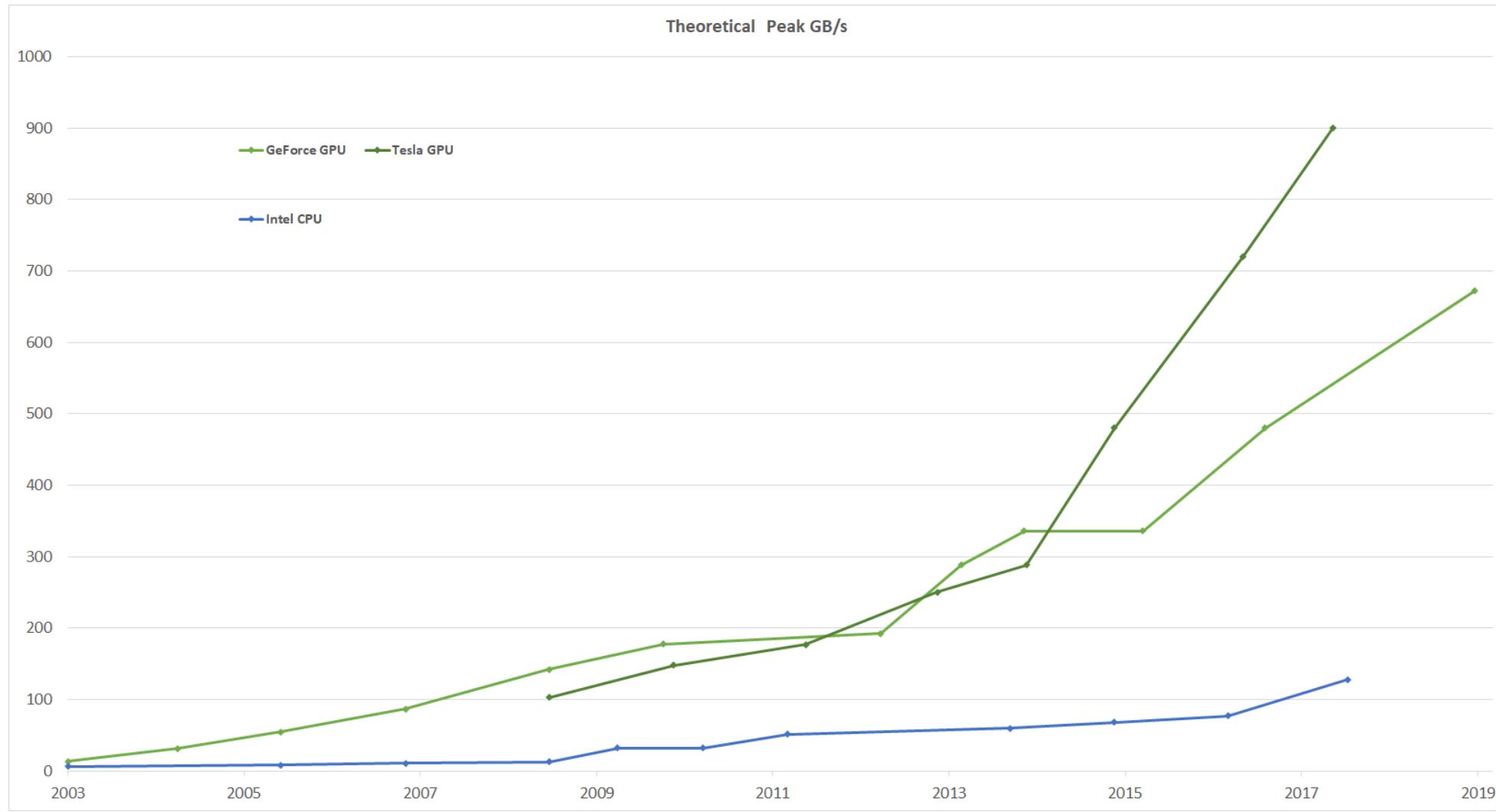
# Bandwidth in a CPU-GPU System



# Info on Previous Slide

- GPU numbers are for an NVIDIA Pascal V100
- PCI numbers are for PCIe3
- Infiniband is assuming Mellanox results from [here](#)
- CPU memory [numbers](#) are for a E7 8890v3, 2.5GHz, 45MB L3
- CPU cache numbers: an average between expected L1 and L3 speed
  - Not exact, but approximate

# Bandwidths, GPU computing w/ NVIDIA



# Speed-ups, GPU Computing: What's reasonable to expect?

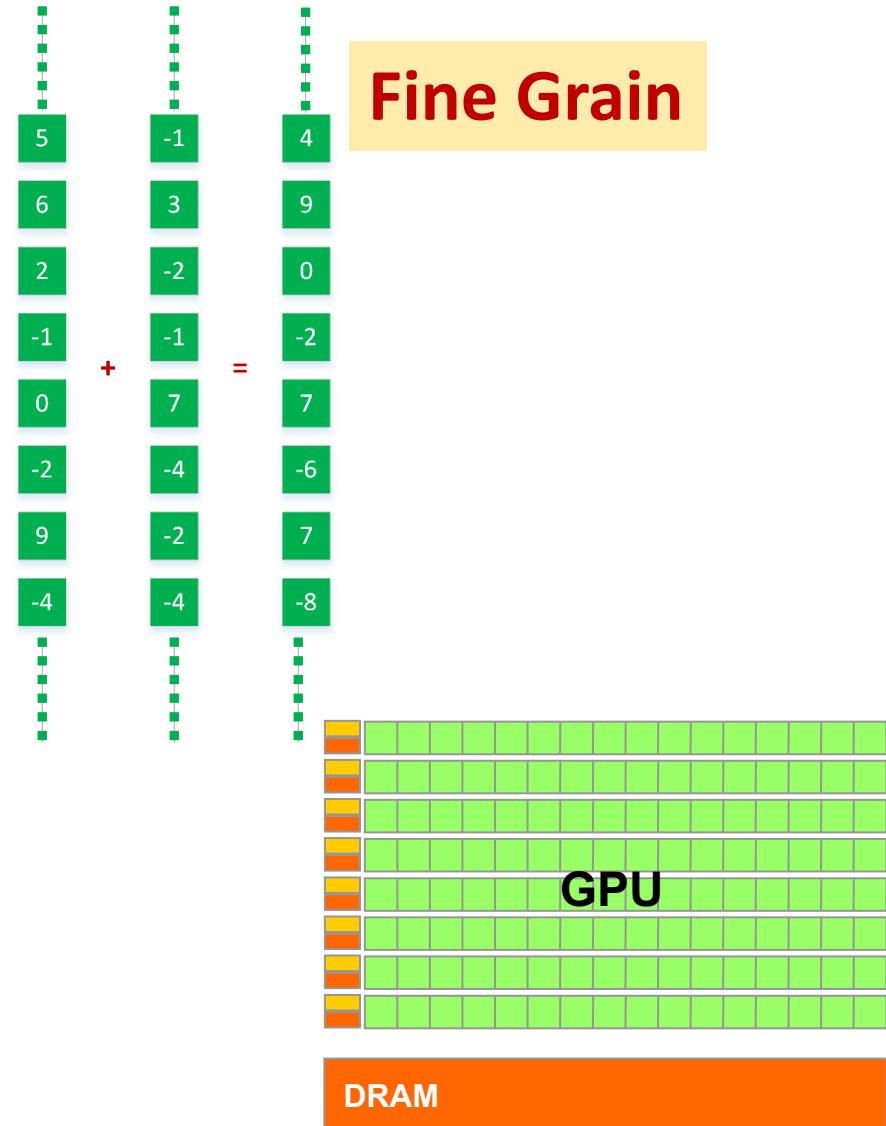
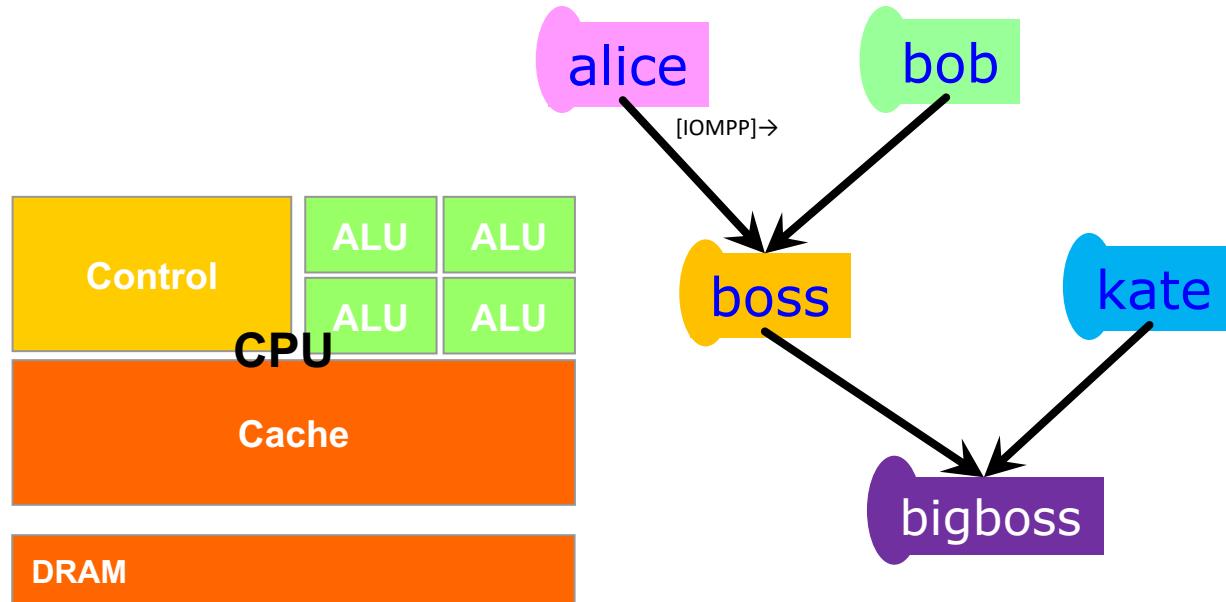
- Since you have higher bandwidth on a GPU, it is likely that your code will run faster on the right problem
- How much faster?
  - Compare the bandwidth/latency of mem accesses on GPU and CPU
  - Ball park: 4-5X speedup
- Other things might come into play
  - Caches and cache size, for instance
  - The very nature of the problem you're trying to solve
    - Good for data parallelism, **fine grain parallelism**

# Coarse Grain vs. Fine Grain Parallelism

```
#pragma omp parallel sections
{
#pragma omp section
    a = alice();
#pragma omp section
    b = bob();
#pragma omp section
    k = kate();
}

double s = boss(a, b);
double result = bigboss(s, k);
```

Coarse Grain



# Coarse Grain vs. Fine Grain Parallelism

- Coarse grain parallelism (good for CPUs)
  - Few tasks
  - Tasks are heterogeneous
  - Tasks are in general complex, lots of control flow
  - Example: baking a cake, making coffee, broiling fish – all at the same time
- Fine grain parallelism (very good for GPU, ok for CPU)
  - Many, many tasks
  - Tasks are basically identical
  - Tasks are in general pretty straightforward, lots of math, not much control flow
  - Example application: image processing – lots of pixels to deal with

# GPGPU Computing: when started, why started

- GPGPU: General Purpose GPU Computing
  - Done in early 2000s using graphics libraries
- Why was GPGPU attractive?
  - GPUs had high bandwidths
  - Something dampening our enthusiasm: data needs to be moved into the GPU to process it
    - 8-16 GB/s is typical today (PCIe)
    - NVLink: 5-12 times faster than PCIe 3

# GPGPU Computing: how carried out

- The idea is to use the GPU as a co-processor
  - Farm out big parallel jobs to the GPU
  - CPU stays busy with the control of the execution and “corner” tasks
  - Data moved down into the GPU, and then results fetched back  
(idea works ok when data transfer overhead overshadowed by the number crunching done using that data)
- NOTE: up until 1.5 years ago the “Data moved down into the GPU” was the responsibility of the programmer. Not the case anymore as of recently (from Pascal on).
- IMPORTANT: for now, we’ll still manage CPU↔GPU data movement. Revisit issue in six lectures

# CUDA: Making the GPU Tick...

- “Compute Unified Device Architecture” – freely distributed by NVIDIA
- When introduced (in 2006) it eliminated the graphics-constraints associated with GPGPU
- CUDA enables a general purpose programming model
  - User kicks off batches of threads on the GPU to execute a function (kernel)
- Targeted software stack
  - Scientific computing oriented drivers, language, and tools
  - Interface designed for compute (i.e., graphics-free API)
  - Explicit GPU memory management

# CUDA Programming Model: GPU as a Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a co-processor to the CPU or host
  - Has its own memory (device memory, in CUDA parlance)
  - Runs many threads in parallel
- Data-parallel portions of an app. run on the device as kernels executed in parallel by many threads
- Differences between GPU and CPU threads
  - GPU threads are **extremely lightweight**
    - Very little creation overhead
  - GPU **needs 1000s of threads** for full efficiency
    - Multi-core CPU needs only a few heavy ones

# Compute Capability [of a Device] vs. CUDA Version

- “**Compute Capability of a Device**” refers to hardware
  - Defined by a major revision number and a minor revision number
  - Example:
    - Tesla C1060 is compute capability 1.3
    - Fermi architecture is capability 2
    - Kepler architecture is capability 3
    - Maxwell architecture is capability 5
    - Pascal architecture is capability 6 (available on Euler)
    - Turing and Volta are compute capability 7 (available on Euler)
  - A higher compute capability indicates an more able piece of hardware
- The “**CUDA Version**” indicates what version of the software you are using to run on the hardware
- In a perfect world
  - You would run the most recent CUDA (version 10.2) software release
  - You would use the most recent architecture (compute capability 7.0)

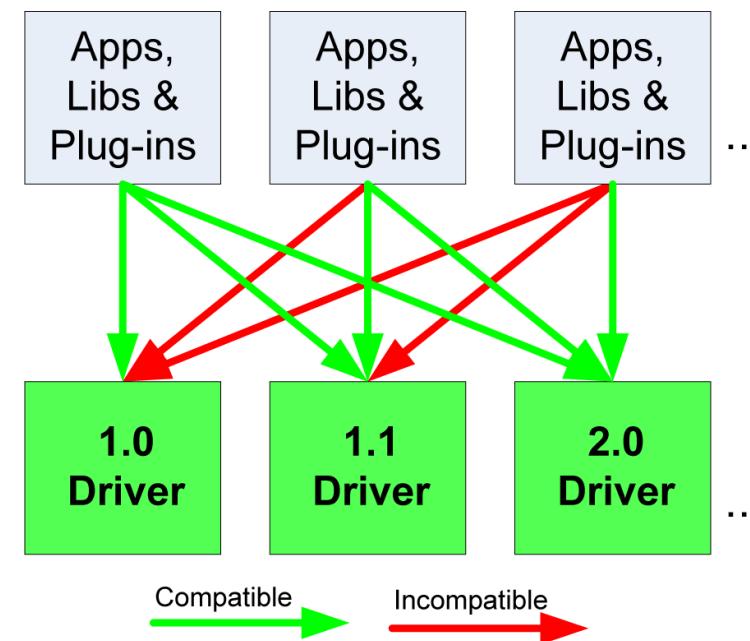
# NVIDIA Compute Capability



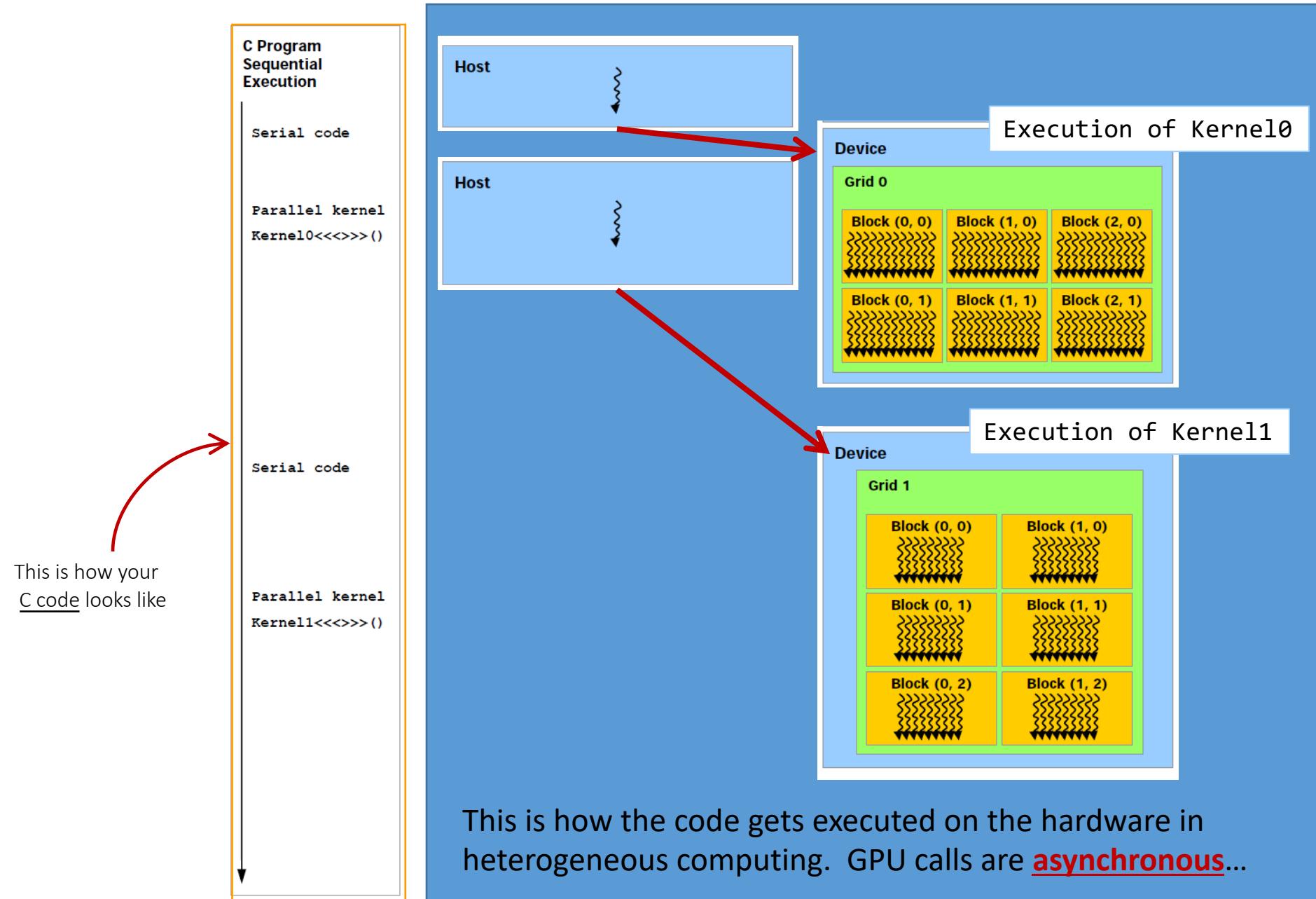
GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255 <sup>1</sup>
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB

# Compatibility Issues, software-wise

- The basic rule: the CUDA Driver API is backward, but not forward compatible
  - The functionality in later versions increased, was not there in previous versions
  - Code you ran ok w/ CUDA 5.0 will work w/ CUDA 10.0 as well
    - Not the other way around though



# The CUDA Execution Model



# The CUDA Execution Model

- Data needs to be copied into GPU device memory and the results need to be fetched back
  - This happens over a PCI-Express 3.0 connection
  - Can also happen over NVLink (5-12 times faster than PCI-E)
    - NVLink: Available on one Euler node only
- **IMPORTANT:**
  - For GPU computing to pay off, the data transfer overhead should be overshadowed by the GPU number crunching that draws on that data
- The CUDA kernel calls and copying to/from GPU are managed by the CUDA runtime in a **separate stream** associated w/ the GPU execution

# The CUDA Execution Model: Three Opportunities for Asynchronous Execution

- The GPU and CPU work in asynchronous mode
  - The CPU moves on **right away** after a kernel launch
  - The GPU works at the same time the CPU works
- Another level of asynchronicity: the GPU has three engines that can work at the same time
  - copy-in engine + copy-out engine + execution engine
  - More later (when we talk about streams)
- Another level of asynchronicity yet: Multiple GPUs can work at the same time on one host

# Languages Supported in CUDA

- CUDA: very good friends with C
  - Yet minor extensions are needed to flag the fact that a function actually represents a kernel, that there are functions that will only run on the device, etc.
    - You end up working in “C with extensions”
- FOTRAN is supported
- There is [growing] support for C++ programming (operator overload, lambda functions, etc.)

# The CUDA-enabled ecosystem for GPU Computing

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT, cuBLAS, cuRAND, cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL, SVM, OpenCurrent	PhysX, OptiX, iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java, Python, Wrappers	DirectCompute	Directives (e.g., OpenACC)	

# The NVIDIA CUDA-enabled hardware

CUDA-enabled NVIDIA GPUs				
Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series	Quadro RTX Series	Tesla T Series
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Tesla V Series
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series
	EMBEDDED	CONSUMER DESKTOP, LAPTOP	PROFESSIONAL WORKSTATION	DATA CENTER

## Remarks:

- There is no CC of gen 4
- Fermi was CC 2
- The name “Tesla” used for high-end GPUs for non-graphics apps

# CUDA, First Example

```
#include<cuda.h>
#include<iostream>

__global__ void simpleKernel(int* data)
{
    //this adds a value to a variable stored in global memory
    data[threadIdx.x] += 2*(blockIdx.x + threadIdx.x);
}

int main()
{
    const int numElems = 4;
    int hostArray[numElems], *devArray;

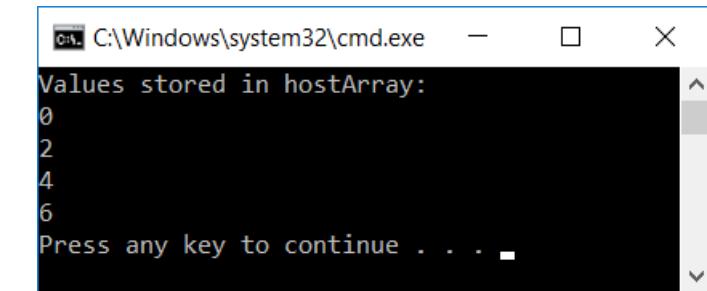
    //allocate memory on the device (GPU); zero out all entries in this device array
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1,numElems>>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}
```



# CUDA Function Declarations: (the “C with extensions” part)

	Executed on the:	Only callable from the:
<code>__device__ float myDeviceFunc ()</code>	device	device
<code>__global__ void myKernelFunc ()</code>	device	host
<code>__host__ float myHostFunc ()</code>	host	host

- `__global__` defines a kernel function, launched by host, executed on the device
  - Must return `void`
- NOTE: can combine `__device__` with `__host__`
- See CUDA Reference Manual <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

# Following Up on our Example: Segue into the “Execution Configuration”

- In our code, we invoked the kernel like this:  
`simpleKernel<<<1,4>>>(devArray)`
  - What happens if we invoke the kernel like this:  
`simpleKernel<<<1,12>>>(devArray)`

# GPU Execution Configuration

# Review of Nomenclature

- The **HOST**
  - This is your CPU executing the “master” thread
- The **DEVICE**
  - This is the GPU card, connected to the HOST through a PCIe connection
- The **HOST** (the master CPU thread) instructs the **DEVICE** to execute **KERNEL**
- When launching a **KERNEL**, the **HOST** has to inform the **DEVICE** how many threads should each execute **KERNEL**
  - This is called “defining the **execution configuration**”

# [New Topic] The concept of Execution Configuration

- A kernel function must be called with an **execution configuration**:

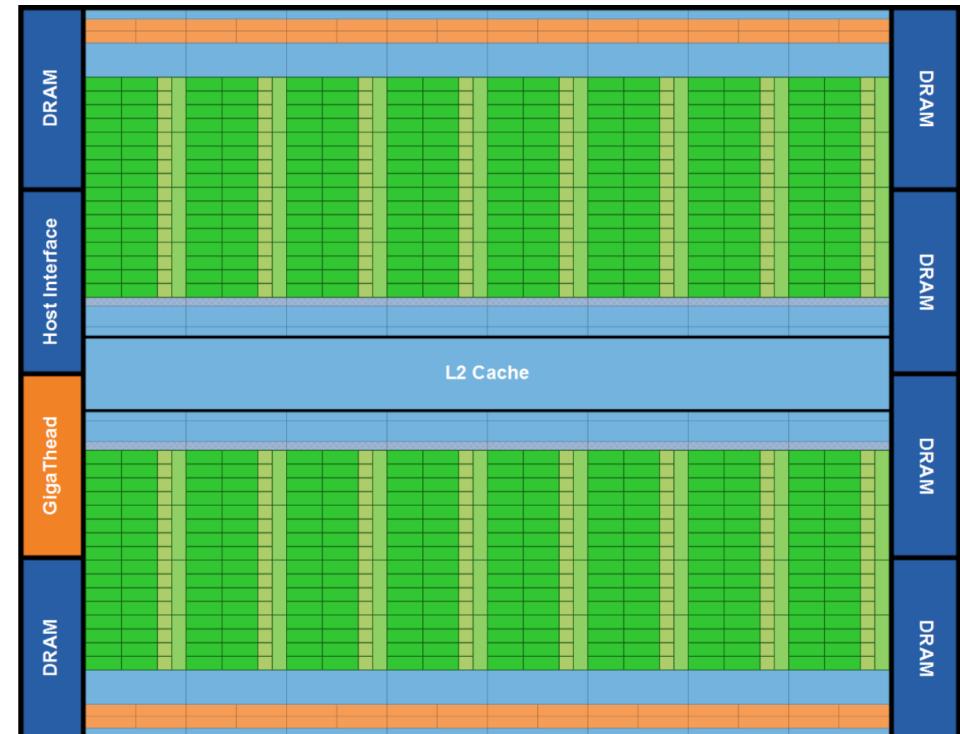
```
__global__ void kernelFoo(...); // declaration
dim3 DimGrid(100, 50);        // 2D grid structure, w/ total of 5000 thread blocks
dim3 DimBlock(4, 8, 8);       // 3D block structure, with 256 threads per block
kernelFoo<<< DimGrid, DimBlock>>>(...your arg list comes here...);
```

# Example

- The host call below instructs the GPU to execute the function (kernel) “`foo`” using 25,600 threads
  - Two arguments are passed down to each thread executing the kernel “`foo`”

```
foo<<<100, 256>>>(p_matrixD, p_vectorD);
```

- In this execution configuration, the host instructs the device it is supposed to run 100 blocks each having 256 threads in it
- The concept of block is important since it represents the entity that gets executed by an SM (stream multiprocessor)



# Execution Configuration Constraints

- There is a limitation on the number of blocks in a grid:
  - The grid of blocks can be organized as a 3D structure: max of  $2^{31}-1$  by 65,535 by 65,535 grid of blocks
- Threads in each block:
  - The threads can be organized as a 3D structure (x,y,z)
  - Maximum x- or y-dimension of a block is 1024
  - Maximum z-dimension of a block is 64
  - The maximum number of threads in each block is 1024

# More on the Execution Configuration

- Bottom line: max number of threads a kernel can be invoked with

$$2^{31} \times 2^{16} \times 2^{16} \times 2^{10} = 2^{73}$$

Max number of blocks in x-direction  
Max number of blocks in y-direction  
Max number of blocks in z-direction  
Max number of threads in a block

- Max count: about  $9.444732965739290e+21$  threads can execute a kernel
- What if you want more threads to execute that kernel?
  - You call the kernel again
- Putting things in perspective:
  - The kernel is executed by the threads you specify through your execution configuration

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 10

02/12/2020

# Quote of the day

“Weaseling out of things is important to learn. It's what separates us from the animals... except the weasel.”

-- Homer Simpson, Safety Inspector [1989 - ]. Regarding his son upbringing.

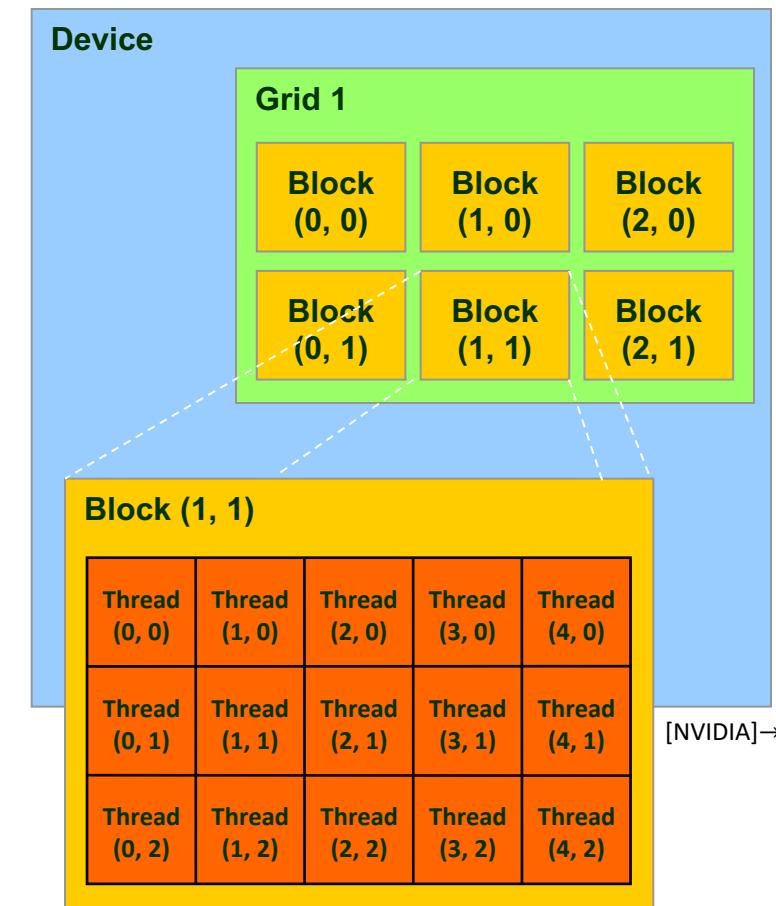
# Before we get going...

- Last time: The basics of GPU computing
  - The unit of scaling in GPU computing is the SM (see pic, for Fermi)
    - Not a lot of CU brain, but lots of ALU muscle
  - GPU computing happens through kernel execution
    - A collection of threads executes the kernel
    - Each thread executes the same kernel
  - Execution configuration: how many threads execute your kernel
  - GPU kernel execution is asynchronous
    - One program calls the GPU to execute kernelFOO, then later on kernelBAR, then later yet kernelWrapUp, etc.
- Today
  - More on execution configuration
  - GPU “scheduling for execution” issues
- Other tidbits:
  - Dan leaving town: today lecture short & no lecture on Friday
  - Assignment due on Th at 9 pm
  - Tonight on-line office hours: Lijing will be in charge
  - Anonymous feedback: <https://forms.gle/ZUCbyKX87mJvK61M7>



# Block and Thread Index (Idx)

- Threads and blocks have indices
  - Used by each thread to decide what data to work on (more later)
  - Block Index: a triplet of uint
  - Thread Index: a triplet of uint
- Why this 3D layout?
  - Simplifies memory addressing when processing multidimensional data
    - Handling matrices
    - Solving PDEs on 3D subdomains
    - ...



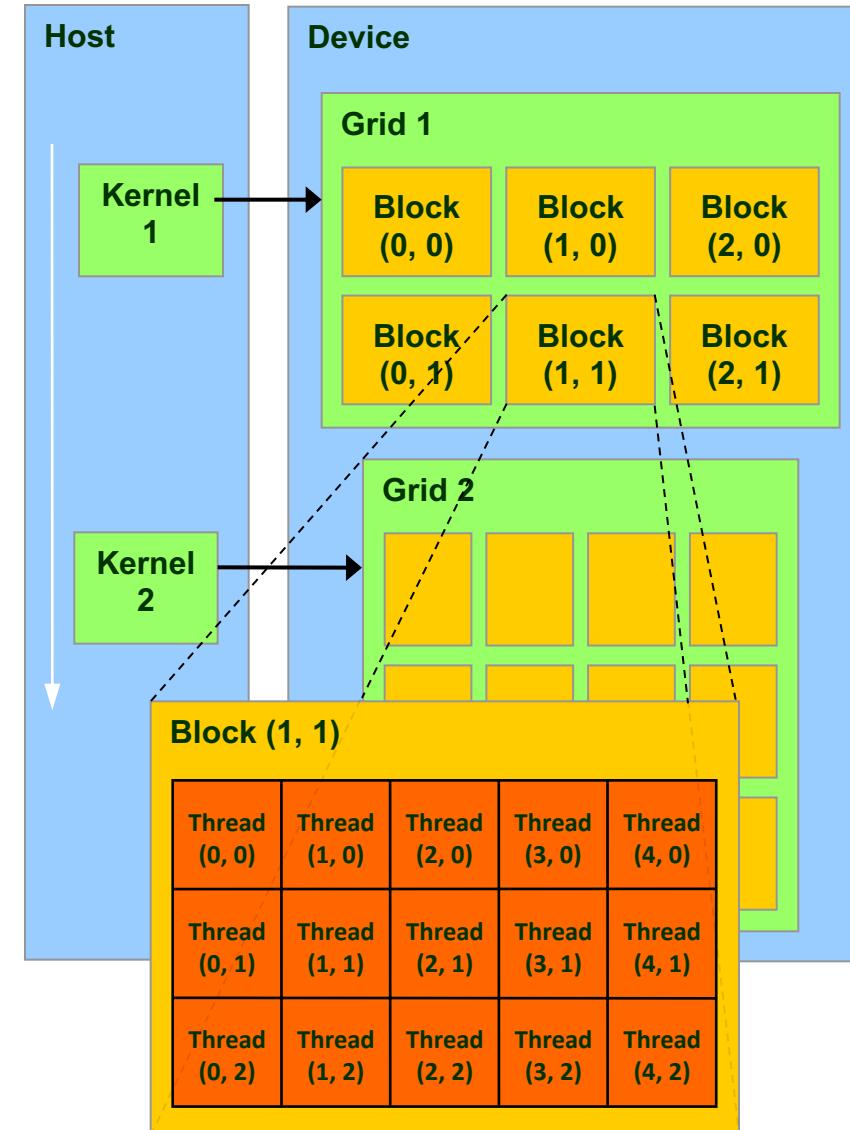
# A Couple of Built-In Variables

[Critical in supporting the SIMD parallel computing paradigm]

- Each thread can find out the grid and block dimensions and its block index and thread index
  - This info used to figure out what work the thread needs to do
- Each thread when executing a kernel has access to the following read-only built-in variables
  - `threadIdx` (`uint3`) – contains the thread index within a block
  - `blockDim` (`dim3`) – contains the dimension of the block
  - `blockIdx` (`uint3`) – contains the block index within the grid
  - `gridDim` (`dim3`) – contains the dimension of the grid
  - [ `warpSize` (`uint`) – provides warp size, we'll talk about this later... ]

# Check your understanding

- How was the grid defined for this pic?
- I.e., how many blocks in X and Y directions?
- How was a block defined in this pic?



## Matrix Multiplication Example

# Simple Example: Matrix Multiplication

- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA
- There's a HW based on this
  - Use only global memory (don't bring shared memory into picture yet)
  - Matrix will be of small dimension, job can be done using one block of threads
  - Concentrate on
    - Thread ID usage
    - Memory data transfer API between host and device

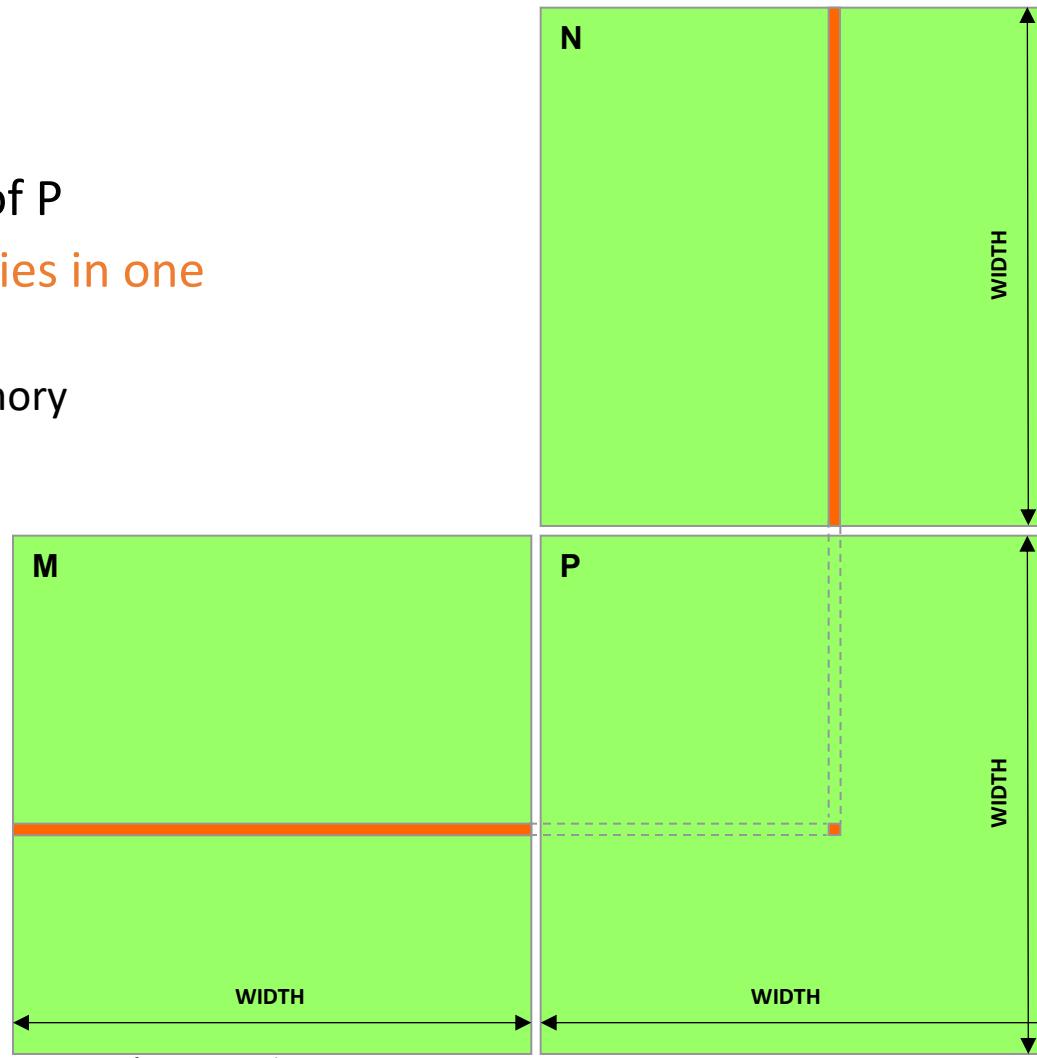
# Prelude: on the Matrix Data Structure

- The following data structure will come in handy
  - Purpose: store matrix-related data
  - Note that the matrix is stored in row-major order in a one dimensional array pointed to by “elements”

```
// IMPORTANT - Matrices are stored in row-major order:  
// M(row, col) = M.elements[row * M.width + col]  
  
typedef struct {  
    int width;  
    int height;  
    float* elements;  
} Matrix;
```

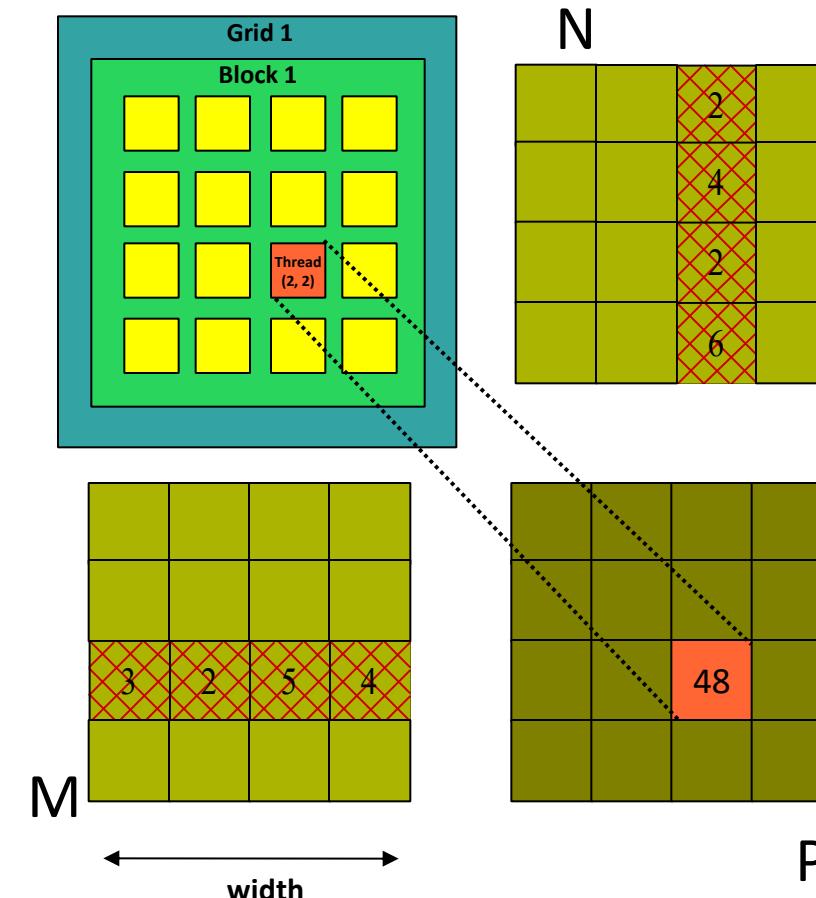
# Square Matrix Multiplication Example

- Compute  $P = M * N$ 
  - The matrices P, M, N are of size  $\text{WIDTH} \times \text{WIDTH}$
  - Assume  $\text{WIDTH}$  was defined to be 32
- Software Design Decisions:
  - One **thread** handles one **element** of P
  - **Each thread will access all the entries in one row of M and one column of N**
    - $2 * \text{WIDTH}$  read accesses to global memory
    - One write access to global memory



# Multiply Using One Thread Block

- One Block of threads computes matrix P
  - Each thread computes one element of P
- Each thread does a dot product
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1
    - Not that good...
- Size of matrix limited by the number of threads allowed in a thread block



# Matrix Multiplication: sequential approach, coded in C

```
// Matrix multiplication on the (CPU) host in double precision;

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i) {
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k]; //march along a row of M
                double b = N.elements[k * N.width + j]; //march along a column of N
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
    }
}
```

# Step 1: Matrix Multiplication, Host-side. Main Program Code

```
int main(void) {
    // Allocate and initialize the matrices.
    // The last argument in AllocateMatrix: should an initialization with
    // random numbers be done? Yes: 1. No: 0 (everything is set to zero)
    Matrix M = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix N = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix P = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);

    return 0;
}
```

## Step 2: Matrix Multiplication [host-side code]

```
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);

    // Setup the execution configuration
    dim3 dimGrid(1, 1, 1);
    dim3 dimBlock(WIDTH, WIDTH);

    // Launch the kernel on the device
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```

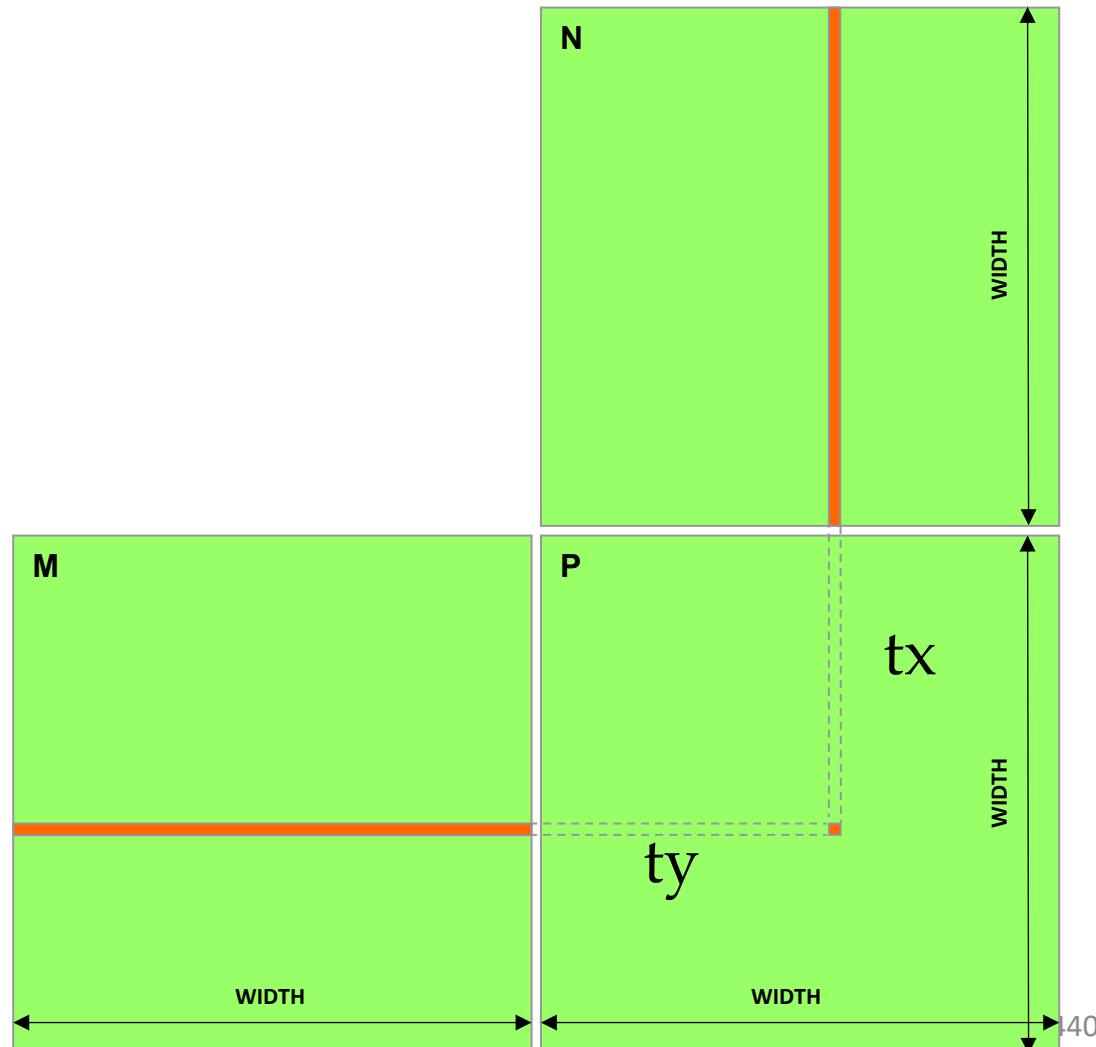
# Step 3: Matrix Multiplication - Device-side Kernel Function

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P) {
    // 2D Thread Index; computing P[ty][tx]...
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue will end up storing the value of P[ty][tx].
    // That is, P.elements[ty * P.width + tx] = Pvalue
    float Pvalue = 0;

    for (int k = 0; k < M.width; ++k) {
        float Melement = M.elements[ty * M.width + k];
        float Nelement = N.elements[k * N.width + tx];
        Pvalue += Melement * Nelement;
    }
    // Write matrix to device memory; each thread one element
    P.elements[ty * P.width + tx] = Pvalue;
}
```

Note: Pvalue is a local value; stored in a register. Access global memory once at the very end



## Step 4: Some Loose Ends

```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M) {
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost) {
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size, cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice) {
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size, cudaMemcpyDeviceToHost);
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}
void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

# Before Moving On...

[Some Words of Wisdom]

- In GPU computing you use as many threads as data items (tasks, jobs) you have to perform
    - This replaces the purpose in life of the “`for`” loop
    - Number of threads & blocks is established at run-time
  - In many cases, the rule is **Number of threads = Number of data items**
    - You’ll have to come up with a rule to match a thread to data item that this thread needs to process
    - Most common source of errors and frustration in GPU computing
      - It never fails to deliver (frustration)
- :-(

# CUDA, Another Example (1/2)

[Highlighted aspect: a thread figuring out its work order]

- Multiply, pairwise, two arrays of 3 million integers

```
1. int main(int argc, char* argv[])
2. {
3.     const int arraySize = 3000000;
4.     int *hA, *hB, *hC;
5.     setupHost(&hA, &hB, &hC, arraySize);
6.
7.     int *dA, *dB, *dC;
8.     setupDevice(&dA, &dB, &dC, arraySize);
9.
10.    cudaMemcpy(dA, hA, sizeof(int) * arraySize, cudaMemcpyHostToDevice);
11.    cudaMemcpy(dB, hB, sizeof(int) * arraySize, cudaMemcpyHostToDevice);
12.
13.    const int threadsPerBlock = 512;
14.    const int blocksPerGrid = (arraySize + threadsPerBlock - 1)/threadsPerBlock;
15.    multiply_ab<<<blocksPerGrid, threadsPerBlock>>>(dA, dB, dC, arraySize);
16.    cudaMemcpy(hC, dC, sizeof(int) * arraySize, cudaMemcpyDeviceToHost);
17.
18.    cleanupHost(hA, hB, hC);
19.    cleanupDevice(dA, dB, dC);
20.    return 0;
21. }
```

## CUDA, Another Example (2/2)

```
1. __global__ void multiply_ab(int* a, int* b, int* c, int size)
2. {
3.     int whichEntry = threadIdx.x + blockIdx.x * blockDim.x;
4.     if( whichEntry<size )
5.         c[whichEntry] = a[whichEntry] * b[whichEntry];
6. }
```

```
1. void setupDevice(int** pdA, int** pdB, int** pdC, int arraySize)
2. {
3.     cudaMalloc((void**) pdA, sizeof(int) * arraySize);
4.     cudaMalloc((void**) pdB, sizeof(int) * arraySize);
5.     cudaMalloc((void**) pdC, sizeof(int) * arraySize);
6. }
7.
8. void cleanupDevice(int *dA, int *dB, int *dC)
9. {
10.    cudaFree(dA);
11.    cudaFree(dB);
12.    cudaFree(dC);
13. }
```

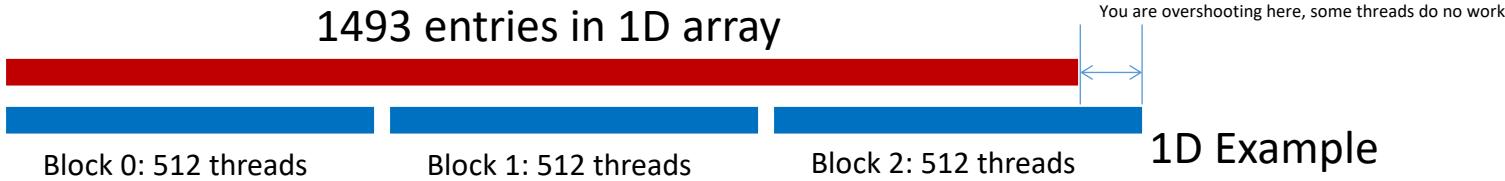
# `if( whichEntry<size )`: Why Do I Need this Test?

- Important observation before answering question above
  - All blocks launched have the same number of threads
- Answer to our question: the `if` is needed to prevent out of bounds indexing
  - Sometimes the product of the number of blocks  $\times$  number of threads per block is not exactly how many jobs need to be done

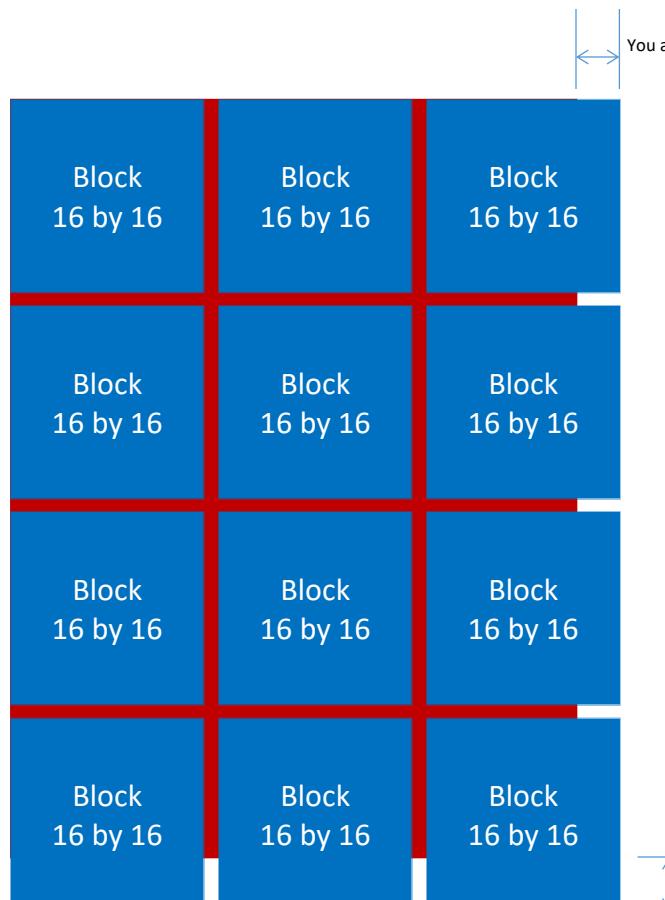
# Execution Configuration Related

- Assume arrays of 1493 elements (instead of 3000000)
- Largest CUDA block that you can get has 1024 threads
- Also, 1493 is a prime number
- You'll have to use at least two CUDA blocks of threads
  - Perhaps you want to use three blocks of 512 each...
- You'll have some threads that do no work (there'll be an `if` statement in there)
- Problem becomes more interesting in two dimension when you have square blocks to pad a matrix and end up overshooting in two directions

1493 entries in 1D array



1D Example



2D Example

You are overshooting here as well

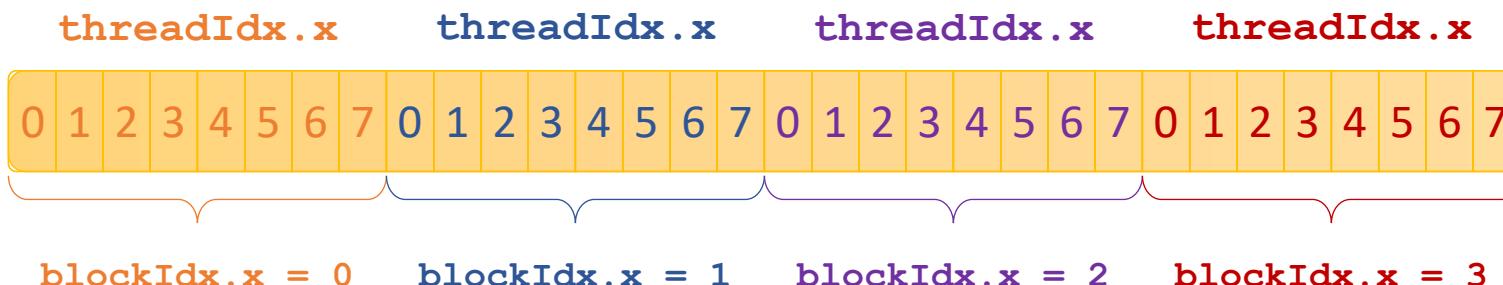
# Array Indexing Issues

- Next three slides: revisit how multiple blocks are used together
  - Fundamental question that a thread asks in GPU computing:  
**What work, or which task, do I have to do?**
- First, recall this: there is a limit on the number of threads you can squeeze in a CUDA block – you can have up to 1024 threads
- Note: In the vast majority of applications you need to use many blocks (each of which contains the same number of threads, say  $M$ ) to get a job done. Next example puts things in perspective

# Array Indexing Demo: The multi-block case

[Important to grasp: shows **thread-to-task mapping**]

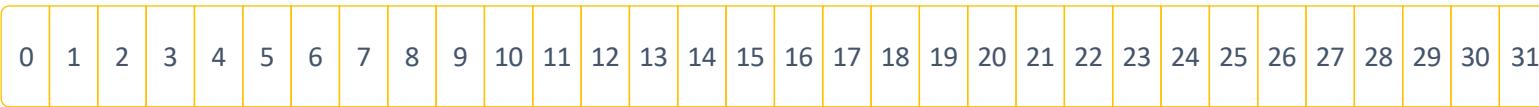
- Multiple blocks – you will likely be using more than just **threadIdx.x**
  - Consider indexing into an 1D array, **one thread accessing one element**
  - Assume you have **M=8** threads per block and the array is 32 entries long



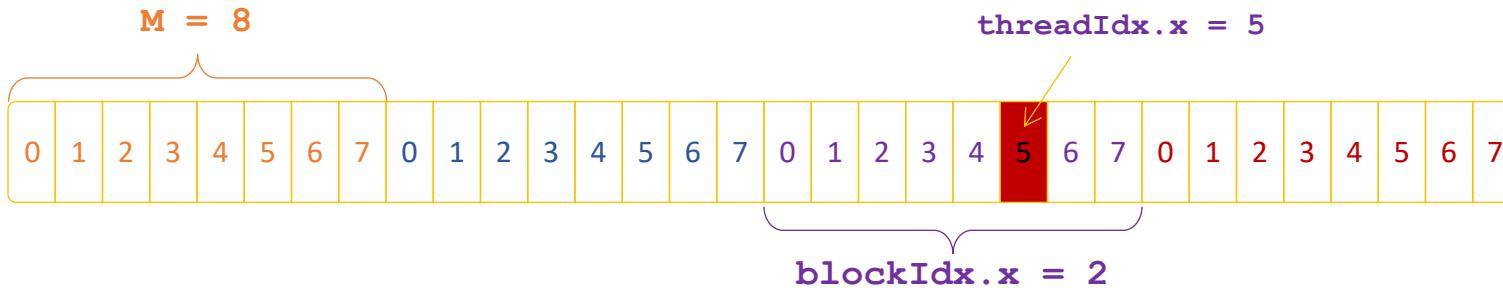
- With **M** threads per 1D CUDA block, the array **index** that a thread works on is:

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

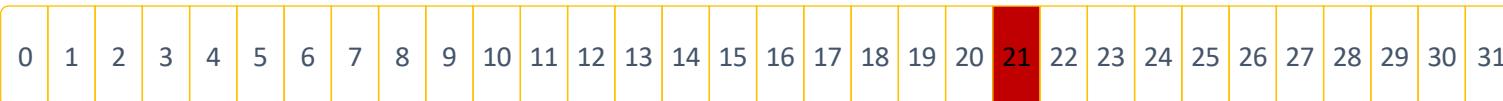
# Example: Array Indexing



- What is the array index that thread of index 5 in block of index 2 will work on?



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



# Recurring Theme in CUDA Programming

[summarizing previous discussion, started after CUDA Example]

- Imagine **you are one of many threads; you have your thread index & block index**
- Two things need to be addressed
  1. You need to figure out the work, or the job, you need to take care of
    - Just like we did on previous slide, where thread 5 in block 2 realized it needed to work on entry 21
  2. You have to make sure you actually need to do that work
    - In many problems there are threads that need not do any work
    - Example: you launch 2 blocks with 512 threads; your array is only 1000 elements long; 24 threads at the end might do nothing
      - “24 threads at the end might do nothing” – depending on the implementation

# [short topic]: Timing Your Application

- Timing support – part of the CUDA API
  - You pick it up as soon as you include `<cuda.h>`
  - Provides cross-platform compatibility
  - Deals with the asynchronous nature of the device calls by relying on events and forced synchronization
- Reports time in milliseconds, accurate within 0.5 microseconds
- From NVIDIA CUDA Library Documentation:

*Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns cudaErrorInvalidValue. If either event has been recorded with a non-zero stream, the result is undefined.*

# Timing Example: timing a GPU call

```
#include<iostream>
#include<cuda.h>

int main() {
    cudaEvent_t startEvent, stopEvent;
    cudaEventCreate(&startEvent);
    cudaEventCreate(&stopEvent);

    cudaEventRecord(startEvent, 0);

    yourKernelCallComesHere<<<NumBlk,NumThrds>>>(args);


    cudaEventRecord(stopEvent, 0);
    cudaEventSynchronize(stopEvent);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);
    std::cout << "Time to get device properties: " << elapsedTime << " ms\n";

    cudaEventDestroy(startEvent);
    cudaEventDestroy(stopEvent);
    return 0;
}
```

# Execution Scheduling Issues

# [short/important topic]: Thread Index vs. Thread ID

[critical in (i) understanding how SIMD is supported in CUDA, and (ii) understanding the concept of “warp”]

- Each block organizes its threads in a 3D structure defined by its three dimensions:  $D_x$ ,  $D_y$ , and  $D_z$  that you specify.
- A block cannot have more than 1024 threads  $\Rightarrow D_x \times D_y \times D_z \leq 1024$ .
- Each thread in a block can be identified by a unique index  $(x, y, z)$ , and

$$0 \leq x < D_x \quad 0 \leq y < D_y \quad 0 \leq z < D_z$$

- A triplet  $(x, y, z)$ , called the thread index, is a high-level representation of a thread in the economy of a block. Under the hood, the same thread has a simplified and unique id, which is computed as  $t_{id} = x + y * D_x + z * D_x * D_y$ . You can regard this as a “projection” to a 1D representation. The concept of thread id is important in understanding how threads are grouped together in warps (more on “warps” later).
- In general, operating for vectors typically results in you choosing  $D_y = D_z = 1$ . Handling matrices typically goes well with  $D_z = 1$ . For handling PDEs in 3D you might want to have all three block dimensions nonzero.

# Thread Execution Scheduling

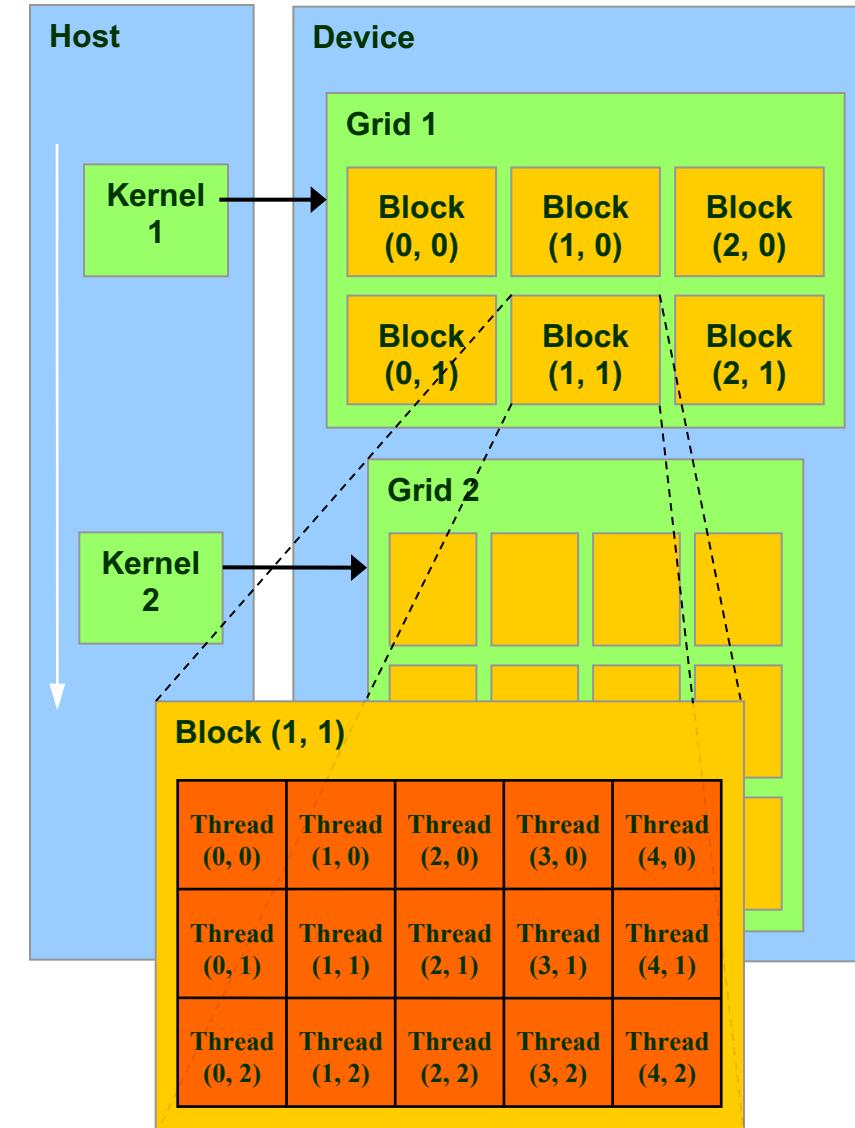
- Starting point observation:
  - You launch on the device many blocks, each containing many threads
- Questions in this segment:
  - In which order are these blocks executed?
  - How is this execution process managed?
  - When/How are the threads in a block executed?
  - Etc...

# High-level Perspective

- There are two schedulers at work in GPU computing
  - A **device-level** scheduler: assigns blocks to SM that indicate at a given time “excess capacity”
    - This scheduler called by NVIDIA the “GigaThread engine”
  - An **SM-level** scheduler: schedules the execution of the threads in a block onto the SM functional units
    - This is the more interesting scheduler

# Device-Level Scheduler

- Grid is launched on the device
- Thread Blocks are distributed to the SMs
  - Potentially more than one block per SM
  - There is a limit on the number of blocks an SM can take.
- As Thread Blocks complete kernel execution, resources are freed  
Device-level scheduler can launch next block(s) in line
- This is the first levels of scheduling:  
For running [desirably] a large number of blocks (thousands) on a relatively small number of SMs (60/16/14/etc.)
- Limits, for **resident** blocks/SM:
  - 32 blocks on the Maxwell SM, Pascal SM, and Volta SM
  - 16 blocks can be resident on a Kepler SM
  - 8 blocks can be resident on a Fermi & Tesla SM

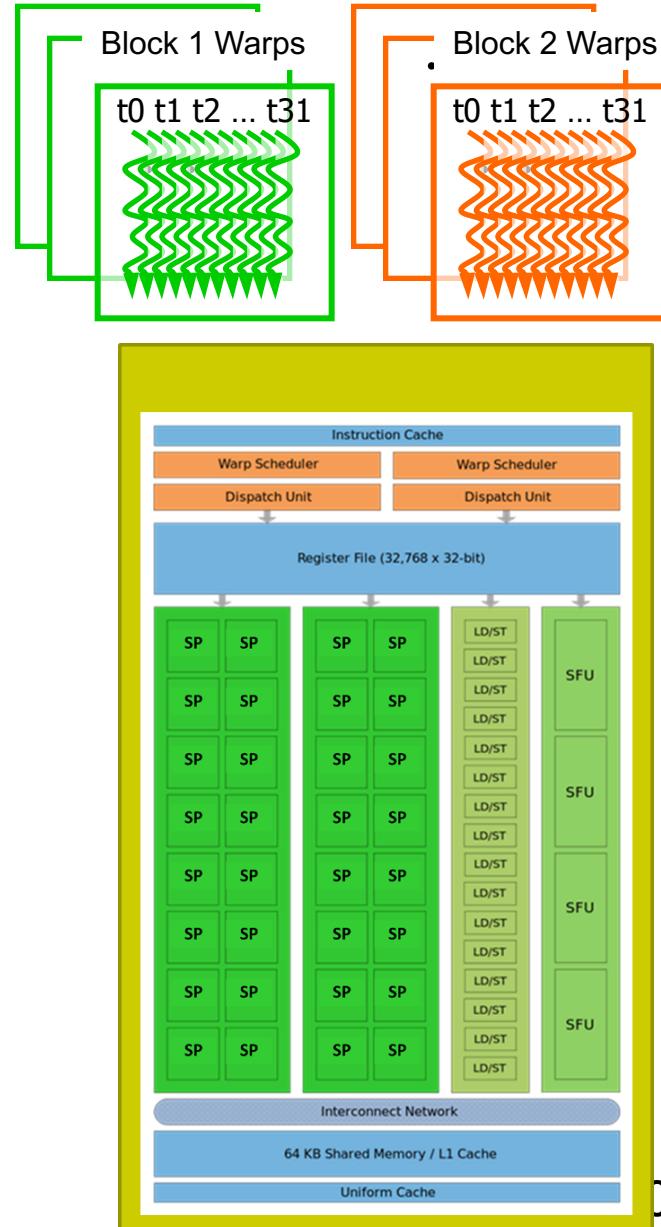


# Device-Level Scheduler, Comments

- Once a block is picked up for execution by one SM, the block does not leave that SM before all threads in that block finish executing the kernel
- Once a block is finished & retired, only then can another block land for execution on that SM
  - The Device-Level Scheduler dispatches the next block in line to the SM that indicates that it has excess capacity
- This explains why it's good to have many SM in your GPU
  - More SMs → more expensive card

# SM-Level Schedulers

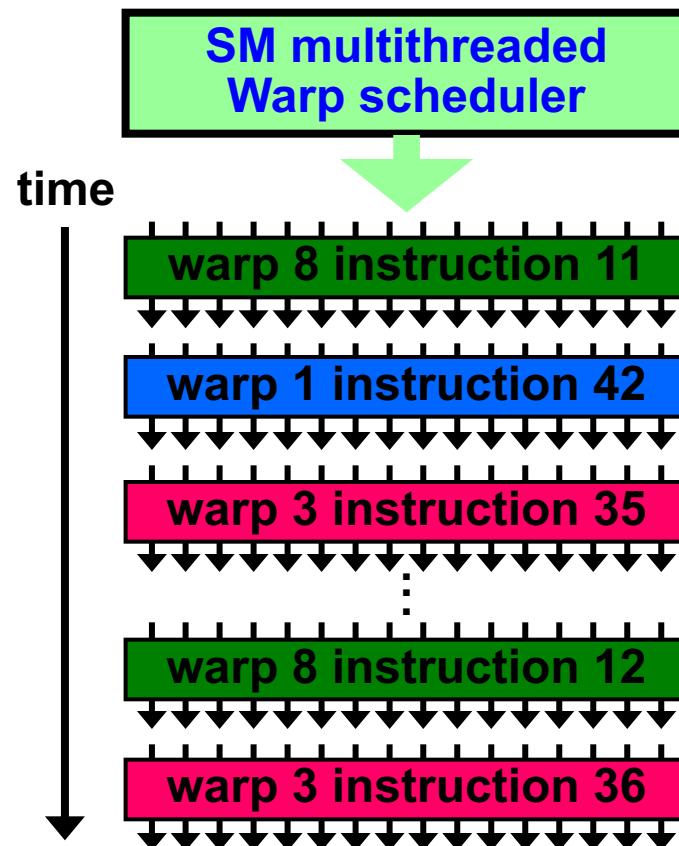
- Each block of threads divided in 32-thread **warps**
  - “32”: selected by NVIDIA, programmer has no say
  - Warp: A group of 32 threads of consecutive IDs
- Warps are the basic scheduling unit on the SM
- Limits, number of resident warps on an SM:
  - 64: on Kepler, Maxwell, Pascal, Volta (i.e., 2048 resident threads)
  - 48: on Fermi (i.e., 1536 resident threads)
  - 32: on Tesla (i.e., 1024 resident threads)



# Quiz

- If 3 blocks are processed by an SM and each Block has 256 threads, how many warps are managed by the SM?
  - Each block of threads is divided into  $256/32 = 8$  warps
  - There are  $8 * 3 = 24$  warps
  - At any point in time, there are 24 warps that can be selected for execution

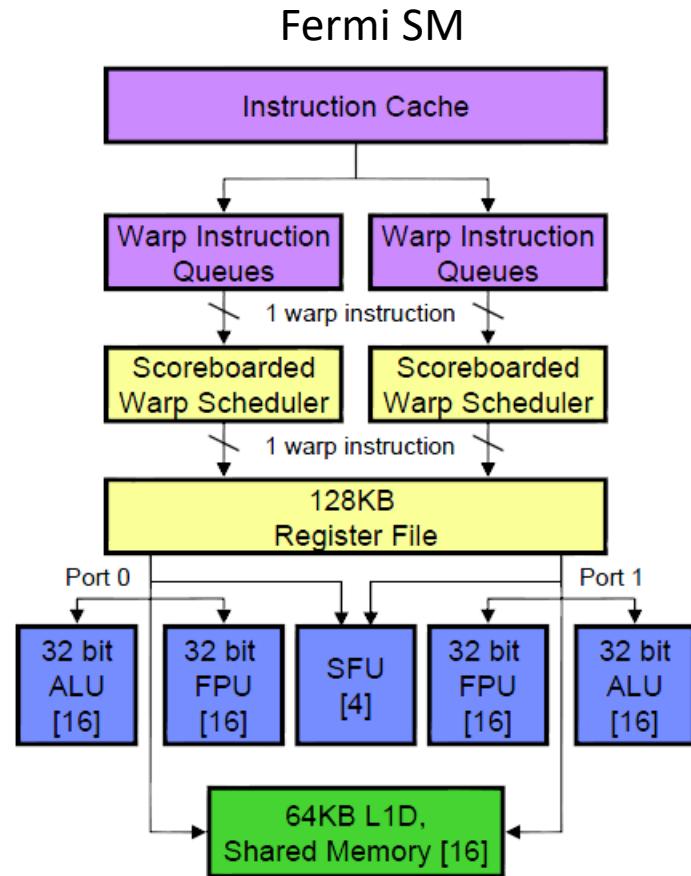
# SM Warp Scheduling, more details



- SM hardware implements almost zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute same instruction

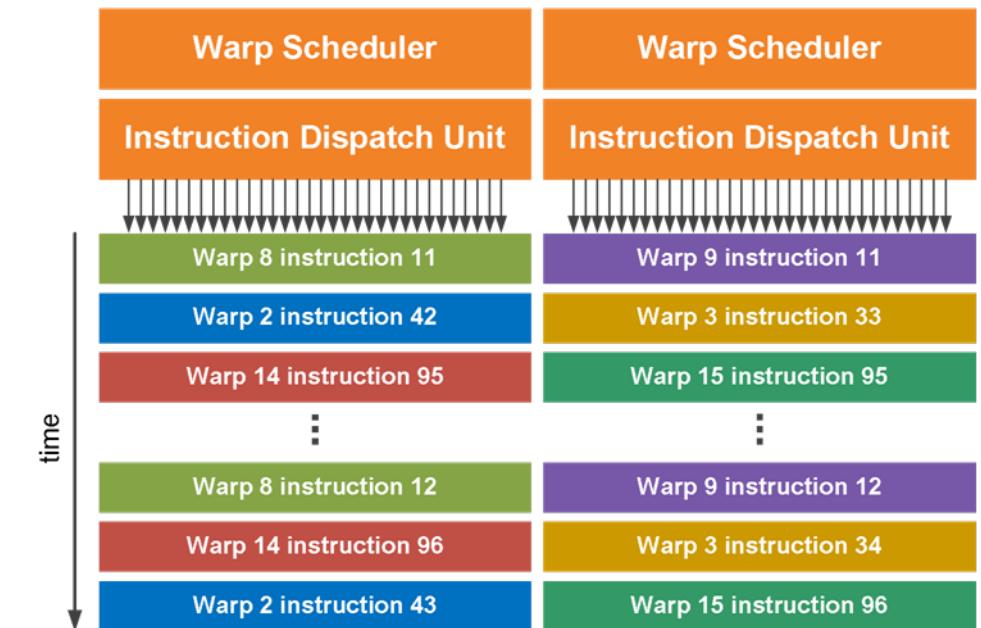
# Fermi Specifics

- There are two schedulers that issue warps of “ready-to-go” threads
- One warp issued at each clock cycle by each scheduler
- During no cycle can more than 2 warps be dispatched for execution on the SM’s functional units
- Scoreboarding used to figure out which warp is ready



# Quick Remarks

- ILP support: relatively limited
- GPU architecture is pipelined
  - At each clock cycle one instruction can be retired
- Fermi has no out-of-order execution smarts
- To the best of my knowledge
  - There is no prefetching
  - There is no speculative execution



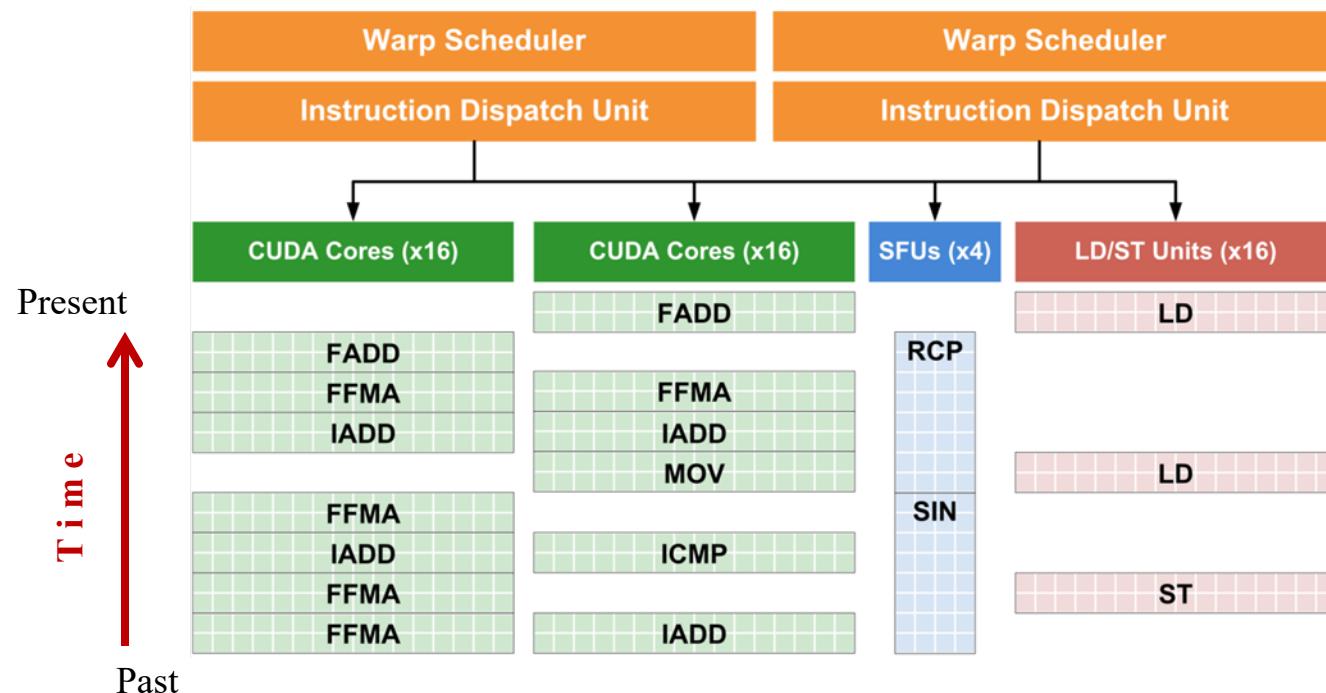
# Example: Fermi Related

- Scheduler works at 607 MHz
- Functional units work at 1215MHz
- Question:
  - What is the peak flop rate of GTX480?
  - $15 \text{ SMs} * 32 \text{ SPs} * 1215 * 2 \text{ (Fused Multiplied Add)} = 1166400 \text{ Mflops}$
  - That is, 1.166 Tflops, single precision



# Fermi Specifics

- As shown in picture, at no time can we see more than 2 warps being dispatched for execution during a cycle
- Note that at any given time we might have more than two functional units working though (which is actually very good, device kept busy)



# More Recent SM Generations

Volta SM

Maxwell SMX



Pascal SM



Motivated by AI & Machine Learning

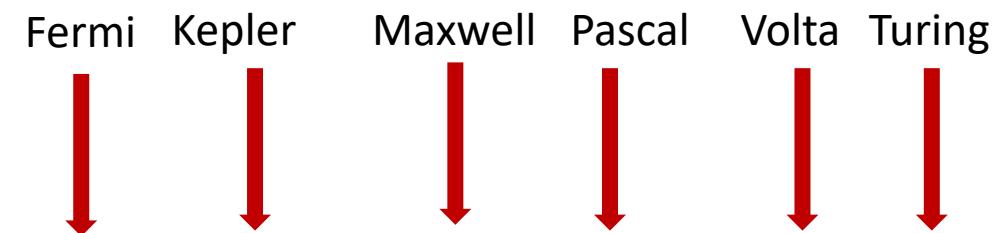


# What you can do with one SM (of the many you have on 1 GPU)



# SM Architecture Specifications

Fermi Kepler Maxwell Pascal Volta Turing



Architecture specifications	Compute capability (version)														
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5	3.7	5.0	5.2	6.0	6.1, 6.2	7.0, 7.2	7.5
Number of ALU lanes for integer and single-precision floating-point arithmetic operations	8 <sup>[36]</sup>				32	48	192		128	64	128		64		
Number of special function units for single-precision floating-point transcendental functions	2				4	8	32			16	32		16		
Number of texture filtering units for every texture address unit or <i>render output unit</i> (ROP)	2				4	8	16					8 <sup>[37]</sup>			
Number of warp schedulers	1				2		4			2		4			
Max number of instructions issued at once by a single scheduler	1				2 <sup>[38]</sup>					1					
Number of tensor cores					N/A							8 <sup>[37]</sup>			
Size in KB of unified memory for data cache and shared memory per multi processor	t.b.d.											128	96 <sup>[39]</sup>		

# Technical Specifications and Features

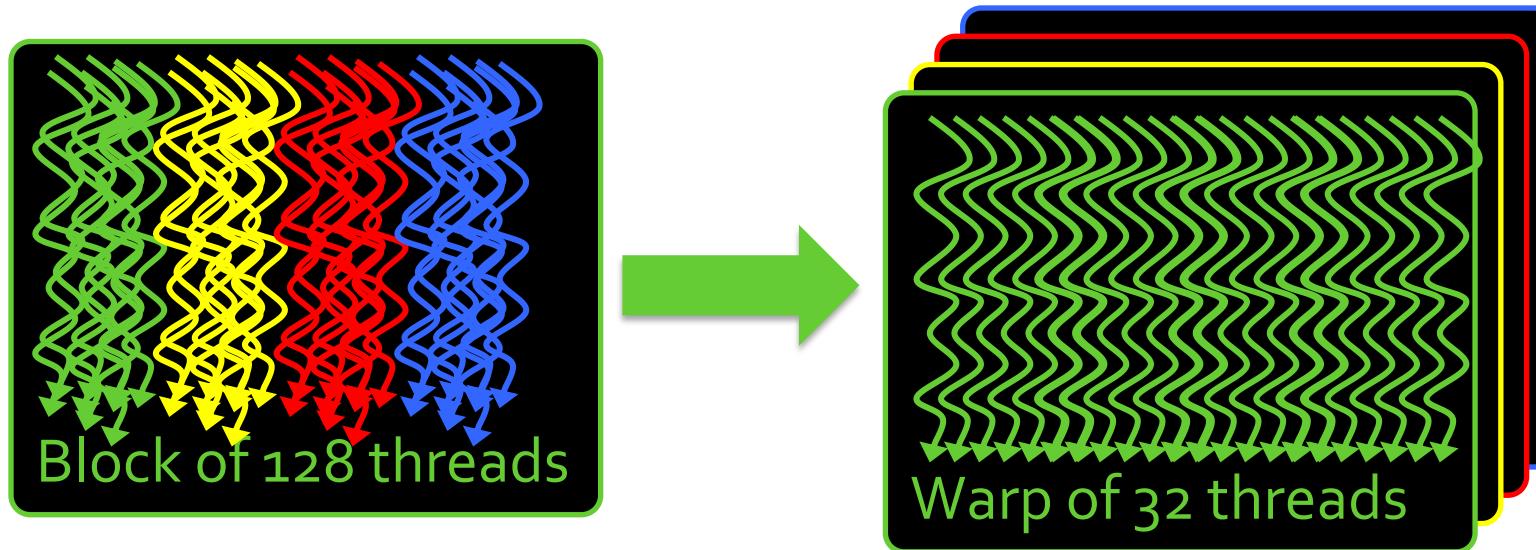
Technical specifications	Compute capability (version)																																	
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0 (7.2?)	7.5																	
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16	4	32				16	128	32	16	128																			
Maximum dimensionality of grid of thread blocks	2				3																													
Maximum x-dimension of a grid of thread blocks	65535				$2^{31} - 1$																													
Maximum y-, or z-dimension of a grid of thread blocks	65535																																	
Maximum dimensionality of thread block	3																																	
Maximum x- or y-dimension of a block	512				1024																													
Maximum z-dimension of a block	64																																	
Maximum number of threads per block	512				1024																													
Warp size	32																																	
Maximum number of resident blocks per multiprocessor	8				16				32				16																					
Maximum number of resident warps per multiprocessor	24	32	48	64												32																		
Maximum number of resident threads per multiprocessor	768	1024	1536	2048												1024																		

# Organizing Threads into Warps

- Thread IDs within a warp are **consecutive** and **increasing**
  - Related to the 1D projection from thread index to thread ID
  - Recall: in multidimensional blocks, the **x** thread index runs fastest, then **y**, then **z**
  - Threads with IDs (0...31) combine into warp 0, threads (32...63) into warp 1, etc.
- Partitioning of threads into warps is always the **same**
  - You can use this in control flow
  - Warp size has always been 32 and is unlikely to change soon
- While you can rely on ordering among threads, **DO NOT** rely on any ordering among warps
  - Warp scheduling is not under user control in CUDA

# Threads are Organized & Executed as Warps

- Each thread block split into one or more warps
- If thread block size is not multiple of warp size, unused lanes go wasted
  - Example: block w/ 50 threads – 64 lanes, of which last 14 lanes go wasted

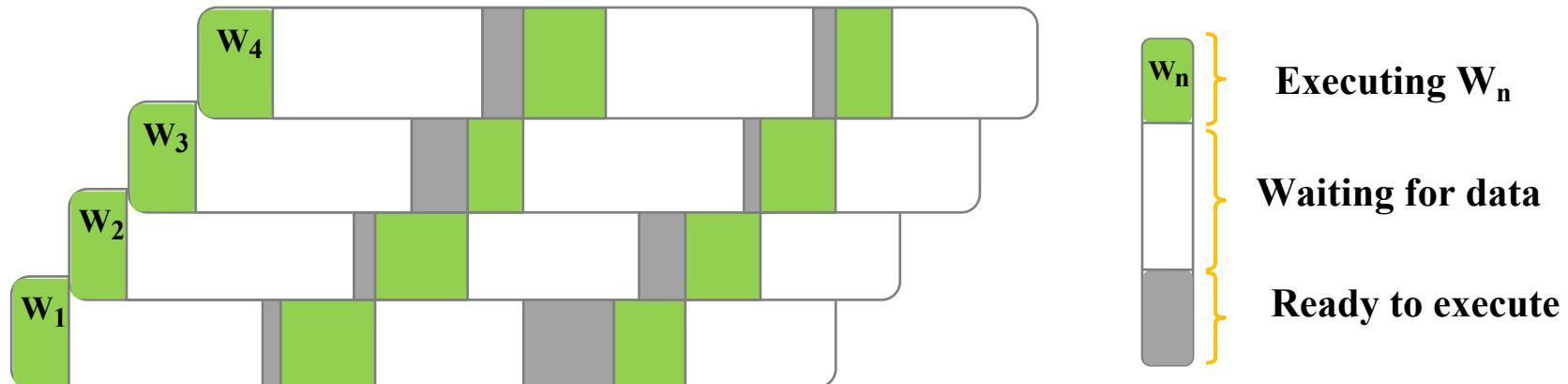


- The hardware schedules each warp independently
- Warps within a thread block execute independently

# Why many warps in flight is a good thing

- An SM can switch between warps with almost no overhead
- Warps with instruction whose inputs are ready are eligible to execute, and will be considered when scheduling
- When a warp is selected for execution, all [active] threads execute the same instruction in **lockstep** fashion

**(pre-Volta; not anymore for Volta & Turing)**



# Execution Configuration, revisiting the concept

- Prefer thread block sizes that result in mostly full warps

**Bad:** `kernel<<<N, 1>>> ( ... )`

**Okay:** `kernel<<<(N+31) / 32, 32>>>( ... )`

**Better:** `kernel<<<(N+127) / 128, 128>>>( ... )`

- Many threads per block provide SM with many warps to switch between
  - This is how the GPU hides memory access latency
- Resource like amount of ShMem & size of RegFile may constrain number of threads per block

# Scheduling: Summing It Up...

- When host invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity
- Number of resident blocks on one SM
  - Up to 8 blocks (on Fermi)
  - Up to 16 on Kepler & Turing
  - Up to 32 on Maxwell, Pascal, and Volta
- When a block of threads is executed on an SM, its threads are grouped in warps. The SM manages several warps at the same time
  - Up to 48 warps can be managed by an SM on Fermi
  - Up to 64 warps can be managed on Kepler, Maxwell, Pascal, and Volta
  - Up to 32 warps can be managed on Turing
- When a thread block finishes, a new block is launched on the vacated SM

# Thread Divergence [pre-Volta]

# Thread divergence issues

- Consider the following code:

```
__global__ void odd_even(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( (i & 0x01) == 0 )
    {
        x[i] = x[i] + 1;
    }
    else
    {
        x[i] = x[i] + 2;
    }
}
```

- Half the threads (even *i*) in the warp execute the **if** clause, the other half (odd *i*) the **else** clause

# Thread Divergence in “if-then-else”

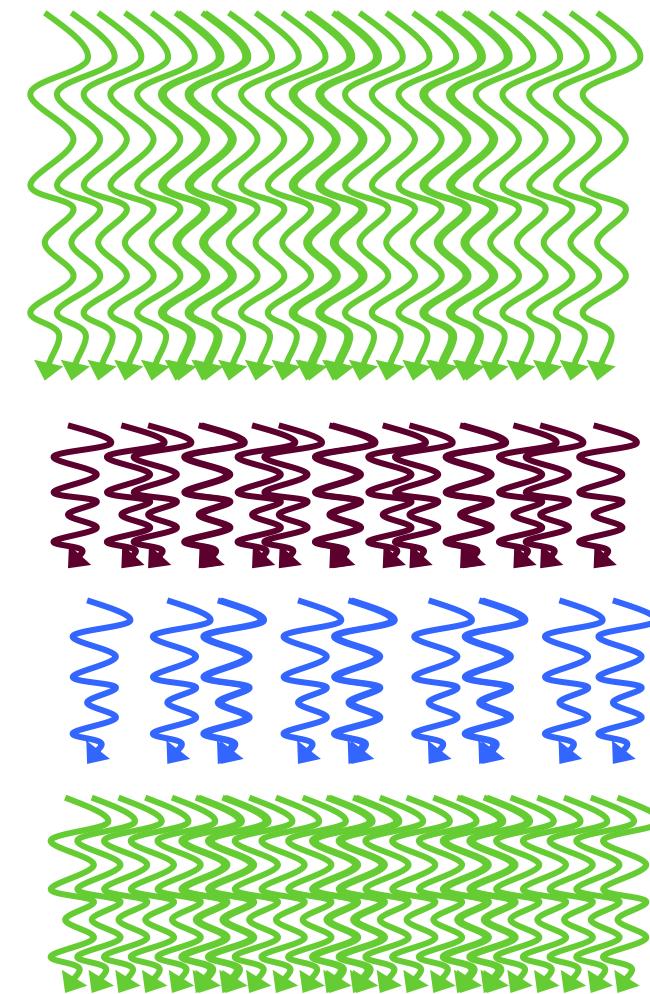
- Handling of an **if-then-else** construct in CUDA
  - First, a subset of threads of the warp execute one branch
    - Execution moves forward for these threads in lockstep fashion
  - Next, the rest of the threads in the warp execute the other branch
    - Execution moves forward for these threads in lockstep fashion
- After the **if-then-else** statement all threads of the warp move on in lockstep fashion

# Thread divergence issues

- The system automatically handles **control flow divergence**, conditions in which threads within a warp execute different paths through a kernel
- Often, this requires that the hardware execute multiple paths through a kernel for a warp
  - For example, both the **if** clause and the corresponding **else** clause

# Thread divergence issues

```
__global__ void kv(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int t;
    bool b = f(x[i]);
    if( b )
    {
        t = g(x[i]); // g(x)
    }
    else
    {
        t = h(x[i])); // h(x)
    }
    y[i] = t;
}
```



# Thread divergence issues

- Nested branches are handled similarly
  - Deeper nesting results in more threads being temporarily disabled
- In general, one does not need to consider divergence when reasoning about the correctness of a program\*
  - Certain code constructs, such as those involving schemes in which threads within a warp spin-wait on a lock, can cause deadlock
- In general, one does need to consider divergence when reasoning about the performance of a program
- NVIDIA calls execution model **SIMT** (Single Instruction Multiple Threads) to differentiate from actual SIMD where threads really are in lockstep

# Performance of Divergent Code (1/2)

- Performance decreases with degree of divergence in warps
- Here's an extreme example...

```
__global__ void dv(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    switch (i % 32)
    {
        case 0 : x[i] = a(x[i]);
        break;
        case 1 : x[i] = b(x[i]);
        break;
        ...
        case 31: x[i] = v(x[i]);
        break;
    }
}
```

# Performance of Divergent Code (2/2)

- Compiler and hardware can detect when **all threads in a warp** branch in the same direction
  - Example: all take the **if** clause, or all take the **else** clause
  - The hardware is optimized to handle these cases without loss of performance

```
if (threadIdx.x / WARP_SIZE >= 2) { }
```

- Creates two different control paths for threads in a block
- Branch granularity is a whole multiple of warp size; all threads in any warp follow the same path. No warp divergence...

- The compiler can also compile short conditional clauses to use predicates (bits that conditionally convert instructions into null ops)
  - Avoids some branch divergence overheads and is more efficient
  - Often acceptable performance with short conditional clauses

- See forum post here:
- <https://devtalk.nvidia.com/default/topic/1031723/cuda-programming-and-performance/warp-divergence-triggered-by-for-loop/post/5249002/?offset=3#5249003>

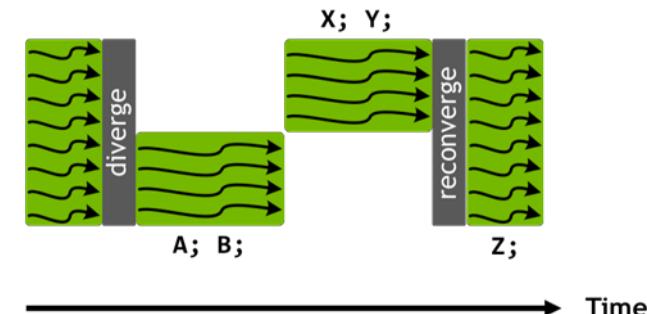
# Thread divergence, VOLTA and TURING

- Volta GV100 is the first GPU to support independent thread scheduling, which enables finer-grain synchronization and cooperation between parallel threads in a program
- Consequence
  - Greater flexibility in thread cooperation, leading to higher efficiency for fine-grained parallel algorithms

# Handling thread divergence, pre-Volta (Pascal and before)

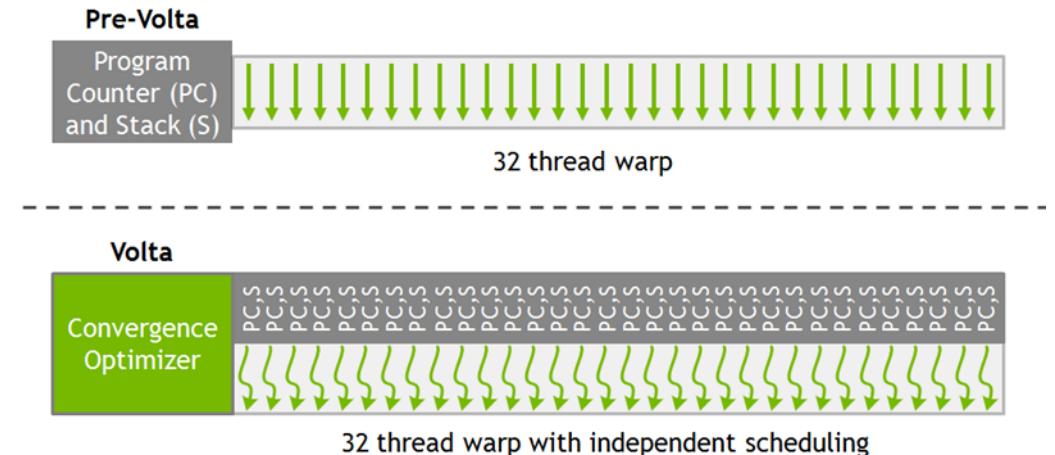
- Pascal and earlier NVIDIA GPUs execute groups of 32 threads (warps) in SIMD
- The Pascal warp uses a single program counter shared amongst all 32 threads, combined with an active mask that specifies which threads of the warp are active at any given time
- Loss of concurrency means that threads from the same warp in divergent regions or different states of execution cannot signal each other or exchange data
  - This is because some threads just sit and do nothing, they're not active

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```



# SIMT Warp Execution Model of Pascal and Earlier GPUs

- Volta enables equal concurrency between all threads, regardless of warp
- Volta maintains execution state (PC and S) per thread, see pic at the right
- Pascal and before: one PC and S per warp



- PC: program counter
- S: call stack

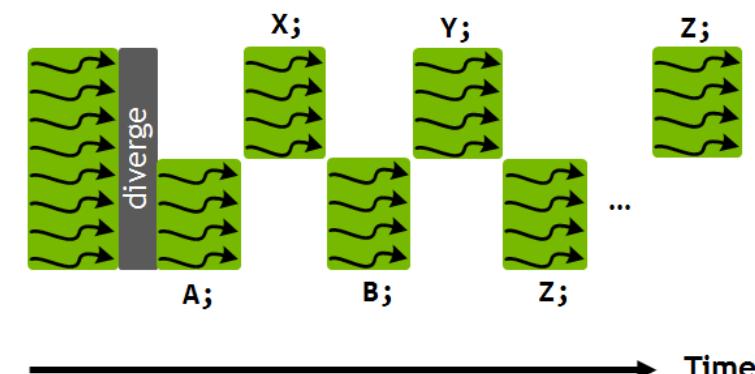
# The schedule optimizer

- To maximize parallel efficiency, Volta includes a schedule optimizer which determines how to group active threads from the same warp together into SIMT units
- Threads can now diverge and re-converge at sub-warp granularity
- While the scheduler supports independent execution of threads, it optimizes non-synchronizing code to maintain as much convergence as possible
  - The “convergence optimizer” will still group together threads which are executing the same code and run them in parallel
    - Flow control simpler, might lead to better efficiency?

# Handling thread divergence, Volta and Turing

- Instructions from the **if** and **else** branches can now be interleaved in time, as shown in the pic.
  - NOTE: what's shown in the pic is one possibility. How the scheduling takes place depends on the code being executed and be different than what shown in the pic
- **IMPORTANT:** execution is still SIMT – at any given clock cycle, the same instruction is executed for all **active** threads in a warp *just as before Volta*
- Why bother with understanding this?
  - Ability to independently schedule threads within a warp makes it possible to implement complex, fine-grained algorithms and data structures in a more natural way

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```



# Example: Volta thread scheduling helps algorithm implementation

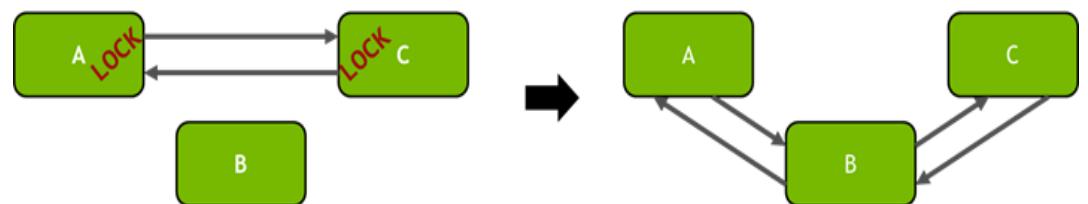
- “Starvation-free algorithm” – a parallel algorithm that works on the presumption that a thread that participates in the implementation of the algorithm cannot be starved
- NVIDIA GPUs prior to Volta: didn’t support starvation avoidance
  - Indeed, one or more threads may repeatedly acquire and release a mutex while starving another thread from ever successfully acquiring the mutex
- Example at right shows how Volta independent thread scheduling implements a starvation-free algorithm
  - Inserting nodes into a doubly linked list in a multithreaded application

```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

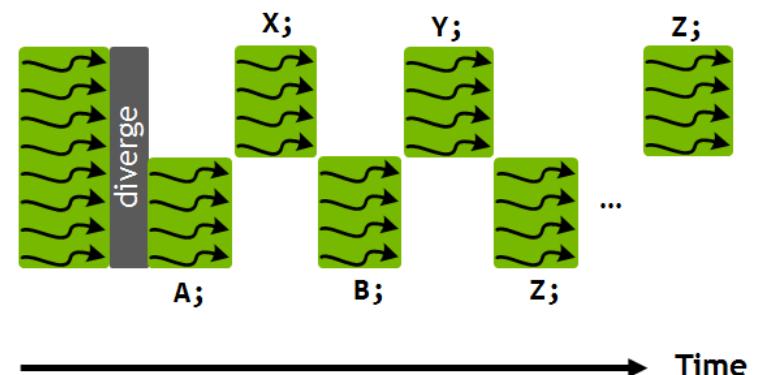


Per-node locks are acquired (left) before inserting node B into the list (right).

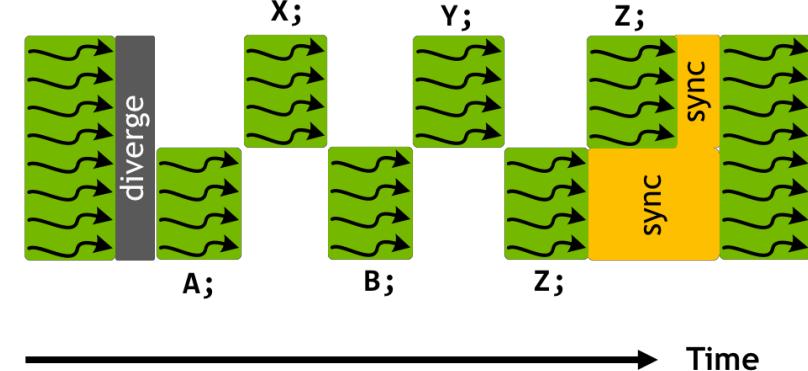
# Handling thread divergence, Volta and Turing

- Note that instruction Z is not executed by the same warp threads at the same time
- DN: In the “white blocks”, it might be that the scheduler executes other threads? Not clear
- Call CUDA 9 warp synchronization function `syncwarp()` to force reconvergence (lower pic)
  - The divergent portions of the warp might not execute Z together, but all execution pathways from threads within a warp will complete before any thread reaches the statement after the `syncwarp()`
- Placing the call to `syncwarp()` before the execution of Z would force reconvergence before executing Z, potentially enabling greater SIMT efficiency if the developer knows that this is safe for their application

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```



```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;  
__syncwarp()
```



# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 11

02/17/2020

# Quote of the day

**“Computer science is no more about computers than astronomy is about telescopes”**

-- Edsger W. Dijkstra, Dutch systems scientist, programmer, software engineer, science essayist, and pioneer in computing science [ [1930-2002].

# Before we get going...

- Last time:
  - More on execution configuration
  - GPU “scheduling for execution” issues
- Today:
  - Wrap up, scheduling for execution on the GPU
  - The GPU memory ecosystem
- Other tidbits:
  - Assignment due on **Friday** at 9 pm
  - Anonymous feedback:  
<https://forms.gle/ZUCbyKX87mJvK61M7>



# Execution Scheduling Issues

# [short/important topic]: Thread Index vs. Thread ID

[critical in (i) understanding how SIMD is supported in CUDA, and (ii) understanding the concept of “warp”]

- Each block organizes its threads in a 3D structure defined by its three dimensions:  $D_x$ ,  $D_y$ , and  $D_z$  that you specify.
- A block cannot have more than 1024 threads  $\Rightarrow D_x \times D_y \times D_z \leq 1024$ .
- Each thread in a block can be identified by a unique index  $(x, y, z)$ , and

$$0 \leq x < D_x \quad 0 \leq y < D_y \quad 0 \leq z < D_z$$

- A triplet  $(x, y, z)$ , called the thread index, is a high-level representation of a thread in the economy of a block. Under the hood, the same thread has a simplified and unique id, which is computed as  $t_{id} = x + y * D_x + z * D_x * D_y$ . You can regard this as a “projection” to a 1D representation. The concept of thread id is important in understanding how threads are grouped together in warps (more on “warps” later).
- In general, operating for vectors typically results in you choosing  $D_y = D_z = 1$ . Handling matrices typically goes well with  $D_z = 1$ . For handling PDEs in 3D you might want to have all three block dimensions nonzero.

# Thread Execution Scheduling

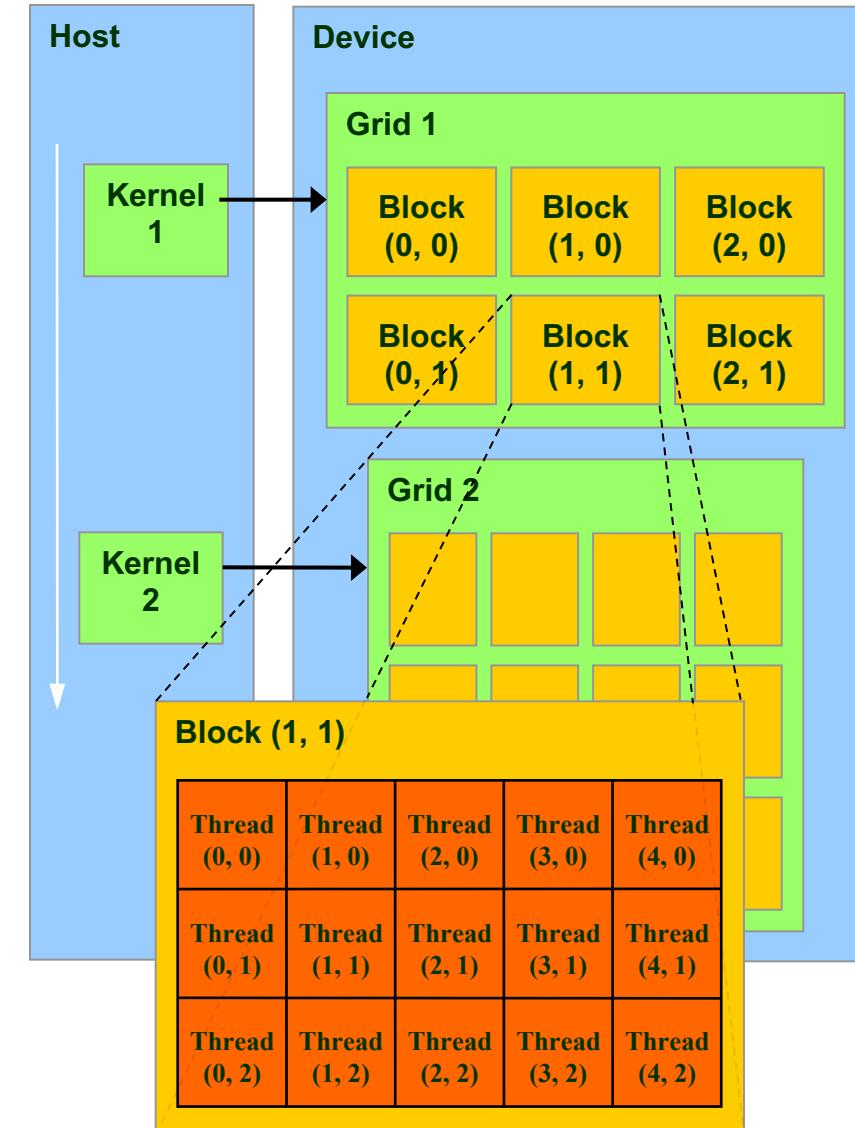
- Starting point observation:
  - You launch on the device many blocks, each containing many threads
- Questions in this segment:
  - In which order are these blocks executed?
  - How is this execution process managed?
  - When/How are the threads in a block executed?
  - Etc...

# High-level Perspective

- There are two schedulers at work in GPU computing
  - A **device-level** scheduler: assigns blocks to SM that indicate at a given time “excess capacity”
    - This scheduler called by NVIDIA the “GigaThread engine”
  - An **SM-level** scheduler: schedules the execution of the threads in a block onto the SM functional units
    - This is the more interesting scheduler

# Device-Level Scheduler

- Grid is launched on the device
- Thread Blocks are distributed to the SMs
  - Potentially more than one block per SM
  - There is a limit on the number of blocks an SM can take.
- As Thread Blocks complete kernel execution, resources are freed  
Device-level scheduler can launch next block(s) in line
- This is the first levels of scheduling:  
For running [desirably] a large number of blocks (thousands) on a relatively small number of SMs (60/16/14/etc.)
- Limits, for **resident** blocks/SM:
  - 32 blocks on the Maxwell SM, Pascal SM, and Volta SM
  - 16 blocks can be resident on a Kepler SM
  - 8 blocks can be resident on a Fermi & Tesla SM



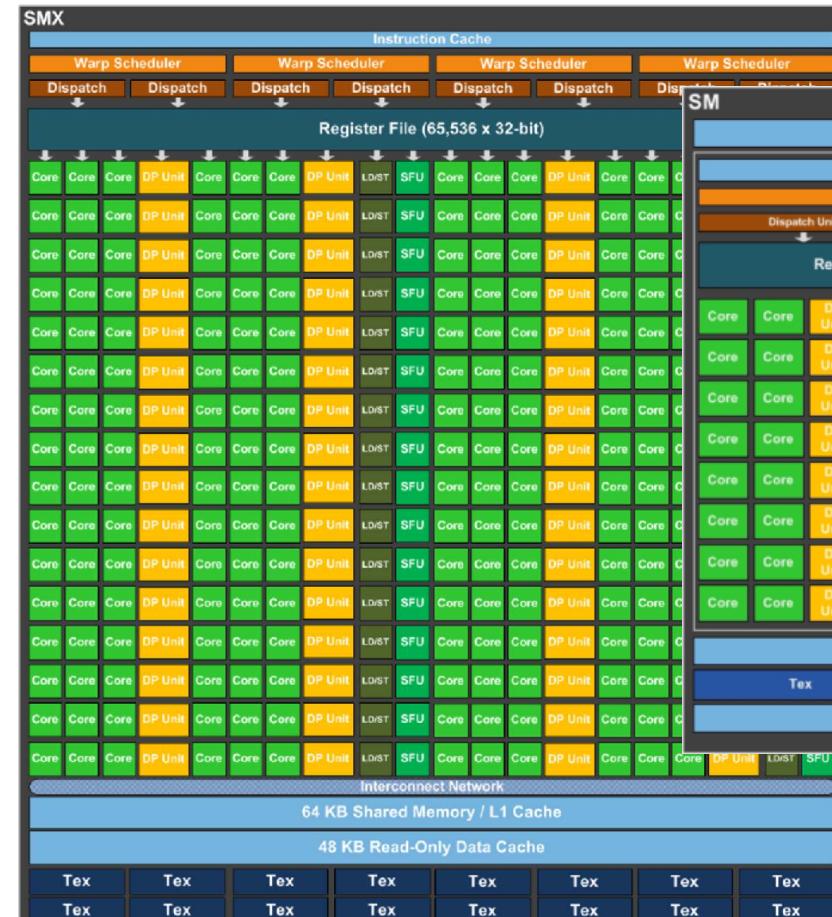
# Device-Level Scheduler, Comments

- Once a block is picked up for execution by one SM, the block does not leave that SM before all threads in that block finish executing the kernel
- Once a block is finished & retired, only then can another block land for execution on that SM
  - The Device-Level Scheduler dispatches the next block in line to the SM that indicates that it has excess capacity
- This explains why it's good to have many SM in your GPU
  - More SMs → more expensive card

# More Recent SM Generations

Volta SM

Maxwell SMX



Pascal SM



Motivated by AI & Machine Learning



# SM-Level Schedulers

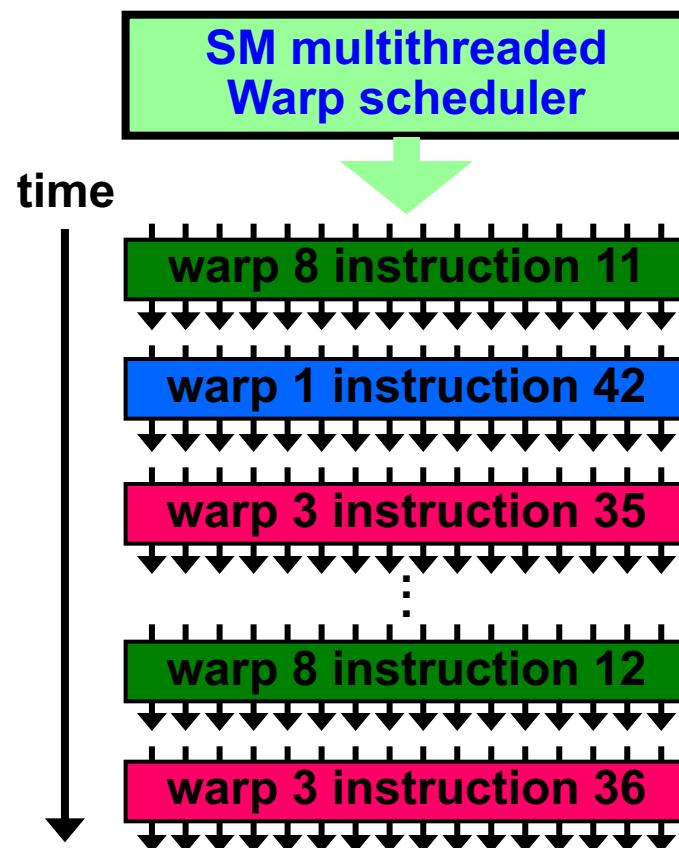
- Each block of threads divided in 32-thread **warps**
  - “32”: selected by NVIDIA, programmer has no say
  - Warp: A group of 32 threads of consecutive IDs
- Warps are the basic scheduling unit on the SM
- Limits, number of resident warps on an SM:
  - 64: on Kepler, Maxwell, Pascal, Volta (i.e., 2048 resident threads)
  - 48: on Fermi (i.e., 1536 resident threads)
  - 32: on Tesla (i.e., 1024 resident threads)



# Quiz

- If 3 blocks are processed by an SM and each Block has 256 threads, how many warps are managed by the SM?
  - Each block of threads is divided into  $256/32 = 8$  warps
  - There are  $8 * 3 = 24$  warps
  - At any point in time, there are 24 warps that can be selected for execution

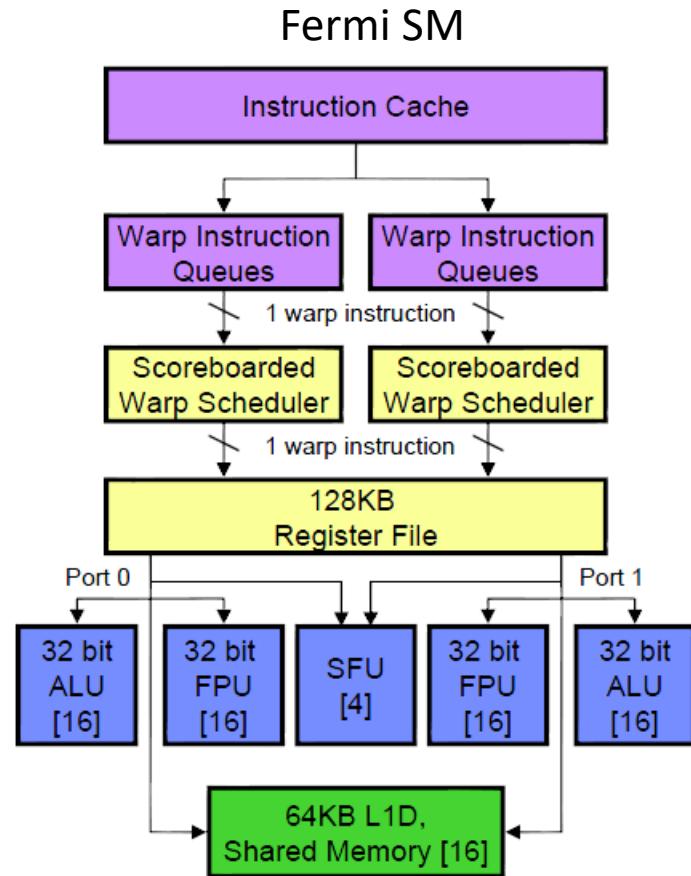
# SM Warp Scheduling, more details



- SM hardware implements almost zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute same instruction

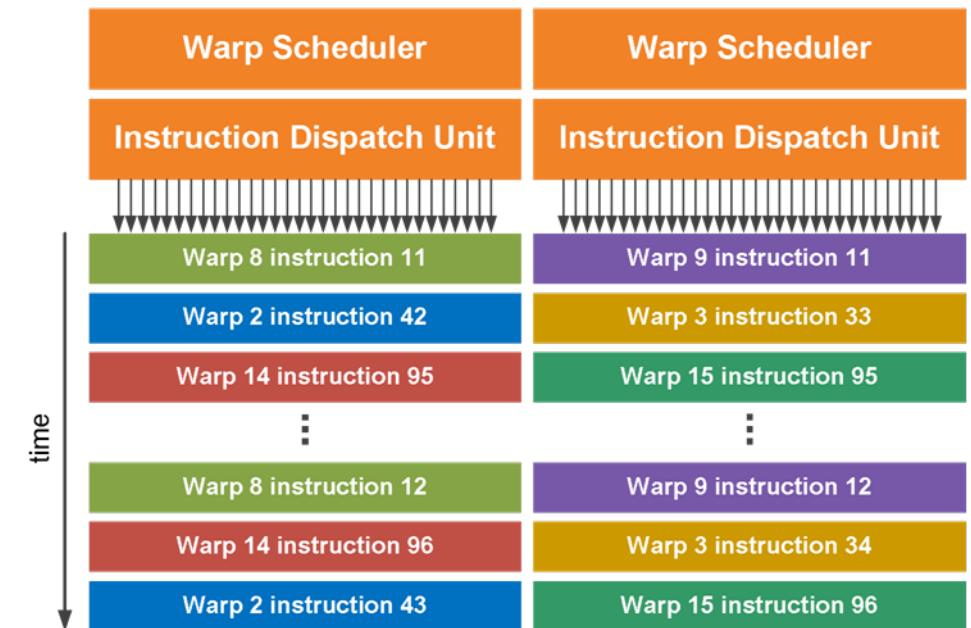
# Fermi Specifics

- There are two schedulers that issue warps of “ready-to-go” threads
- One warp issued at each clock cycle by each scheduler
- During no cycle can more than 2 warps be dispatched for execution on the SM’s functional units
- Scoreboarding used to figure out which warp is ready



# Quick Remarks

- ILP support: relatively limited
- GPU architecture is pipelined
  - At each clock cycle one instruction can be retired
- Fermi has no out-of-order execution smarts
- To the best of my knowledge
  - There is no prefetching
  - There is no speculative execution



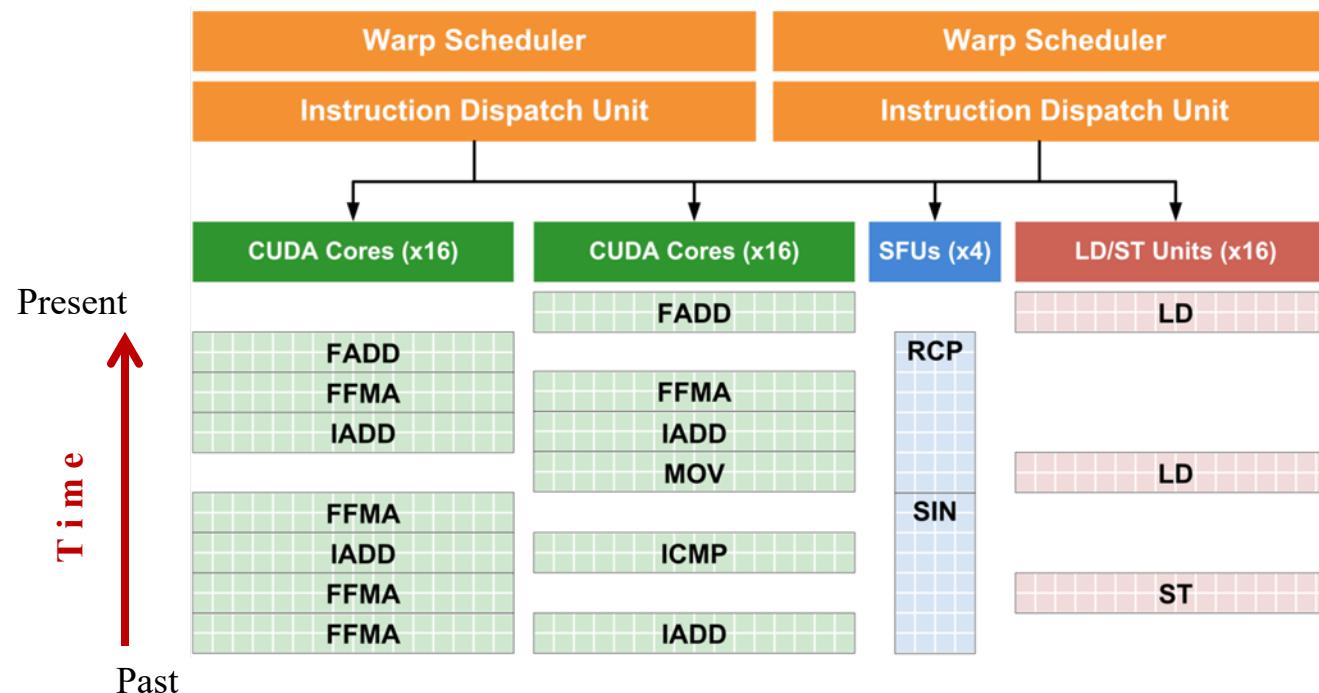
# Example, establishing Flop/s performance level: Fermi

- Scheduler works at 607 MHz
- Functional units work at 1215MHz
- Question:
  - What is the peak flop rate of GTX480?
  - $15 \text{ SMs} * 32 \text{ SPs} * 1215 * 2 \text{ (Fused Multiplied Add)} = 1166400 \text{ Mflops}$
  - That is, 1.166 Tflops, single precision



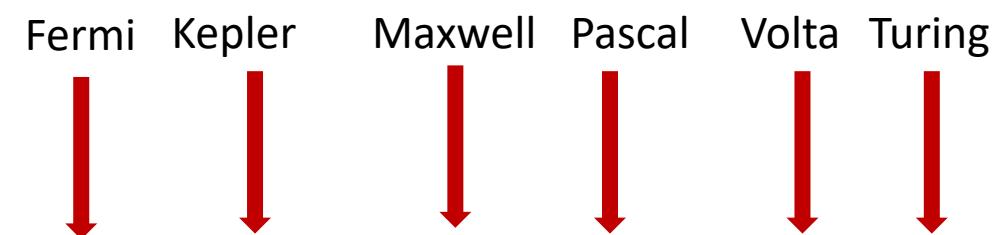
# Fermi Specifics

- As shown in picture, at no time can we see more than 2 warps being dispatched for execution during a cycle
- Note that at any given time we might have more than two functional units working though (which is actually very good, device kept busy)



# SM Architecture Specifications (data below refers to ONE SM)

Fermi   Kepler   Maxwell   Pascal   Volta   Turing



Architecture specifications	Compute capability (version)														
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5	3.7	5.0	5.2	6.0	6.1, 6.2	7.0, 7.2	7.5
Number of ALU lanes for integer and single-precision floating-point arithmetic operations	8				32	48	192		128	64	128		64		
Number of special function units for single-precision floating-point transcendental functions	2				4	8	32			16	32		16		
Number of texture filtering units for every texture address unit or <i>render output unit</i> (ROP)	2				4	8	16					8			
Number of warp schedulers	1				2		4			2		4			
Max number of instructions issued at once by a single scheduler	1				2		2			1					
Number of tensor cores							N/A					8			
Size in KB of unified memory for data cache and shared memory per multi processor	t.b.d.											128	96		

# Technical Specifications and Features

Technical specifications	Compute capability (version)																															
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0 (7.2?)	7.5															
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4	32				16	128	32	16	128																
Maximum dimensionality of grid of thread blocks	2				3																											
Maximum x-dimension of a grid of thread blocks	65535				$2^{31} - 1$																											
Maximum y-, or z-dimension of a grid of thread blocks	65535																															
Maximum dimensionality of thread block	3																															
Maximum x- or y-dimension of a block	512				1024																											
Maximum z-dimension of a block	64																															
Maximum number of threads per block	512				1024																											
Warp size	32																															
Maximum number of resident blocks per multiprocessor	8				16				32				16																			
Maximum number of resident warps per multiprocessor	24	32	48	64												32																
Maximum number of resident threads per multiprocessor	768	1024	1536	2048												1024																

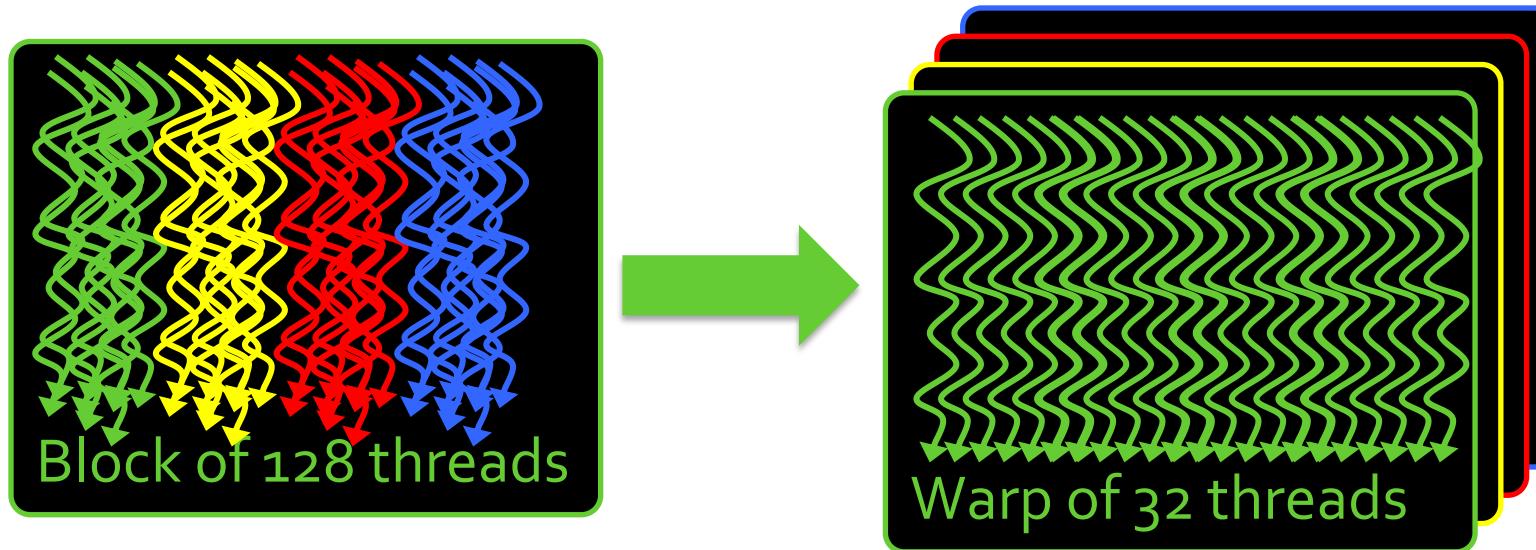


# Organizing Threads into Warps

- Thread IDs within a warp are **consecutive** and **increasing**
  - Related to the 1D projection from thread index to thread ID
  - Recall: in multidimensional blocks, the **x** thread index runs fastest, then **y**, then **z**
  - Threads with IDs (0...31) combine into warp 0, threads (32...63) into warp 1, etc.
- Partitioning of threads into warps is always the **same**
  - You can use this in control flow
  - Warp size has always been 32 and is unlikely to change soon
- While you can rely on ordering among threads, **DO NOT** rely on any ordering among warps
  - Warp scheduling is not under user control in CUDA

# Threads are Organized & Executed as Warps

- Each thread block split into one or more warps
- If thread block size is not multiple of warp size, unused lanes go wasted
  - Example: block w/ 50 threads – 64 lanes, of which last 14 lanes go wasted

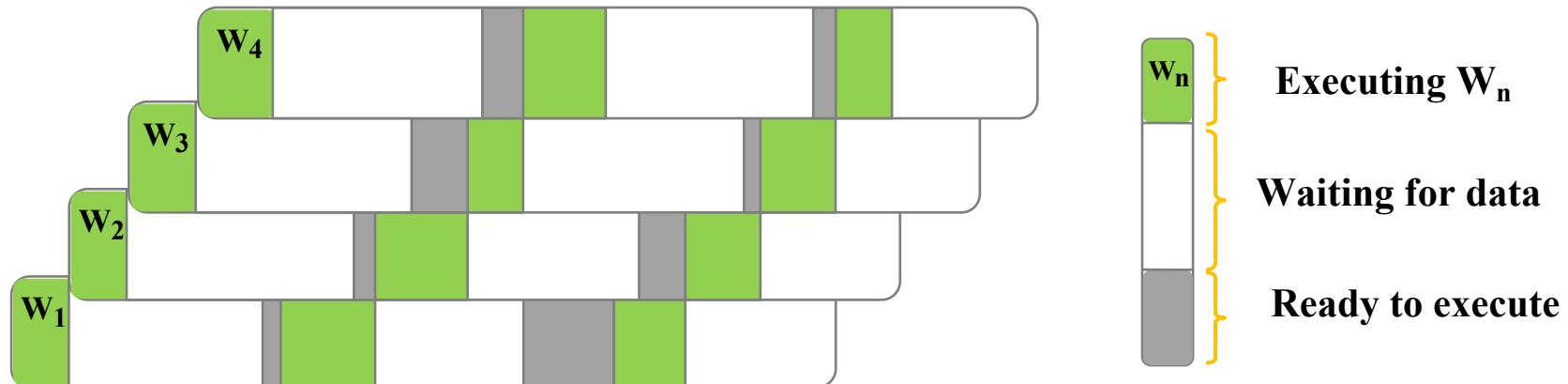


- The hardware schedules each warp independently
- Warps within a thread block execute independently

# Why many warps in flight is a good thing

- An SM can switch between warps with almost no overhead
- Warps with instruction whose inputs are ready are eligible to execute, and will be considered when scheduling
- When a warp is selected for execution, all [active] threads execute the same instruction in **lockstep** fashion

**(pre-Volta; not anymore for Volta & Turing)**



# Execution Configuration, revisiting the concept

- Prefer thread block sizes that result in mostly full warps

**Bad:** `kernel<<<N, 1>>> ( ... )`

**Okay:** `kernel<<<(N+31) / 32, 32>>>( ... )`

**Better:** `kernel<<<(N+127) / 128, 128>>>( ... )`

- Many threads per block provide SM with many warps to switch between
  - This is how the GPU hides memory access latency
- Resource like amount of ShMem & size of RegFile may constrain number of threads per block

# Scheduling: Summing It Up...

- When host invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity
- Number of resident blocks on one SM
  - Up to 8 blocks (on Fermi)
  - Up to 16 on Kepler & Turing
  - Up to 32 on Maxwell, Pascal, and Volta
- When a block of threads is executed on an SM, its threads are grouped in warps. The SM manages several warps at the same time
  - Up to 48 warps can be managed by an SM on Fermi
  - Up to 64 warps can be managed on Kepler, Maxwell, Pascal, and Volta
  - Up to 32 warps can be managed on Turing
- When a thread block finishes, a new block is launched on the vacated SM

# Thread Divergence [pre-Volta]

# Thread divergence issues

- Consider the following code:

```
__global__ void odd_even(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( (i & 0x01) == 0 )
    {
        x[i] = x[i] + 1;
    }
    else
    {
        x[i] = x[i] + 2;
    }
}
```

- Half the threads (even *i*) in the warp execute the **if** clause, the other half (odd *i*) the **else** clause

# Thread Divergence in “if-then-else”

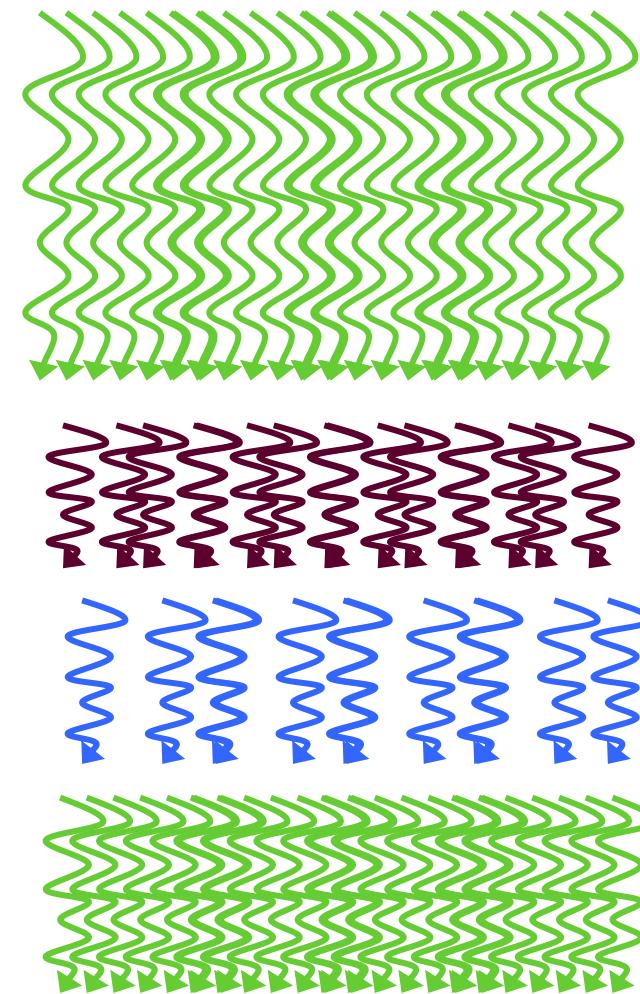
- Handling of an **if-then-else** construct in CUDA
  - First, a subset of threads of the warp execute one branch
    - Execution moves forward for these threads in lockstep fashion
  - Next, the rest of the threads in the warp execute the other branch
    - Execution moves forward for these threads in lockstep fashion
- After the **if-then-else** statement all threads of the warp move on in lockstep fashion

# Thread divergence issues

- The system automatically handles **control flow divergence**, conditions in which threads within a warp execute different paths through a kernel
- Often, this requires that the hardware execute multiple paths through a kernel for a warp
  - For example, both the **if** clause and the corresponding **else** clause

# Thread divergence issues

```
__global__ void kv(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int t;
    bool b = f(x[i]);
    if( b )
    {
        t = g(x[i]); // g(x)
    }
    else
    {
        t = h(x[i])); // h(x)
    }
    y[i] = t;
}
```



# Thread divergence issues

- Nested branches are handled similarly
  - Deeper nesting results in more threads being temporarily disabled
- In general, one does not need to consider divergence when reasoning about the correctness of a program\*
  - Certain code constructs, such as those involving schemes in which threads within a warp spin-wait on a lock, can cause deadlock
- In general, one does need to consider divergence when reasoning about the performance of a program
- NVIDIA calls execution model **SIMT** (Single Instruction Multiple Threads) to differentiate from actual SIMD where threads really are in lockstep

# Performance of Divergent Code (1/2)

- Performance decreases with degree of divergence in warps
- Here's an extreme example...

```
__global__ void dv(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    switch (i % 32)
    {
        case 0 : x[i] = a(x[i]);
                    break;
        case 1 : x[i] = b(x[i]);
                    break;
        ...
        case 31: x[i] = v(x[i]);
                    break;
    }
}
```

# Performance of Divergent Code (2/2)

- Compiler and hardware can detect when **all threads in a warp** branch in the same direction
  - Example: all take the **if** clause, or all take the **else** clause
  - The hardware is optimized to handle these cases without loss of performance

```
if (threadIdx.x / WARP_SIZE >= 2) { }
```

- Creates two different control paths for threads in a block
- Branch granularity is a whole multiple of warp size; all threads in any warp follow the same path. No warp divergence...

- The compiler can also compile short conditional clauses to use predicates (bits that conditionally convert instructions into null ops)
  - Avoids some branch divergence overheads and is more efficient
  - Often acceptable performance with short conditional clauses

- See forum post here:
- <https://devtalk.nvidia.com/default/topic/1031723/cuda-programming-and-performance/warp-divergence-triggered-by-for-loop/post/5249002/?offset=3#5249003>

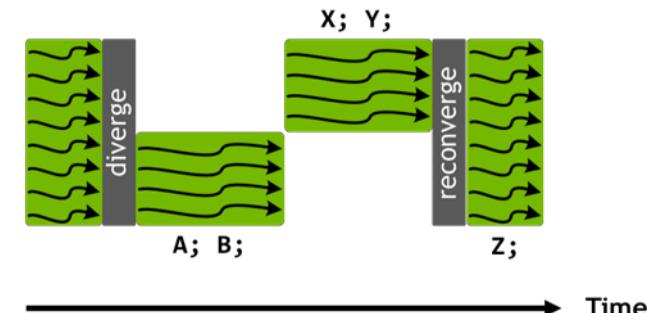
# Thread divergence, VOLTA and TURING

- Volta GV100 is the first GPU to support independent thread scheduling, which enables finer-grain synchronization and cooperation between parallel threads in a program
- Consequence
  - Greater flexibility in thread cooperation, leading to higher efficiency for fine-grained parallel algorithms

# Handling thread divergence, pre-Volta (Pascal and before)

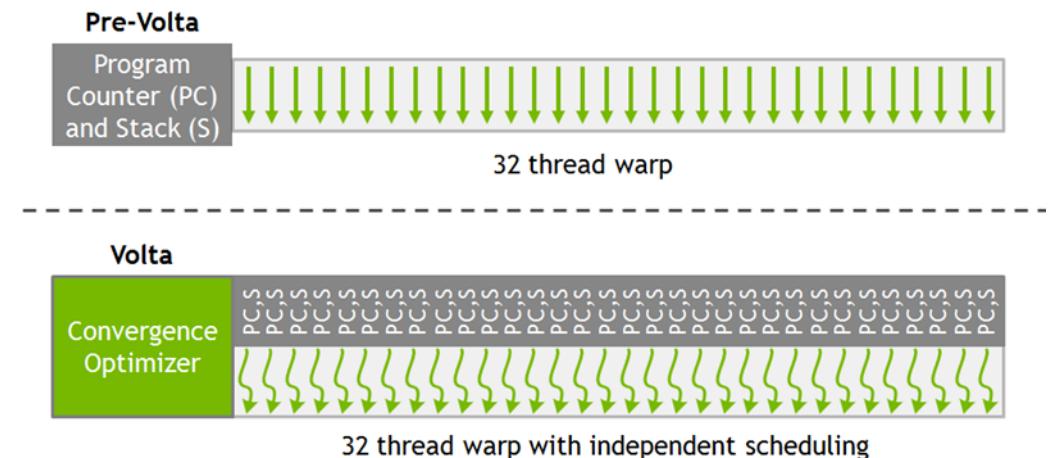
- Pascal and earlier NVIDIA GPUs execute groups of 32 threads (warps) in SIMD
- The Pascal warp uses a single program counter shared amongst all 32 threads, combined with an active mask that specifies which threads of the warp are active at any given time
- Loss of concurrency means that threads from the same warp in divergent regions or different states of execution cannot signal each other or exchange data
  - This is because some threads just sit and do nothing, they're not active

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```



# Warp Execution Model of Volta and later GPUs

- Volta enables equal concurrency between all threads, regardless of warp
- Volta maintains execution state (PC and S) per thread, see pic at the right
- Pascal and before: one PC and S per warp



- PC: program counter
- S: call stack

# The schedule optimizer

- To maximize parallel efficiency, Volta includes a schedule optimizer which determines how to group active threads from the same warp together into SIMT units
- Threads can now diverge and re-converge at sub-warp granularity
- While the scheduler supports independent execution of threads, it optimizes non-synchronizing code to maintain as much convergence as possible
  - “convergence optimizer” still groups together threads that execute the same code to run them in parallel

# Handling thread divergence, Volta and Turing

- Instructions from the **if** and **else** branches can now be interleaved in time, as shown in the pic.

- NOTE: what's shown in the pic is one possibility. How the scheduling takes place depends on the code being executed and be different than what shown in the pic

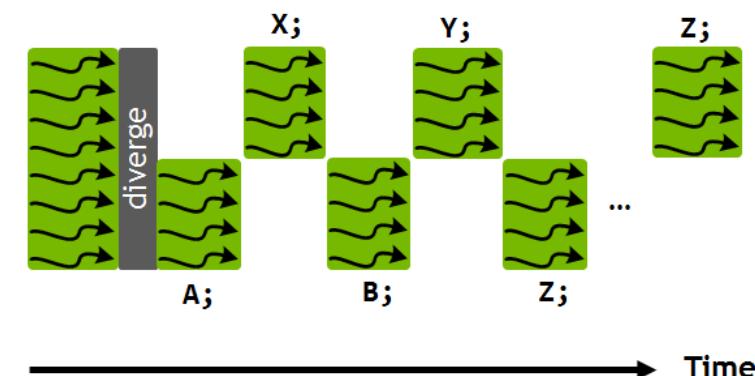
- IMPORTANT:** execution is still SIMT – at any given clock cycle, the same instruction is executed for all **active** threads in a warp *just as before Volta*

- NOTE: this is why the “schedule optimizer” tries to converge all threads
    - If all converged, the 32 threads get to execute an instruction. Otherwise, only a subset of threads get to execute an instruction

- Why bother with understanding this?

- Ability to independently schedule threads within a warp makes it possible to implement complex, fine-grained algorithms and data structures in a more natural way

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```



# Example: Volta thread scheduling helps algorithm implementation

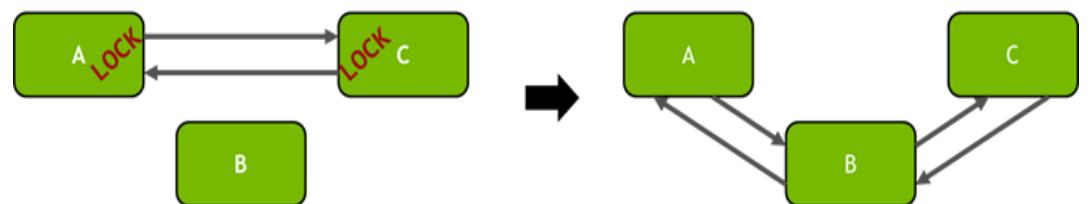
- “Starvation-free algorithm” – a parallel algorithm that works on the presumption that a thread that participates in the implementation of the algorithm cannot be starved
- NVIDIA GPUs prior to Volta: didn’t support starvation avoidance
  - Indeed, one or more threads may repeatedly acquire and release a mutex while starving another thread from ever successfully acquiring the mutex
- Example at right shows how Volta independent thread scheduling implements a starvation-free algorithm
  - Inserting nodes into a doubly linked list in a multithreaded application

```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```



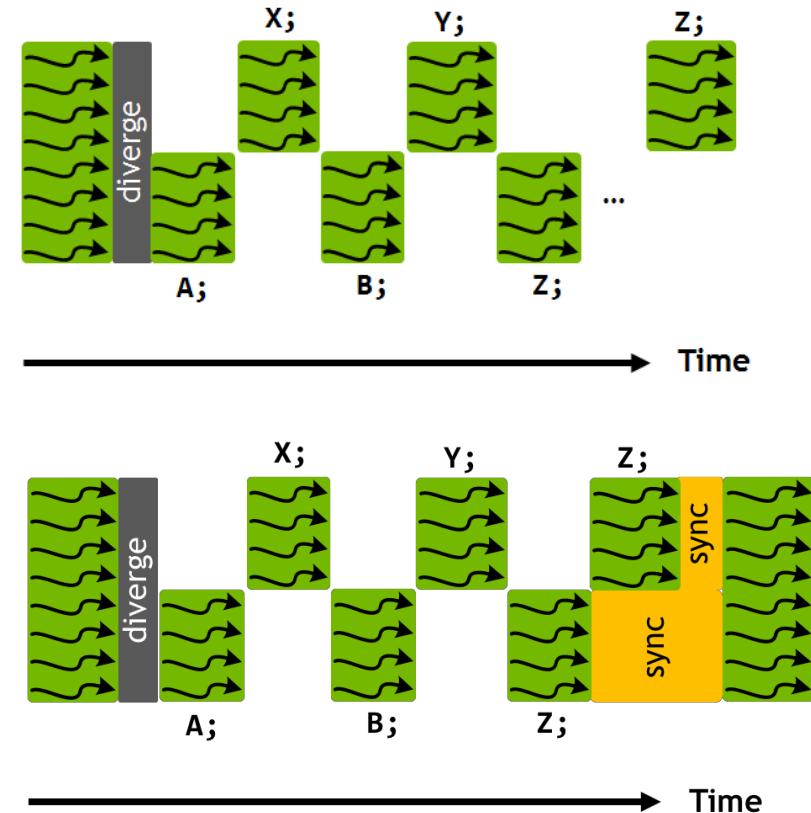
Per-node locks are acquired (left) before inserting node B into the list (right).

# Handling thread divergence, Volta and Turing

- Note that instruction Z is not executed by the same warp threads at the same time
- Call CUDA 9 warp synchronization function `syncwarp()` to force reconvergence (lower pic)
  - The divergent portions of the warp might not execute Z together, but all execution pathways from threads within a warp will complete before any thread reaches the statement after the `syncwarp()`
- Placing the call to `syncwarp()` before the execution of Z would force reconvergence before executing Z, potentially enabling greater SIMT efficiency if the developer knows that this is safe for their application

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```

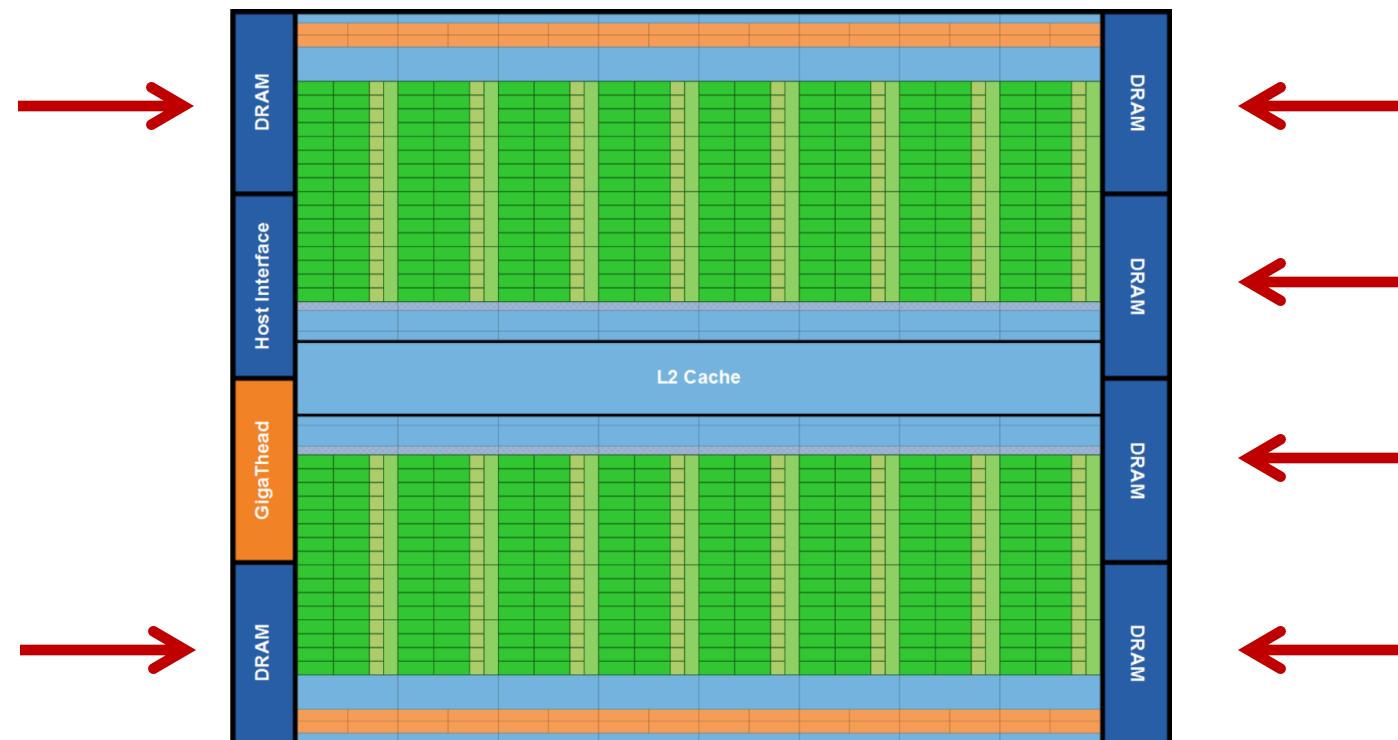
```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;  
__syncwarp()
```



# The NVIDIA GPU Memory Ecosystem

# Fermi: Global Memory

- Up to 6 GB of “global memory”, Kepler goes up to 12 GB, Pascal up to 16 GB, Volta up to 32 GB
- “Global” in the sense that it doesn’t belong to an SM but rather all SMs can access it

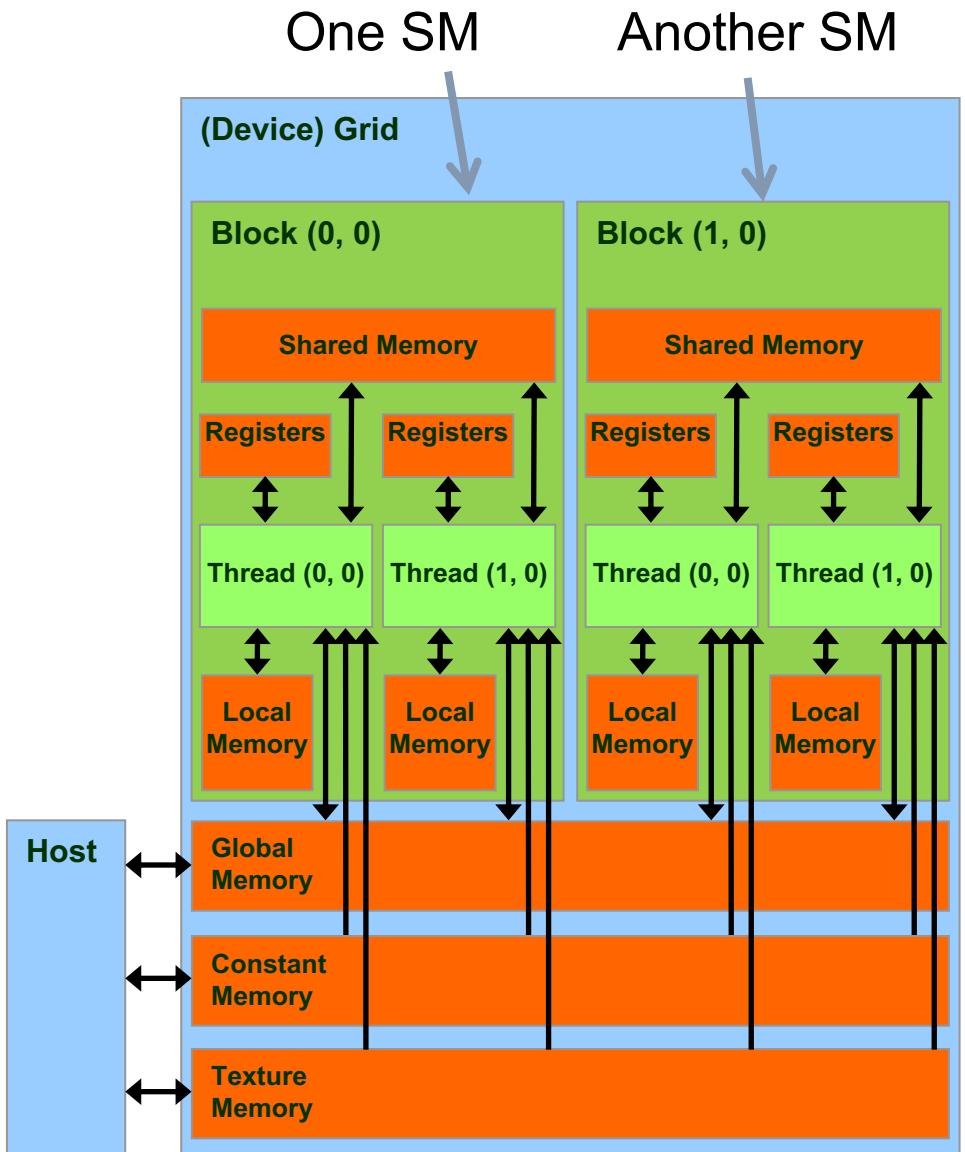


# CUDA Memory Ecosystem, high vantage point

[Note: picture assumes two blocks, each with two threads]

- Image shows the memory hierarchy that a block sees while running on an SM
- Each thread can:
  - R/W per-thread **registers**
  - R/W per-thread **local memory**
  - R/W per-block **shared memory**
  - R/W per-grid **global memory**
  - Read only per-grid **constant memory**
  - Read only per-grid **texture memory**
- Host can R/W **global**, **constant**, and **texture** memory

**IMPORTANT NOTE:** **Global**, **constant**, and **texture** memory spaces are **persistent** between kernels called by the same host application.

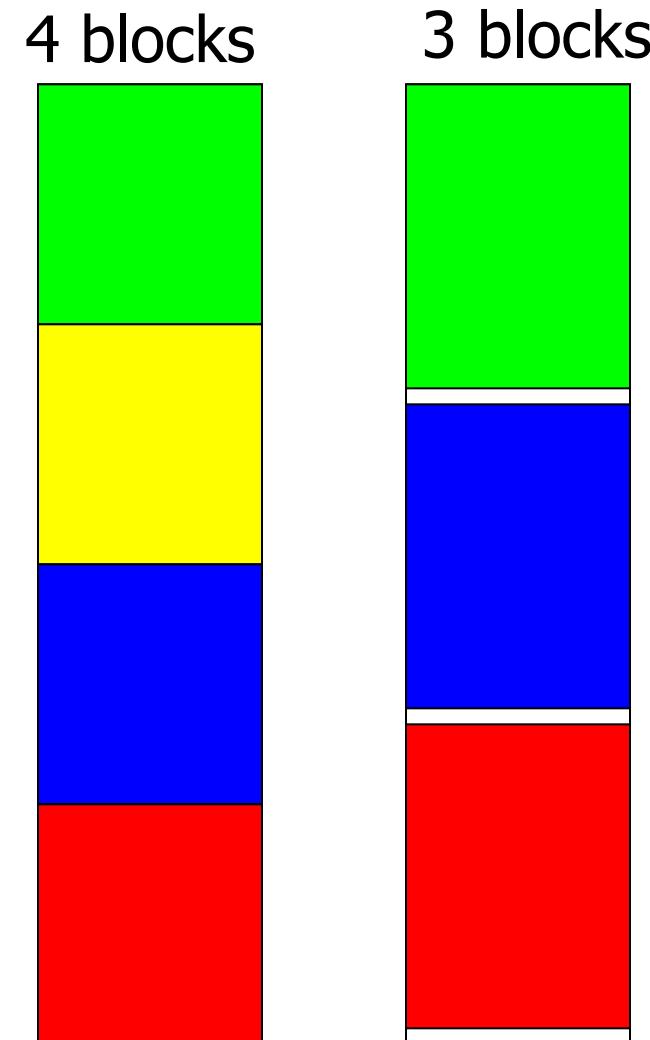


# What “register” means in CUDA

- Any variable that you are using in a kernel eats up a register
  - The variable that you use, it needs to “live” somewhere
- Then every single variable you use in a kernel will burn (at some point in the execution flow) a register
  - NOTE: even a variable stored in ShMem, for it to be operated on, it’ll eat up a register at some point in the execution
- As execution advances through a kernel, the number of registers occupied keeps changing
  - What’s relevant: max number of registers required during the execution of the kernel

# Programmer View of Register File

- Number of **32 bit** registers in one SM:
  - 8K registers in each SM in G80
  - 16K on Tesla
  - 32K on Fermi
  - 64K on Kepler and Maxwell and Pascal and Volta
- Registers are dynamically partitioned across all Blocks assigned to the SM
- Once assigned to a Block, these registers are NOT accessible by threads in other Blocks
- A thread in a Block can only access registers assigned to itself
  - **IMPORTANT:** A thread can have assigned by the compiler up to 255 registers



Possible per-block partitioning scenarios of the RF available on the SM

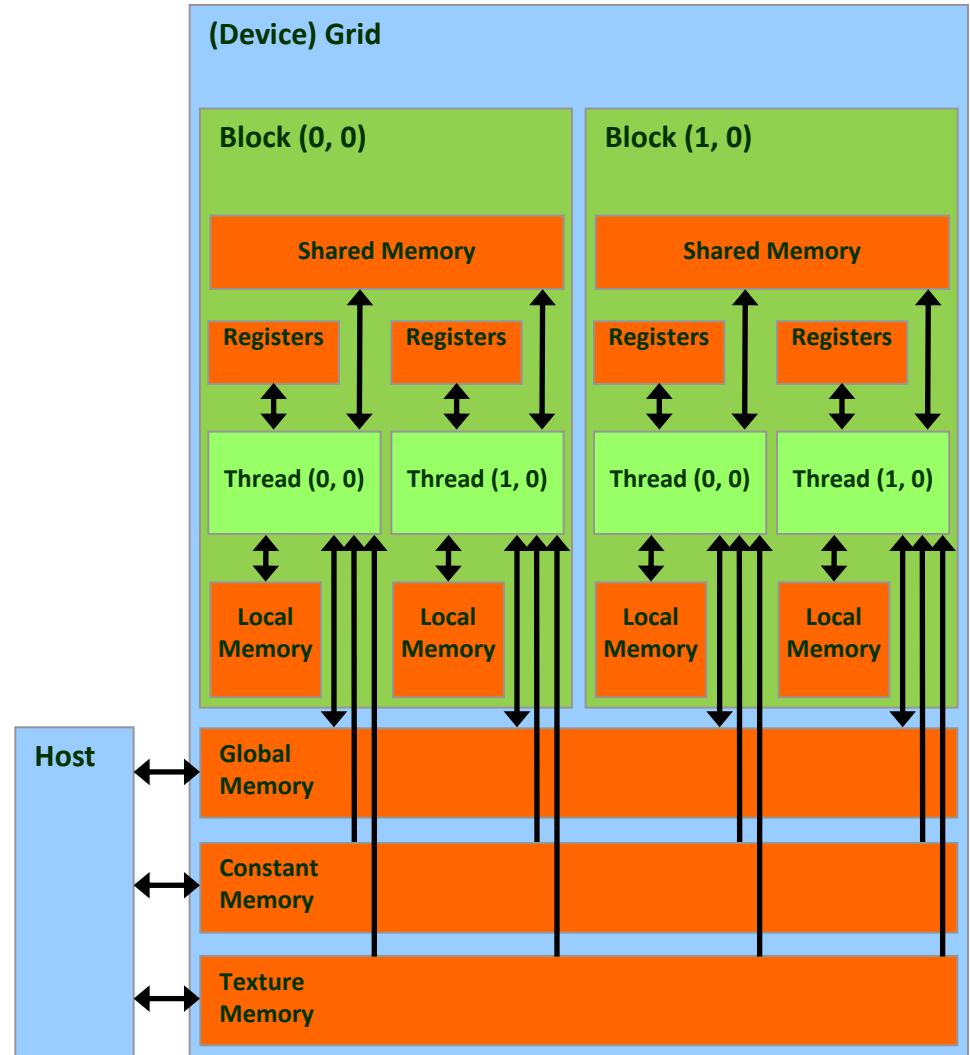
# Register related: a “level of parsimony” issue

- Size of a register: 4 bytes
- Is it that if you use a `char` in a kernel, then you don’t use a full register but rather  $\frac{1}{4}$  of it?
- Not really, unfortunately; i.e., there is **no parsimony** in usage of a register
  - Any data type that requires less than or equal to 4 bytes eats up a register
    - One `char` thus eats up an entire register
  - If you use a `double`, it eats up two registers

# The Concept of Local Memory

[“local”: misnomer]

- Local memory does not physically exist
- Data stored in “local memory” is actually placed in cache or the global memory at run time or by the compiler
  - If too many registers are needed for computation (“high register pressure”) the ensuing data overflow is stored in local memory
  - “Local” means that it’s got local scope; i.e., it’s specific to one thread and not visible to any other thread
  - Long access times for local memory unless cached



# Caches & the Shared Memory

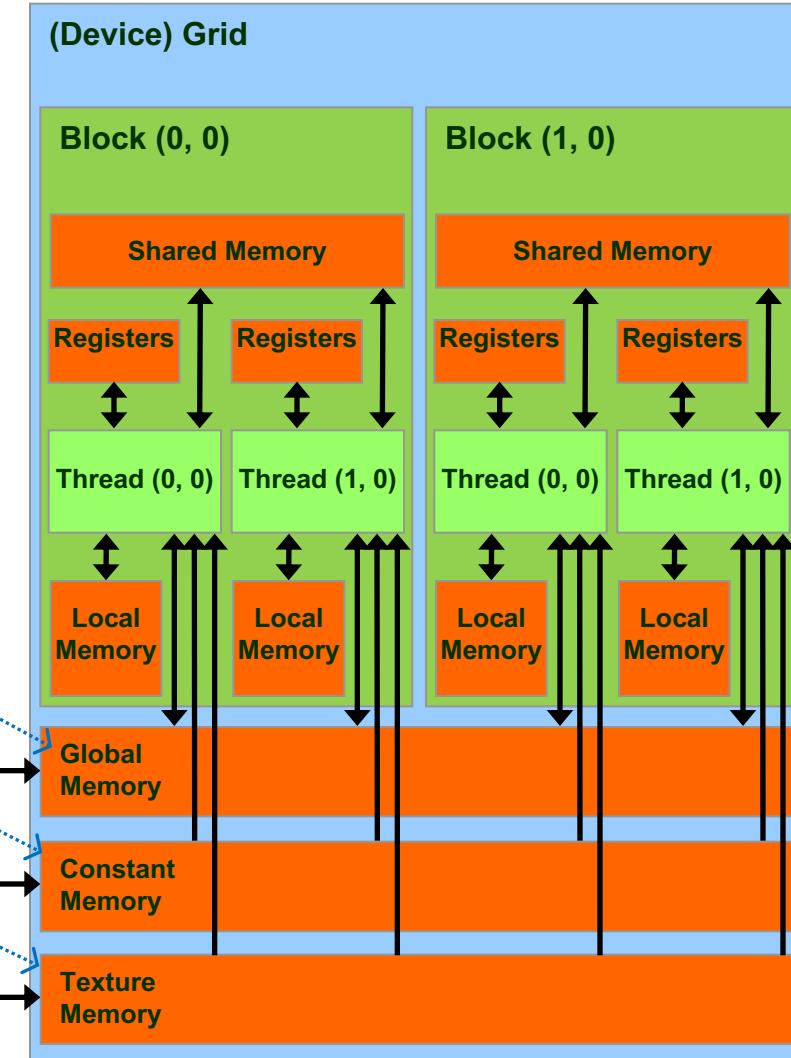


- Fermi
  - Each SM: 64 KB L1 cache & shared memory
  - All SMs tap into a common 768 KB L2 cache memory
- Pascal:
  - Each SM: Unified 24 KB L1 cache
  - Each SM: 64 KB shared memory
  - All SMs tap into 4096 KB of L2 cache memory
- Volta:
  - L1 cache + Shared Mem = 128 KB per SM
  - One can carve out of 128KB shared memory as follows:
    - 0 KB, 8 KB, 16 KB, 32 KB, 64 KB, or 96 KB
  - What is not taken by the ShMem becomes L1 cache:
    - 128 KB, 120 KB, 112 KB, 96 KB, 64 KB, 32 KB
  - All SMs tap into 6144 KB of L2 cache memory
- **IMPORTANT:** Numbers reported are per SM

# Global, Constant, and Texture Memories

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads
- Global, texture and constant memories are accessible by host
  - Done at high latency, low bandwidth

NOTE: We will not emphasize texture in ME759.



# Memory Access Times

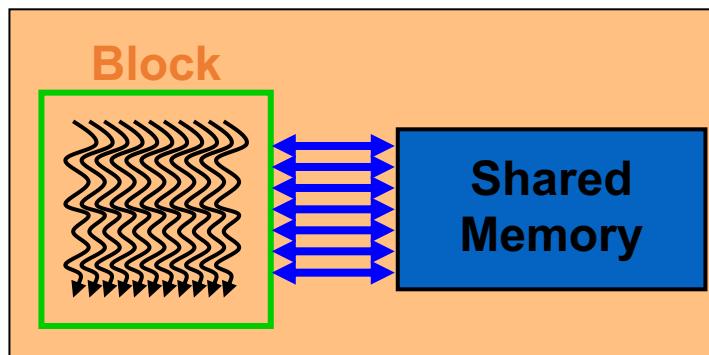
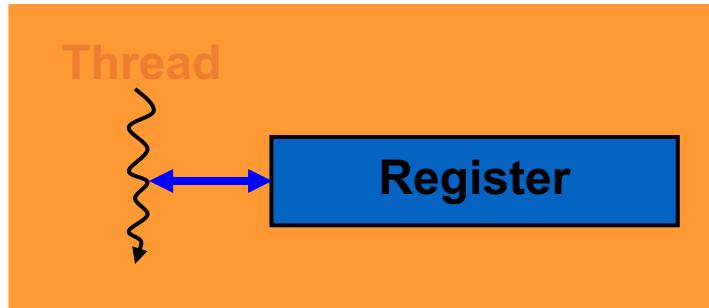
- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW – a few cycles
- Local Memory – DRAM, (if not in cache - \*slow\*; if cached – relatively fast)
- Global Memory – DRAM, no cache - \*slow\*; cached, is fast...
- Constant Memory – DRAM, cached, 10s...100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 10s...100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

# Storage Locations

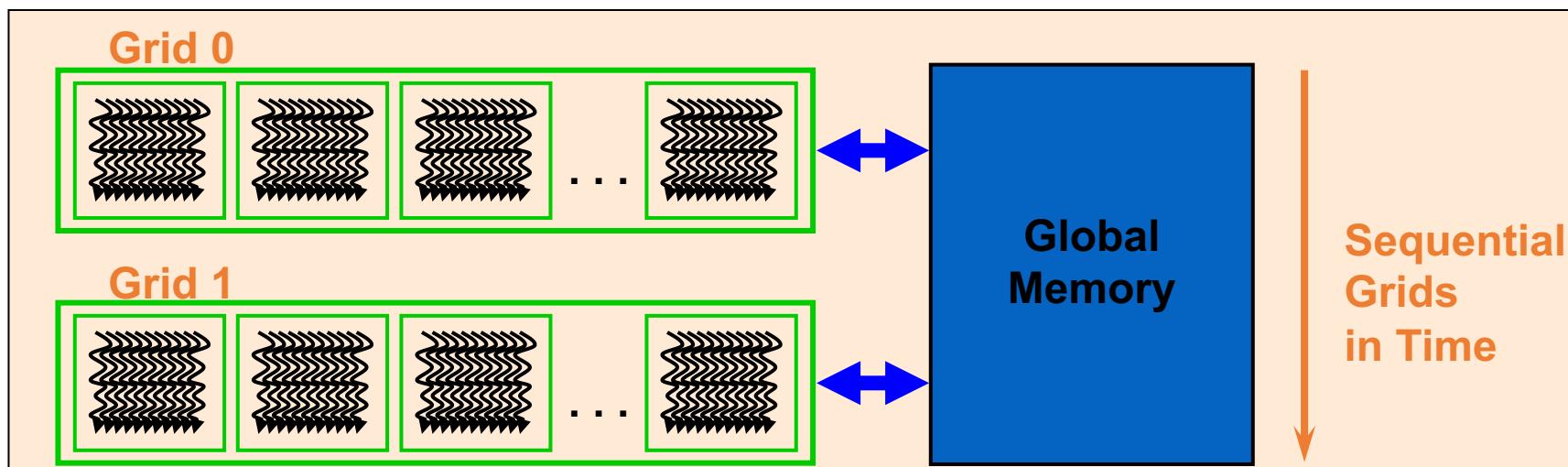
Memory	Location	Cached	Access	Who
Register	On-chip	N/A	Read/write	One thread only
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	Yes	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

Off-chip means on the GPU but not on the SM,  
which translates into high latency.

# Putting things in perspective: The 3 most important GPU memory spaces



- Register: per-thread basis
  - Private per thread
  - Can spill into local memory (potential performance hit unless cached)
- Shared Memory: per-block basis
  - Shared by threads of the same block
  - Used for: intra-block inter-thread communication
- Global Memory: per-application basis
  - Available for use by all threads
  - Used for: global access, all threads
  - Also used for inter-grid communication

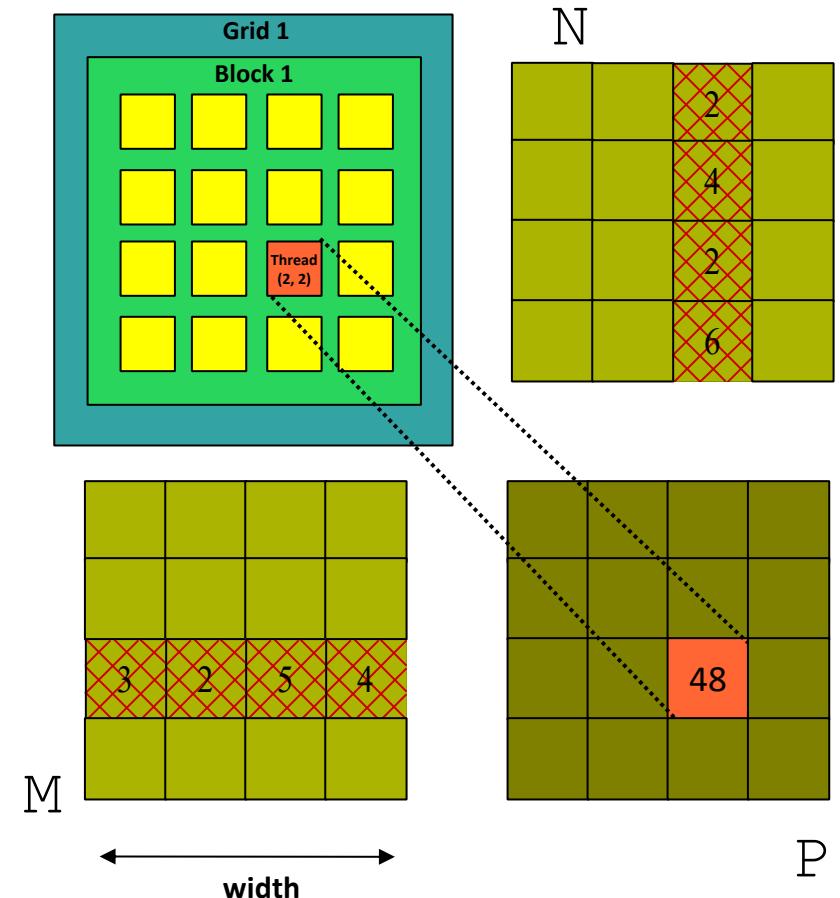


# Example: Matrix Multiplication, Revisited

- Purpose
  - See an example where the use of multiple blocks of threads plays a central role
  - Highlight the use/role of the shared memory
  - Point out the `__syncthreads()` function call
- NOTE: A one dimensional array stores the entries in the matrix

# A Critique of the Old Matrix Multiplication Example

- One row of  $M$  is loaded  $N.\text{width}$  times
  - This happens for each row of  $M$
  - Moral of the story: the matrix  $M$  is loaded  $N.\text{width}$  times
- 
- One column of  $N$  is loaded  $M.\text{height}$  times
  - This happens for each column of  $N$
  - Moral of the story: the matrix  $N$  is loaded  $M.\text{height}$  times
- 
- Loading  $M$  and  $N$  over and over → not a good idea, memory transactions are expensive



# Why Revisit the Old Example?

- In the naïve first implementation the ratio of arithmetic computation to memory transaction (“**arithmetic intensity**”) very **low**
  - Each arithmetic computation required one fetch from global memory
  - The entries in  $M$  are copied from global memory to the device  $N.width$  times
  - The entries in  $N$  are copied from global memory to the device  $M.height$  times
- No brownie points for moving data around
  - Hundreds of cycles required for bringing data from global memory
  - When solving a problem the goal is to go through the machine instructions in your executable, not to wait for data to come your way

# How to answer an important question: “Could I use Shared Memory?”

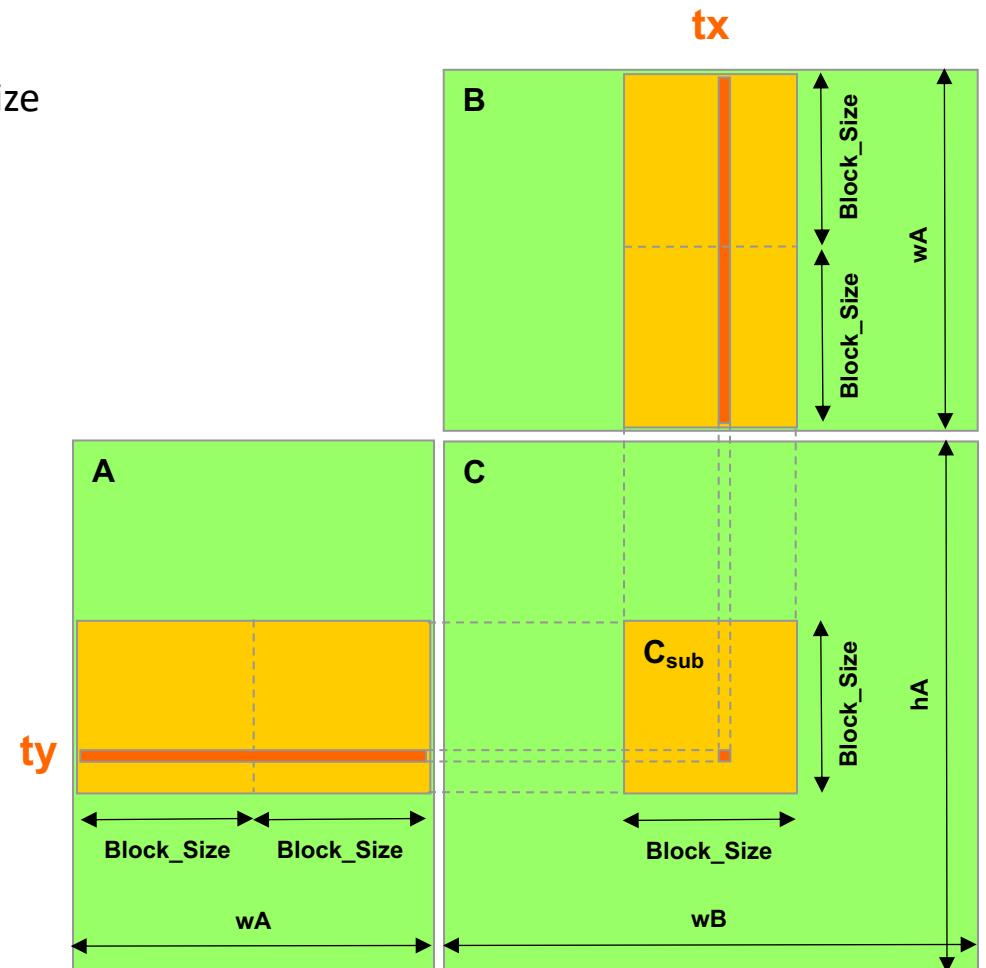
- Imagine you are a thread executing the kernel
- The ShMem test: If data that you, as a thread, use can also be used by another thread in your block, then you should consider using shared memory
- Note: regardless of the above test, you can use shared memory as scratch pad memory
  - Don't let it go wasted, use for storing often-used variables even if they're not used by multiple threads

# A Common CUDA Programming Pattern: Tiling w/ Shared Memory

- Global memory access - much slower when compared to shared memory access
- No brainer: Use shared memory, if at all possible
  - Partition data into data subsets (**tiles**) that each fits into shared memory
  - Handle each data subset (tile) with one thread block by:
    - Loading the tile from global memory into shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the tile from shared memory; each thread can efficiently multi-pass over any data element of the tile

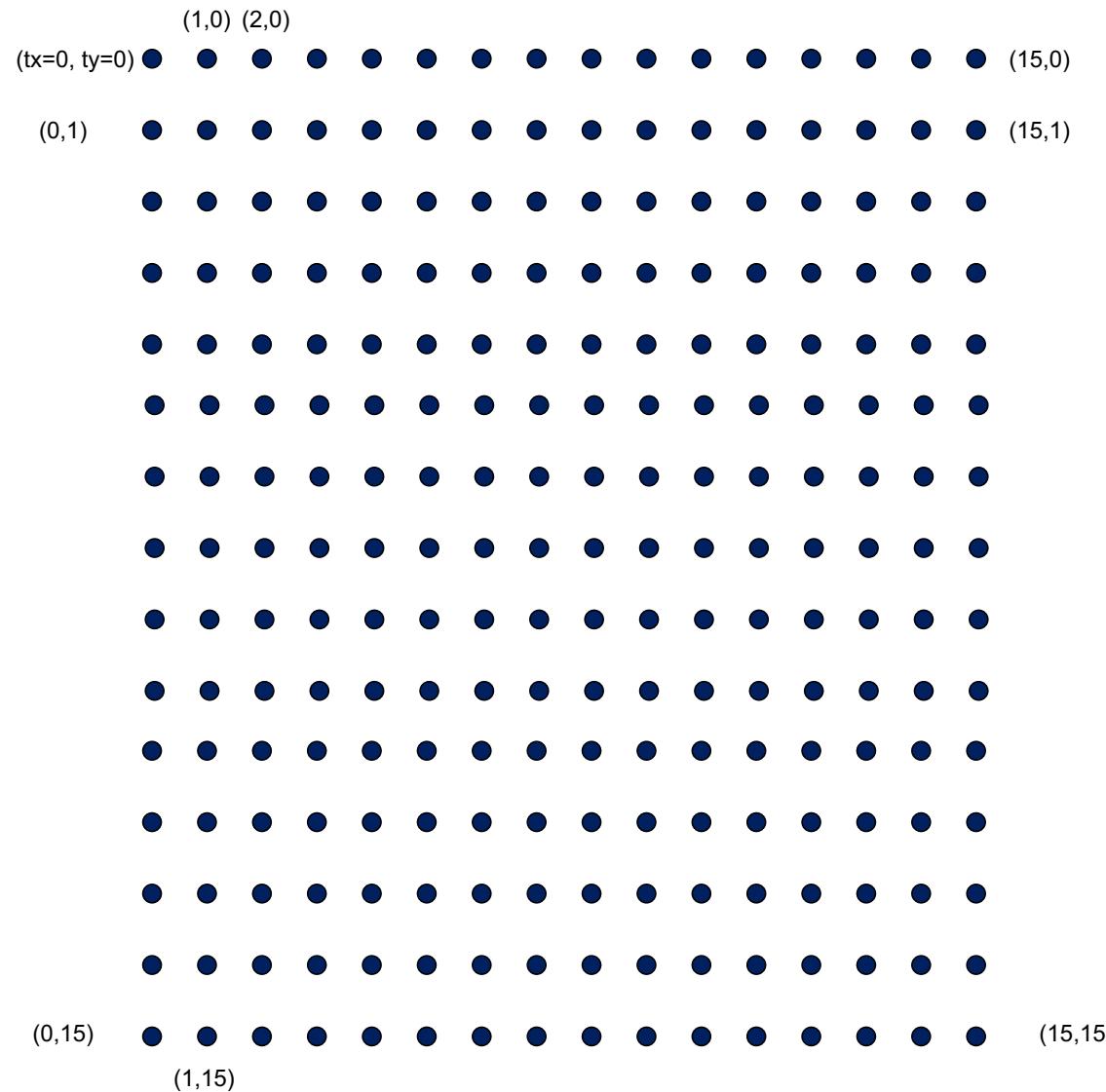
# Tiling used for Matrix-Matrix multiplication

- One **block** computes one square sub-matrix  $C_{sub}$  of size **Block\_Size**
- One **thread** computes one entry of  $C_{sub}$
- **Assumption:** **A** and **B** are *square matrices* and their dimensions of are *multiples of Block\_Size*
  - Doesn't have to be like this, but keeps example simpler and focused on the concepts of interest
  - In this example work with **Block\_Size=16x16**



NOTE: A similar technique is used on CPUs to improve cache hits. See slide "Blocking Example" at  
<http://cseweb.ucsd.edu/classes/fa10/cse240a/pdf/08/CSE240A-MBT-L15-Cache.ppt.pdf>

# A Block of 16 X 16 Threads



```

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A,const float* B,int hA,int wA,int wB,
float* C)
{
    // Load A and B to the device
    float* Ad;
    int size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);

    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
}

```

```

// Allocate C on the device
float* Cd;
size = hA * wB * sizeof(float);
cudaMalloc((void**)&Cd, size);

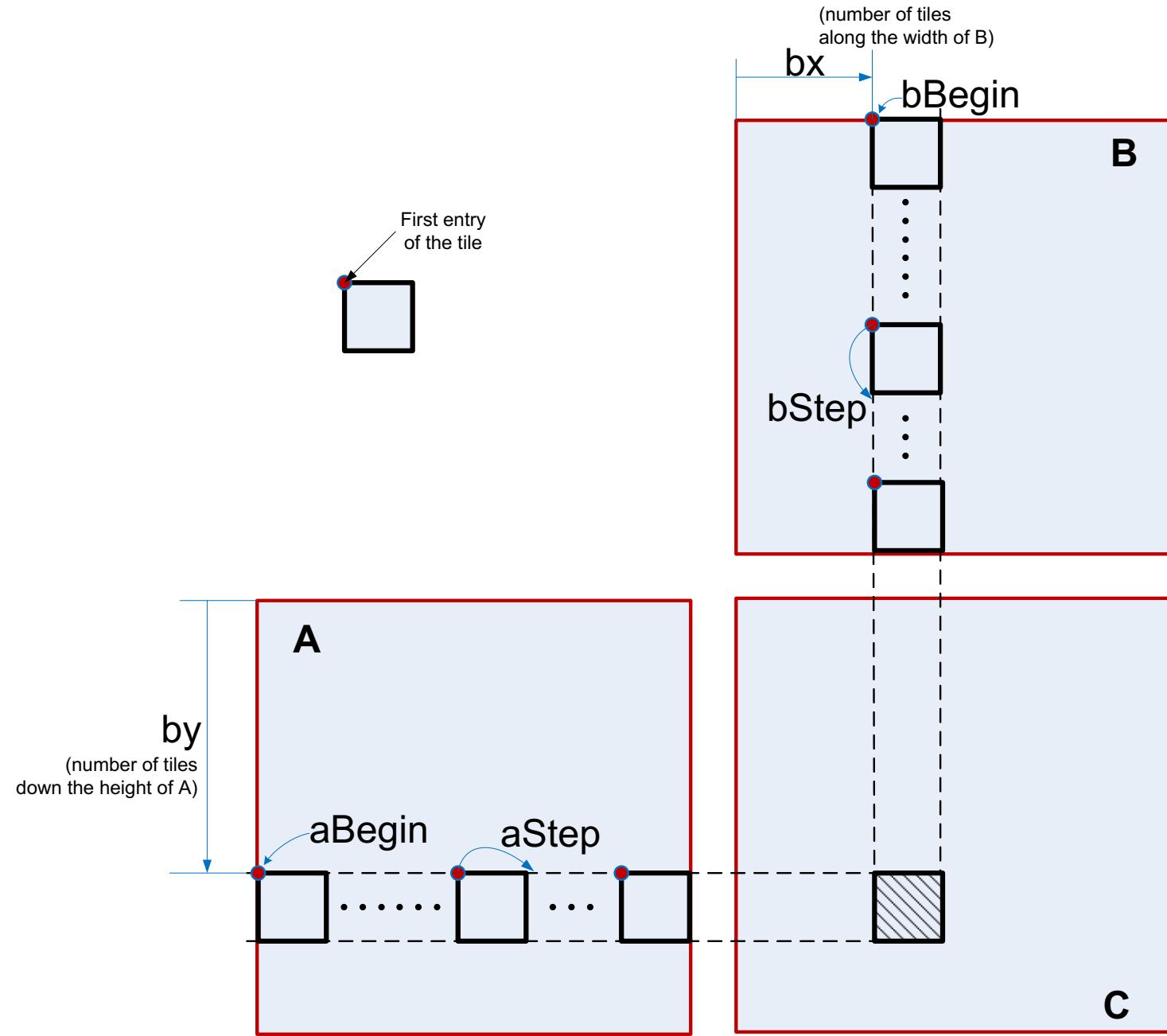
// Compute the execution configuration *assuming*
// the matrix dimensions are multiples of BLOCK_SIZE
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid( wB/dimBlock.x , hA/dimBlock.y );

// Launch the device computation
Muld<<<dimGrid, dimBlock>>>(Ad,Bd,wA,wB,Cd);

// Read C from the device
cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(Ad);
cudaFree(Bd);
cudaFree(Cd);
}

```



```

// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x; //the B (and C) matrix sub-block column index
    int by = blockIdx.y; //the A (and C) matrix sub-block row index

    // Thread index
    int tx = threadIdx.x; //the column index in the sub-block
    int ty = threadIdx.y; //the row index in the sub-block

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;
}

```

```

// Shared memory for the sub-matrices (tiles) of A and B
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

// Loop over all the sub-matrices (tiles) of A and B required to
// compute the block sub-matrix; moving in A left to right in
// a row, and in B from top to bottom in a column
for (int a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep) {

    // Load tiles from global memory into shared memory; each
    // thread loads one element of the two tiles from A & B
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    // Synchronize to make sure the matrices are loaded
    __syncthreads();

    // Each thread in this block computes one element
    // of the block sub-matrix (tile). Thread with indexes
    // ty and tx computes in this tile the entry [ty][tx].
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += As[ty][k] * Bs[k][tx];

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
}

// Write the block sub-matrix to global memory;
// each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}

```

# Synchronization Function

- Synchronization accomplished through a device lightweight function
  - `void __syncthreads();`
- Synchronizes all threads in a block (acts as barrier for all block threads)
  - Does **not** synchronize threads from two blocks
- **One more condition** for execution to commence once all threads in a block reached the sync point:
  - All global & shared mem. transactions made by these threads prior to `__syncthreads()` are visible to all block threads
  - Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory (more later)

# [warning] \_\_syncthreads() & thread divergence

```
#include<cuda.h>
#include<iostream>

__global__ void badIdeaSynchronization(int* data)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if ((i & 0x01) == 0)
    {
        data[i] = data[i] + i; // if not odd, come here
    }
    else
    {
        data[i] = data[i] + 2 * i; // come here if you're odd
        __syncthreads();
    }
}

int main()
{
    const int numElems = 4;
    int hostArray[numElems], *devArray;

    //allocate memory on the device (GPU); zero out all entries in this device array
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    hangs<<<1, numElems >>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}
```

- Be very cautious when using a `__syncthreads()` in a conditional
- The actual behavior is not clear
  - Some difference between the Programming Guide and what happens in reality
- See here for discussion of how/when code deadlocks:
  - <https://stackoverflow.com/questions/6666382/can-i-use-syncthreads-after-having-dropped-threads>
  - <https://stackoverflow.com/questions/15146886/conditional-syncthreads-deadlock-or-not>

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 12

02/19/2020

# Quote of the day

“You don't need a weatherman to know which way the wind blows.”

-- Bob Dylan, singer-songwriter, author, and visual artist [ 1941- ].

# Student Feedback quotes of the day

[Fake]

“You are the best. We really like you a lot.”

“Esti cel mai tare.”

“You are smarter than all other faculty at UW-Madison. Combined.”

“You hair style is amazing. Thank you!”

“I wish I could speak English with your accent.”

2020 ME759 anonymous feedback

# Before we get going...

- Last time:
  - Wrapped execution configuration
    - Example: Volta V100
      - 2048 threads in flight per SM for 100% occupancy
      - Up to  $84 \times 2048 = 172,032$  can be in flight at one instance in time
  - The GPU memory ecosystem
    - Register, local, cache, shared, global, constant, texture
- Today:
  - Wrap up memory ecosystem
    - Shared memory issues
    - Good global memory accesses
- Other tidbits:
  - Assignment due on **Friday** at 9 pm
  - Please read assigned readings
    - Some docs are long, I know – good to have on your radar
  - From now on, Dan has office hours online in Canvas each Wd at 7 PM
  - Next time:
    - Quick discussion about Final Project proposal
    - Touch on some points you raised in your feedback



# 3 Ways to Set Aside Shared Memory

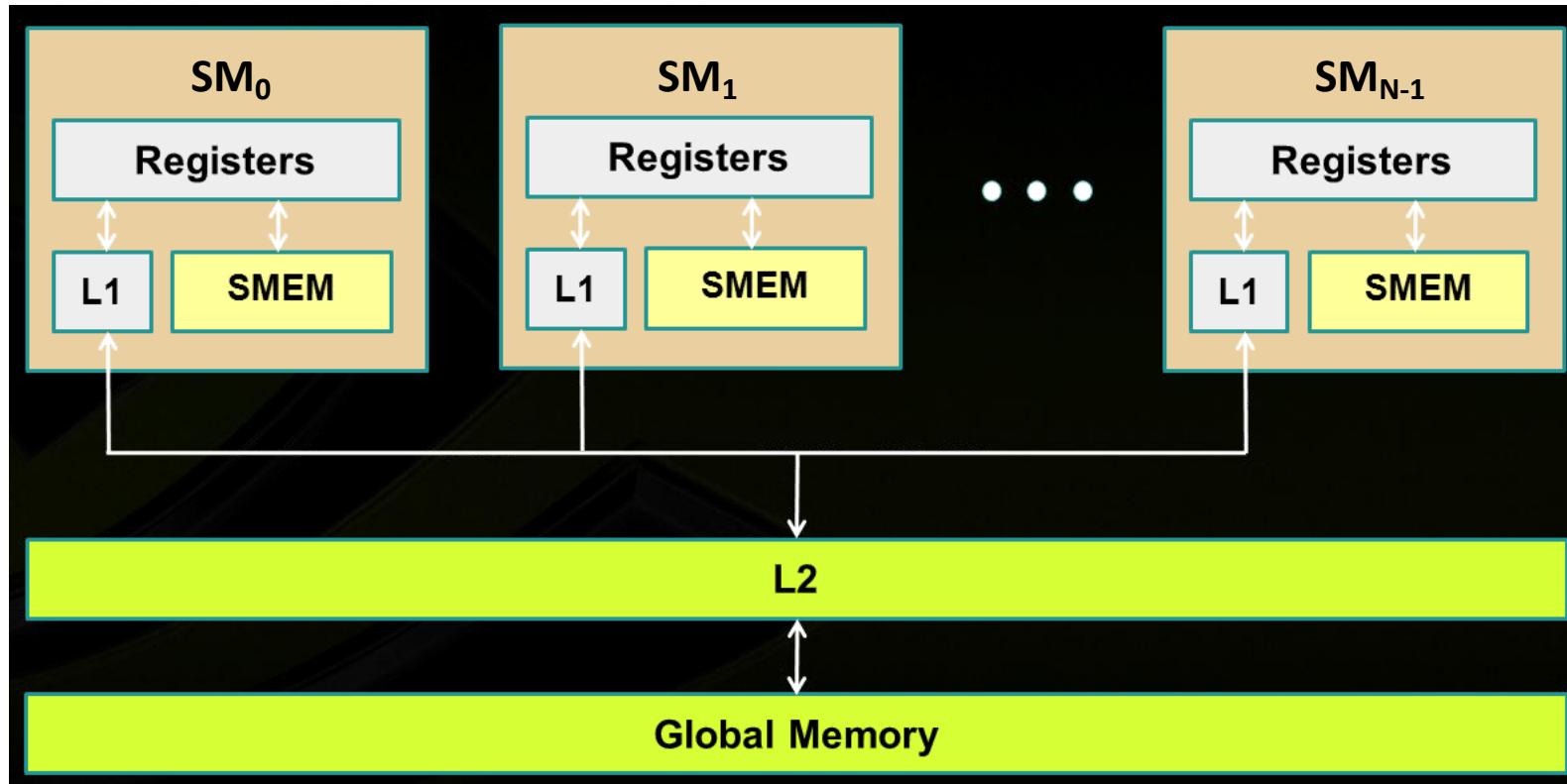
- First way: Statically, declared inside a kernel
  - See matrix multiplication example w/ shared memory use
- Second way: Through the execution configuration
  - **Ns** below indicates size (in bytes) to be allocated in shared memory

```
_global_ void MyFunc(float*) // __device__ or __global__ function
{
    extern __shared__ float shMemArray[];
    // Size of shMemArray determined through the execution configuration
    // You can use shMemArray as you wish here...
}

// invoke like this
MyFunc<<< Dg, Db, Ns >>>(parameter);
```

- Third way: Dynamically, via CUDA Driver API
  - Advanced feature, uses API function cuFuncSetSharedSize(), not discussed here

# Memory Layout, Fermi and more recent architectures



# GPU Cache Line

- All global memory accesses are cached
- A cache line is **128 bytes**
  - It maps to a 128-byte aligned segment in device memory
  - Note: it so happens that  $128 \text{ bytes} = 32 \text{ (warp size)} * 4 \text{ bytes}$
  - In other words, 32 floats or 32 ints fit in one cache line (and can be brought in L2 or L1 in one trip)
- If the size of the type accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently

# L1 & L2 Cache Issues

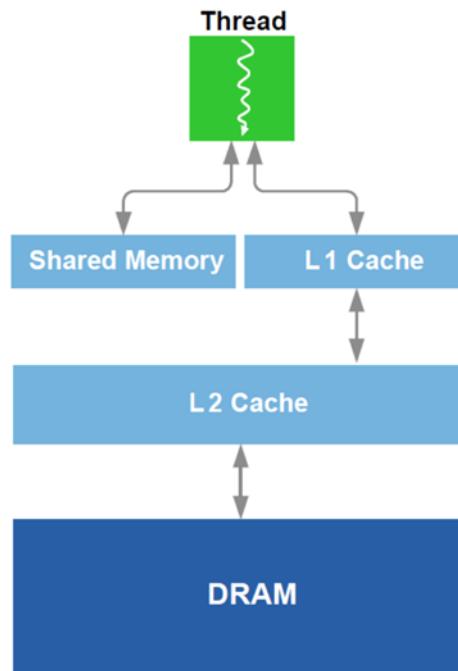
[two slide detour – 1/2]

- L1 and L2 cache used to cache accesses to
  - Local memory
  - Global memory
  - (texture & constant as well)
- L2 cache (Volta): 6144 KB – one big pool available to \*all\* SMs on the device
- Whether reads are cached in [L1 & L2] or in [L2 only] can be partially configured on a per-access basis using modifiers to the load or store instruction

# L1 Cache vs. Shared Memory

[two slide detour – 2/2]

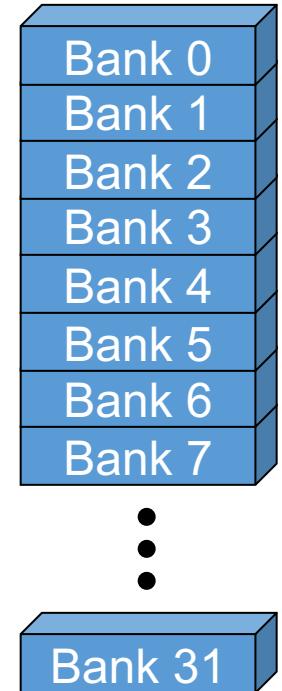
- Starting w/ Fermi architecture, you can split some fast memory between shared memory and cache



- Example, Fermi: you can go 48/16 or 16/48 KB for Shared Mem/Cache
- OPTION 1: you choose to go w/ “Lots of Cache & Little Shared Mem”
  - Cache handled for you by the scheduler
  - No control over it
  - Can’t have too many blocks of threads running if blocks use Shared Mem
- OPTION 2: you choose to go w/ “Lots of Shared Mem & Little Cache”
  - Good in tiling, you have full control
  - Shared Mem pretty cumbersome to manage

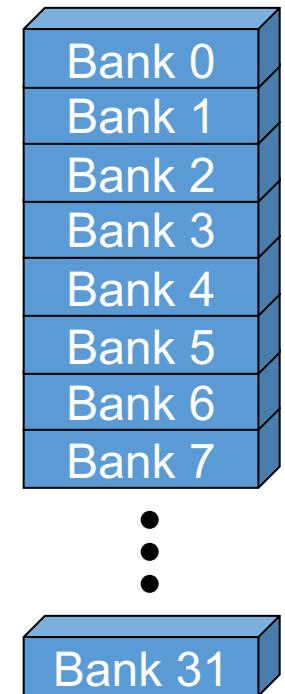
# On the architecture of the GPU's Shared Memory [1/2]

- GPU computing: many threads access the ShMem at the same time
  - To service more than one thread, ShMem is divided into independent **banks**
  - This layout essential to achieve high bandwidth
- Each SM has ShMem organized in 32 Memory banks
- Successive 32-bit words map to successive banks
- Each bank has a bandwidth of 32 bits per clock cycle



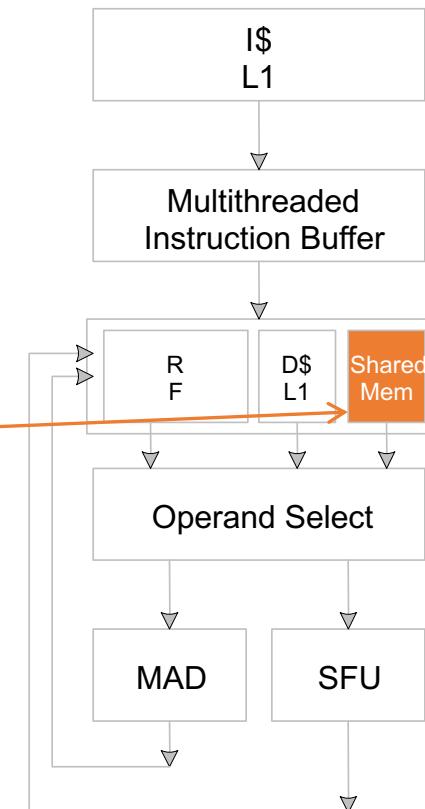
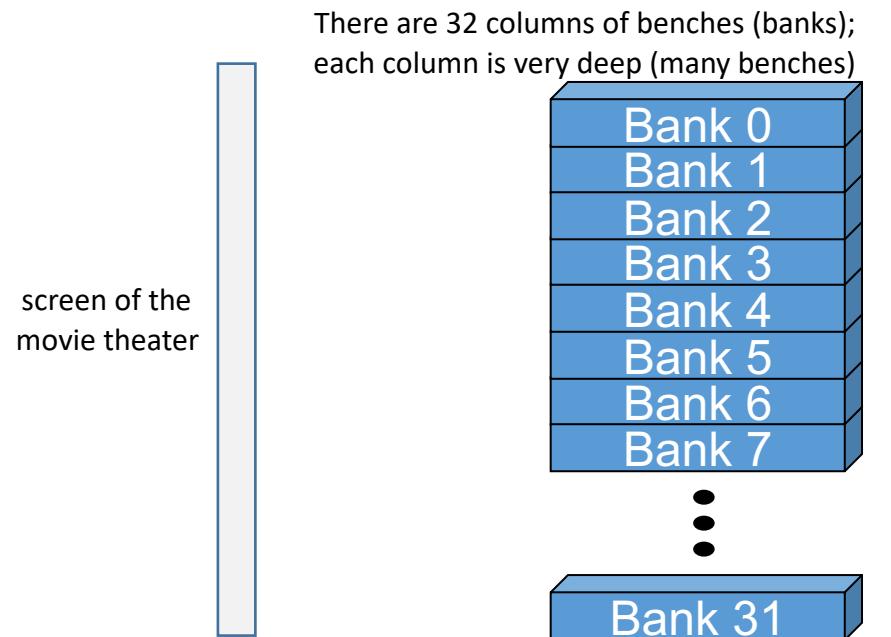
# On the architecture of the GPU's Shared Memory [1/2]

- ShMem and L1 cache draw on the same physical memory inside an SM
  - The cache: managed by the CUDA runtime
  - ShMem: managed by you (scratchpad memory)
- Example → Volta:
  - L1 cache + Shared Mem = **128 KB per SM**
  - One can carve out of 128KB shared memory as follows:
    - 0 KB, 8 KB, 16 KB, 32 KB, 64 KB, or 96 KB
  - What is not taken by the ShMem becomes L1 cache:
    - 128 KB, 120 KB, 112 KB, 96 KB, 64 KB, 32 KB



# On the architecture of the GPU's Shared Memory [2/2]

- The 32 ShMem banks organized like benches in a movie theater
  - You have multiple rows of benches
  - Each row has 32 benches separated; each bench belongs to a long column
  - A bank: a column of benches in the movie theater, perpendicular to screen
  - In each bench you can “seat” a family of four bytes (32 bits total)
- Each bank has a bandwidth of 32 bits per two **clock cycles**



# Shared Memory: Transaction Rules & Bank Conflicts

- When reading in four-byte words, 32 threads in a warp attempt to access shared memory simultaneously
- Bank conflict: two different threads access *\*different\** words in different rows of the same bank
  - Note that there is no conflict if different threads access any bytes within the same word (same row)
- Bank conflicts enforce the hardware to serialize a ShMem access, which adversely impacts bandwidth

# Shared Memory Bank Conflicts

- If there are no bank conflicts:
  - Shared memory access is fast, but not quite as fast as register access
  - On the bright side, latency is roughly 100x lower than global memory latency
- ShMem operation, the fast case:
  - If all threads of a warp access different banks, there is no bank conflict
  - If all threads of a warp access an identical address for a fetch operation, there is no bank conflict (broadcast)
- ShMem memory operation, the slow case:
  - Worst case: 32 threads access 32 different words in the same bank
  - Must serialize all the accesses
  - In general, cost = max # of simultaneous accesses to a single bank

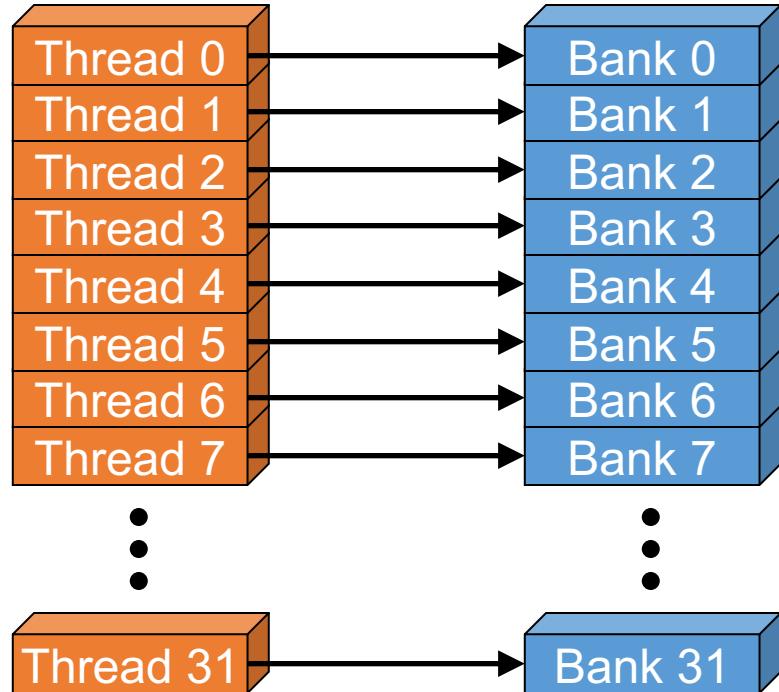
# Example: How Addresses Map to Banks, for array of float

- An allotted chunk of shared memory starts at an address that is at least a multiple of 128 (if not 256)
- Bank you work with = (address of offset) % 32
  - Example: 1D shared mem array, `myShMemVar`, of 1024 `floats`
    - `myShMemVar[4]`: accesses bank #4 (physically, the fifth one – first row)
    - `myShMemVar[31]`: accesses bank #31 (physically, the last one – first row)
    - `myShMemVar[50]`: access bank #18 (physically, the 19<sup>th</sup> one – second row)
    - `myShMemVar[128]`: access bank #0 (physically, the first one – fifth row)
    - `myShMemVar[178]`: access bank #18 (physically, the 19<sup>th</sup> one – sixth row)
  - If, for instance, the third thread in a warp accesses `myShMemVar[50]` and the eight thread in the warp accesses `myShMemVar[178]`, then you have a two-way bank conflict and the two transactions get serialized
- IMPORTANT: There is no such thing as “bank conflicts” between threads belonging to different warps

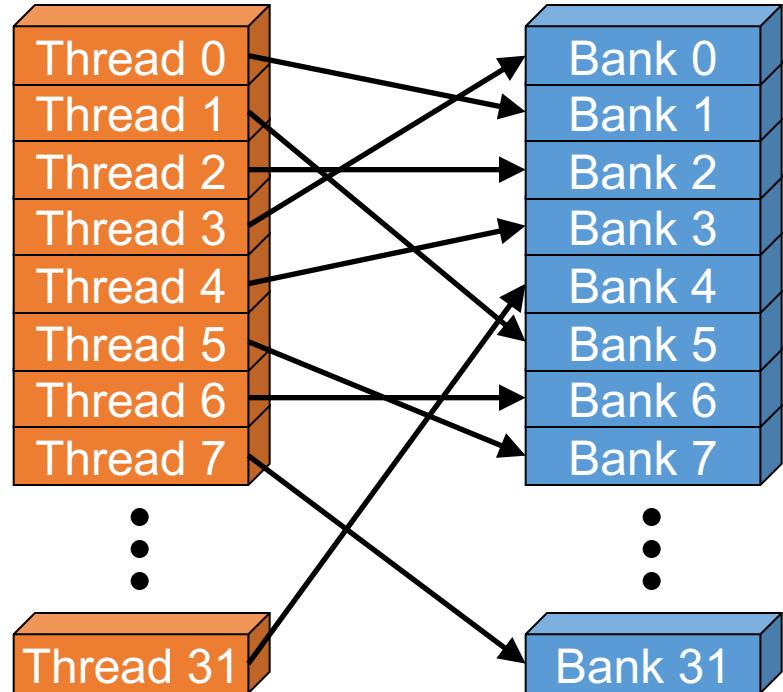
# Bank Addressing Examples

Transactions Involving 4 Byte Words

- No Bank Conflicts
  - Linear addressing stride == 1



- No Bank Conflicts
  - Random 1:1 Permutation

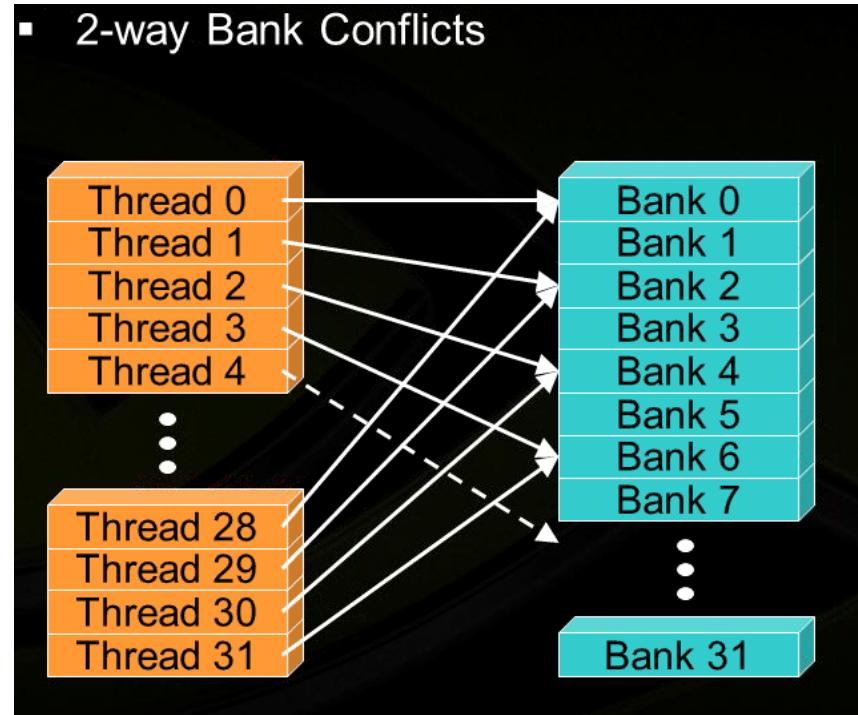


# Bank Addressing Examples

Transactions Involving 4 Byte Words

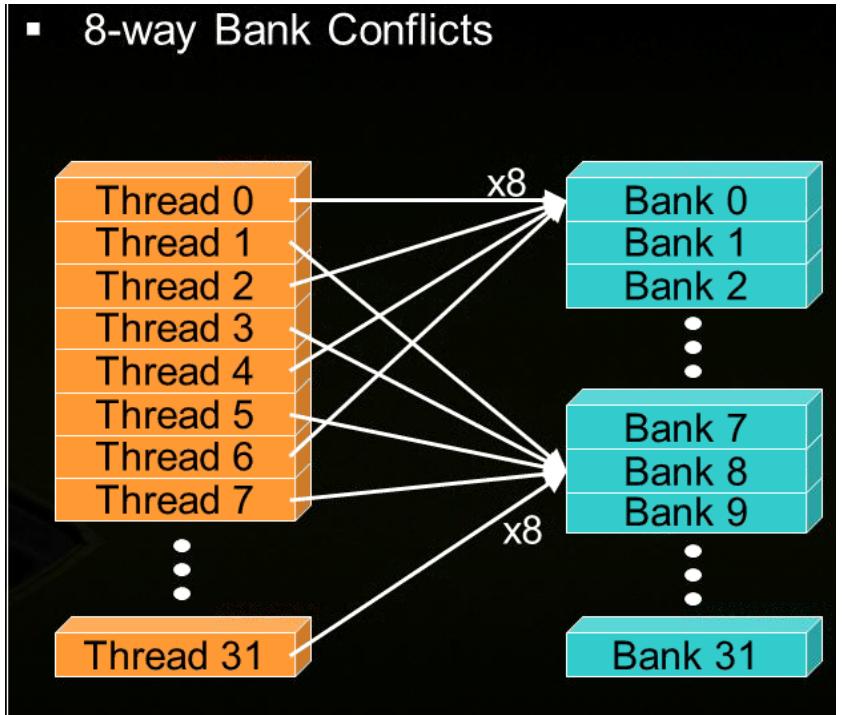
A case in which only the even banks end up being accessed, and the warps access different words in each bank

- 2-way Bank Conflicts



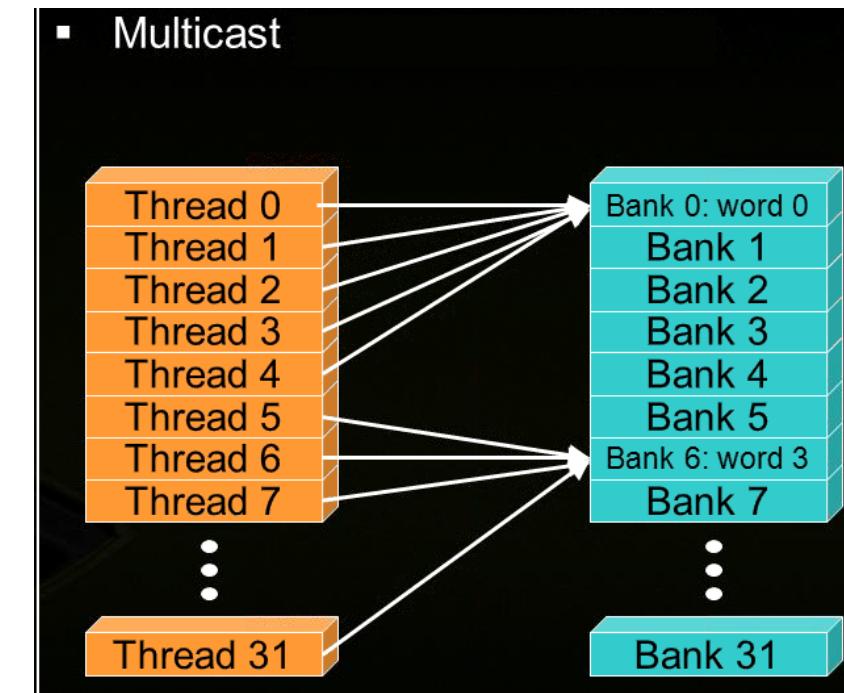
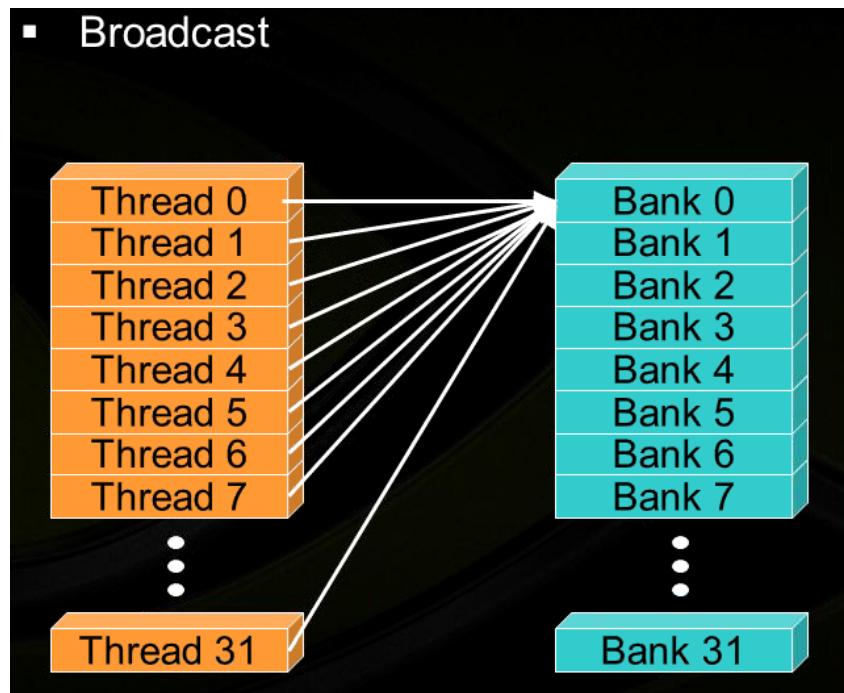
A case in which the 32 threads in a warp access only banks 0, 8, 16, and 24 (and different words in these four banks)

- 8-way Bank Conflicts



# Other Examples

- Two “no conflict” scenarios, reading operations:
  - Broadcast: all threads in a warp access the same word in a bank
  - Multicast: several threads in a warp access the same word in the same bank



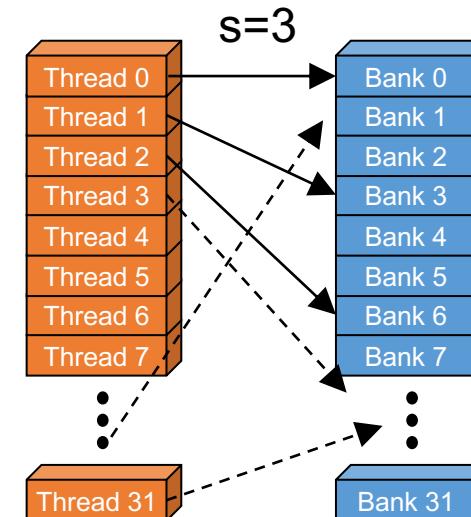
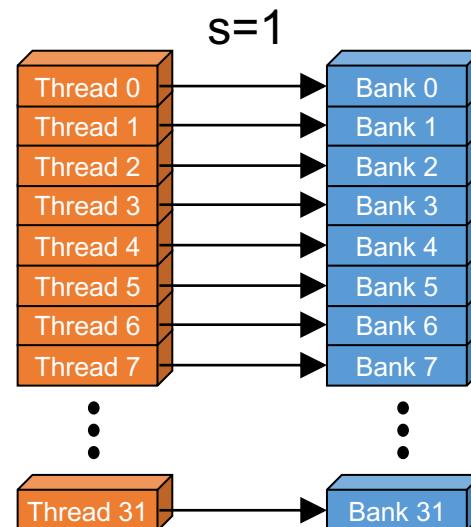
# Linear Addressing

- Given:

```
shared float sharedM[256];  
float foo = sharedM[baseIndex + s * threadIdx.x];
```



- This is bank-conflict-free if  $s$  shares no common factors with the number of banks
  - Conclusion: you are fine if  $s$  is odd



Note: baseIndex assumed 0 (zero) in these two pics

# Data types and bank conflicts

- No conflicts if `shrd` is a 32-bit data type:

```
foo = shrd[baseIndex + threadIdx.x]
```

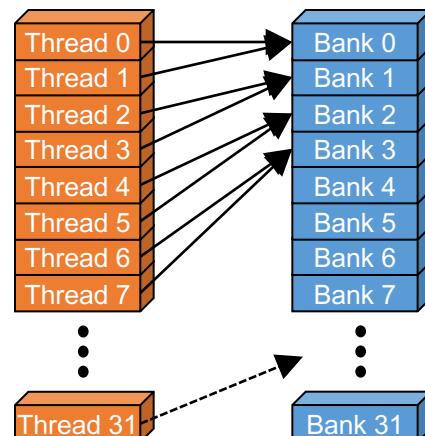
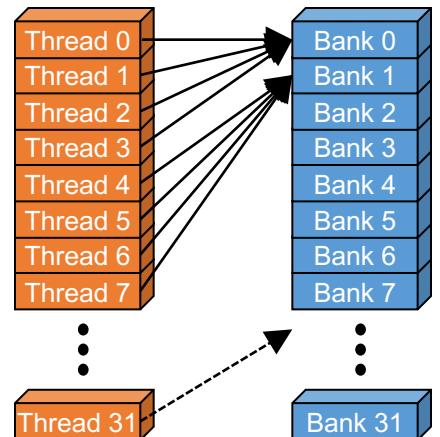
- Also if accessing one byte-per-thread, no conflict since \*different\* bytes of the same word are accessed

- No conflicts:

```
extern __shared__ char shrd[];  
foo = shrd[baseIndex + threadIdx.x];
```

- No conflicts:

```
extern __shared__ short shrd[];  
foo = shrd[baseIndex + threadIdx.x];
```

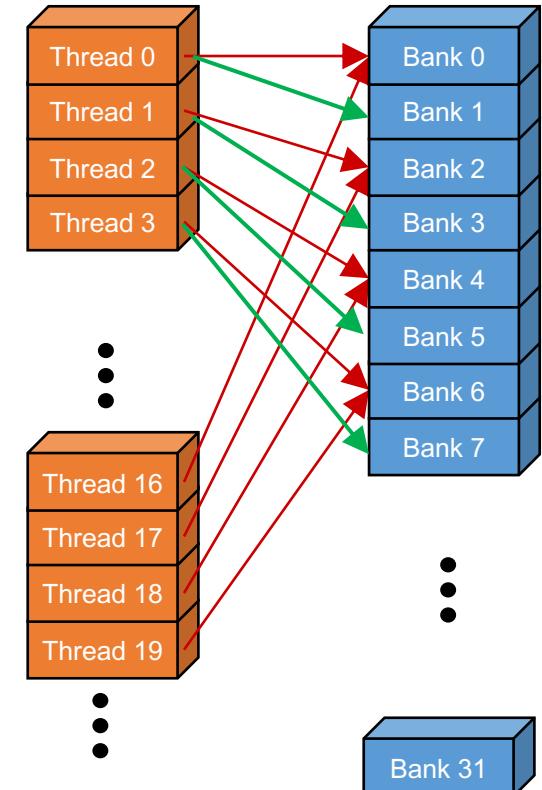


# Exercise: Is ShMem access below good or bad?

- Snippet from a kernel, each thread loads two floats into ShMem:

```
int tid = threadIdx.x;
shared[2*tid] = global[offset+2*tid];
shared[2*tid+1] = global[offset+2*tid+1];
```

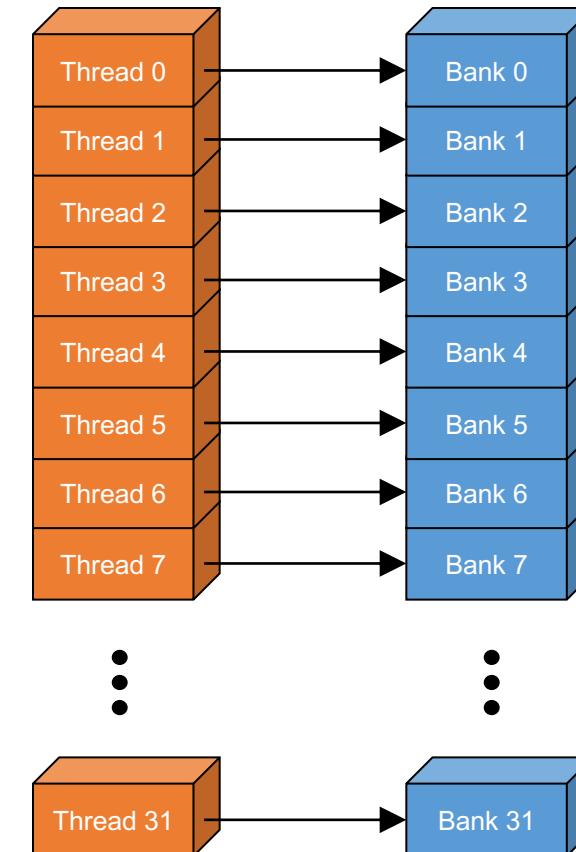
- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic
  - Doesn't make sense in shared memory usage where there is no cache line effects but banking effects
  - 2-way-interleaved loads result in 2-way bank conflicts
- Adding insult to injury: you don't have coalesced global memory loads – basically you are halving the device mem bandwidth (more on this later)



# A better array access pattern: Revisiting example on previous slide

- Here's a better way of doing it
  - Each thread loads one element in every consecutive group of `blockDim` elements.

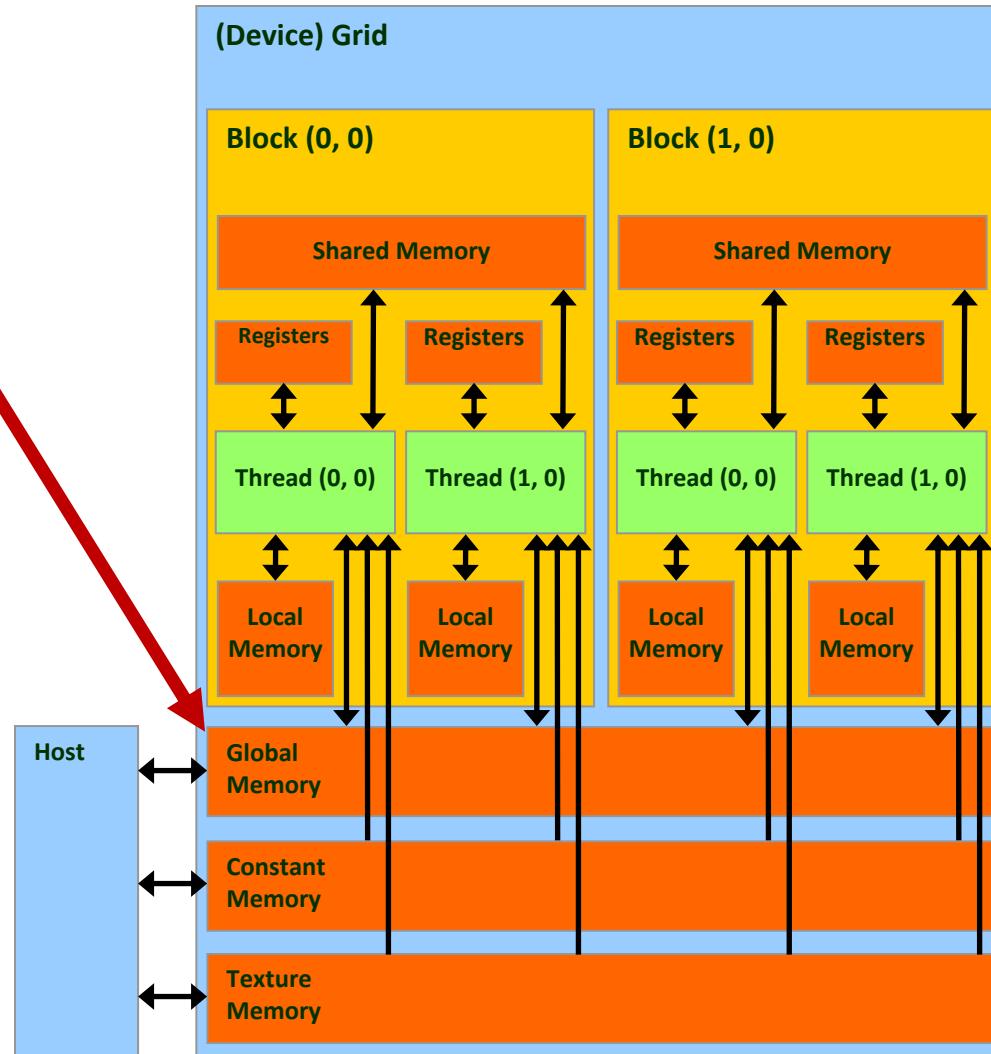
```
shared[tid] = global[offset + tid];  
shared[tid + blockDim.x] = global[offset + tid + blockDim.x];
```



# Global Memory Access Issues

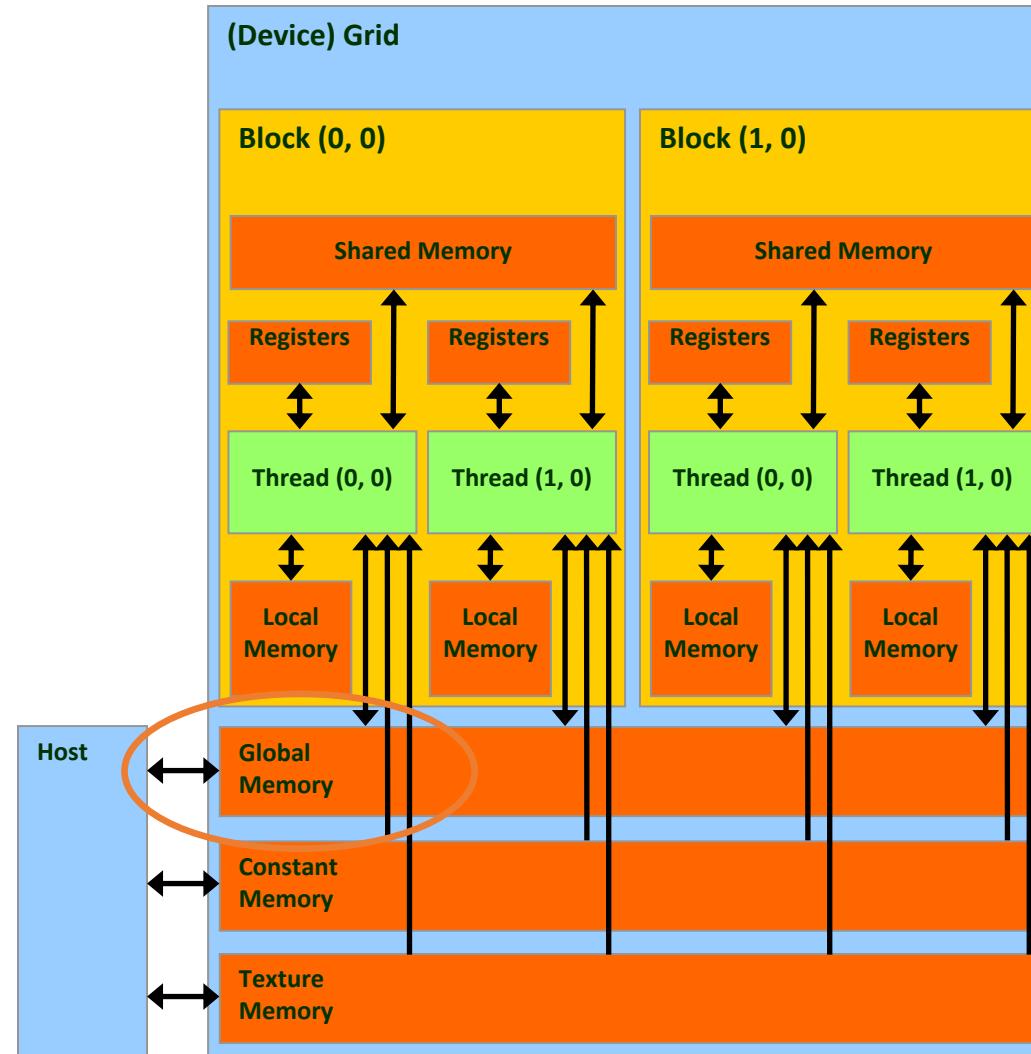
# CUDA Device Memory Allocation/Deallocation

- `cudaMalloc()`
  - Allocates mem in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object
- `cudaFree()`
  - Frees mem from device Global Memory
    - Requires a pointer to freed object



# CUDA Host-Device Data Transfer

- `cudaMemcpy()`
  - Memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Things happen over a PCIe connection
  - Up to 16 GB/s (PCIe 3.0 x16, in each direction)



# CUDA Host-Device Data Transfer (cont.)

- Example:
  - Transfer a number of “size” bytes
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are CUDA-defined constants

```
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);
```

- Note: if you use gdb to debug, don't expect to be able to read memory off a device pointer
  - Debugging for GPU computing relies on cuda-gdb

# Ruminations, memory related

- Looking ahead (today and next lecture), ruminations on two topics
- • Memory operations, getting the result right (broad discussion, for parallel computing, not only GPU)
- Memory operations, getting the result fast (narrower discussion, for GPU computing)

```

#include<cuda.h>
#include<iostream>

__global__ void odd_even(int* data) {
    int i = threadIdx.x;
    if ((i & 0x01) == 0)
    {
        data[i] = data[i] + i; // if even, come here
    }
    else
    {
        data[i] = data[i] + 2*i; // come here if you're odd
    }
}

int main() {
    const int numElems = 4;
    int hostArray[numElems], *devArray;

    //allocate memory on the device (GPU); zero out all device array entries
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    odd_even<<<1, 4 >>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}

```

Linux, P100

The terminal window shows the command `nvcc -arch=sm\_60 odd\_even.cu` being run, followed by the execution of the resulting binary `./a.out`. The output displays the values stored in the host array: 0, 2, 2, 6.

```

Cmder
hopper ~/CodeBits> nvcc -arch=sm_60 odd_even.cu
hopper ~/CodeBits> ./a.out
Values stored in hostArray: result to confirm that
0      std::cout << "Values stored in hostArray"
2      for (int i = 0; i < numElems; i++)
2          std::cout << hostArray[i] << std::endl
6
hopper ~/CodeBits>

```

Windows, GTX1080

The command prompt window shows the command `C:\Windows\system32\cmd.exe` being run. The output displays the values stored in the host array: 0, 2, 2, 6, followed by a prompt to press any key to continue.

```

C:\Windows\system32\cmd.exe
Values stored in hostArray:
0
2
2
6
Press any key to continue . . .

```

```

#include<cuda.h>
#include<iostream>

__global__ void wicked(int* data) {
    int i = threadIdx.x;
    if ((i & 0x01) == 0)
    {
        data[i+1] = data[i+1] + i; // if even, come here
    } else
    {
        data[i] = data[i] + 2*i; // come here if you're odd
    }
}

int main() {
    const int numElems = 4;
    int hostArray[numElems], *devArray;

    //allocate memory on the device (GPU); zero out all device array entries
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    wicked<<<1, 4 >>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}

```

Linux, P100

```

hopper ~/CodeBits> nvcc -arch=sm_60 wicked.cu
hopper ~/CodeBits> ./a.out
Values stored in hostArray:
0
2
0
6
hopper ~/CodeBits> |

```

Windows, GTX1080

```

C:\Windows\system32\cmd...>
Values stored in hostArray:
0
2
0
6
Press any key to continue . . .

```

# Data Hazards in Parallel Computing

- Backdrop:
  - Threads  $i$  and  $j$  execute two streams of instructions in parallel and operate at some point on the same memory entry
  - The expectation is that  $i$  executes before  $j$
- Three types of data hazards
  - Read-After-Write (RAW) – think of it as a violation of “*j* ought to Read only After the Write in *i* occurred”
    - $j$  reads a source before  $i$  writes it, so  $j$  incorrectly gets the old value
    - Perhaps most common of them three hazards
  - Write-After-Read (WAR) – think of it as a violation of “*j* ought to Write only After the Read in *i* occurred”
    - $j$  writes a destination before it is read by  $i$ , so  $i$  gets the new (and wrong) value
  - Write-After-Write (WAW) – think of it as a violation “*j* ought to Write only After another Write occurred in *i*”
    - $j$  writes an operand before it is written by  $i$ . The writes end up being performed in the wrong order, leaving the value written by  $i$  rather than the value written by  $j$  in the destination
- Moral of the story: The *ordering* of memory operations is important in getting the result right

# Parallel computing, bottom line

- Parallel computing: previous slide, two threads led to race conditions – in absence of special measures, there is no way to know who executes what and when
  - A matter of “racing of threads”
- “special measures”, example: use of `__syncthreads()`
- Can we at least say something about the **order in which memory operations** are carried out?
  - A matter of “**sequence of memory ops**”, pertains one thread (with side effects when a 2<sup>nd</sup> thread present)

# The concept of “memory consistency”

- Backdrop
  - Thread\_1 executes writeXY()
  - Thread\_2 executes readXY()
- Question: what values can A and B assume?
- For sequential consistency (“strongly-ordered mem. model”), the only possible scenarios are
  - A=2 and B=1
  - A=2 and B=10
  - A=20 and B=10
- For weak consistency (“weakly-ordered memory model”), it is possible to have this
  - A=20 and B=1

```
__device__ volatile int X = 1, Y = 2;  
  
__device__ void writeXY()  
{  
    X = 10;  
    Y = 20;  
}  
  
__device__ void readXY()  
{  
    int A = Y;  
    int B = X;  
}
```

# Flavors of memory consistency

- Sequential consistency: all reads and all writes are in-order
- Relaxed consistency: some types of reordering are allowed
  - Loads can be reordered after loads (for better working of cache coherency, better scaling)
  - Loads can be reordered after stores
  - Stores can be reordered after stores
  - Stores can be reordered after loads
- Weak consistency: reads & writes arbitrarily reordered
  - Programmer expected to step in and use explicit memory barriers if a certain order needed to be observed

# Not a GPU computing but rather a parallel computing issue

- Example, memory ordering for some non-GPU architectures

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	z/Architecture
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	

# The CUDA \_\_threadfence function, backdrop

- What we have just learned:
  - If a memory transaction was posted, it does not imply that it has been completed before another memory transaction that follows it
- Need a guarantee that memory transaction has completed before another one is started
  - Comes through functionality associated w/ threadfence calls

# The CUDA \_\_threadfence family of functions

**\_\_threadfence\_block()** ;

- Execution of the kernel by the calling thread pauses until all **global** and **shared memory** outstanding writes are visible to all threads in block

**\_\_threadfence()** ;

- Execution of kernel by a calling thread ensures all **global** and **shared memory** outstanding writes are visible to all threads in block AND all other threads in flight for global data
- NOTE: the fences are NOT a synchronization mechanism
  - Not all threads should reach the fence function for the thread calling the fence to be allowed to move past the fence
  - It's just a way to enforce that memory transactions for ONE thread can be seen by other threads

# The CUDA \_\_threadfence family of functions

**\_\_threadfence\_system()** ;

- A thread stops in its execution of the kernel until global and shared memory accesses prior to the **\_\_threadfence\_system()** are visible to
  - all threads in the block for ShMem accesses
  - all threads in the device for global memory accesses,
  - host threads for page-locked host memory accesses

# The nitty-gritty, has to do with enforcing ordering...

- The fine print, for `_threadfence_block()`, two statements:
  - All writes to all memory made by the calling thread before the call to `_threadfence_block()` are observed by all threads in the block of the calling thread as occurring before all writes to all memory made by the calling thread after the call to `_threadfence_block()`
  - All reads from all memory made by the calling thread before the call to `_threadfence_block()` are ordered before all reads from all memory made by the calling thread after the call to `_threadfence_block()`
- Similar semantics for `_threadfence()` & `_threadfence_system()`

```

#include<cuda.h>
#include<iostream>

__global__ void divergence(int* data)
{
    int i = threadIdx.x;
    if ((i & 0x01) == 0)
    {
        data[i+1] = data[i+1] + i; // if even, come here
    } else
    {
        data[i] = data[i] + 2*i; // if odd, come here
    }
}

int main()
{
    const int numElems = 4;
    int hostArray[numElems], *devArray;

    //allocate memory on the device (GPU); zero out all entries in this device array
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    divergence <<<1, 4 >>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}

```

Linux, P100

```

hopper ~/CodeBits> nvcc -arch=sm_60 wicked.cu
hopper ~/CodeBits> ./a.out
Values stored in hostArray:
0
2
0
6
hopper ~/CodeBits> |

```

Windows, GTX1080

```

C:\Windows\system32\cmd...>
Values stored in hostArray:
0
2
0
6
Press any key to continue . . .

```

```

#include<cuda.h>
#include<iostream>

__global__ void divergence(int* data)
{
    int i = threadIdx.x;
    if ((i & 0x01) == 0)
    {
        data[i+1] = data[i+1] + i; // if even, come here
        __threadfence_block();
    }
    else
    {
        data[i] = data[i] + 2*i; // if odd, come here
    }
}

int main()
{
    const int numElems = 4;
    int hostArray[numElems], *devArray;

    //allocate memory on the device (GPU); zero out all entries in this device array
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    divergence <<<1, 4 >>>(devArray);

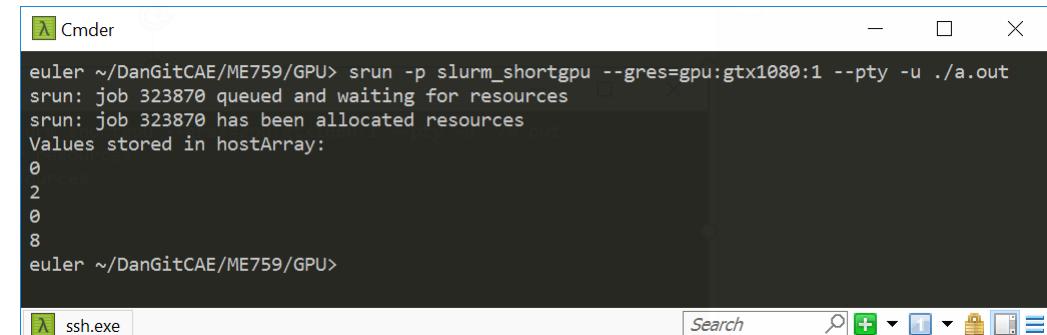
    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}

```

## Linux, P100



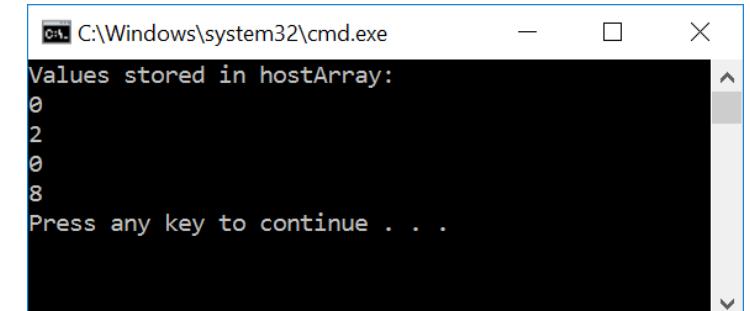
The terminal window shows the command `srun -p slurm\_shortgpu --gres=gpu:gtx1080:1 --pty -u ./a.out` being run. The output indicates the job is queued and waiting for resources, then has been allocated resources. The values stored in the hostArray are printed as 0, 2, 0, 8.

```

euler ~/DanGitCAE/ME759/GPU> srun -p slurm_shortgpu --gres=gpu:gtx1080:1 --pty -u ./a.out
srun: job 323870 queued and waiting for resources
srun: job 323870 has been allocated resources
Values stored in hostArray:
0
2
0
8
euler ~/DanGitCAE/ME759/GPU>

```

## Windows, GTX1080



The terminal window shows the command `C:\Windows\system32\cmd.exe` being run. The output shows the values stored in the hostArray as 0, 2, 0, 8, followed by a prompt to press any key to continue.

```

C:\Windows\system32\cmd.exe
Values stored in hostArray:
0
2
0
8
Press any key to continue . . .

```

# The **volatile** qualifier

- From CUDA Programming Guide
  - Compiler is “free to optimize reads and writes to global or shared memory” provided
    - It respects the memory ordering semantics of memory fence functions
    - It respects memory visibility semantics of synchronization functions
  - What is the compiler allowed to do?
    - Example: store global reads into registers or L1 cache to speed up the computation
  - These “behind your back” optimizations disabled using the **volatile** keyword
  - CUDA: “If a variable located in global or shared memory is declared as **volatile**, the compiler assumes that its value can be changed or used at any time by another thread and therefore any reference to this variable compiles to an actual memory read or write instruction.”

# The **volatile** qualifier: how things go south, ShMem example

- To speed execution, compiler places a variable that should make it into ShMem into a register
- A register is specific to a thread; another thread cannot see what's stored in that register
- Perhaps your code relies on thread B reading something that was deposited in ShMem by thread A
- Thread A's variable didn't make it all the way to the ShMem since the compiler wrote that variable in a register, to speed up the computation
  - **volatile** prevents the compiler from playing these games and forces any variable to be written to the location in the memory where it ought to be stored – no funny business
- Note: if in your kernel you do not expect any thread B to read from ShMem values deposited by thread A, then there is no need to use **volatile**

# Shared Memory: Historical Fact

- It used to be that any access to Shared Memory was a direct access (in compute capability 1.x)
- Fermi (2.x) and after: the load/store solution embraced can keep data into registers instead of moving them to ShMem
  - No guarantee for coherence between the shared memory block and the value stored in the register
- Problem is typically addressed by making that shared memory **volatile**:
  - In 1.x, this was always ok:  
`__shared__ int myShVars[256];`
  - Fermi and after, you might have to do this (prevents compiler from optimizing instructions related to **myShVars**):  
`volatile __shared__ int myShVars[256];`

```

#include<cuda.h>
#include<iostream>

_global_ void divergence(volatile int* data)
{
    int i = threadIdx.x;
    if ((i & 0x01) == 0)
    {
        data[i+1] = data[i+1] + i; // if even, come here
    }
    else
    {
        data[i] = data[i] + 2*i; // if odd, come here
    }
}

int main()
{
    const int numElems = 4;
    int hostArray[numElems], *devArray;

    //allocate memory on the device (GPU); zero out all entries in this device array
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    divergence <<<1, 4 >>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}

```



Linux, P100

```

euler ~/DanGitCAE/ME759/GPU> nvcc -arch=sm_60 volatileExample.cu
euler ~/DanGitCAE/ME759/GPU> srun -p slurm_shortgpu --gres=gpu:gtx1080:1 --pty -u ./a.out
srun: job 323889 queued and waiting for resources
srun: job 323889 has been allocated resources
Values stored in hostArray:
0
2
0
8
euler ~/DanGitCAE/ME759/GPU>

```

Windows, GTX1080

```

C:\Windows\system32\cmd.exe
Values stored in hostArray:
0
2
0
8
Press any key to continue . . .

```

# **volatile** vs. **\_\_threadfence()**: how different?

- **volatile** applies equally well to sequential computing
  - Not an appendage of CUDA, exists in the standard C
- **\_\_threadfence()** specific to parallel computing
  - Although this particular syntax is specific to CUDA, the idea of a memory fence is not

# Landmines everywhere???

- Is parallel computing riddled with landmines all over the place?
- If same memory location gets accessed by different threads: keep your eyes peeled
  - Up to you to take necessary steps to ensure proper sequence in memory operations (RAW, WAW, WAR)
  - The tools of the trade (or “tools of the thread”?):
    - Use of `volatile` qualifier
    - Thread synchronization
    - Memory fences
- Note: this transcends GPU computing, applies equally well to multi-core parallel computing

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 13

02/21/2020

# Quote of the day

“Let yourself be silently drawn by the strange pull of what you really love. It will not lead you astray.”

-- Rumi, Persian poet [1207 (Afghanistan) – 1273 (Turkey)] – the best selling poet in the US in 2017.

# Before we get going...

- Last time:
  - Wrap up memory ecosystem
    - Shared memory issues
    - Good global memory accesses
  - Discussion of parallel computing hazards: RAW, WAW, etc.
  - Memory consistency model
  - The use of thread fences & `volatile`
- Today:
  - Memory aspects: speed issues
  - Atomic operations
  - Case study: a reduction operation on the GPU
- Other tidbits:
  - Assignment due tonight at 9 pm
  - Please read assigned readings
  - Dan also has office hours online in Canvas each Wd at 7 PM
  - Next time:
    - Quick discussion about Final Project proposal

```
__device__ volatile int X = 1, Y = 2;

__device__ void writeXY()
{
    X = 10;
    Y = 20;
}

__device__ void readXY()
{
    int A = Y;
    int B = X;
}
```

# Landmines everywhere???

- Is parallel computing riddled with landmines all over the place?
- If same memory location gets accessed by different threads: keep your eyes peeled
  - Up to you to take necessary steps to ensure proper sequence in memory operations (RAW, WAW, WAR)
  - The tools of the trade (or “tools of the thread”?):
    - Use of `volatile` qualifier
    - Memory fences
    - Thread synchronization
- Note: this transcends GPU computing, applies equally well to multi-core parallel computing

# Ruminations, memory related

- Ruminations, on two topics
  - Memory operations, getting the result right (broad discussion, for parallel computing)
  - Memory operations, getting the result fast (narrower discussion, for GPU computing)

# Memory Access Issues

- Issue 1: Not all **global** memory accesses are equally efficient
  - How can you optimize memory access?
  - Question is highly relevant (for fast execution)
- Issue 2: Not all **shared** memory accesses are equally efficient
  - How can we optimize shared memory accesses?
  - Moderately relevant question
  - Discussed already, bottom line: avoid bank conflicts
- NOTE: Getting global memory access right has higher priority

# Data Access “Divergence”

- Concept is similar to thread divergence and often conflated
- Hardware is optimized for accessing contiguous blocks of global memory when performing loads and stores
- “block of global memory”: a 128-byte aligned chunk of 128 byte of memory
  - Becomes a GPU cache line, once it gets read in L2 (and possibly L1) memory

# Global Memory Access

- Two aspects of global memory access are relevant when fetching data into shared memory and/or registers
  - The **layout of the access** to global memory (the pattern of the access)
  - The **alignment** of the data you try to fetch from global memory
- **IMPORTANT:**
  - The two attributes in red above should be interpreted in the context of a memory transaction carried out by one warp of threads

# Comment, on the “layout of the access” attribute

- The basic idea:
  - Suppose each thread in a warp accesses a global memory address for a load operation at some point in the execution of the kernel
  - These threads can access global memory data that is either (a) neatly grouped, or (b) scattered all over the place
  - Case (a) is called a “**coalesced memory access**”
    - If you end up with (b) this will adversely impact the effective bandwidth of your program
- Analogy
  - It’s one thing to get the timber I need from one forest
  - Alternatively, in case I need timber from different forests several trips will be in order to get what I need

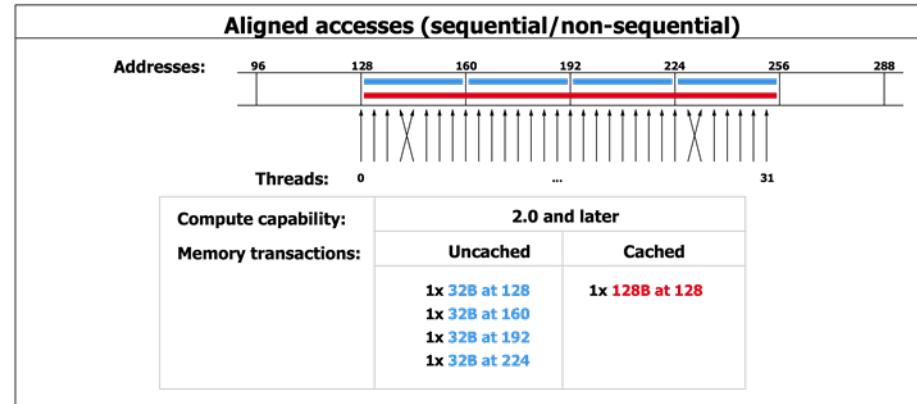
# Comment, on the “alignment” attribute

- Fact: Any address of memory allocated in device memory with `cudaMalloc` is a multiple of 256
- The “alignment” component of this story:
  - It’s a really nice if all threads in a warp access data inside only one memory block

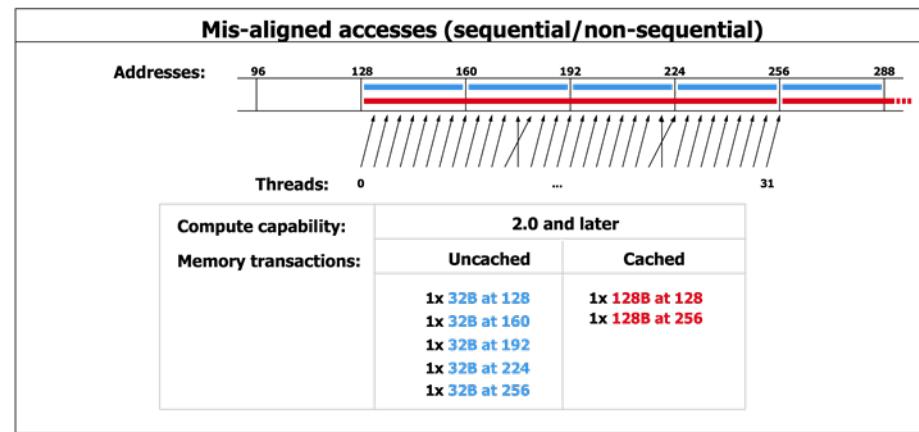
# Hitting the jackpot/winning the lottery

- Best memory accesses are simultaneously: (1) coalesced and (2) properly aligned

# Scenario A: Coalesced and aligned – great!



# Scenario B: Coalesced, but not aligned



# Why is this important?

- In **Scenario B** you have half the effective bandwidth you get in **Scenario A**
  - Just because of the alignment of your data access
- If your code is memory bound and dominated by this type of access, you might see a sizeable slow down...
- The moral of the story:
  - When you reach out to fetch data from global memory, [visualize how a full warp reaches out for access.](#)
    - Is the access coalesced and well aligned?

# A note on the effective bandwidth

- If you need data from multiple 128 byte-memory blocks, effective bandwidth goes down
- Reduction in effective bandwidth (assuming caching doesn't come into play)
  - Reduction equal to the number of 128-byte memory blocks you are hitting on
- Side effect: if you work with type **double**, effective bandwidth halves (vs. **float** or **int**)
  - **float & int**: 4 bytes ( $4 \times 32$  threads/warp = 128 bytes)
  - **double**: 8 bytes ( $8 \times 32$  threads/warp = 256 bytes → likely hitting at least 2 mem. blocks)

# Example: Adding Two Matrices

- You have two matrices A and B of dimension NxN (N=32)
- You want to compute C=A+B in parallel
- Code provided below (some details omitted, such as **#define N 32**)

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

# Test Your Understanding

- Given that the  $x$  field of a thread index changes the fastest, is the array indexing scheme on the previous slide good or bad?
  - HINT: think how threads are mapped into a set of 1D sequence of consecutive IDs
- The “good or bad” refers to how data is accessed in the device’s global memory
- In other words should we have

or...

$$C[i][j] = A[i][j] + B[i][j]$$

$$C[j][i] = A[j][i] + B[j][i]$$

# Reflecting on Accessing Global Memory in CUDA

- Assume `dA` is a device array, holding `ints`
- Assume `z []` is an array of `unsigned ints`
- Which of these global memory accesses are good/bad? Why?
  1. `dA[threadIdx.x]`
  2. `dA[threadIdx.x+2]`
  3. `dA[3*threadIdx.x+2]`
  4. `dA[z[threadIdx.x]]`
- This might be helpful: When answering 1-4 above, imagine you're a thread and try to understand what the other 31 threads in the warp do when the warp executes some memory instruction

# How Do You Avoid Bad Memory Accesses?

- Design your data structures in a way that leads to advantageous memory accesses
  - Choosing a design: done before you write any line of code
  - Reflect on your algorithm
    - What data it needs, how much of it, where you read from/write to, how often
- A good data structure design for parallel computing is likely different than a good data structure design for sequential computing
- I learn programming exclusively thinking about **correctness**
  - Other aspects to consider when writing code:
    - speed of execution & convenience & productivity & code legacy & large team development & portability

# Related to Data Organization

- Say you use in your program complex data constructs that could be organized using C-structures
- For GPU computing, how is it more advantageous to store data in global memory?
  - Alternative A: as an array of structures (AoS)
  - Alternative B: as a structure of arrays (SoA)
- If you use all the data in a `struct` once you bring over, question not that relevant (although there are aspects that come into play)
- However, if you only use one field from a `struct`, stick w/ SoA

# SoA or AoS?

- Collection of  $N = 10,000$  points in 3D
- Need to compute difference between the  $x$  coordinates of a bunch of points:

$$|x_i - x_j| \text{ for } 0 \leq i, j < N$$

- However, you don't use the  $y$  and  $z$  coordinates of your points

- Should I work w/ AoS or SoA?
- What's a good data structure for this problem?

# Latency vs. Bandwidth, or how the CPU and GPU go about hiding memory access overhead

- CPU uses the low latency of the cache to hide memory access overhead
  - Caches have small latency; bandwidth though is that of the system memory
    - Why? Caches are small, 256 KB, doesn't make sense to talk about bandwidth when all you have is 0.000256 GB
- GPU hides memory latency by using a combination of two things: SM oversubscription & high bandwidths
- Caveat: data needs to find its way into the GPU global memory
  - PCIe – a good I/O bus, but not a good memory bus
  - NVLINK: a good memory bus

# Atomic Operations

# Coordinating Memory Operations to Avoid Data Hazards

- Accesses to shared locations (global memory & shared memory) need to be correctly choreographed (orchestrated) to avoid race conditions
- In many common shared memory multithreaded programming models, one uses coordination mechanisms such as [locks](#) to choreograph accesses to shared data
- CUDA has a scalable coordination mechanism called “[atomic memory operation](#)”

# Before diving in

- Next slides: several examples that show “race conditions”

# Recall a CUDA Early Example

```
#include<cuda.h>
#include<iostream>
__global__ void simpleKernel(int* data)
{
    //this adds a value to a variable stored in global memory
    data[threadIdx.x] = data[threadIdx.x] + 2*(blockIdx.x + threadIdx.x);
}

int main() {
    const int numElems = 4;
    int hostArray[numElems], *devArray;

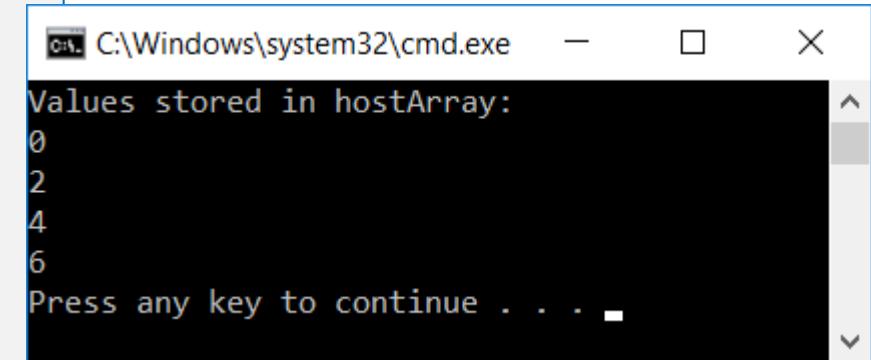
    //allocate memory on the device (GPU); zero out all entries in this device array
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1,4>>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
Values stored in hostArray:
0
2
4
6
Press any key to continue . . .
```

# Old Question...

- In our code, we invoked the kernel like this:

```
simpleKernel<<<1,4>>>(devArray)
```

- What would happen if we invoke the kernel like this:

```
simpleKernel<<<2,4>>>(devArray)
```

# Another Race Condition

- A contrived (cooked up) example...

```
// update.cu
__global__ void update_race(int* x, int* y)
{
    int i = threadIdx.x;
    if (i < 2)
        *x += *y;
    else
        *x += 2*i;
}

// main.cpp
update_race<<<1,4>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

# Example: Inter-Block Issue

- Would this fly?

```
// update.cu
__global__ void updateVals(int* x)
{
    int i = threadIdx.x;
    if (i == 0) *x += blockIdx.x;
}

// main.cpp
// assume d_x starting value is zero
updateVals<<<2,5>>>(d_x);
```



# Atomics, Introduction

- Atomic memory operations (atomic functions) are used to solve **access coordination** problems in parallel computing
- General concept: provide a mechanism for a thread to update a memory location such that the update appears to happen **without interruption** (atomically) with respect to all other threads
- This ensures that all atomic updates issued are performed (**in some unspecified order**) to completion and that all threads can observe all updates

# Atomic Functions, Example [1/3]

- Atomic functions perform read-modify-write operations on data residing in global or shared memory

```
//example of int atomicAdd(int* addr, int val)
__global__ void update(unsigned int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = atomicAdd(x, i);      // j is now old value of x;
}

//-----
// snippet of code in main.cu; mem. was already allocated on device
int x = 0;
cudaMemcpy(&d_x, &x, cudaMemcpyHostToDevice);
update<<<1,128>>>(&d_x);
cudaMemcpy(&x, &d_x, cudaMemcpyDeviceToHost);
```

Side-note question: in example below, are we working w/ global or shared memory?

# Atomic Functions [2/3]

- Atomic functions perform read-modify-write operations on data that can reside in **global** or **shared** memory
- Synopsis of atomic function **atomicOP(a,b)** is typically

```
t1 = *a;           // read
t2 = (*a) OP (*b); // modify
*a = t2;          // write
return t1;
```

- All statements are executed atomically without interruption by any other function/process
- The atomic function returns the initial value, **\*not\*** the final value, stored at the memory location

# Atomic Functions [3/3]

- The order in which concurrent atomic updates are performed is not defined, and may appear arbitrary
- While order is not clear, none of the atomic updates will be lost
- Several different kinds of atomic operations:
  - Add (add), Sub (subtract), Inc (increment), Dec (decrement)
  - And (bit-wise and), Or (bit-wise or) , Xor (bit-wise exclusive or)
  - Exch (Exchange)
  - Min (Minimum), Max (Maximum)
  - Compare-and-Swap
  - Etc.

# Histogram Example

```
// Compute histogram of colors in an image
//
// picturePixels - pointer to picture pixels, each w/ its own color
// picturePixels[i] - takes a value between 0 and 6 (7 colors total)
// bucket - pointer to histogram bucket of size equal to # of colors
//
__global__ void histogram(int n, int* picturePixels, int* bucket)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n)
    {
        int c = picturePixels[i];
        atomicAdd(&bucket[c], 1);
    }
}
```

# Performance Notes

- While very convenient to use, atomics do impact the speed of the code
- Performance poor when many threads attempt to perform atomic operations on a small number of locations
  - Bad case: all threads trying to update the same variable

# No mix-and-match, please...

- Atomic updates are not guaranteed to appear atomic to concurrent accesses that also use loads and stores
  - No mix and match, please

```
__global__ void broken(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0)
    {
        *x = *x + 1;
    }
    else
    {
        int j = atomicAdd(x, 1); // j = *x; *x += i;
    }
}

// main.cpp
broken<<<1,128>>>(128, d_x); // d_x = d_x + {1, 127, 128}
```

# Rules of thumb, atomic ops

- When to use: Cannot fall back on normal memory operations because of possible race conditions
- Use for infrequent, sparse, and/or unpredictable global communication
- Use shared memory and/or customized data structures & algorithms to avoid synchronization whenever reasonable
- Recent compute capabilities (post Maxwell) improved the speed of atomic operations

# Synchronization vs. Coordination

- There is a qualitative difference between a `__syncthreads()` function and an atomic operation
  - `__syncthreads()` has the connotation of barrier; i.e., of synchronization
    - `__syncthreads()` establishes a point in the execution of the kernel that every thread in the **\*\*block\*\*** needs to reach before any block thread can move beyond that point. Also, all memory ops are seen through
  - The “atomic operation” concept instead tied to the idea of coordination in relation to operations that involve `memory` transactions
    - Threads *need not* synchronize their execution, it’s only that a certain memory operation in a kernel is conducted in an atomic fashion

# Resource Management Considerations

# What is “Resource Management”?

- The GPU is a resourceful device
- How do you get to use the card at capacity?
  - “used at capacity”: SM executes the max. number of warps it can possibly host
- The three factors that come into play are
  - How many threads you decide to use in each block
  - What register requirements end up associated with a thread
  - How much shared memory you assign to one block of threads
  - **threads/block = ?**
  - **registers/thread = ?**
  - **shMem/block = ?**

# Some Hard Constraints (1/2)

- Max number of **warps** that one SM can service simultaneously:
  - 32 on Tesla C1060 and Turing (wow)
  - 48 on Fermi
  - 64 on Kepler, Maxwell, Pascal, and Volta
- Max number of **blocks** that one SM can process simultaneously:
  - 8 on Fermi
  - 16 on Kepler and Turing (wow)
  - 32 on Maxwell, Pascal, and Volta

## Some Hard Constraints (2/2)

- The number of **32-bit registers** available on each SM is limited:
  - 16,384 registers (on Tesla C1060)
  - Roughly 48,000 on Fermi
  - Roughly 64,000 on Kepler, Maxwell, Pascal, and Volta
- The amount of **shared memory** available to each SM is limited
  - 16 KB on Tesla 1060
  - 64 KB on Fermi (16/48 or 48/16 configurable between L1 and shared memory)
  - 64 KB on Kepler (16/48 or 48/16 or 32/32 configurable)
  - 64 KB on Maxwell and Pascal, but not split w/ L1
  - 96 KB on Volta (this is split-able again, w/ L1 and texture)
- Note: there is also a max amount of ShMem that a block can get assigned
  - At 48 KB , sometimes 64KB, sometimes 96 KB

# The Concept of Occupancy (1/2)

- Let's talk Pascal on this slide (discussion similar for Volta, Maxwell, Kepler, etc.)
  - On Pascal, you want to have up to 64 warps serviced at the same time by one SM (that is, 64 warps in flight)
    - High warp count: increases likelihood of hiding mem access latencies with useful computation
  - Occupancy examples:
    - Four blocks with 512 threads running together on one SM: 100% occupancy
    - Four blocks of 256 threads each running on one SM: 50% occupancy
    - 96 blocks with 32 threads each – oops, can't have more than 32 blocks on a Volta SM
      - Effectively this scenario gives you at most 50% occupancy (up to 32 blocks w/ 32 threads each)

# The Concept of Occupancy (2/2)

registers	↔	per thread
shared mem	↔	per block

- What prevents you from getting high occupancy?
  - Amount of shared mem demanded by each block
    - Total amount of shared memory in one SM is limited: up to 64 Kb on Pascal
  - Number of registers used by each thread
    - Size of the register file in one SM: 64K four byte registers on Kepler, Maxwell, Pascal, etc.
- How do you find out how many registers and shared memory get used?
  - Solution 1: using the CUDA profiler
  - Solution 2: use the flag `-ptx-options=-v` when you compile your kernel w/ `nvcc`

# Examples, Occupancy of hardware (Fermi example, 48 KB of ShMem)

- **Example 1, Fermi:** If each of your block demands 80 KB of shared memory, the kernel will fail to launch
  - Not enough memory on the SM to run even a block
- **Example 2, Fermi:** If your blocks each uses 15 KB of shared mem, you can have up to three blocks running on one SM (there will be some shared mem that will go unused)
- **Example 3, Fermi:** Like Example 2 above, and you have 512 threads per block, each thread uses 30 registers. Will one SM be able to handle **2** blocks?
  - Total number of registers:  $512 \times 2 \times 30 = 30,720$  out of the 48,000 are used **OK**
  - Amount of shared memory:  $2 \times 15K = 30$  KB, well below 48 KB **OK**
  - Number of warps:  $2 \text{ blocks} \times 512 \text{ threads} = 1024 \text{ threads} = 32 \text{ warps} < \text{max of } 48$  **OK**
  - **Quiz:** Will the SM be able to handle 3 blocks?

# NVIDIA CUDA Occupancy Calculator

- There is an “occupancy calculator” that can tell you what percentage of the HW gets utilized by your kernel
- Comes as an Excel spreadsheet
- Requires the following input
  - Compute capability
  - Threads per block
  - Registers per thread
  - Shared memory per block
  - Selected Shared Memory Size Config (bytes)
  - Selected Global Load Caching Mode
- Search online “CUDA occupancy calculator” to get the excel file

# NVIDIA CUDA Occupancy Calculator

CUDA\_Occupancy\_calculator.xls [Compatibility Mode] - Excel

File Home Insert Page Layout Formulas Data Review View Add-Ins LOAD TEST Acrobat Team Tell me what you want to do...

Normal 2 Normal Bad Good Neutral Calculation Check Cell Explanatory... Followed Hyp... Hyperlink

Conditional Format as Table

General \$ % .,.00 .,.00

Font Alignment Number

Styles Cells Editing

R54 A B C D E F G H I J K L M N O P Q R S T

## CUDA Occupancy Calculator

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

1.) Select Compute Capability (click): 6.0 (Help)

1.b) Select Shared Memory Size Config (bytes): 65536

1.c) Select Global Load Caching Mode: L2 only (cg)

2.) Enter your resource usage:

Threads Per Block: 128 (Help)

Registers Per Thread: 48

Shared Memory Per Block (bytes): 4096

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:

Active Threads per Multiprocessor: 1280 (Help)

Active Warps per Multiprocessor: 40

Active Thread Blocks per Multiprocessor: 10

Occupancy of each Multiprocessor: 63%

Physical Limits for GPU Compute Capability: 6.0

Threads per Warp: 32

Max Warps per Multiprocessor: 64

Max Thread Blocks per Multiprocessor: 32

Max Threads per Multiprocessor: 2048

Maximum Thread Block Size: 1024

Registers per Multiprocessor: 65536

Max Registers per Thread Block: 65536

Max Registers per Thread: 256

Shared Memory per Multiprocessor (bytes): 65536

Max Shared Memory per Block: 49152

Register allocation unit size: 256

Register allocation granularity: warp

Shared Memory allocation unit size: 256

Warp allocation granularity: 2

Allocated Resources: = Allocatable

	Per Block	Limit Per SM	Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	4	64	16
Registers (Warp limit per SM due to per-warp reg count)	4	42	10
Shared Memory (Bytes)	4096	49152	16

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor: Blocks/SM \* Warps/Block = Warps/SM

Limited by Max Warps or Max Blocks per Multiprocessor: 16

Limited by Registers per Multiprocessor: 10 4 40

Limited by Shared Memory per Multiprocessor: 16

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64  
Occupancy = 40 / 64 = 63%

CUDA Occupancy Calculator Version: 7.5 Copyright and License

Impact of Varying Block Size

Impact of Varying Shared Memory Usage Per Block

Impact of Varying Register Count Per Thread

Calculator Help GPU Data Copyright & License

# Example: Occupancy Study

[NOTE: Specific to Fermi]

- For Matrix Multiplication example (with shared memory), should I use 8x8, 16x16, or 64x64 threads per blocks?
  - For 8x8, we have 64 threads per Block. Since each Fermi SM can manage up to 1536 resident threads, it could take up to 32 Blocks. However, each SM is limited to 8 resident Blocks, so only 512 threads will go into each SM!
  - For 16x16, we have 256 threads per Block. Since each Fermi SM can take up to 1536 resident threads, it can take up to 6 Blocks unless other resource considerations overrule.
    - Next you need to see how much shared memory and how many registers get used in order to understand whether you can actually have four blocks per SM
  - 64x64 is a no starter, you can only have up to 1024 threads in a block, the tile cannot be this big

# Occupancy != Performance

[yet a pretty good proxy]

- Increasing occupancy does not necessarily increase performance
  - If you want to read more about this, there is a Volkov paper, assigned reading
  - What controls the damage is the Instruction Level Parallelism (ILP) that the compiler can capitalize on (better efficiency in pipelining, instruction reordering, etc.)

HOWEVER,

- Low-occupancy multiprocessors are likely to have a hard time when it comes to hiding latency on memory-bound kernels
  - This latency hiding draws on Thread Level Parallelism (TLP); i.e., having enough threads (warps, that is) that are ready for execution

# Parameterize Your Application

- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
  - # of SMs
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Max. threads per block
  - Max. number of warps per SM
- You can even make apps self-tuning (like FFTW and ATLAS)
  - “Experiment” mode discovers and saves optimal configuration

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 14

02/24/2020

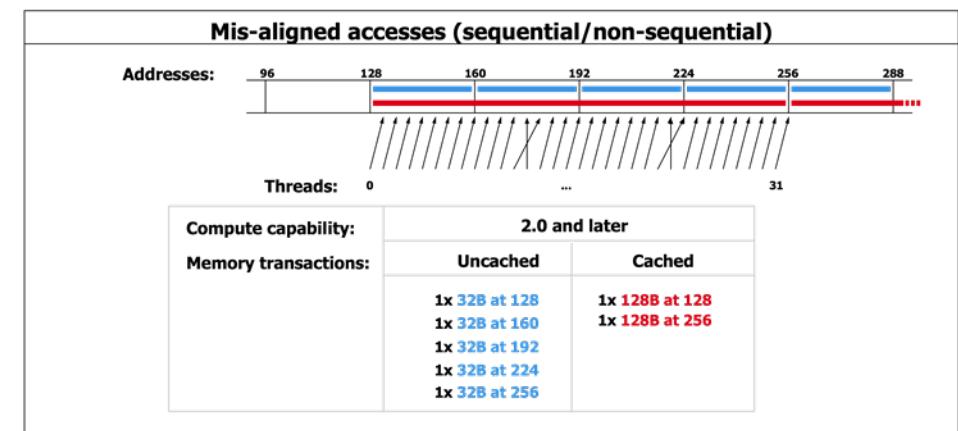
# Quote of the day

“You see things; you say, 'Why?' But I dream things that never were; and I say 'Why not?’”

-- George Bernard Shaw, Irish playwright, 1925 Nobel prize winner in Literature, [1856 - 1950]

# Before we get going...

- Last time:
  - Memory aspects: speed issues
  - Atomic operations
- Today:
  - Rules of thumb, GPU computing optimization
  - Test case: reduction operation
  - Test case: prefix scan operation
- Other tidbits:
  - Assignment due on Thursday at 9 pm
  - Please read assigned readings
  - Dan also has office hours online in Canvas each Wd at 7 PM



# ME759 Final Project

- Final Project Proposal due on March 27 (9 PM, in Canvas)
  - You'll receive feedback by April 3
- There will be a template made available
  - Two pages long proposal
  - Template available in two weeks
- Final Project due at the time when the Final Exam starts
  - Exact date TBA

# Final Project

- Final Project - accounts for 25% of final grade
- Your work on Final Project to start no later than April 4:
  - It is an individual project or outcome of a 2- or 3-student team
  - You choose a problem that suites your research or interests
  - I encourage you to tackle a meaningful problem
    - Attempt to solve a useful problem rather than a problem that you are confident that you can solve
    - Projects that are not successful are ok, provided you aim high enough and demonstrate good progress
      - “demonstrate good progress”: we’ll look at your git history to understand level of effort and progress towards goal

# Final Project: Default Option

- Default Final Project: work on a computational dynamics code that runs in parallel
  - Profile existing code
  - Improve performance of the implementation via parallel computing
- There might be other project[s] to chose from

# CUDA Optimization Rules of Thumb

# Writing CUDA Software: High-Priority Recommendations

1. To get the maximum benefit from CUDA, focus first on finding ways to **parallelize sequential code**. Expose fine grain parallelism
2. **Minimize data transfer** between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU

Very good resource: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#abstract>

# Writing CUDA Software: High-Priority Recommendations

3. Strive to have **aligned and coalesced global memory accesses**. Design your implementation such that global memory accesses are coalesced for that part of the red-hot parts of the code
4. Minimize the use of global memory. Prefer **shared memory** access where possible (consider tiling as a design solution)

# Writing CUDA Software: Medium-Priority Recommendations

1. Accesses to shared memory should be designed to avoid serializing requests due to **bank conflicts**
2. Maintain sufficient numbers of active threads per multiprocessor (i.e., sufficient occupancy)
3. Keep the number of threads per block a **multiple of 32** to avoid wasted lanes

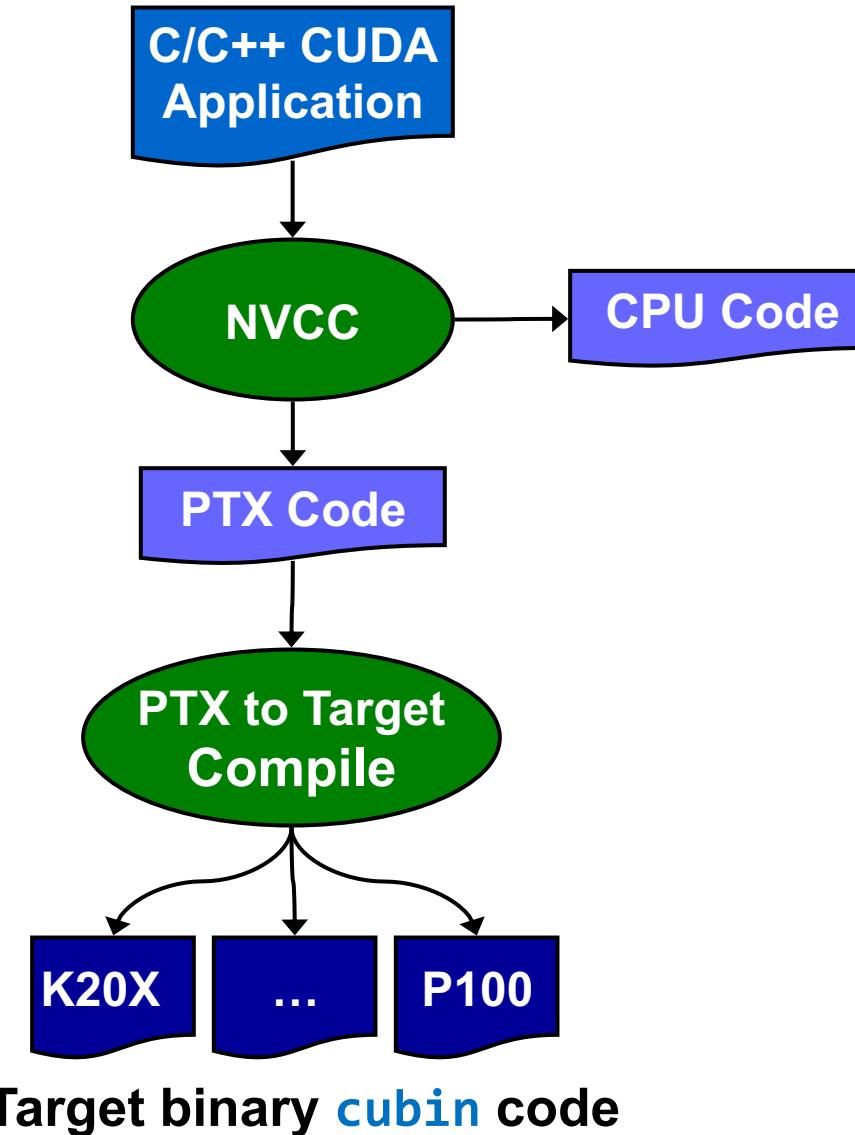
# Writing CUDA Software: Medium-Priority Recommendations

4. Use the **fast math library** whenever speed is very important and you can live with a tiny loss of accuracy
5. Avoid thread divergence

Some nuts and bolts, compiler related and such

# Compiling CUDA Code

[with nvcc driver]



# PTX: Parallel Thread eXecution

- PTX: an assembly language used in CUDA programming environment
- **nvcc** translates code written in CUDA's C into the language-agnostic Intermediate Representation, and then into PTX assembly
- **nvcc** subsequently invokes an assembler which translates the PTX into a binary code which can be run on a certain GPU

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

## PTX for fillKernel

```
.entry _Z10fillKernelPii (
.param .u64 __cudaparm_Z10fillKernelPii_a,
.param .s32 __cudaparm_Z10fillKernelPii_n)
{
.reg .u16 %rh<4>;
.reg .u32 %r<6>;
.reg .u64 %rd<6>;
.reg .pred %p<3>;
.loc      14      5      0
$LDWbegin_Z10fillKernelPii:
mov.u16 %rh1, %ctaid.x;
mov.u16 %rh2, %ntid.x;
mul.wide.u16 %r1, %rh1, %rh2;
cvt.u32.u16 %r2, %tid.x;
add.u32 %r3, %r2, %r1;
ld.param.s32 %r4, [__cudaparm_Z10fillKernelPii_n];
setp.le.s32 %p1, %r4, %r3;
@%p1 bra $Lt_0_1026;
.loc      14      9      0
ld.param.u64 %rd1, [__cudaparm_Z10fillKernelPii_a];
cvt.s64.s32 %rd2, %r3;
mul.wide.s32 %rd3, %r3, 4;
add.u64 %rd4, %rd1, %rd3;
st.global.s32 [%rd4+0], %r3;
$Lt_0_1026:
.loc      14      11      0
exit;
$LDWend_Z10fillKernelPii:
}
```

# More on the nvcc compiler

File suffix	How the nvcc compiler interprets the file
.cu	CUDA source file, containing host and device code
.cuh	CUDA header files, containing host and device code
.cup	Preprocessed CUDA source file, containing host code and device functions
.c	C source file
.cc, .cxx, .cpp	C++ source file
.gpu	GPU intermediate file (device code only)
.ptx	PTX intermediate assembly file (device code only)
.cubin	CUDA device only binary file

# The JIT story

- C code gets converted into an Intermediate Representation (IR) called NVVM IR
- NVVM IR is then converted to PTX assembly
- PTX assembly gets assembled into binary code for some number of GPU targets (“cubin”)
- Why the intermediate steps?
- Brings into the picture the concept of just-in-time (JIT) compiling
  - Compilers for other languages can generate this intermediate representation rather than implementing a dialect of PTX assembly for each frontend language (Clang and CUDA Fortran do this)
  - PTX material can be converted at run-time into cubin instructions
- Why bother with JIT?
  - CON: JIT increases the load-time of an application (since an additional compile step needs to happen)
  - PRO: Allows the app to benefit from newer devices (better compilers, more advanced features supported by the hardware)
    - The only way an app can run on a device that didn’t exist at the time the PTX was generated

# JIT and the magic of –code and –arch nvcc flags

- How does one control how PTX is generated?
- How does one control how the cubin is generated?

# Getting stuck with a CC, via the -code flag

- Compiling to produce cubin binary code for CC 3.5 done like this

```
>> nvcc -code=sm_35 ...otherstuff...
```

- You are stuck with 3.5 CC, you can't run this on a device of CC 4.0
  - Rule: A cubin object generated for CC  $X.y$  will only run on hardware of CC  $X.z$ , where  $z \geq y$
- Using the nvcc command above is ok as long as you know *\*for sure\** that you only run on a certain card (like your personal laptop, office desktop, etc.)
- Not good if you want to distribute the app to be run by other folks with unknown hardware

# A word on PTX compatibility, and the role of the –arch flag

- Some CUDA features supported only on devices of higher CC
  - Example: unified (managed) memory
- You can control the C to PTX translation so that you do/don't pick certain CUDA features
- Example: Warp Shuffle Function available only in CC 3.0 and above
  - Code that contains Warp Shuffle functionality should be compiled with this in mind
    - Accomplished by the use of the –arch=compute\_30 flag on nvcc
    - >> nvcc –arch=compute\_30 ...blahblah...
    - NOTE: the PTX generated by the above can then be compiled to cubin to run on higher CC
      - Recall the cubin generation control by the –code flag

# Example, combining -arch and -code via -gencode

```
>> nvcc x.cu
    -gencode arch=compute_35,code=sm_35
    -gencode arch=compute_50,code=sm_50
    -gencode arch=compute_60,code=\`compute_60,sm_60\`
```

- The cubin fat binary generated by this command embeds the following (it's got bigger footprint to accommodate all):
  - Binary code generated to work on Kepler (CC 3.5)
  - Binary code generated to work on Maxwell (CC 5.0)
  - PTX and binary code to work on Pascal (CC 6.0)
- The arch part says that the binary code in each case should be produced based on PTX code generated as you'd expect: binary for 3.5 comes from PTX for 3.5, binary for 5.0 comes from PTX for 5.0, binary for 6.0 comes from PTX for 6.0
- Shortcut: you can get `-gencode arch=compute_60,code=\`compute_60,sm_60\`` by simply saying :  
`>> nvcc x.cu -arch=sm_60`
- If you don't say anything but "nvcc x.cu" the compile driver will use default arch & code settings

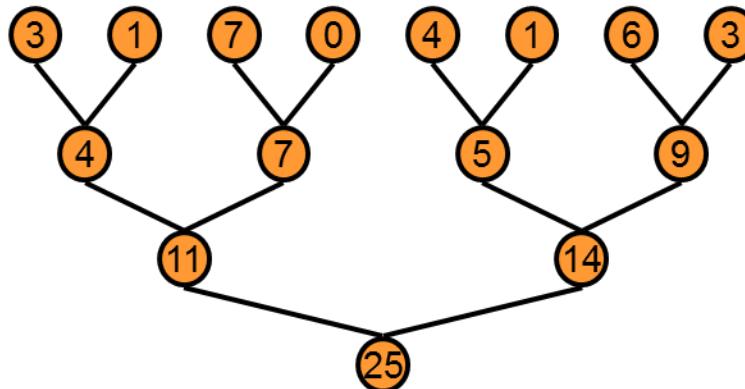
# CUDA Case Study: Parallel Reduction

# Parallel Reduction in CUDA

- Exercise draws on material made available by Mark Harris of NVIDIA
- Parallel Reduction: Common and important data parallel primitive
  - Example: Used to compute the norm of a large vector
- Easy to implement in CUDA
  - Challenging to get it to run fast though
- Serves as a good optimization example
  - Walk step by step through several different versions
  - Demonstrates several important optimization strategies
- Results are for old CC, yet it's instructive to understand how far folks went to get top performance

# Parallel Reduction

- Basic Idea: tree-based approach used within each thread block



- Backdrop: assume you have very large arrays – 100,000 entries and beyond
  - Keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array to one single value
- Q: How do we communicate partial results between thread blocks?

# Problem: Global Synchronization

- If we could synchronize across all thread blocks, could easily reduce very large arrays
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- But CUDA has no global synchronization
  - Only synchronization of threads that belong to the same block
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible overhead

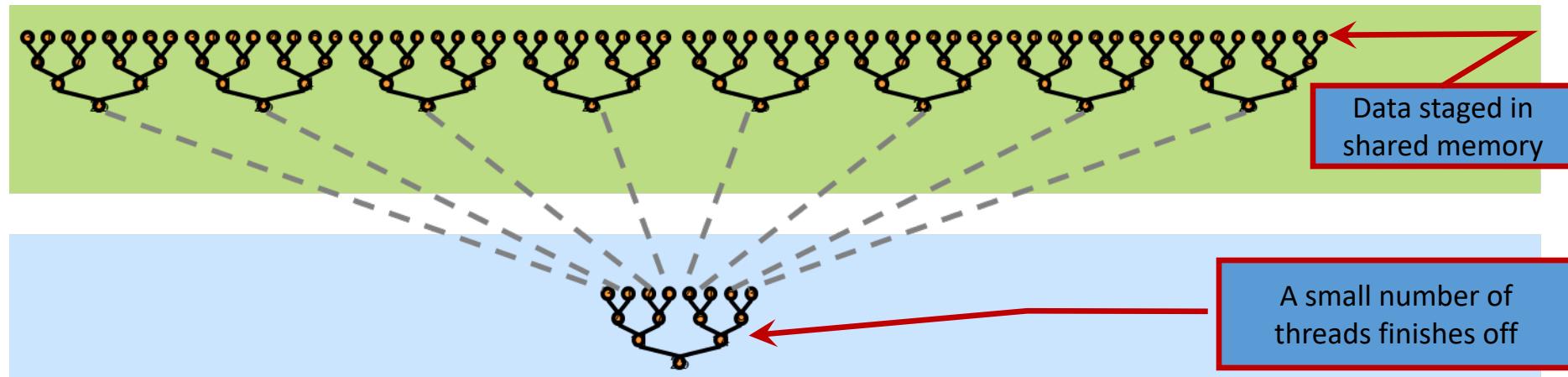
# Multiple Kernel Calls

[An Example, and how it all works out...]

- Imagine you launch a 1D grid in which each 1D block has 256 threads
- Assume that the number of elements in the array is N=100,000
  - Note that  $100,000 = 390 * 256 + 160$ , therefore  $\lceil (N+255) / 256 \rceil = 391$  blocks needed
- For the first stage, you launch 391 blocks of 256 threads
  - At the end of this stage you still have to operate on 391 elements
- For the second stage, you launch two blocks of 256 threads
  - At the end of this stage you only have to operate on two elements
- For the third and last stage, you launch one block of 32 threads
  - Almost all threads will be idle in this case...
- NOTE: after the first stage, each subsequent stage operates on a number of entries equal to the number of blocks in the previous stage

# Vector Reduction: 30,000 Feet Perspective

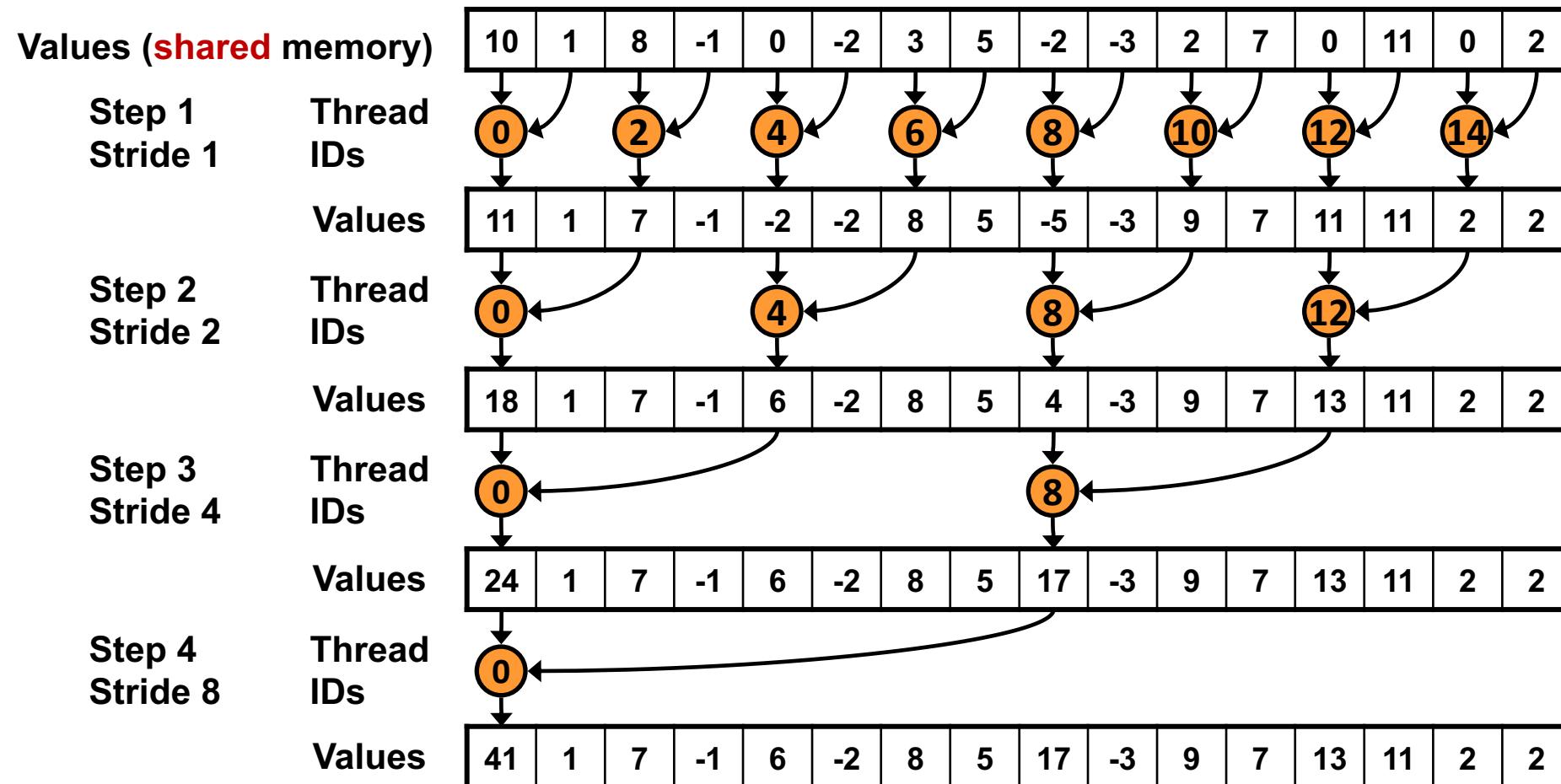
- At the block level: Bring data in **shared memory**, then start adding in parallel
- Fewer and fewer threads of a block participate
- Recall that there is no synchronization between threads in different blocks
- The process is memory bound, low arithmetic intensity...



# What is Our Optimization Goal?

- We should strive to reach GPU peak performance
  - Choose the right metric:
    - GFLOP/s: for compute-bound kernels
    - Bandwidth: for memory-bound kernels
- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)
  - Therefore we should strive for peak bandwidth
- We'll go through an old example - results generated using an old G80 GPU
  - Compute capability (CC) 1.0
  - 384-bit memory interface, 900 MHz DDR
  - $384 * 0.900 * 2 / 8 = 86.4 \text{ GB/s}$
  - Example carries over to other CCs – on NVIDIA hardware this algorithm will always be memory bound

# Parallel Reduction: Interleaved Addressing



**Note: in stage  $s$ , only threads divisible to  $2^{s-1}$  get to do work. Stride:  $2^{s-1}$**

# Reduction #1: Interleaved Addressing

```
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global memory
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Why do we need  
to sync threads?

First, threads w/ ID multiple of 2 do work  
Next, threads w/ ID multiple of 4 do work  
Next, threads w/ ID multiple of 8 do work  
And so on...

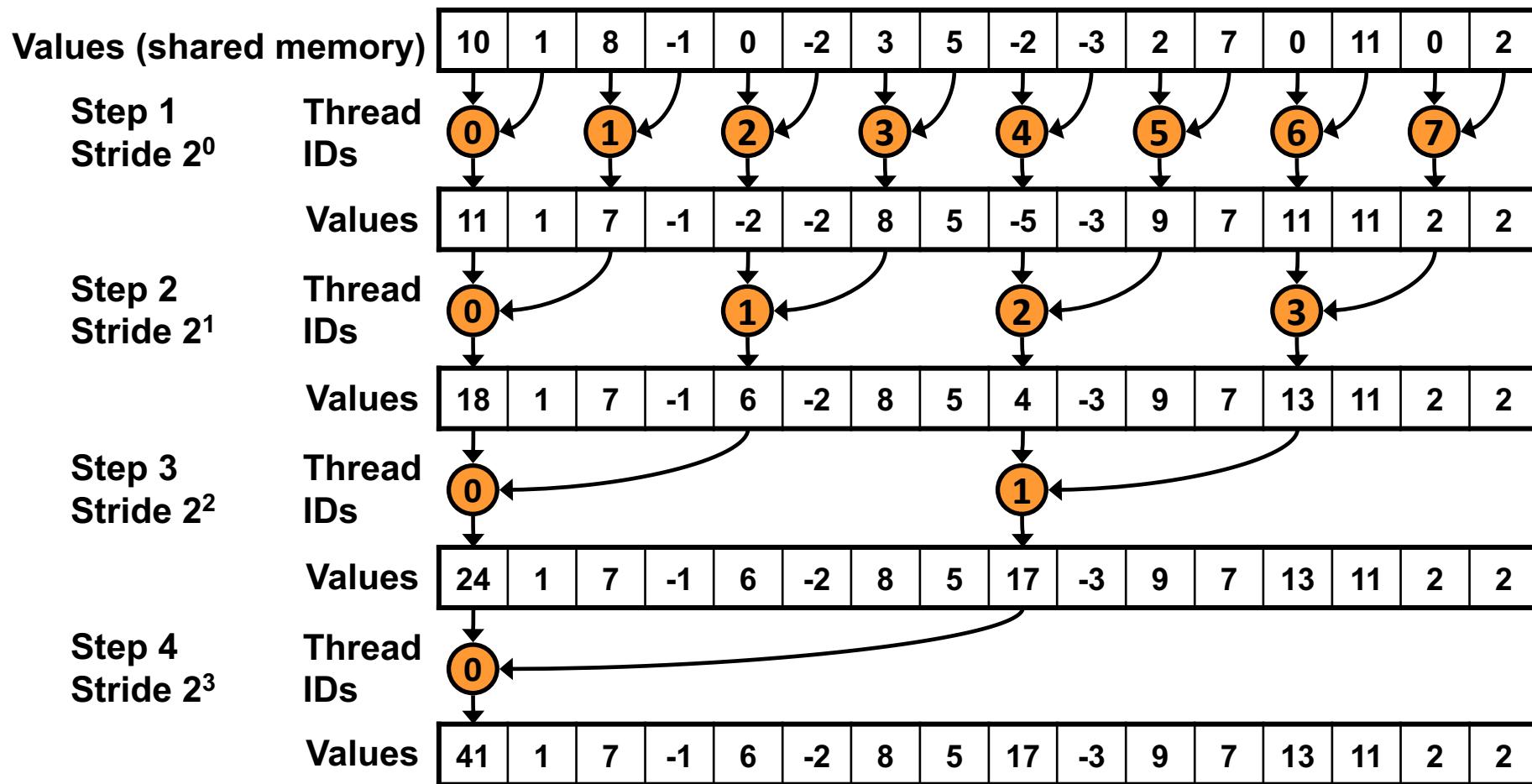
**Problem: highly divergent  
warps are inefficient, and %  
operator is very slow**

# Performance for 4 Million element reduction

	<b>Time (<math>2^{22}</math> ints)</b>	<b>Bandwidth</b>
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>

Note: Block Size = 128 threads for all tests

# Parallel Reduction: Interleaved Addressing



New Problem: Shared Memory Bank Conflicts

# Reduction #2: Interleaved Addressing

Just replace divergent branch in inner loop...

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

...with strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

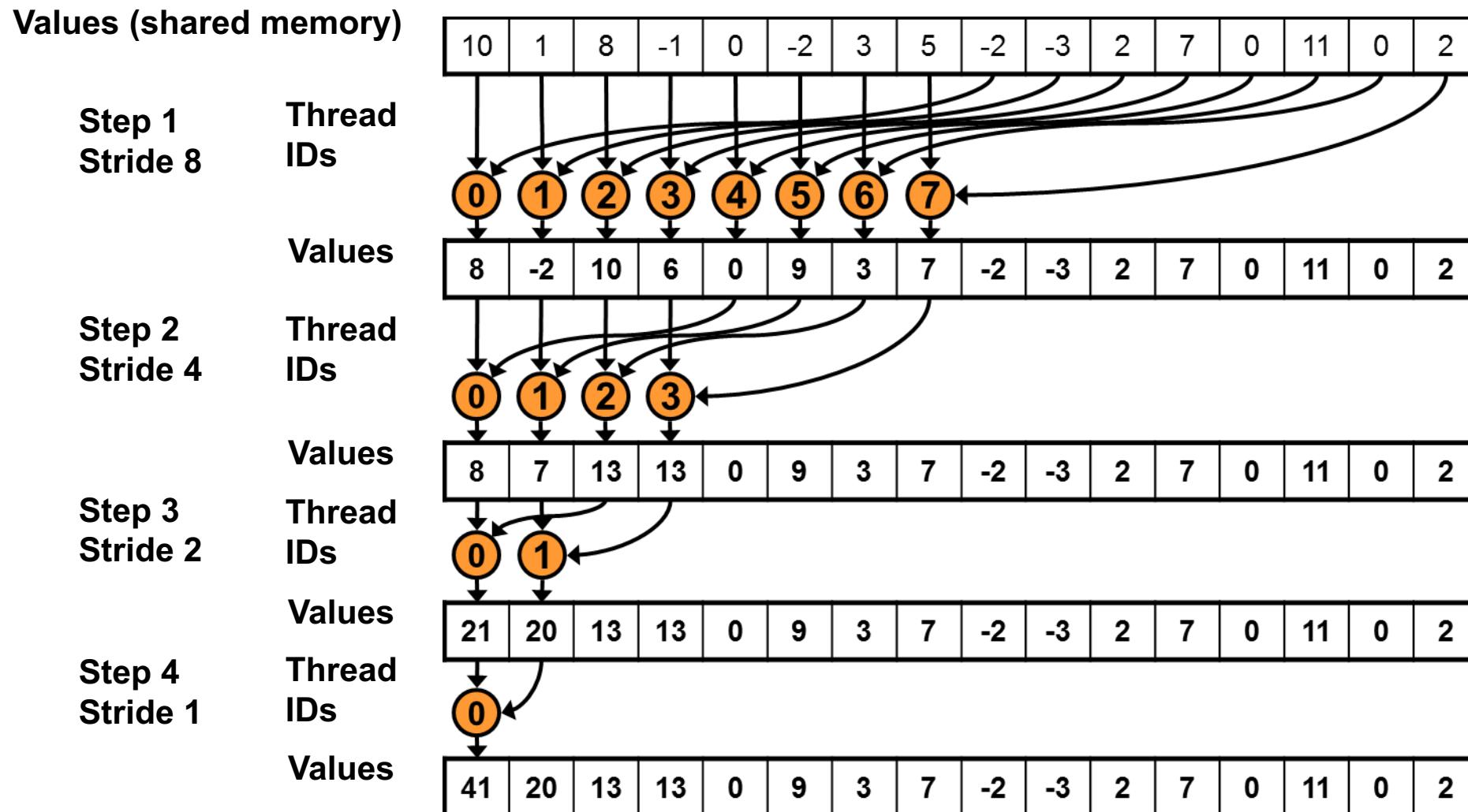
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

Previous case: one thread works, several next to it don't.  
This case: thread A works & A+1 works too, except that A+1 goes far from the memory location where thread A goes

# Performance for 4M element reduction

	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>

# Parallel Reduction: Sequential Addressing



Sequential addressing is Shared Mem. conflict free

# Reduction #3: Sequential Addressing

Just replace strided indexing in inner loop...

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

...with **reversed** loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

- First you go out 1/2 the block size to match a thread with its second operand
- Then you go out 1/4 of the block size
- Then you go out 1/8 of the block size
- Etc.

# Performance for 4M element reduction

	Time (2 <sup>22</sup> ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>

# Idle Threads...

Current solution:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Half of the threads are idle on first loop iteration. This is wasteful...

## Reduction #4: First Add During Load

Replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i   = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

...With two loads and first add of the reduction:

```
// perform first level of reduction upon reading from
// global memory and writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i   = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

**Side-effect: number of blocks needed half of what it used to be...**

# Performance for 4M element reduction

	Time (2 <sup>22</sup> ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>

# Instruction Bottleneck

- At 17 GB/s, we're far from bandwidth bound
- Therefore a likely bottleneck is instruction overhead
  - Ancillary instructions that are not loads, stores, or core arithmetic
  - In other words: **address arithmetic** and **loop overhead**
- Strategy: unroll loops

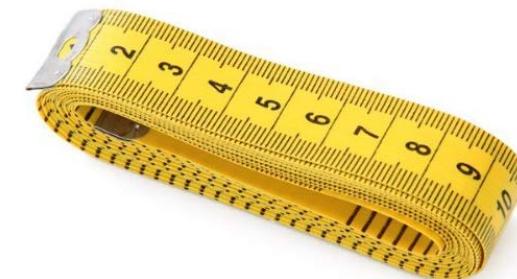
# Unrolling the Last Warp

- As reduction proceeds, the number of “active” threads decreases
  - When `s <= 32`, we have only one warp left
- Instructions executed in lockstep fashion within a warp (not on Volta, though!)
- That means when `s <= 32`:
  - We don’t need to `_syncthreads()`
  - We don’t need “`if (tid < s)`” because it doesn’t save any work
- The key idea: unroll the last 6 iterations of the inner loop, which involve 32 or less threads

“unroll the last 6 iterations of the inner loop, which involve 32 or less threads”

Legend:

- Entry we care about, when last thread works on data
- Entry that is read in order to update a “blue” entry



[<https://www.amazon.com/ZXUY-Measurement-Also-Centimetre-Reverse/dp/B008U4E7RU>] →

## Reduction #5: Unroll the Last Warp

```
// and use later like this...
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32) warpReduce(sdata, tid);
```

This used to be:  
s>0

```
device void warpReduce(volatile int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```

IMPORTANT: For this to  
be correct, we must use  
the "volatile" keyword!

**Note: This saves useless work in *all* warps, not just the last one!**

Without unrolling, all warps execute every iteration of the for loop and if statement

# Performance for 4M element reduction

	Time (2 <sup>22</sup> ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
<b>Kernel 3:</b> sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
<b>Kernel 4:</b> first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
<b>Kernel 5:</b> unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x

# Complete Unrolling

- If we knew the number of iterations (or equivalently, of threads in a block) at compile time, we could completely unroll the reduction
  - Luckily, the block size on G80 is limited by the GPU to 512 threads
    - 1024 on Fermi GPUs and newer
  - Also, we are sticking to power-of-2 block sizes
- Basic idea: unroll everything for a fixed block size
  - But we need to be generic – how can we unroll for block sizes that we don't know at compile time?
- Use of templates can solve this issue...
  - CUDA supports C++ template parameters on device and host functions

# Unrolling with Templates

- Specify block size as a function template parameter
- The kernel is parameterized:

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata)
```

# Reduction #6: Completely Unrolled

This is the key part of the kernel

```
if (blockSize >= 512) {  
    if(tid < 256){ sdata[tid] += sdata[tid + 256]; } __syncthreads();}  
if (blockSize >= 256) {  
    if(tid < 128){ sdata[tid] += sdata[tid + 128]; } __syncthreads();}  
if (blockSize >= 128) {  
    if(tid < 64){ sdata[tid] += sdata[tid + 64]; } __syncthreads();}  
  
if (tid < 32) warpReduce<blockSize>(sdata, tid); // last warp only
```

This is a helper function (device only)

```
template <unsigned int blockSize>  
__device__ void warpReduce(volatile int* sdata, int tid) {  
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];  
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];  
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];  
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];  
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];  
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];  
}
```

- All code in RED will be evaluated at compile time. Results in a very efficient inner loop.
- For Fermi and newer CC, you'd have one more `if` statement that covers the case when `blockSize>=1024`
- You can call the `warpReduce` function only when you got to one warp. Reason: you don't have to synchronize at that point.

# Invoking Template Kernels

- Don't we still need block size at compile time?
  - There is a way out, use a switch statement for 10 possible block sizes:

```
switch (threads) {  
    case 512:  
        reduce6<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 256:  
        reduce6<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 128:  
        reduce6<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 64:  
        reduce6< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 32:  
        reduce6< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 16:  
        reduce6< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 8:  
        reduce6< 8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 4:  
        reduce6< 4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 2:  
        reduce6< 2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 1:  
        reduce6< 1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
}
```

**This is code on the host, calling the appropriate kernel**

# Performance for 4 Million element reduction

	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>	<b>15.01x</b>
<b>Kernel 6:</b> completely unrolled	<b>0.381 ms</b>	<b>43.996 GB/s</b>	<b>1.41x</b>	<b>21.16x</b>

# Parallel Reduction Complexity

- Assume that the number of elements in array is of the form  $N=2^D$
- **Log N** parallel stages, each stage  $S$  requires  $N/2^S$  independent ops
  - Stage Complexity is  $O(\log N)$
- For  $N=2^D$ , approach requires a total of  $\sum_{S \in [1..D]} 2^{D-S} = N-1$  operations
  - Work Complexity is  $O(N)$  – It is **work-efficient**
  - That is, it does not perform more operations than a sequential algorithm
- **Time complexity**, for  $P$  threads physically in parallel ( $P$  processors):  $O(N/P + \log N)$ 
  - Compare to  $O(N)$  for sequential reduction
  - In a thread block,  $N=P$ , so  **$O(\log N)$**

# What About Cost?

- *Cost* of a parallel algorithm is processors  $\times$  time complexity
  - Allocate threads instead of processors:  $O(N)$  threads
  - Time complexity is  $O(\log N)$ , so *cost* is  $O(N \log N)$  : **not cost efficient!**
- Brent's theorem suggests  $O(N/\log N)$  threads
  - Each thread does  $O(\log N)$  sequential work
  - Then all  $O(N/\log N)$  threads cooperate for  $O(\log N)$  stages
  - Cost =  $O((N/\log N) * \log N) = O(N) \rightarrow$  cost efficient
- Sometimes called *algorithm cascading*
  - Can lead to significant speedups in practice

# Algorithm Cascading

- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory
- Brent's theorem says each thread should sum  $O(\log N)$  elements
  - i.e. 1024 or 2048 elements per block vs. 256
- Probably beneficial to push it even further
  - Possibly better latency hiding with more work per thread
  - More threads per block reduces levels in tree of recursive kernel invocations
  - High kernel launch overhead in last levels with few blocks
- On G80, best performance with 64-256 blocks of 128 threads
  - 1024-4096 elements per *thread*

# Kernel 7, Comments

- For the first six kernels a large number of blocks was used to “tile” the array
- Kernel 7: reduce the number of blocks and have a thread do more work than just fetch something to shared memory
- Example [cooked up, not related to actual CUDA warp size, typical CUDA block dim, etc.]:
  - Say you have 1024 elements stored in an array; you need to reduce that array
  - You start with 32 blocks, each with 4 threads
  - Then, 128 threads total. It means that a thread, say in block 11, would have to add two numbers, then two numbers, then two numbers, then two more numbers.
  - At this point, everything is in the union of the shared memory associated with the 32 blocks. At this point proceed like before with kernel 6.

# Reduction #7: Multiple Adds / Thread

**Replace load and add of two elements:**

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

**With a while loop to add as many as necessary:**

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

Note: gridSize loop stride to  
maintain coalescing!

# Performance for 4 Million element reduction

	Time (2 <sup>22</sup> ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>	<b>15.01x</b>
<b>Kernel 6:</b> completely unrolled	<b>0.381 ms</b>	<b>43.996 GB/s</b>	<b>1.41x</b>	<b>21.16x</b>
<b>Kernel 7:</b> multiple elements per thread	<b>0.268 ms</b>	<b>62.671 GB/s</b>	<b>1.42x</b>	<b>30.04x</b>

**Kernel 7 on 32M elements: 73 GB/s!**

# Final Kernel...

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}

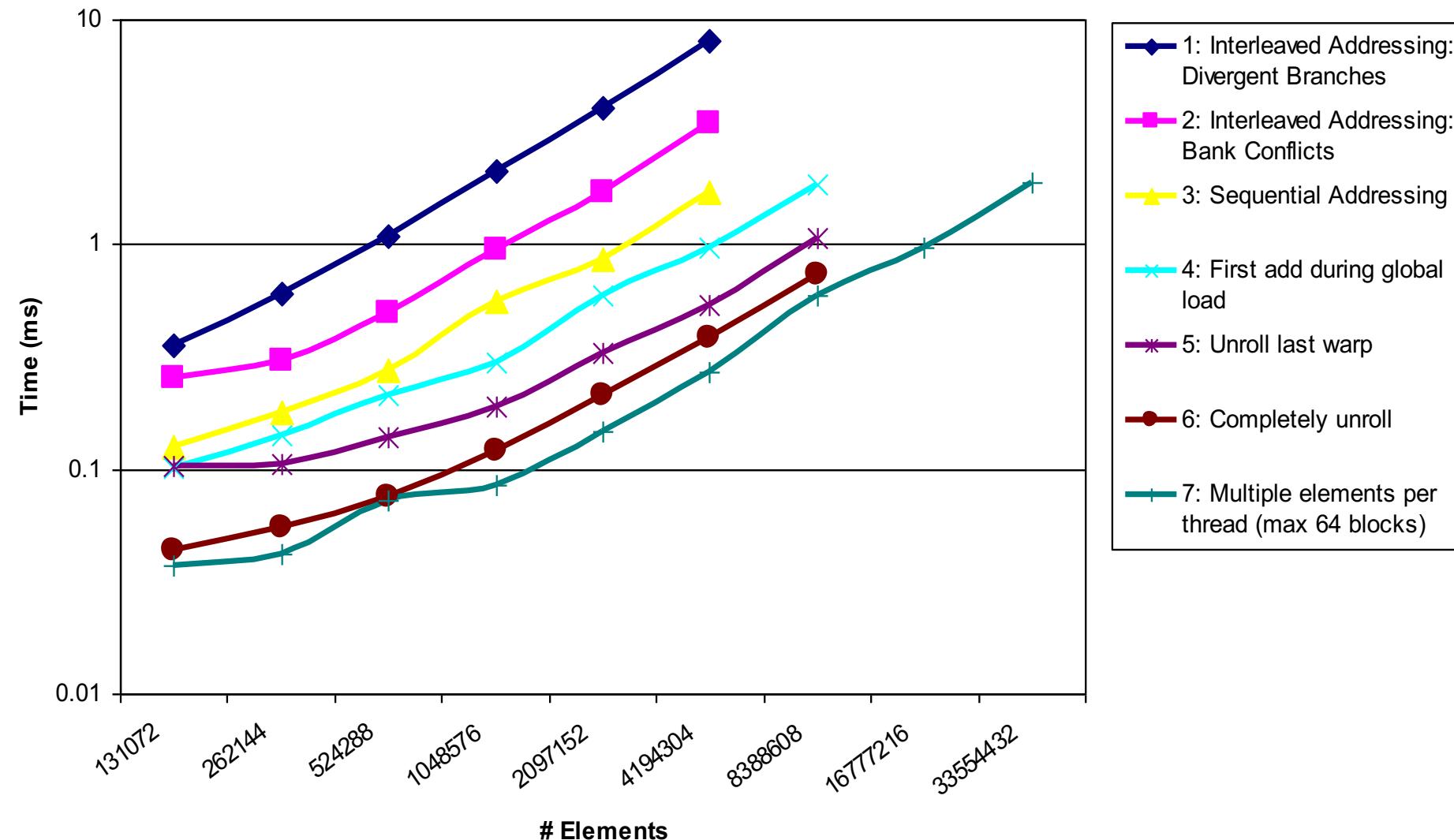
template <unsigned int blockSize>
__global__ void reduce7(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

# Performance Comparison



# Sources of Efficiency Improvement

- Algorithmic optimizations
  - Changes to addressing, algorithm cascading
  - 11.84x speedup, combined
- Code optimizations
  - Loop unrolling
  - 2.54x speedup, combined

# Lessons Learned, Vector Reduction

- Understand CUDA performance characteristics
  - Memory coalescing
  - Warp divergence
  - Bank conflicts
  - Latency hiding
- Use *peak performance metrics* to guide optimization
- Know how to identify type of bottleneck
  - E.g. memory, core computation, or instruction overhead
- Optimize your algorithm, *then* unroll loops
- Use template parameters to generate optimal code
- Understand parallel algorithm complexity theory

# CUDA Case Study: Parallel Prefix Scan on the GPU

# Software Design Exercise: Parallel Prefix Scan

- Implementation of prefix sum
  - Serial implementation – assigned as HW early in the semester
  - Parallel implementation: topic of next assignment
- Goal 1: Getting additional exposure to CUDA programming and thinking that goes behind it
- Goal 2: Understand that
  - Different algorithmic designs lead to different performance levels
  - Hardware constraints come into play in your applications and/or design solutions
- Goal 3: Identify design patterns that can result in superior parallel performance
  - Understand that there are patterns and it's worth being aware of them
    - To a large extend, patterns are shaped up by the underlying hardware

# Parallel Prefix Sum (Scan)

- Definition:

The all-prefix-sums operation takes a binary associative operator  $\oplus$  with identity  $I$ , and an array of  $n$  elements

$$[a_0, a_1, \dots, a_{n-1}]$$

and returns the ordered set

$$[I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})].$$

- Example:

If  $\oplus$  is addition, then scan on the set

$$[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$$

returns the set

$$[0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22]$$

Exclusive scan: last input element is not included in the result

# Scan on the CPU

```
void scan( float* scanned, float* input, int length)
{
    scanned[0] = 0;
    for(int i = 1; i < length; ++i)
    {
        scanned[i] = scanned[i-1] + input[i-1];
    }
}
```

- Just add each element to the sum of the elements before it
- Trivial, but sequential
  - Tempted to say that algorithms don't come more sequential than this...
- Requires exactly  $n-1$  adds

# Applications of Scan

- Scan is a simple and useful parallel building block

- Convert recurrences from sequential ...

```
out[0] = f(0)
for(j=1; j<n; j++)
    out[j] = out[j-1] + f(j);
```

- ... into parallel:

```
forall(j) in parallel
    temp[j] = f(j);
scan(out, temp);
```

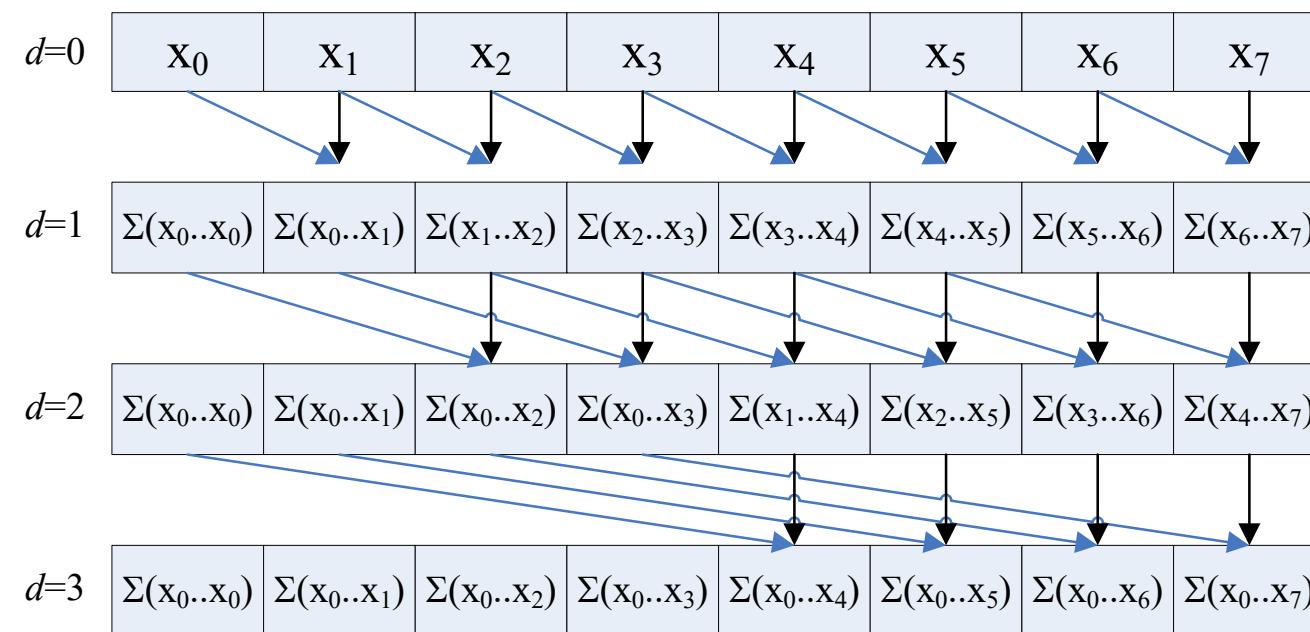
- Useful in implementation of several parallel algorithms:

- Radix sort
- Quicksort
- String comparison
- Lexical analysis
- Stream compaction

- Polynomial evaluation
- Solving recurrences
- Tree operations
- Histograms
- Etc.

# Parallel Scan Algorithm Solution #1: Hillis & Steele (1986)

- Implementation of the algorithm shown requires two buffers of length  $n$  (shown is the case  $n=8=2^3$ )
- Assumption: the number  $n$  of elements is a power of 2:  $n=2^M$



# The Plain English Perspective

- First iteration, I go with stride  $1=2^0$ 
  - Start at  $x[2^M]$  and apply this stride to all the array elements before  $x[2^M]$  to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have  $2^M - 2^0$  additions
- Second iteration, I go with stride  $2=2^1$ 
  - Start at  $x[2^M]$  and apply this stride to all the array elements before  $x[2^M]$  to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have  $2^M - 2^1$  additions
- Third iteration: I go with stride  $4=2^2$ 
  - Start at  $x[2^M]$  and apply this stride to all the array elements before  $x[2^M]$  to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have  $2^M - 2^2$  additions
- ... (and so on)

# The Plain English Perspective

- Consider the  $k^{\text{th}}$  iteration (where  $1 < k < M-1$ ): I go with stride  $2^{k-1}$ 
  - Start at  $x[2^M]$  and apply this stride to all the array elements before  $x[2^M]$  to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have  $2^M - 2^{k-1}$  additions
- ...
- $M^{\text{th}}$  iteration: I go with stride  $2^{M-1}$ 
  - Start at  $x[2^M]$  and apply this stride to all the array elements before  $x[2^M]$  to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have  $2^M - 2^{M-1}$  additions
- NOTE: There is no  $(M+1)^{\text{th}}$  iteration since this would automatically put me beyond the bounds of the array (if you apply an offset of  $2^M$  to " $\&x[2^M]$ " it places you right before the beginning of the array – not good...)

# Hillis & Steele Parallel Scan Algorithm

- Algorithm looks like this:

```
for d := 0 to M-1 do
    forall k do in parallel
        if k - 2d ≥ 0 then
            x[out][k] := x[in][k] + x[in][k - 2d]
        else
            x[out][k] := x[in][k]
    endforall
    swap(in,out)
endfor
```

Double-buffered version of the sum scan

# Hillis & Steele, Operation Count

- The number of operations tally:

$$(2^M - 2^0) + (2^M - 2^1) + \dots + (2^M - 2^{k-1}) + \dots + (2^M - 2^{M-1})$$

- Final operation count:

$$M \cdot 2^M - (2^0 + \dots + 2^{M-1}) = M \cdot 2^M - 2^M + 1 = n(\log(n) - 1) + 1$$

- This is an algorithm with  $O(n * \log(n))$  work

- Concluding remarks: is this approach good or not?

- Sequential scan algorithm only needs  $n-1$  additions
- A factor of  $\log_2(n)$  might hurt: 20x more work for  $10^6$  elements!
  - Homework requires a scan of about 16 million elements
- One more drawback: you need two buffers...

# Hillis & Steele: Kernel Function

```
__global__ void scan(float *g_odata, float *g_idata, int n) {
    extern volatile __shared__ float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int pout = 0, pin = 1;
    // load input into shared memory.
    // **exclusive** scan: shift right by one element and set first output to 0
    temp[thid] = (thid == 0) ? 0: g_idata[thid-1];
    __syncthreads();

    for( int offset = 1; offset<n; offset *= 2 ) {
        pout = 1 - pout; // swap double buffer indices
        pin = 1 - pout;

        if (thid >= offset)
            temp[pout*n+thid] = temp[pin*n+thid] + temp[pin*n+thid - offset];
        else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads(); // I need this here before I start next iteration
    }

    g_odata[thid] = temp[pout*n+thid]; // write output
}
```

# Hillis & Steele: Kernel Function, Quick Remarks

- The kernel is very simple, which is good
- Note the pin/pout trick that was used to alternate the destination buffer
- $O(N \log_2 N)$  algorithm – significant overhead when  $N$  gets large
- The kernel only works when the entire array is processed by one block
  - One block in CUDA has at the most 1024 threads
  - In this setup we cannot handle yet 16 million entries, which is what your assignment will call for

# Parallel Scan Algorithm: Solution #2: Harris-Sengupta-Owen (2007)

- A common parallel algorithm pattern:

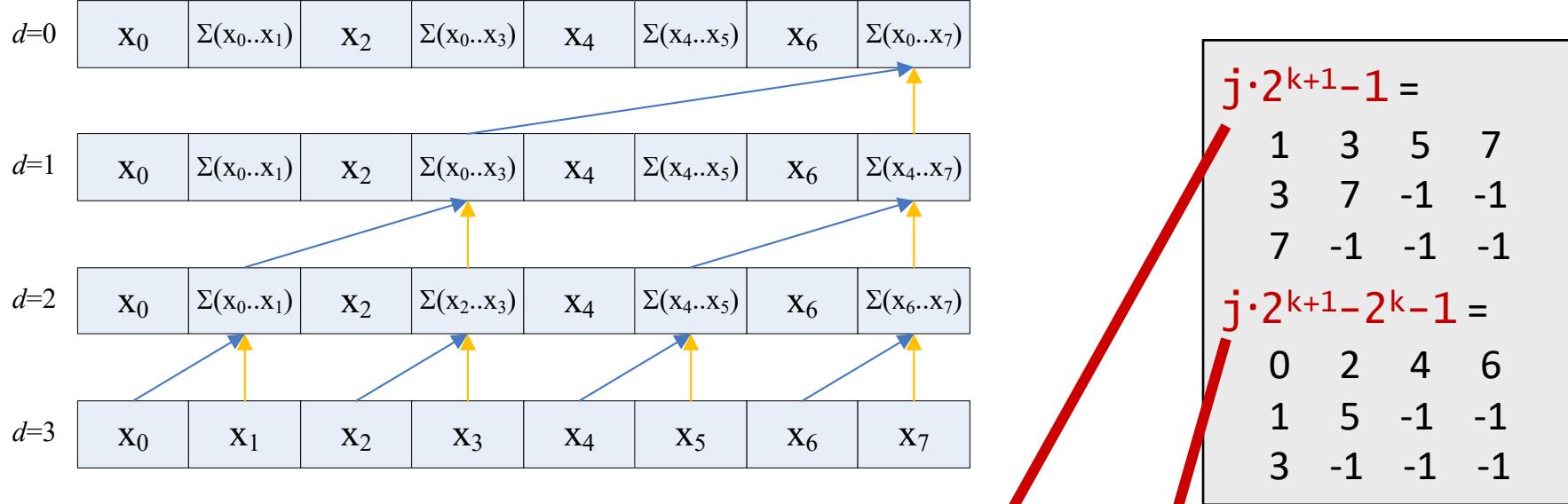
## Balanced Trees

- Build a balanced binary tree on the input data and sweep it to the root and then back into the leaves
- Tree is not an actual data structure, but a concept to determine what each thread does at each step

- For scan:

- Traverse from leaves to root building partial sums at internal nodes in the tree
  - Root holds sum of all leaves → nice, this is a reduction algorithm
- Traverse the tree back building the scan from the partial sums
  - Called down-sweep phase

# Picture and Pseudocode: The Reduction Step (“sweep to root”)



```

for k=0 to M-1
    offset =  $2^k$ 
    for j=1 to  $2^{M-k-1}$  do in parallel
         $x[j \cdot 2^{k+1}-1] = x[j \cdot 2^{k+1}-1] + x[j \cdot 2^{k+1}-2^k-1]$ 
    endfor
endfor

```

NOTE: “-1” entries indicate no-ops

# Operation Count, Reduce Phase

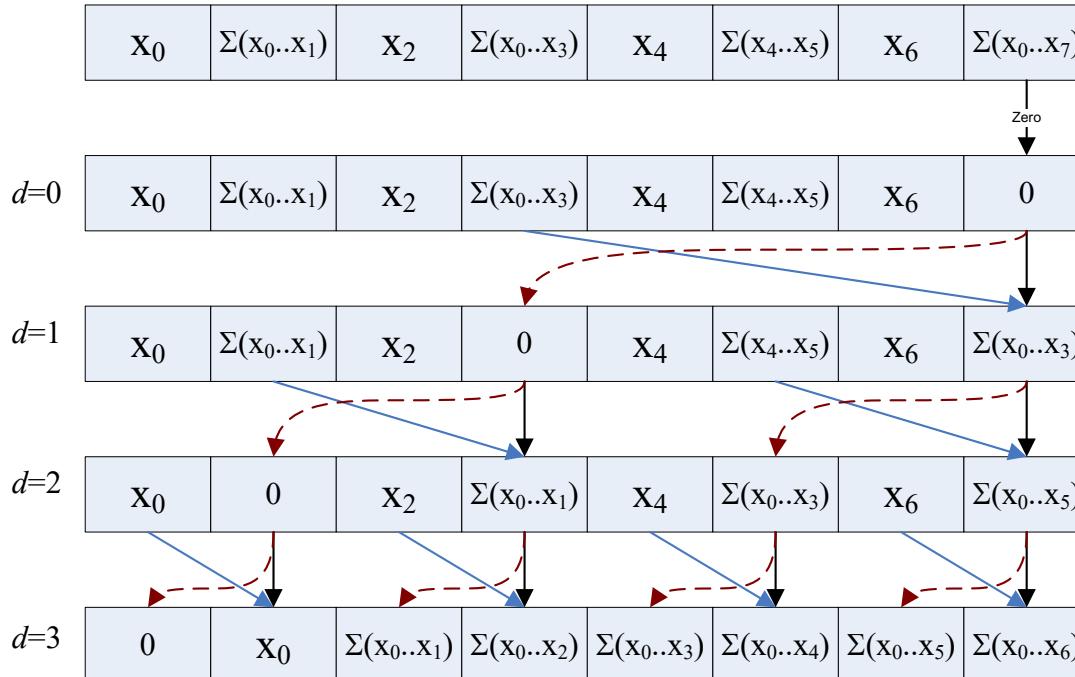
```
for k=0 to M-1
    offset = 2k
    for j=1 to 2M-k-1 do in parallel
        x[j · 2k+1-1] = x[j · 2k+1-1] + x[j · 2k+1-2k-1]
    endfor
endfor
```

By inspection:

$$\sum_{k=0}^{M-1} 2^{M-k-1} = 2^M - 1 = n - 1$$

Looks promising in terms of operation count...

# The “root to leaves” sweep



NOTE: This is just a mirror image of the reduction stage. Easy to come up with the indexing scheme...

```

for k=M-1 to 0
    offset = 2k
    for j=1 to 2M-k-1 do in parallel
        dummy = x[j · 2k+1-2k-1] ← Set aside
        x[j · 2k+1-2k-1] = x[j · 2k+1-1] ← Overwrite op.
        x[j · 2k+1-1] = x[j · 2k+1-1] + dummy ← Use
    endfor
endfor

```

Set aside  
Overwrite op.  
Use

# Down-Sweep Phase, Remarks

- Number of operations for the down-sweep phase:
  - Additions:  $n-1$
  - Overwrite ops. :  $n-1$  (each overwrite shadows an addition)
- Total number of operations associated with this algorithm
  - Additions:  $2n-2$
  - Overwrite ops:  $n-1$
  - Looks comparable with the work load in the sequential solution
- Convolved algorithm, indexing tricky to figure out
  - Kernel shown on next slide

Argument list:

g\_odata – pointer to output data, stored in device global memory

g\_idata – pointer to input data, stored in device global memory

n – size of array on which prefix scan performed (assumed a power of 2)

```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern volatile __shared__ float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int offset = 1;

    temp[2*thid] = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();

        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            temp[bi] += temp[ai];
        }
        offset <= 1; //multiply by 2 implemented as bitwise operation
    }

    if (thid == 0) { temp[n - 1] = 0; } // clear the last element

    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();

        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
    }

    __syncthreads();

    g_odata[2*thid] = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

# Going Beyond 2048 Entries

[1/3]

- Upon first invocation of the kernel (kernel #1), each will bring into shared memory 2048 elements:
  - 1024 “lead” elements (see vertical arrows  on slide 6), and...
  - 1024 mating elements (the blue, oblique arrows  on slide 6)
  - Two consecutive “lead” elements are separated by a stride of  $k=2^1$
  - A “lead” element and its “mating” element are separated by a stride of  $k/2=1$
- Suppose you take 6 reduction steps in this first kernel and bail out after writing into the global memory the preliminary data that you computed and stored in shared memory
- The next kernel invocation should pick up the unfinished business where the previous kernel left...
  - Call this a “flawless reentry requirement”

# Going Beyond 2048 Entries

[2/3]

- Upon the second kernel call, each block will bring into shared memory 2048 elements:
  - 1024 “lead” elements, and...
  - 1024 “mating” elements
  - Two consecutive “lead” elements that you bring in will be separated in global memory by a stride of  $k=2^6$
  - A “lead” element and its “mating” element are separated by a stride of  $k/2=2^5$ 
    - Thus, when bringing in data from global memory, you are not going to bring over a contiguous chunk of memory of size 2048, rather you'll have to jump  $2^5$  locations between successive “lead and mating element” pairs
  - However, once you bring data in shared memory, you process as before
  - Before you exit kernel #2 you have to write back data from shared memory into global memory
    - Again, you have to choreograph this shared to global memory store since there is a  $2^5$  stride that comes into play
  - If you exit kernel #2 after say 4 more reduction steps, the next time you re-enter the kernel (#3) you will have  $k=2^{10}$

# Going Beyond 2048 Entries

[3/3]

- You will continue the reduction stage until the stride is  $2^{M-1}$ 
  - At this point you are ready to start the root-to-leaves sweep phase
  - “root-to-leaves” sweep phase carried out in a similar fashion: we will have to invoke the kernel several times
  - Always work in shared memory and copy back data to global memory before bailing out
- The challenges here are:
  - Understanding the indexing into the global memory to bring data into ShMem
  - How to loop across the data in shared memory
- Numerous shared memory bank conflicts since strides are powers of 2
  - Shared memory bank conflicts: discussed next
  - Advanced topic: get rid of the bank conflict through padding

# Concluding Remarks, Parallel Scan

- Intuitively, the scan operation is not the type of procedure ideally suited for parallel computing
  - Even if it doesn't fit like a glove, leads to good speedup:

# elements	CPU Scan (ms)	GPU Scan (ms)	Speedup
1024	0.002231	0.079492	0.03
32768	0.072663	0.106159	0.68
65536	0.146326	0.137006	1.07
131072	0.726429	0.200257	3.63
262144	1.454742	0.326900	4.45
524288	2.911067	0.624104	4.66
1048576	5.900097	1.118091	5.28
2097152	11.848376	2.099666	5.64
4194304	23.835931	4.062923	5.87
8388688	47.390906	7.987311	5.93
16777216	94.794598	15.854781	5.98

Source: 2007 paper of Harris, Sengupta, Owens

# Concluding Remarks, Parallel Scan

- Hillis-Steele (HS) solution simple, but suboptimal
- Harris-Sengupta-Owen (HSO) solution convoluted, yet  $O(N)$  scaling
  - Algorithm is complex (particularly if implementation avoids bank conflicts)
- Problem not solved yet: we only looked at the case when our array has up to 2048 elements
  - How do we handle the  $16,777,216 = 2^{24}$  elements case?
  - Likewise, how would you implement the case when the number of elements is not a power of 2?

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 15

02/26/2020

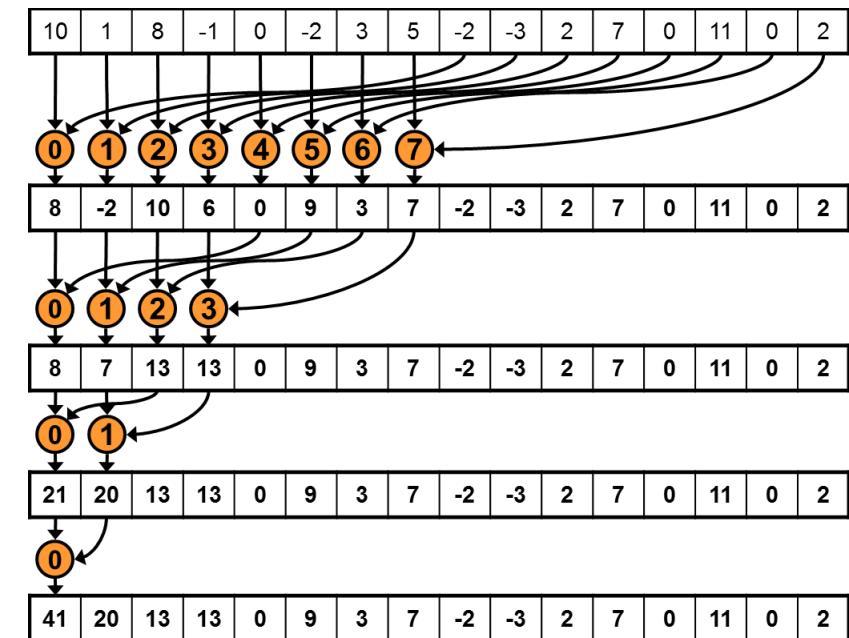
# Quote of the day

“Roads are just a suggestion Marge, just like pants.”

-- Homer Simpson, Safety Inspector [1989 - ]

# Before we get going...

- Last time:
  - Rules of thumb, GPU computing optimization
  - Test case: reduction operation
- Today:
  - Test case: scan operation
  - Quick overview, stencil op
  - CUDA streams (?)
- Other tidbits:
  - Assignment due on Thursday at 9 pm
    - In my opinion, this and next are the toughest assignments of the semester
  - Assigned readings – don't neglect them
  - Dan has office hours online each Wd@7 PM



# CUDA Case Study: Parallel Prefix Scan on the GPU

# Software Design Exercise: Parallel Prefix Scan

- Implementation of prefix sum
  - Serial implementation – assigned as HW early in the semester
  - Parallel implementation: topic of next assignment
- Goal 1: Getting additional exposure to CUDA programming and thinking that goes behind it
- Goal 2: Understand that
  - Different algorithmic designs lead to different performance levels
  - Hardware constraints come into play in your applications and/or design solutions
- Goal 3: Identify design patterns that can result in superior parallel performance
  - Understand that there are patterns and it's worth being aware of them
    - To a large extend, patterns are shaped up by the underlying hardware

# Parallel Prefix Sum (Scan)

- Definition:

The all-prefix-sums operation takes a binary associative operator  $\oplus$  with identity  $I$ , and an array of  $n$  elements

$$[a_0, a_1, \dots, a_{n-1}]$$

and returns the ordered set

$$[I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})].$$

- Example:

If  $\oplus$  is addition, then scan on the set

$$[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$$

returns the set

$$[0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22]$$

Exclusive scan: last input element is not included in the result

# Scan on the CPU

```
void scan( float* scanned, float* input, int length)
{
    scanned[0] = 0;
    for(int i = 1; i < length; ++i)
    {
        scanned[i] = scanned[i-1] + input[i-1];
    }
}
```

- Just add each element to the sum of the elements before it
- Trivial, but sequential
  - Tempting to say that algorithms don't come more sequential than this...
- Requires exactly  $n - 1$  adds

# Applications of Scan

- Scan is a simple and useful parallel building block

- Convert recurrences from sequential ...

```
out[0] = f(0)
for(j=1; j<n; j++)
    out[j] = out[j-1] + f(j);
```

- ... into parallel:

```
forall(j) in parallel
    temp[j] = f(j);
scan(out, temp);
```

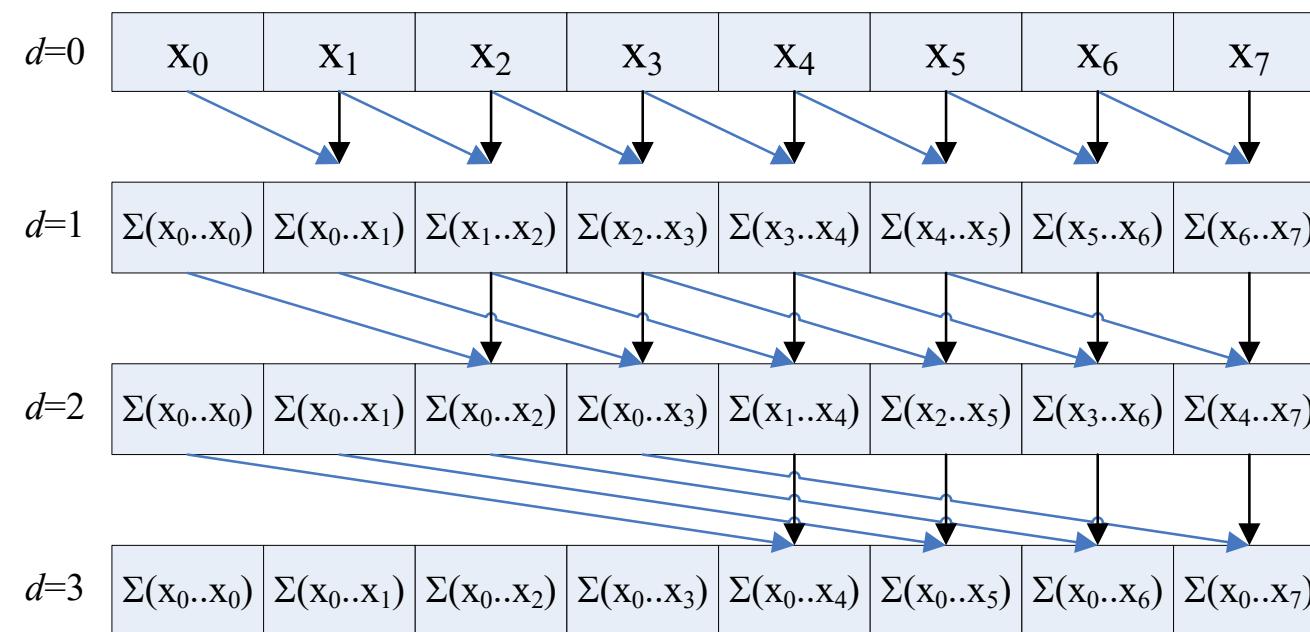
- Useful in implementation of several parallel algorithms:

- Radix sort
- Quicksort
- String comparison
- Lexical analysis
- Stream compaction

- Polynomial evaluation
- Solving recurrences
- Tree operations
- Histograms
- Etc.

# Parallel Scan Algorithm Solution #1: Hillis & Steele (1986)

- Implementation of the algorithm shown requires two buffers of length  $n$  (shown is the case  $n=8=2^3$ )
- Assumption: the number  $n$  of elements is a power of 2:  $n=2^M$



# The Plain English Perspective

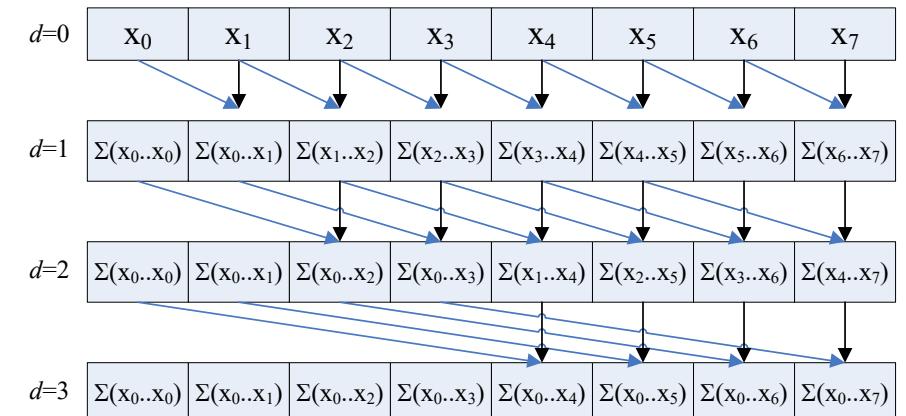
- First iteration, I go with stride  $1=2^0$ 
  - Start at  $x[2^M]$  and apply this stride to all the array elements before  $x[2^M]$  to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have  $2^M - 2^0$  additions
- Second iteration, I go with stride  $2=2^1$ 
  - Start at  $x[2^M]$  and apply this stride to all the array elements before  $x[2^M]$  to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have  $2^M - 2^1$  additions
- Third iteration: I go with stride  $4=2^2$ 
  - Start at  $x[2^M]$  and apply this stride to all the array elements before  $x[2^M]$  to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have  $2^M - 2^2$  additions
- ... (and so on)

# The Plain English Perspective

- Consider the  $k^{\text{th}}$  iteration (where  $1 < k < M-1$ ): I go with stride  $2^{k-1}$ 
  - Start at  $x[2^M]$  and apply this stride to all the array elements before  $x[2^M]$  to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have  $2^M - 2^{k-1}$  additions
- ...
- $M^{\text{th}}$  iteration: I go with stride  $2^{M-1}$ 
  - Start at  $x[2^M]$  and apply this stride to all the array elements before  $x[2^M]$  to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have  $2^M - 2^{M-1}$  additions
- NOTE: There is no  $(M+1)^{\text{th}}$  iteration since this would automatically put me beyond the bounds of the array (if you apply an offset of  $2^M$  to "& $x[2^M]$ " it places you right before the beginning of the array – not good...)

# Hillis & Steele Parallel Scan Algorithm

- Algorithm looks like this:



```
for d := 0 to M-1 do
    forall k do in parallel
        if k - 2d ≥ 0 then
            x[out][k] := x[in][k] + x[in][k - 2d]
        else
            x[out][k] := x[in][k]
    endforall
    swap(in,out)
endfor
```

Double-buffered version of the sum scan

# Hillis & Steele, Operation Count

- The number of operations tally:

$$(2^M - 2^0) + (2^M - 2^1) + \dots + (2^M - 2^{k-1}) + \dots + (2^M - 2^{M-1})$$

- Final operation count:

$$M \cdot 2^M - (2^0 + \dots + 2^{M-1}) = M \cdot 2^M - 2^M + 1 = n(\log(n) - 1) + 1$$

- This is an algorithm with  $O(n * \log(n))$  work

- Concluding remarks: is this approach good or not?

- Sequential scan algorithm only needs  $n-1$  additions
- A factor of  $\log_2(n)$  might hurt: 20x more work for  $10^6$  elements!
  - Homework requires a scan of about 16 million elements
- One more drawback: you need two buffers...

# Hillis & Steele: Kernel Function

```
__global__ void scan(float *g_odata, float *g_idata, int n) {
    extern volatile __shared__ float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int pout = 0, pin = 1;
    // load input into shared memory.
    // **exclusive** scan: shift right by one element and set first output to 0
    temp[thid] = (thid == 0) ? 0: g_idata[thid-1];
    __syncthreads();

    for( int offset = 1; offset<n; offset *= 2 ) {
        pout = 1 - pout; // swap double buffer indices
        pin = 1 - pout;

        if (thid >= offset)
            temp[pout*n+thid] = temp[pin*n+thid] + temp[pin*n+thid - offset];
        else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads(); // I need this here before I start next iteration
    }

    g_odata[thid] = temp[pout*n+thid]; // write output
}
```

# Hillis & Steele: Kernel Function, Quick Remarks

- The kernel is very simple, which is good
- Note the pin/pout trick that was used to alternate the destination buffer
- $O(N \log_2 N)$  algorithm – significant overhead when  $N$  gets large
- The kernel only works when the entire array is processed by one block
  - One block in CUDA has at the most 1024 threads
  - Assignment calls for a multi-block implementation of this, handling up to 1 billion entries (later in semester)

# Parallel Scan Algorithm: Solution #2: Harris-Sengupta-Owen (2007)

- A common parallel algorithm pattern:

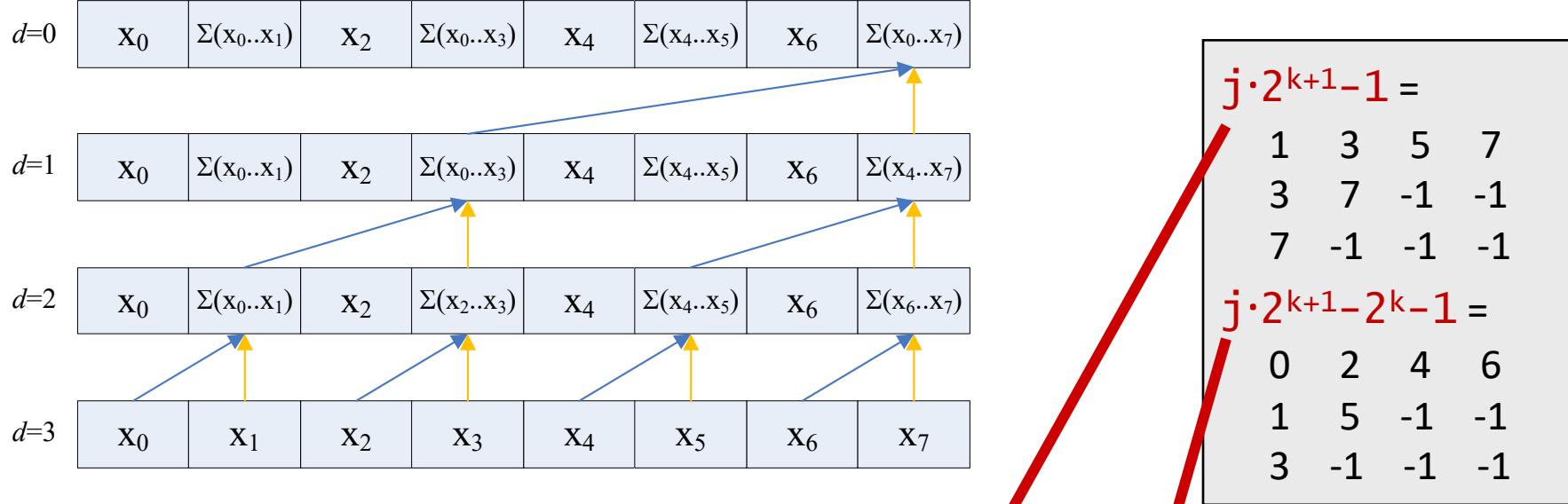
## Balanced Trees

- Build a balanced binary tree on the input data and sweep it to the root and then back into the leaves
- Tree is not an actual data structure, but a concept to determine what each thread does at each step

- For scan:

- Traverse **from leaves to root** building partial sums at internal nodes in the tree
  - Root holds sum of all leaves → nice, this is a reduction algorithm
- Traverse the tree back, **from root to leaves**, building the scan from the partial sums
  - Called down-sweep phase

# Picture and Pseudocode: The Reduction Step (“leaves to root”)



```

for k=0 to M-1
    offset = 2k
    for j=1 to 2M-k-1 do in parallel
        x[j · 2k+1-1] = x[j · 2k+1-1] + x[j · 2k+1-2k-1]
    endfor
endfor

```

NOTE: “-1” entries indicate no-ops

# Operation Count, Reduce Phase

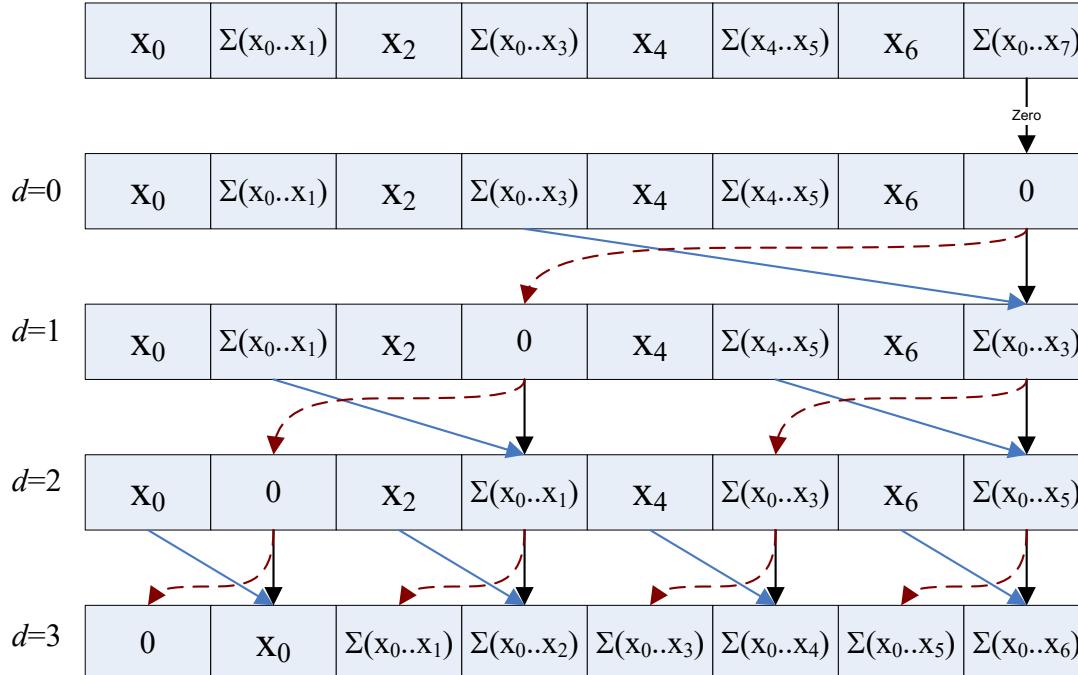
```
for k=0 to M-1
    offset = 2k
    for j=1 to 2M-k-1 do in parallel
        x[j · 2k+1-1] = x[j · 2k+1-1] + x[j · 2k+1-2k-1]
    endfor
endfor
```

By inspection:

$$\sum_{k=0}^{M-1} 2^{M-k-1} = 2^M - 1 = n - 1$$

Looks promising in terms of operation count...

# The “root-to-leaves” sweep



NOTE: This is just a mirror image of the reduction stage. Easy to come up with the indexing scheme...

```

for k=M-1 to 0
    offset = 2k
    for j=1 to 2M-k-1 do in parallel
        dummy = x[j · 2k+1-2k-1] ← Set aside
        x[j · 2k+1-2k-1] = x[j · 2k+1-1] ← Overwrite op.
        x[j · 2k+1-1] = x[j · 2k+1-1] + dummy ← Use
    endfor
endfor

```

Set aside  
Overwrite op.  
Use

# Root-to-leaves phase, remarks

- Number of operations for the down-sweep phase:
  - Additions:  $n-1$
  - Overwrite ops. :  $n-1$  (each overwrite shadows an addition)
- Total number of operations associated with this algorithm
  - Additions:  $2n-2$
  - Overwrite ops:  $n-1$
  - Looks comparable with the work load in the sequential solution
- Convolved algorithm, indexing tricky to figure out
  - Kernel shown on next slide

Argument list:

g\_odata – pointer to output data, stored in device global memory

g\_idata – pointer to input data, stored in device global memory

n – size of array on which prefix scan performed (assumed a power of 2)

```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern volatile __shared__ float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int offset = 1;

    temp[2*thid] = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();

        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            temp[bi] += temp[ai];
        }
        offset <= 1; //multiply by 2 implemented as bitwise operation
    }

    if (thid == 0) { temp[n - 1] = 0; } // clear the last element

    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();

        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
    }

    __syncthreads();

    g_odata[2*thid] = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

# Going beyond 2048 entries handled by one CUDA block

[1/3]

- Upon first invocation of the kernel (kernel #1), each block will bring into shared memory 2048 elements:
  - 1024 “lead” elements (see vertical arrows  on slide 6), and...
  - 1024 mating elements (the blue, oblique arrows  on slide 6)
  - Two consecutive “lead” elements are separated by a stride of  $k=2^1$
  - A “lead” element and its “mating” element are separated by a stride of  $k/2=1$
- Suppose you take 6 reduction steps in this first kernel and bail out after writing into the global memory the preliminary data that you computed and stored in shared memory
- The next kernel invocation should pick up the unfinished business where the previous kernel left...
  - Call this a “flawless reentry requirement”

# Going beyond 2048 entries handled by one CUDA block

[2/3]

- Upon the second kernel call, each block will bring into shared memory 2048 elements:
  - 1024 “lead” elements, and...
  - 1024 “mating” elements
  - Two consecutive “lead” elements that you bring in will be separated in global memory by a stride of  $k=2^6$
  - A “lead” element and its “mating” element are separated by a stride of  $k/2=2^5$ 
    - Thus, when bringing in data from global memory, you are not going to bring over a contiguous chunk of memory of size 2048, rather you'll have to jump  $2^5$  locations between successive “lead and mating element” pairs
  - However, once you bring data in shared memory, you process as before
  - Before you exit kernel #2 you have to write back data from shared memory into global memory
    - Again, you have to choreograph this shared to global memory store since there is a  $2^5$  stride that comes into play
  - If you exit kernel #2 after say 4 more reduction steps, the next time you re-enter the kernel (#3) you will have  $k=2^{10}$

# Going beyond 2048 entries handled by one CUDA block

[3/3]

- You will continue the reduction stage until the stride is  $2^{M-1}$ 
  - At this point you are ready to start the root-to-leaves sweep phase
  - “root-to-leaves” sweep phase carried out in a similar fashion: we will have to invoke the kernel several times
  - Always work in shared memory and copy back data to global memory before bailing out
- The challenges here are:
  - Understanding the indexing into the global memory to bring data into ShMem
  - How to loop across the data in shared memory
- Numerous shared memory bank conflicts since strides are powers of 2
  - Shared memory bank conflicts: discussed next
  - Advanced topic: get rid of the bank conflict through padding

# Concluding Remarks, Parallel Scan

- Intuitively, the scan operation is not the type of procedure ideally suited for parallel computing
  - Even if it doesn't fit like a glove, leads to good speedup:

# elements	CPU Scan (ms)	GPU Scan (ms)	Speedup
1024	0.002231	0.079492	0.03
32768	0.072663	0.106159	0.68
65536	0.146326	0.137006	1.07
131072	0.726429	0.200257	3.63
262144	1.454742	0.326900	4.45
524288	2.911067	0.624104	4.66
1048576	5.900097	1.118091	5.28
2097152	11.848376	2.099666	5.64
4194304	23.835931	4.062923	5.87
8388688	47.390906	7.987311	5.93
16777216	94.794598	15.854781	5.98

Source: 2007 paper of Harris, Sengupta, Owens

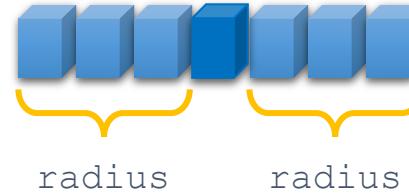
# Concluding Remarks, Parallel Scan

- Hillis-Steele (HS) solution simple, but suboptimal
- Harris-Sengupta-Owen (HSO) solution convoluted, yet  $O(N)$  scaling
  - Algorithm is complex (particularly if implementation avoids bank conflicts)
- Problem not solved yet: we only looked at the case when our array has up to 2048 elements
  - How do we handle the  $16,777,216 = 2^{24}$  elements case?
  - Likewise, how would you implement the case when the number of elements is not a power of 2?

## Case studies: Performing a 1D Stencil

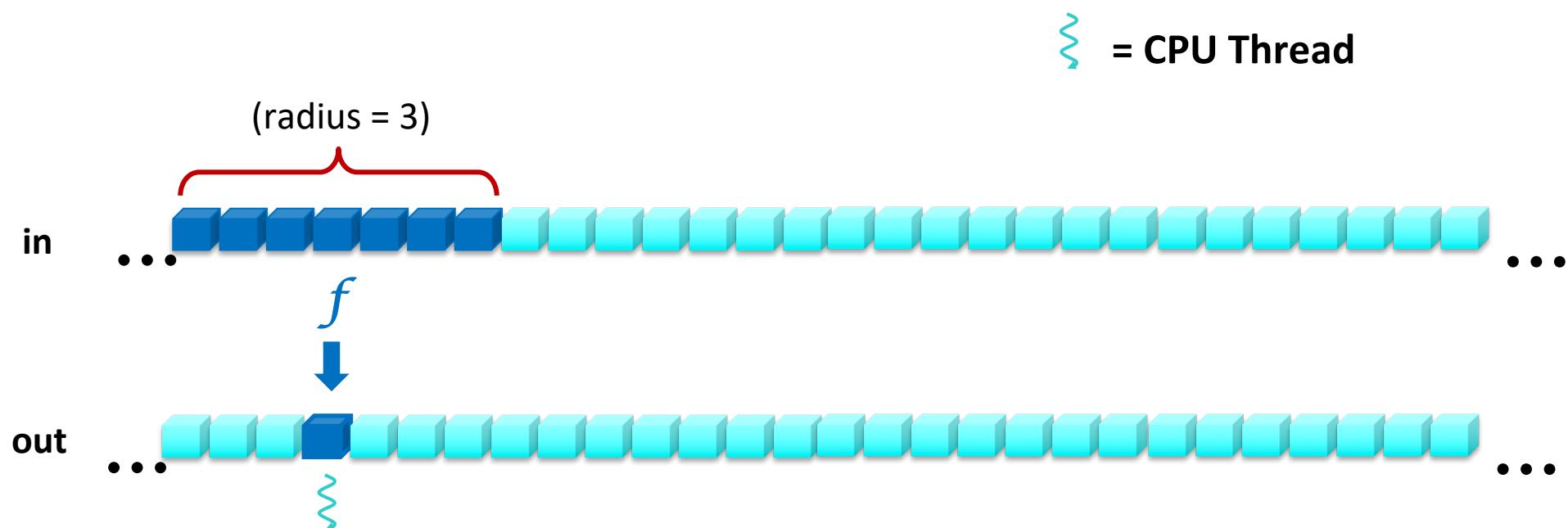
# Performing a 1D Stencil: problem setup

- Applying a 1D stencil to a 1D array of elements
  - Function of input elements within a radius

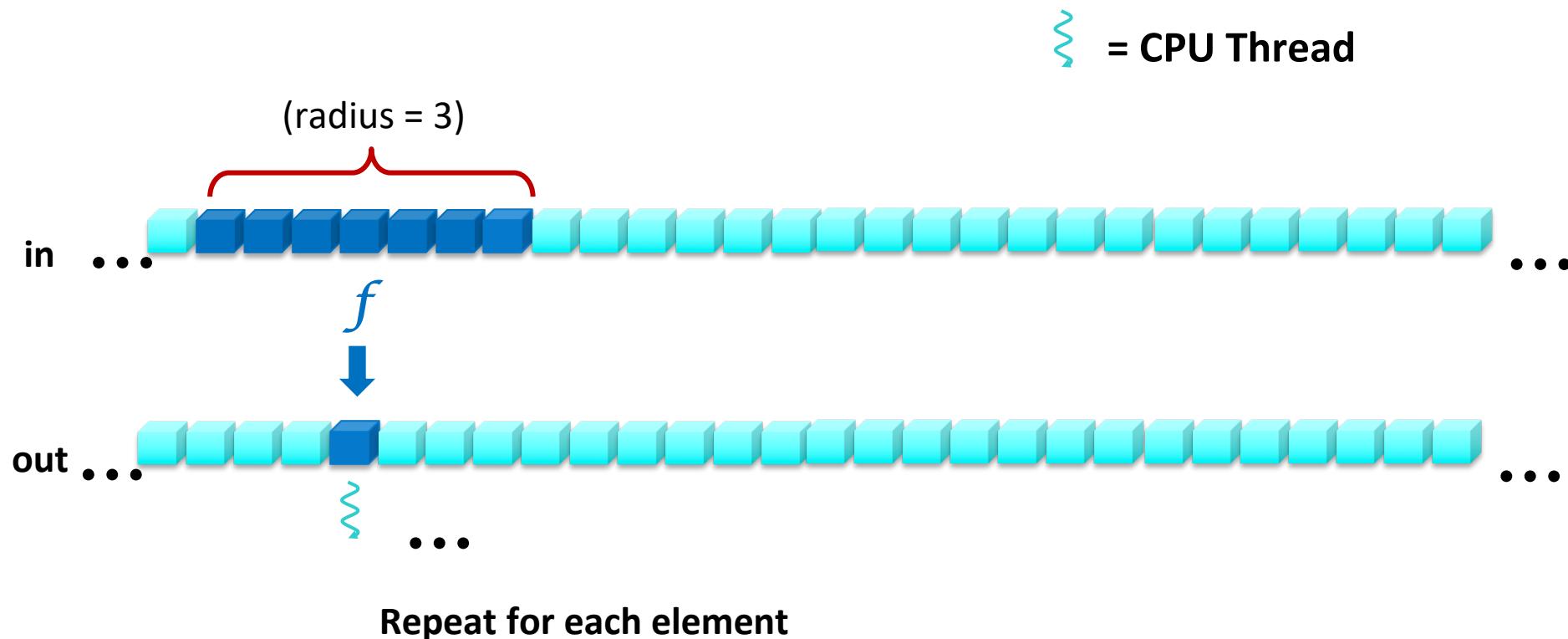


- Operation is fundamental to many algorithms
  - Standard discretization methods, interpolation, convolution, filtering, etc.
- This example will use weighted arithmetic mean

# Serial Algorithm



# Serial Algorithm



# Serial Implementation

[assume N=16 mil.]

```
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS, N-RADIUS, weights, in, out);

    //free resources
    free(weights); free(in); free(out);
}
```

```
void applyStencil1D(int sIdx, int eIdx, const
                     float *weights, float *in, float *out) {

    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

# Serial Implementation

```
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS,N-RADIUS,weights,in,out);

    //free resources
    free(weights); free(in); free(out);
}
```

```
void applyStencil1D(int sIdx, int eIdx, const
    float *weights, float *in, float *out) {

    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

# Serial Implementation

```
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS,N-RADIUS,weights,in,out);

    //free resources
    free(weights); free(in); free(out);
}
```

For each element...

```
void applyStencil1D(int sIdx, int eIdx, const
                    float *weights, float *in, float *out) {

    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

Weighted mean over radius

# Serial Implementation

```
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS,N-RADIUS,weights,in,out);

    //free resources
    free(weights); free(in); free(out);
}
```

```
void applyStencil1D(int sIdx, int eIdx, const
                     float *weights, float *in, float *out) {

    for (int i = sIdx; si < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

CPU	MElements/s
i7-930	30

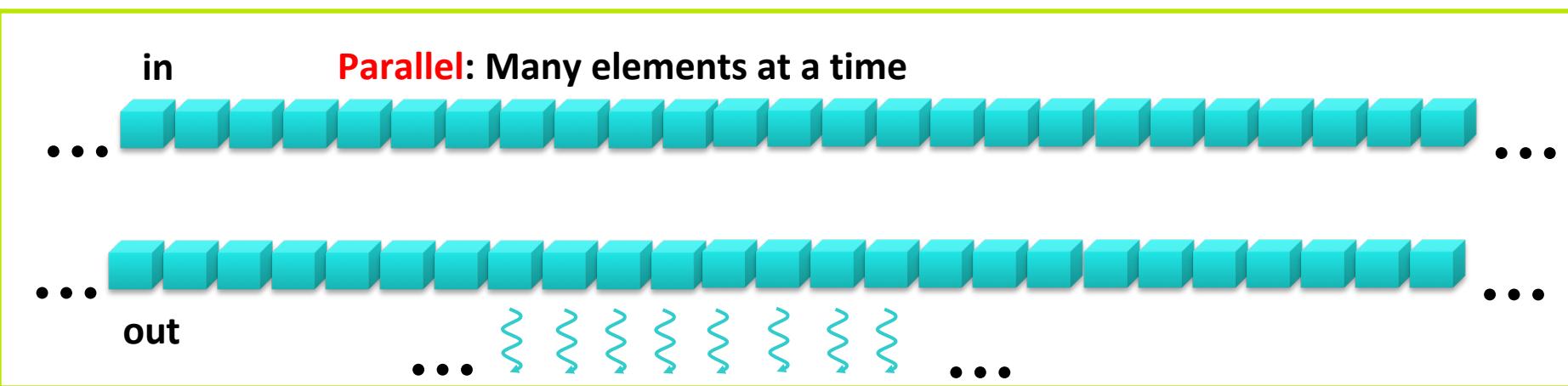
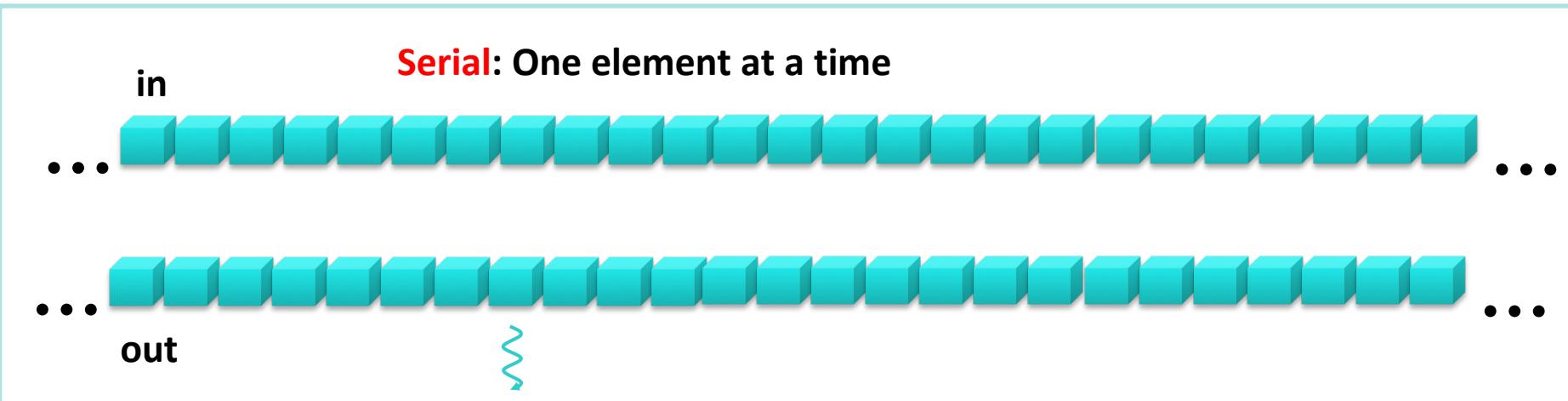
# Application Optimization Process



- Identify Optimization Opportunities
  - 1D stencil algorithm
- Parallelize with CUDA and confirm functional correctness
  - **cuda-gdb, cuda-memcheck**
  - Note: **cuda-memcheck** useful for memory debugging
    - Out of bounds accesses
    - Accessing misaligned data
    - Race conditions
    - Memory leaks
- Optimize
  - ...dealing with this next, using **nvvp**

# Parallel Algorithm

 = Thread



# The Parallel Implementation

The GPU kernel

```
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

```
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights;  cudaMalloc(&d_weights, wsize);
    float *d_in;       cudaMalloc(&d_in, size);
    float *d_out;      cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights,weights,wsize,cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

# The Parallel Implementation

```
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights;  cudaMalloc(&d_weights, wsize);
    float *d_in;       cudaMalloc(&d_in, size);
    float *d_out;      cudaMalloc(&d_out, size);

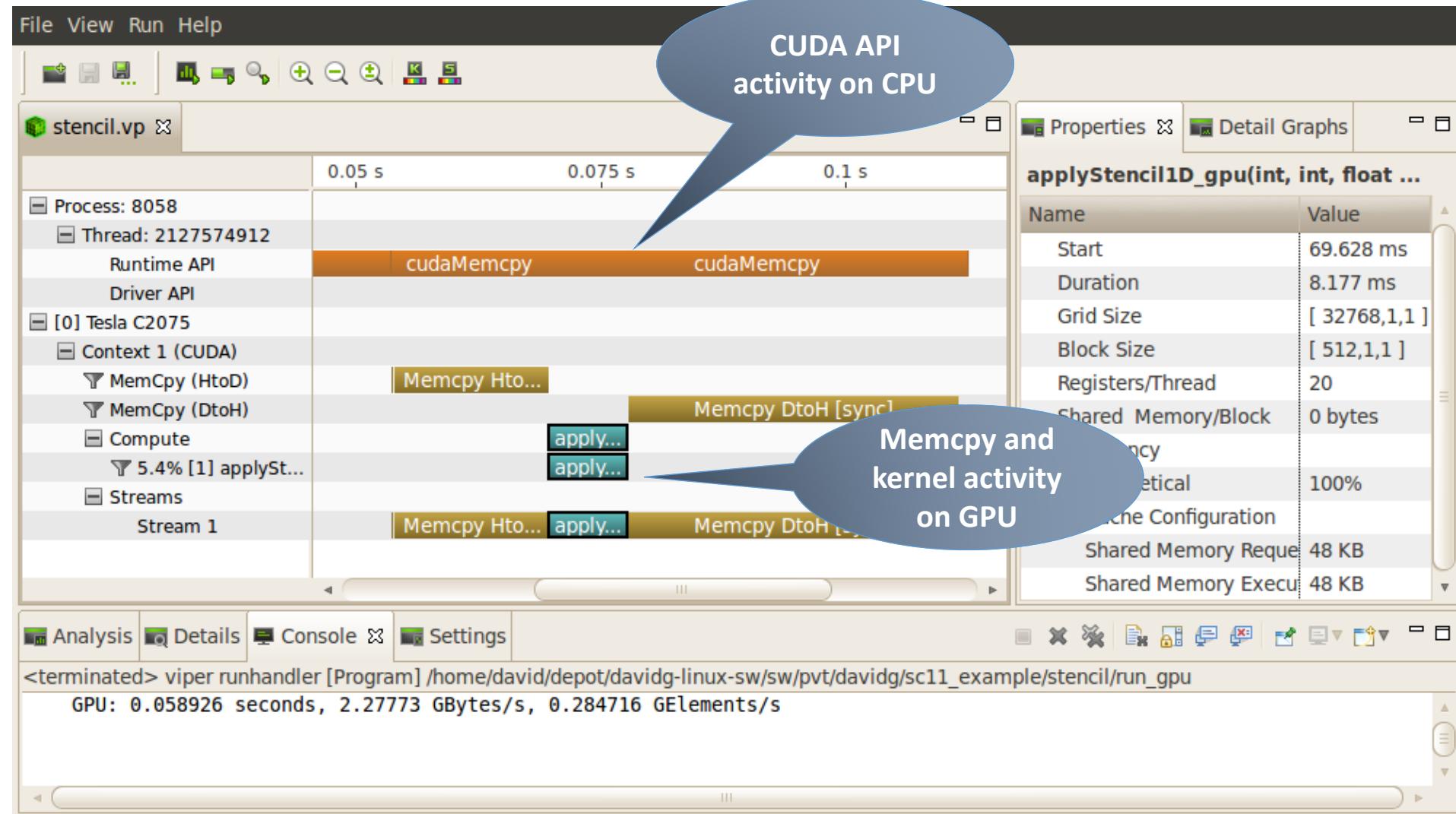
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
```

Device	Algorithm	MElements/s	Speedup
i7-930*	Optimized & Parallel	130	1x
Tesla C2075	Simple	285	2.2x

```
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

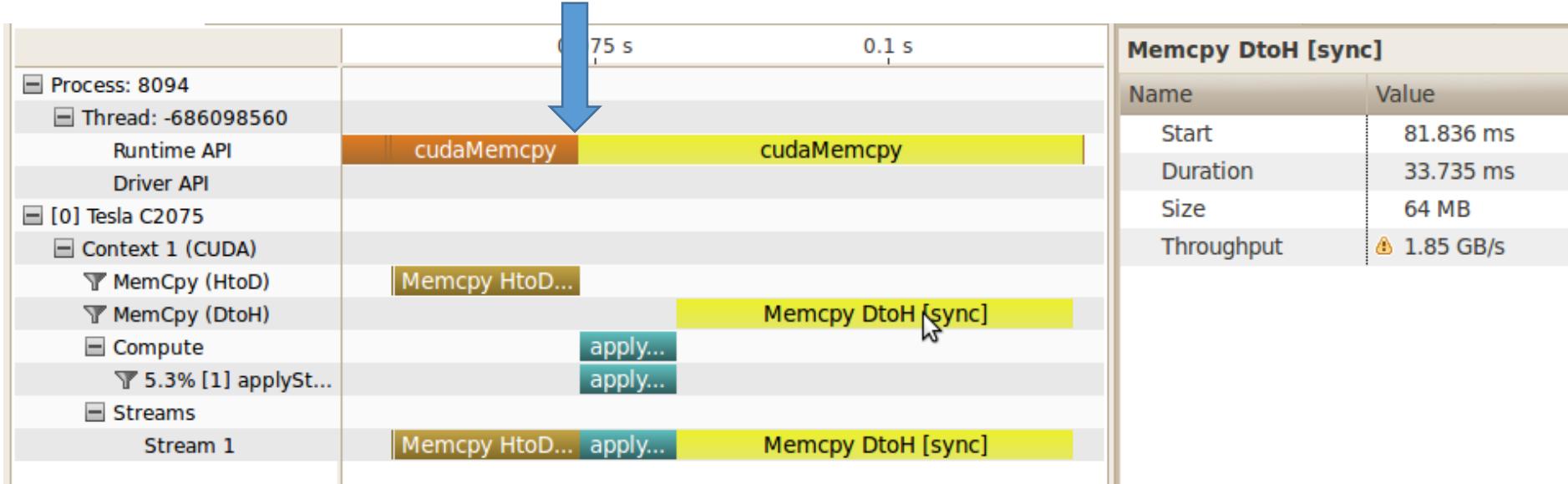
# NVIDIA Visual Profiler

The interesting part,  
done on the GPU

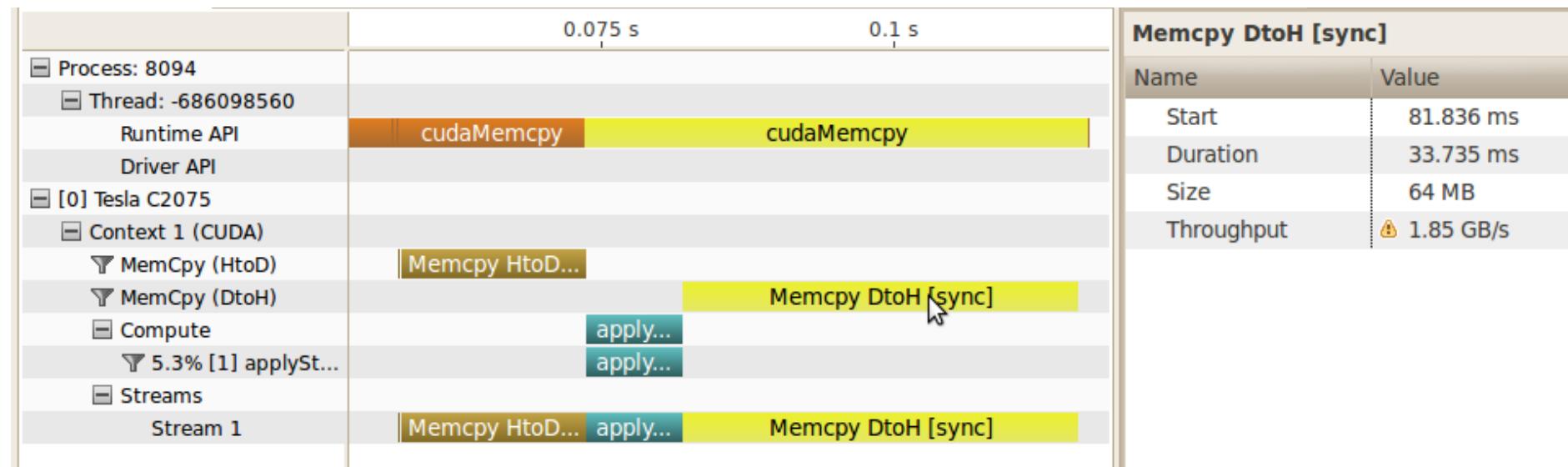


# Quiz: Why is CPU thread shown with two contiguous cudaMemcpy?

Why contiguous on the CPU side?



# Detecting Low Memory Throughput

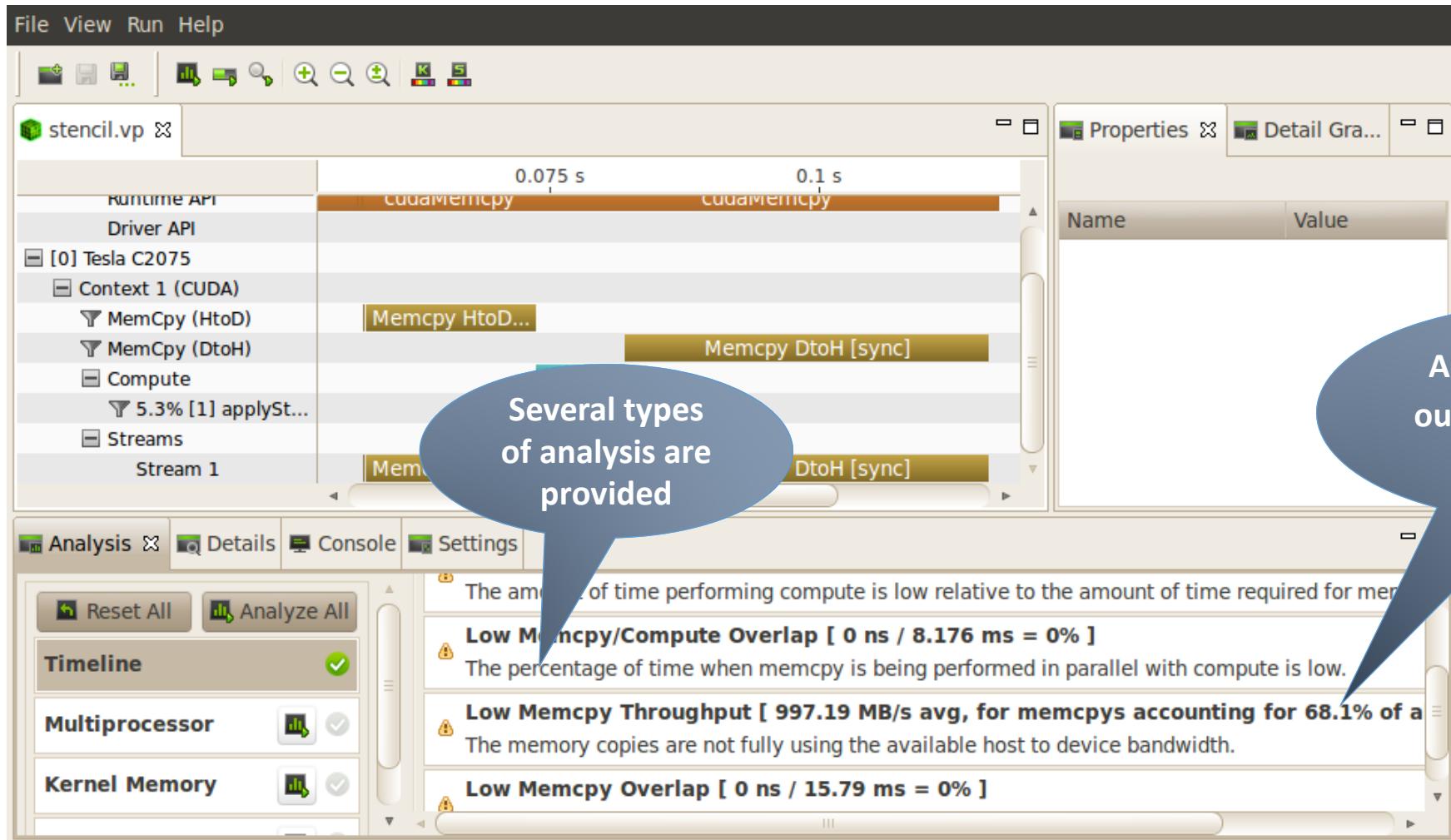


- Spent majority of time in data transfer
  - Often can be overlapped with preceding or following computation
- From timeline can see that throughput is low
  - PCIe x16 can sustain significantly more than this

# Visual Profiler Analysis

- How do we know when there is an optimization opportunity?
  - Timeline visualization seems to indicate an opportunity
  - Documentation gives guidance and strategies for tuning
    - CUDA Best Practices Guide
    - CUDA Programming Guide
- Visual Profiler analyzes your application
  - Uses timeline and other collected information
  - Highlights specific guidance from Best Practices
  - Like having a customized Best Practices Guide for your application

# Visual Profiler Analysis



# Online Optimization Help

**Low Memcpy Throughput [ 997.19 MB/s avg, for memcpys accounting for 68.1% of all memcpy time ]**

The memory copies are not fully using the available host to device bandwidth.

[More...](#)

Search:  Go Scope: All topics

Content: [Up](#) [Down](#) [Edit](#) [New](#) [Delete](#)

**Visual Profiler Optimization Guide** > [Memory Optimization Between Host and Device](#)

**Pinned Memory**

Page-locked or pinned memory transfers attain the highest bandwidth between host and device. On PCIe ×16 Gen2 cards, for example, pinned memory can achieve greater than 5 GBps transfer rates.

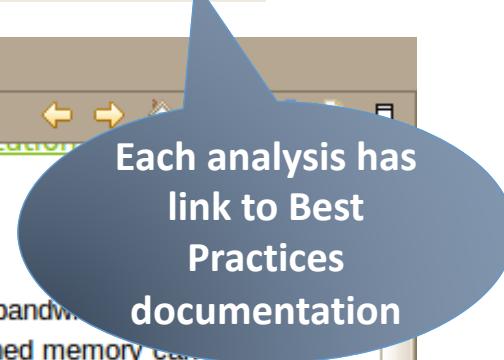
Pinned memory is allocated using the `cudaMallocHost()` or `cudaHostAlloc()` functions in the Runtime API. The `bandwidthTest.cu` program in the CUDA SDK shows how to use these functions as well as how to measure memory transfer performance.

Pinned memory should not be overused. Excessive use can reduce overall system performance because pinned memory is a scarce resource. How much is too much is difficult to tell in advance, so as with all optimizations, test the applications and the systems they run on for optimal performance parameters.

**Parent topic:** [Data Transfer Between Host and Device](#)

Copyright © 2011 NVIDIA Corporation | [www.nvidia.com](#)

NVIDIA



# Pinned CPU Memory Implementation

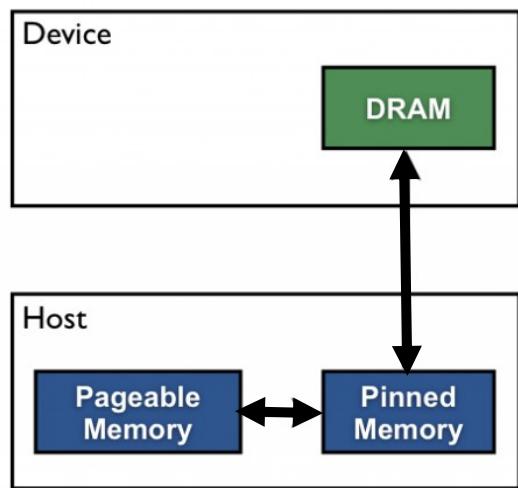
```
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights; cudaMallocHost(&weights, wsize);
    float *in;        cudaMallocHost(&in, size);
    float *out;       cudaMallocHost(&out, size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights);
    float *d_in;      cudaMalloc(&d_in);
    float *d_out;     cudaMalloc(&d_out);
    ...
}
```

CPU allocations use pinned memory to enable fast memcpy

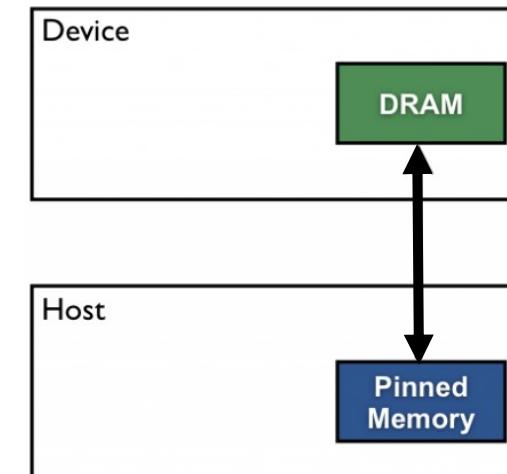
No other changes

# CUDA: Pageable vs. Pinned Data Transfer

***Pageable Data Transfer***



***Pinned Data Transfer***



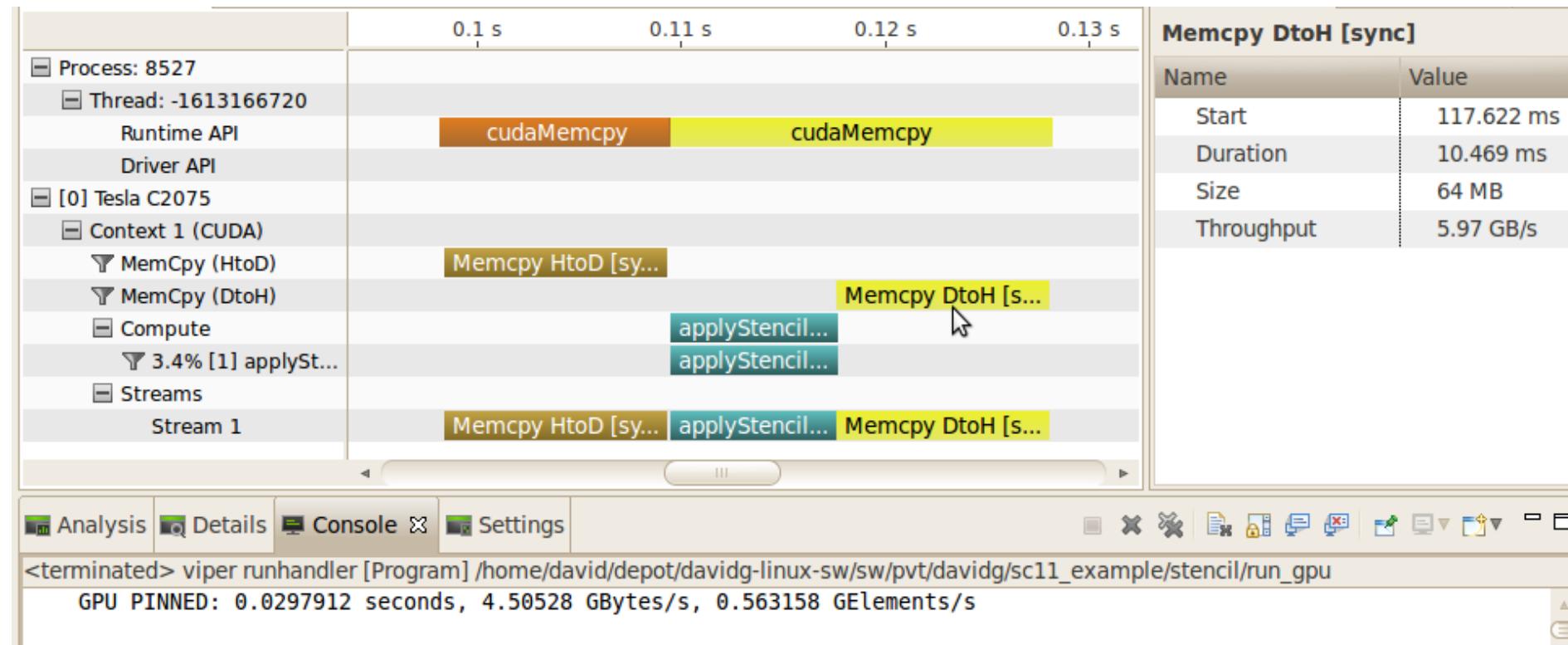
# Pinned Memory

- DMA (direct memory access) uses physical addresses
- The OS could accidentally page out the data that is being read or written by a DMA and page in another virtual page into the same location
- Pinned memory cannot not be paged out
- If a source or destination of a `cudaMemcpy()` in the host memory is not pinned, it needs to be first copied to a pinned memory – extra overhead
- `cudaMemcpy` faster with pinned host memory source or destination

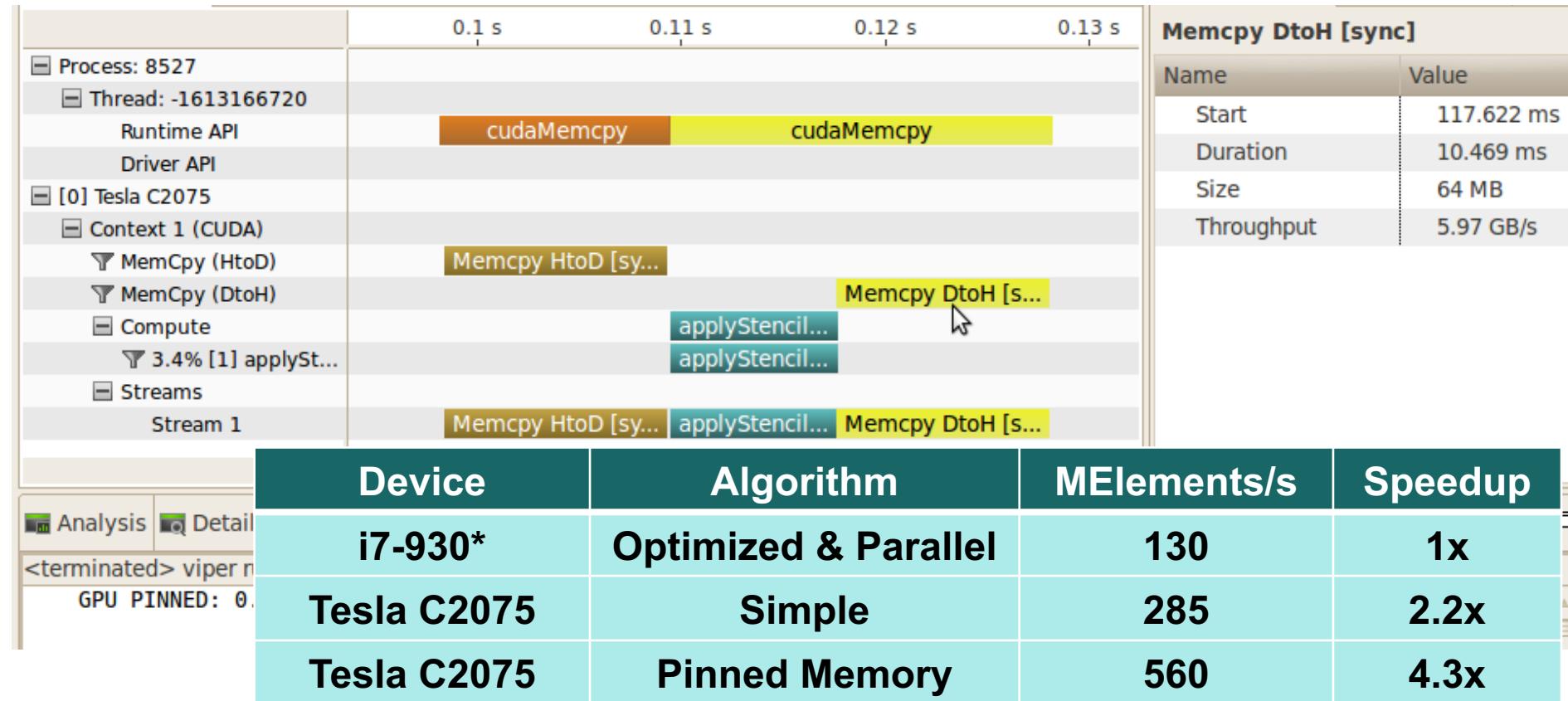
# Allocate/Free Pinned Memory (a.k.a. Page Locked Memory)

- `cudaHostAlloc()`
  - Three parameters
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use `cudaHostAllocDefault` for now
- `cudaFreeHost()`
  - One parameter
  - Pointer to the memory to be freed

# Pinned CPU Memory Result



# Pinned CPU Memory Result



# Application Optimization Process

[Revisited]



- Identify Optimization Opportunities
  - 1D stencil algorithm
- Parallelize with CUDA, confirm functional correctness
  - Debugger
  - Memory Checker
- Optimize
  - Profiler (pinned memory)

# Application Optimization Process

[Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm
- Parallelize with CUDA, confirm functional correctness
  - Debugger
  - Memory Checker
- Optimize
  - Profiler (pinned memory)

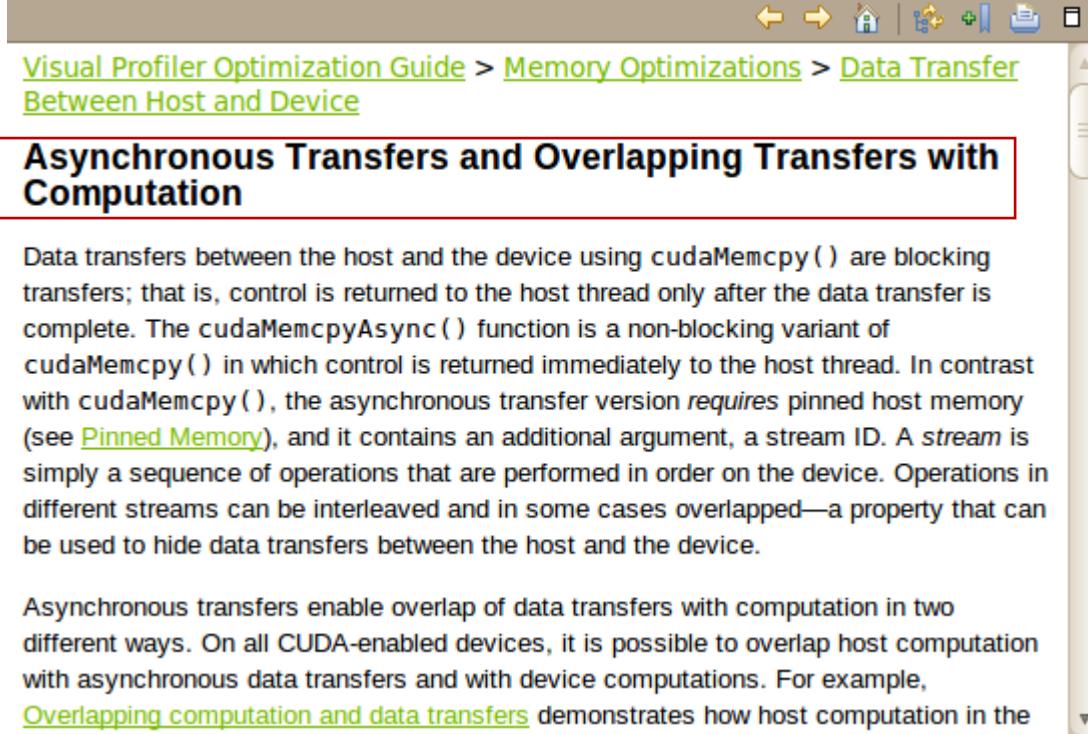


- Advanced optimization
  - Larger time investment
  - Potential for larger speedup

⚠ **Low Memcpy/Compute Overlap [ 0 ns / 8.176 ms = 0 % ]**

The percentage of time when memcpy is being performed in parallel with compute is low.

[More...](#)



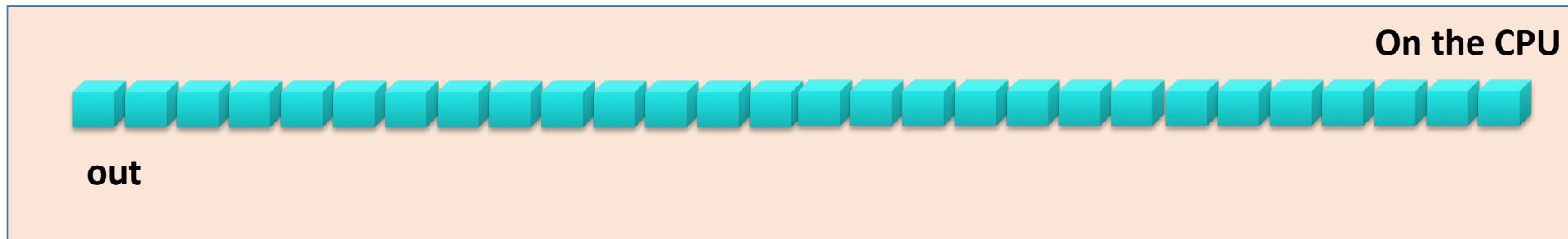
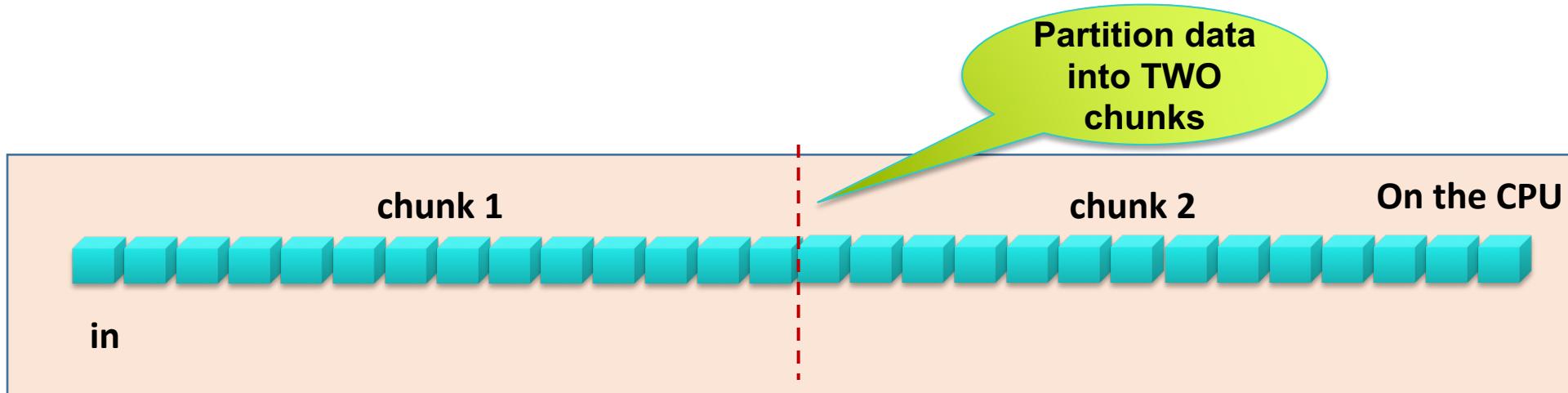
Visual Profiler Optimization Guide > Memory Optimizations > Data Transfer Between Host and Device

## Asynchronous Transfers and Overlapping Transfers with Computation

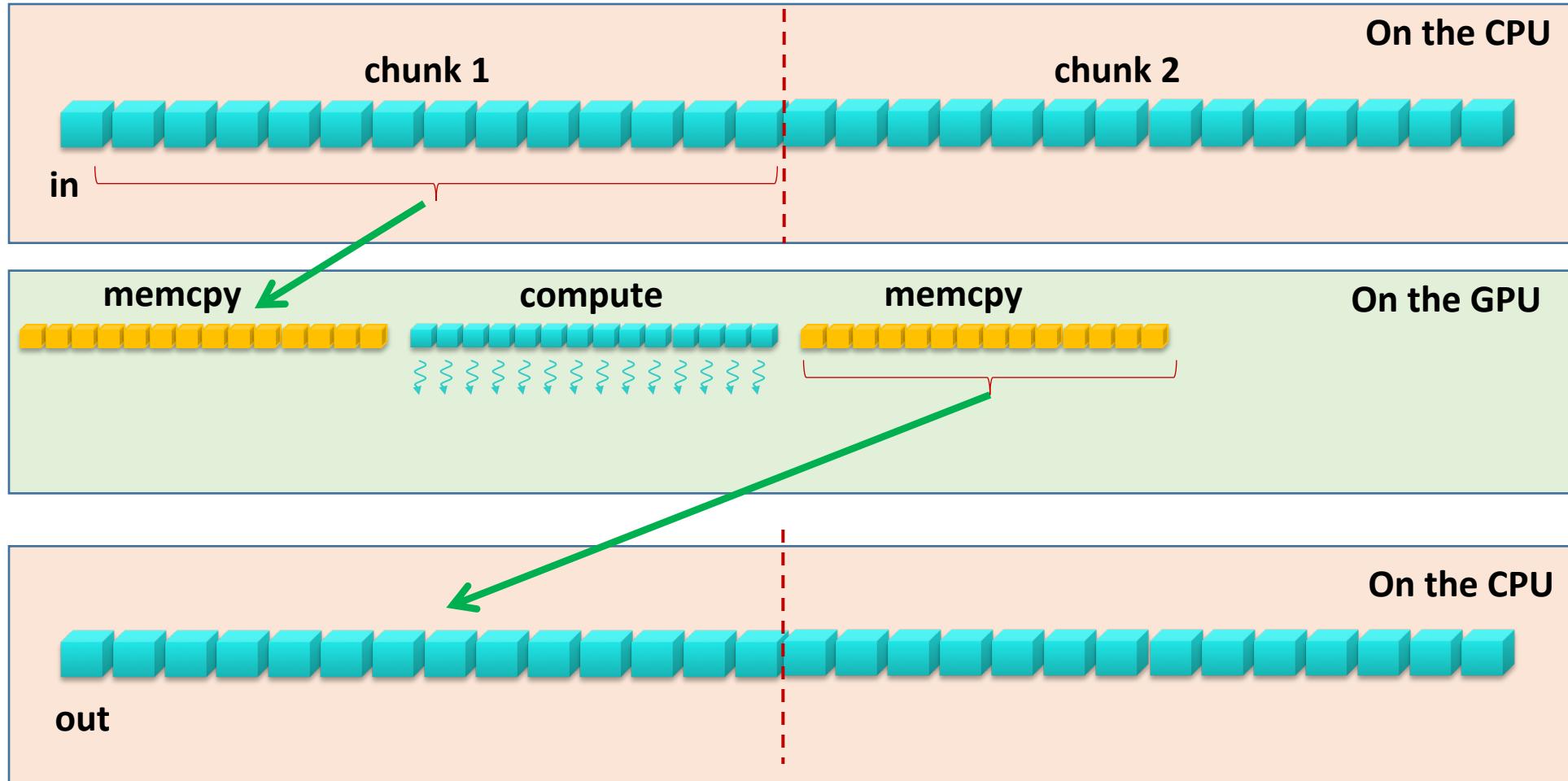
Data transfers between the host and the device using `cudaMemcpy()` are blocking transfers; that is, control is returned to the host thread only after the data transfer is complete. The `cudaMemcpyAsync()` function is a non-blocking variant of `cudaMemcpy()` in which control is returned immediately to the host thread. In contrast with `cudaMemcpy()`, the asynchronous transfer version *requires* pinned host memory (see [Pinned Memory](#)), and it contains an additional argument, a stream ID. A *stream* is simply a sequence of operations that are performed in order on the device. Operations in different streams can be interleaved and in some cases overlapped—a property that can be used to hide data transfers between the host and the device.

Asynchronous transfers enable overlap of data transfers with computation in two different ways. On all CUDA-enabled devices, it is possible to overlap host computation with asynchronous data transfers and with device computations. For example, [Overlapping computation and data transfers](#) demonstrates how host computation in the

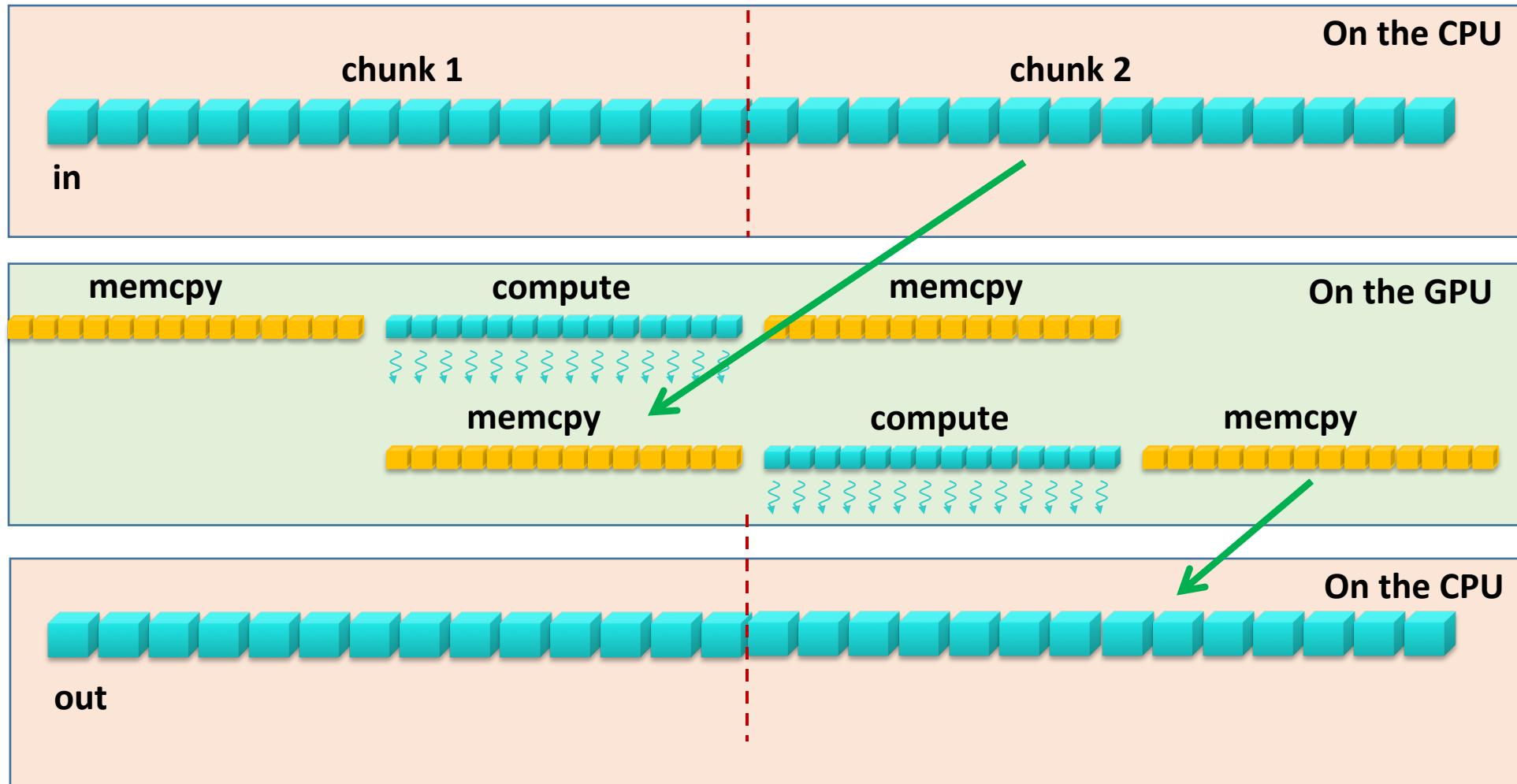
# Data Partitioning Example



# Data Partitioning Example

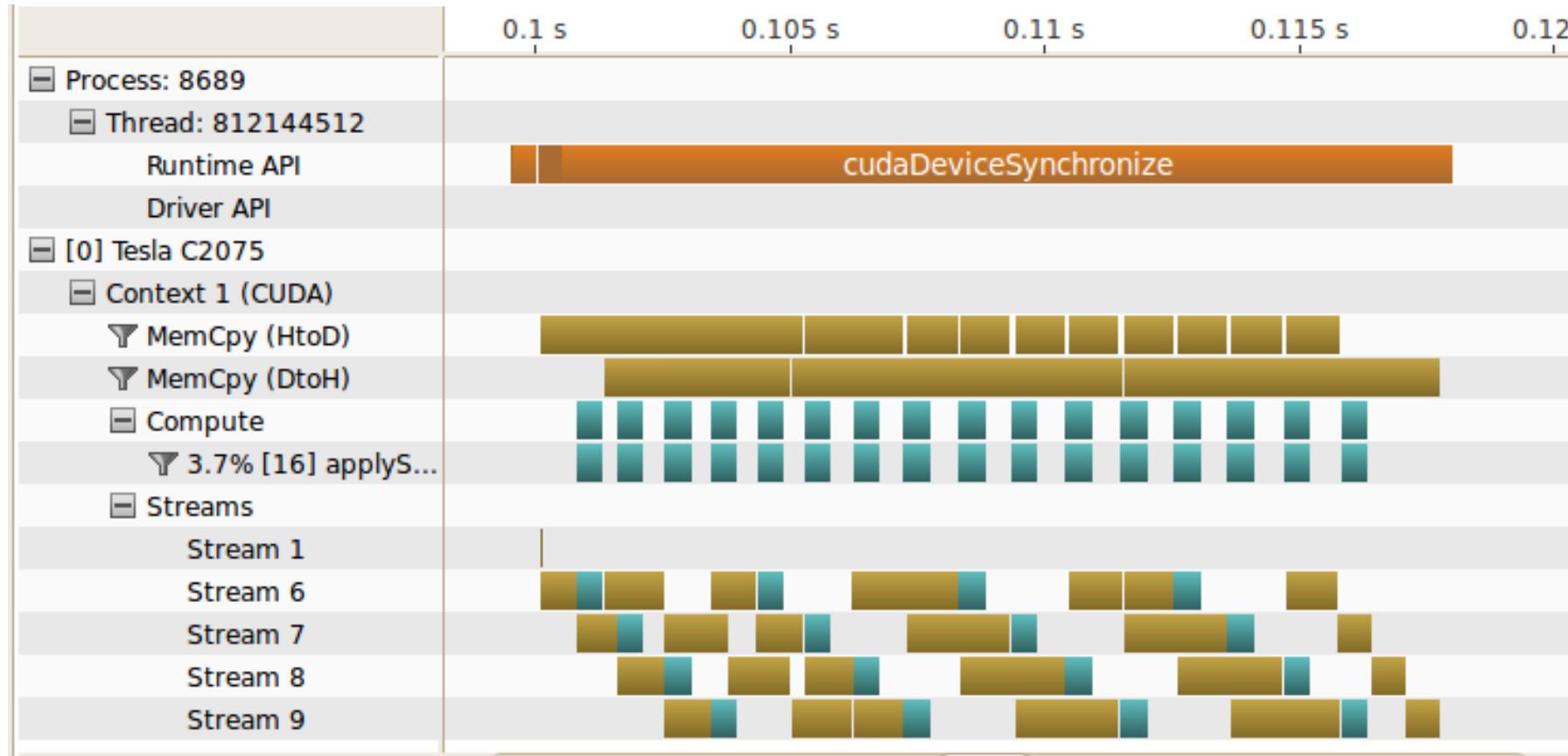


# Data Partitioning Example

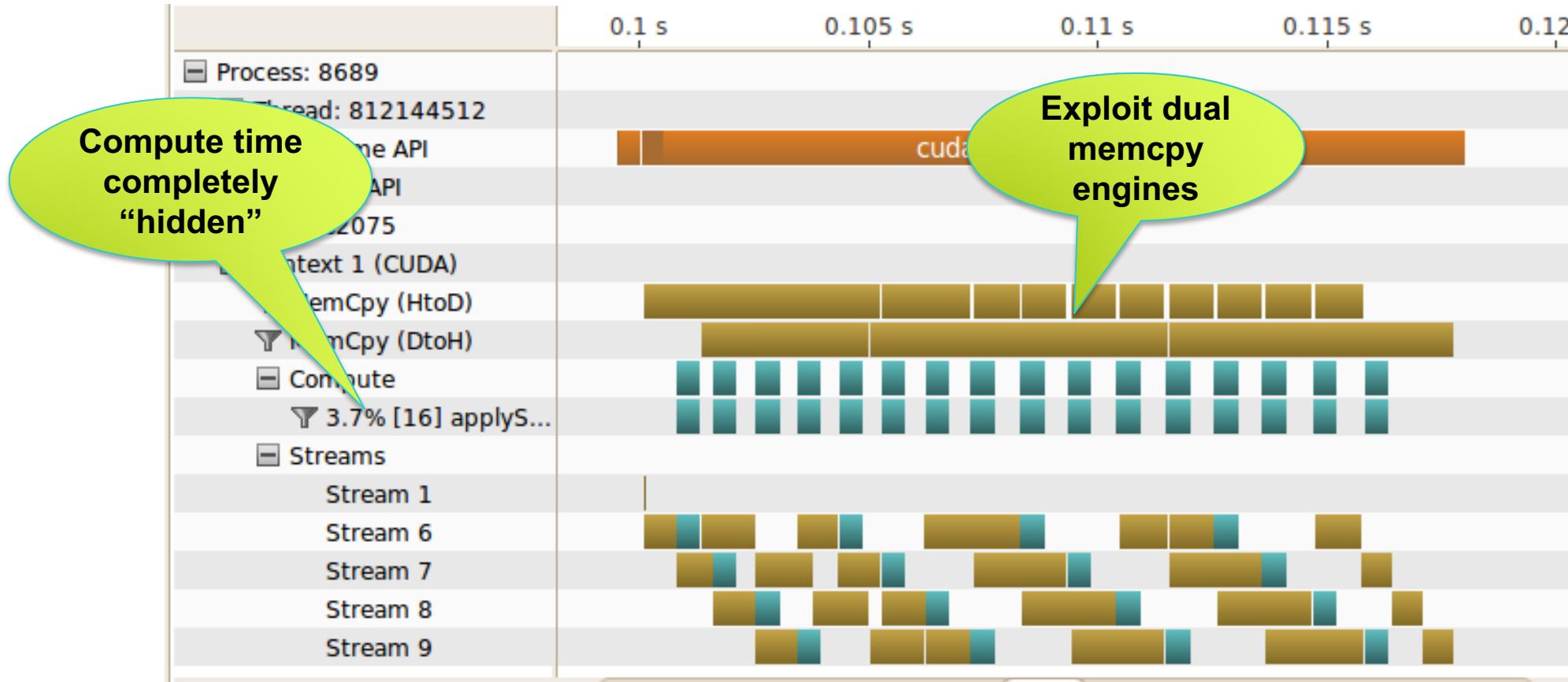


# Overlapped Compute/Memcpy

[problem broken into 16 chunks]



# Overlapped Compute/Memcpy



ME759: Use of multiple streams covered later

# Overlapped Compute/Memcpy

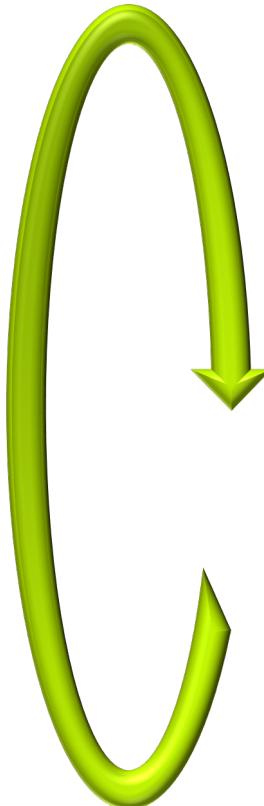
Device	Algorithm	MElements/s	Speedup
i7-930*	Optimized & Parallel	130	1x
Tesla C2075	Simple	285	2.2x
Tesla C2075	Pinned Memory	560	4.3x
Tesla C2075	Overlap	935	7.2x

# Optimization Summary

[Looking Back at 1D Stencil Example...]

- Initial CUDA parallelization
  - Expeditious, kernel almost word-for-word replica of sequential code
  - 2.2x speedup
- Optimize memory throughput
  - Minimal code change, yet need to know about pinned memory
  - 4.3x speedup
- Overlap compute and data movement
  - More involved, need to know about the inner works of CUDA
  - Problem should be large enough to justify mem-transfer/execution
  - 7.2x speedup

# Iterative Optimization

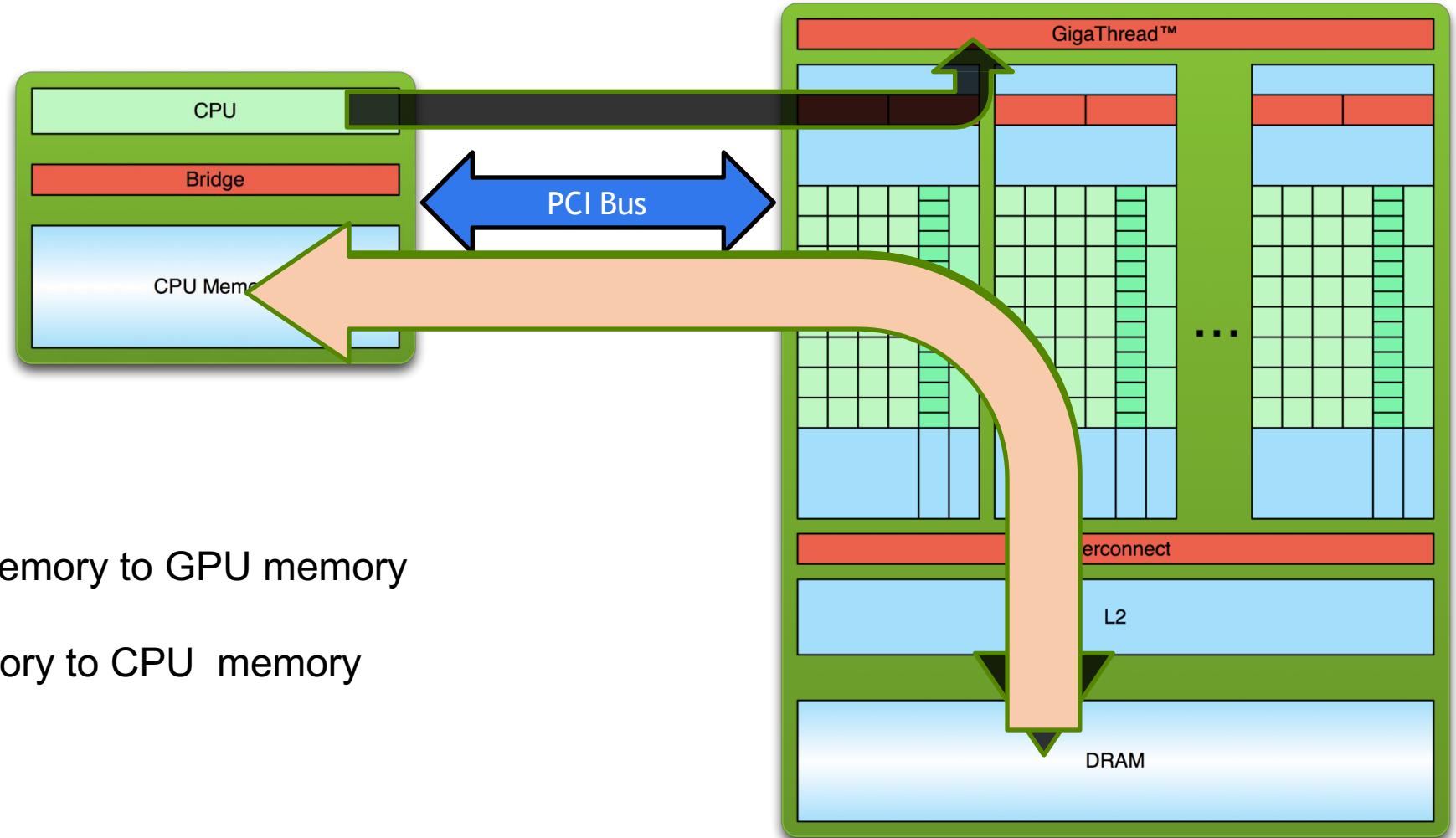


- Identify Optimization Opportunities
- Parallelize
- Optimize

# CUDA Streams

- Read this blog post: <https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency>

# The “GPU computing” process



1. Copy input data from CPU memory to GPU memory
2. Launch a GPU Kernel
3. Copy results from GPU memory to CPU memory
4. Repeat Many Times

# CUDA Streams: Prelude

- A CUDA enabled GPU has two engines
  - An execution engine
  - A copy engine, which has 2 sub-engines that can work simultaneously
    - A H2D copy sub-engine
    - A D2H copy sub-engine
- Goal of the “streams” segment of ME759: Learn how to simultaneously use both GPU engines
- NOTE (in relation to this “streams” segment of the course):
  - The important things happen on the host side, not on the device side

# Asynchronous Concurrent Execution

- In order to facilitate concurrent execution on host and device, some function calls are asynchronous
  - Control is returned to the host thread before the device has completed the requested task
- Examples of asynchronous calls
  - Kernel launches
  - device ↔ device memory copies
  - host ↔ device memory copies of a memory block of 64 KB or less
  - Memory copies performed by functions that are suffixed with Async (e.g., `cudaMemcpyAsync`)
- NOTE: When an application is run via a (`cuda-gdb`, `nvprof`), all launches are synchronous

# Non-blocking Host-Device Data Transfer Issues

- In general, host  $\leftrightarrow$  device data transfers using `cudaMemcpy()` are blocking
  - Control is returned to the host thread only after the data transfer is complete
- There is a non-blocking variant, `cudaMemcpyAsync()`

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
myKernel<<<grid,block>>>(a_d);
cpuFunction();
```

- The host does not wait on the device to finish the mem copy or on the completion of the kernel call for it to start execution of the `cpuFunction()` call
  - The launch of “myKernel” only happens after the mem copy call finishes
- NOTE: the asynchronous transfer version **requires pinned host memory** (allocated with `cudaHostAlloc()`), and it contains an additional argument (a stream ID)
  - The `cudaHostAlloc()` replaces `malloc()` typical call on the host side
- What does this buy us?
  - We have the CPU busy while copying data to/from device

# Overlapping Host $\leftrightarrow$ Device Data Transfer with Device Execution

- When is this overlapping useful?
  - Imagine a kernel executes on the device and only works with a portion (say *lower half*) of the device global memory
    - While kernel works on lower half, you can copy data from host to device into the *upper half* of the device global memory
  - Punch line: The kernel execution and the copy operations can take place simultaneously
- Note that there is an issue with this idea:
  - The device execution stack is FIFO, one function call on the device is not serviced until all the previous device function calls completed
  - This would prevent overlapping execution with data transfer
- This issue was addressed by the use of CUDA “streams”

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 16

02/28/2020

# Quote of the day

“Facts are meaningless. You could use facts to prove anything that's even remotely true.”

-- Homer Simpson, Safety Inspector [1989 - ]

# Before we get going...

- Last time:
  - Test case: scan operation
  - Test case: 1D stencil op
  - CUDA streams
- Today:
  - CUDA streams, wrapping up
  - Tools of the trade: GPU debugging & profiling
- Other tidbits:
  - Assignment due on Thursday at 9 pm
    - First case of cheating this semester :-(
  - Assigned readings – don't neglect them
  - Dan has office hours online each Wd@7 PM
  - Dan's office hours today: Lijing will hold them

# CUDA Streams: Overview

- A programmer can manage *concurrency* through *streams*
  - “*concurrency*” refers to “the copy and the execution engines of the GPU working at the same time” or “multiple different kernels being executed at the same time on the GPU”
- A stream is a sequence of CUDA commands issued by the host that execute on the GPU in issue-order
  - CUDA stream: a queue of GPU operations; i.e., of device work
  - The execution order in a stream defines the order in which the GPU operations are added to the stream (FIFO)
  - **NOTE:** an operation in a stream does not commence prior to the previous operation in that stream being fully completed
    - There is a distinction between queuing an operation in a stream and the moment when it actually starts to be executed on the GPU

# CUDA Streams: Overview

[Cntd.]

- One host thread can define multiple CUDA streams
- What are the typical operations in a stream?
  - Invoking a data transfer
  - Invoking a kernel execution
  - Handling events
- The CUDA run-time executes commands from different streams as it sees fit
  - No inter-stream relative synchronization unless the user explicitly takes care of synchronization process
  - Also, streams can be synchronized at barrier points, but correlation of sequence execution between different streams not supported

# CUDA Streams: Quick notes

- As soon as you invoke a CUDA function you create a default stream (stream 0)
- If you don't explicitly state a stream in the execution configuration of a kernel it is assumed it's launched as part of "Stream 0" (zero)

# CUDA Streams: Creation

- A stream is defined by creating a stream object
  - It is subsequently used by specifying it as the stream parameter to a sequence of kernel launches and host-to/from-device memory copies
- The following code sample creates two streams and allocates an array “hostPtr” of float in page-locked memory
  - hostPtr will be used in asynchronous host ↔ device memory transfers

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Note the length of the array. Recall previous upper/lower side of the memory discussion.

# CUDA Streams: Making Use of Them

- In the code below, each of the two streams is defined as a sequence of
  - One memory copy from host to device,
  - One kernel launch, and
  - One memory copy from device to host

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);

    MyKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);

    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);
}
```

- There are some wrinkles to it, we'll revisit shortly...

# CUDA Streams: Clean Up Phase

- Streams are folded by calling `cudaStreamDestroy()`

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

- `cudaStreamDestroy()` waits for all preceding commands in the given stream to complete before destroying the stream and returning control to the host thread

# CUDA Streams: Caveats

- Two GPU-involving tasks from different streams cannot run concurrently if one of the following operations is issued in-between them by the host thread:
  - A page-locked host memory allocation
  - A device memory allocation
  - A device memory set
  - A device  $\leftrightarrow$  device memory copy
  - A CUDA command to stream 0 (including kernel launches and host  $\leftrightarrow$  device memory copies that do not specify any stream parameter)
  - A switch between the L1/shared memory configurations

# CUDA Stream: More Caveats

- All GPU calls (memcpy, kernel execution, etc.) are placed into default stream unless otherwise specified
- Stream 0 is special (has special synchronization rules)
  - Synchronous with all streams
    - Meaning: Things done in stream 0 cannot overlap other streams
      - Exception: see next bullet
- Streams with non-blocking flag are exception
  - `cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking)`

# CUDA Streams: Synchronization Aspects

**cudaDeviceSynchronize()** halts execution on the host until all preceding commands in all CUDA streams have completed

- Halts execution of the host

**cudaStreamSynchronize()** takes a stream as a parameter and halts execution on the host until all preceding commands in the given CUDA stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device

- Halts execution of the host

**cudaStreamWaitEvent()** takes a CUDA stream and an event as parameters and makes all the commands added to the given stream after the call to **cudaStreamWaitEvent()** delay their execution until the given event has completed.

- Halts execution within a stream

**cudaStreamQuery()** provides applications with a way to know if all preceding commands in a stream have completed

- NOTE: To avoid unnecessary slowdowns, use these synchronization functions sparingly

# Example 1: Using One Stream

[Enable both CPU and GPU to mind their business at the same time]

- Example draws on material presented in the “CUDA By Example” book
  - J. Sanders and E. Kandrot, authors
- What is the purpose of this example?
  - Shows strategy that you can invoke when dealing with applications that require **more memory than you can accommodate on the GPU**
- Remark:
  - In this example the magic happens on the host side. Focus on host code, not on the kernel executed on the GPU (the kernel code is basically irrelevant)

# This Example's Kernel

- Computes some average, it's not important, simply something that gets done and allows us later on to gauge efficiency gains when using \*multiple\* streams (for now dealing with one stream only)
  - Inputs: **a** and **b**
  - Output: **C**

```
#include "../common/book.h"

#define N    1048576 // this is 1024*1024
#define FULL_DATA_SIZE  (N*20)

__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float   as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float   bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2;
    }
}
```

# The “main()” Function

```
01| int main( void ) {
02|     cudaEvent_t      start, stop;
03|     float            elapsedTime;
04|
05|     cudaStream_t     stream;
06|     int *host_a, *host_b, *host_c;
07|     int *dev_a, *dev_b, *dev_c;
08|
09|     // start the timers
10|     cudaEventCreate( &start );
11|     cudaEventCreate( &stop   );
12|
13|     // initialize the stream; only one stream for now...
14|     cudaStreamCreate( &stream );
15|
16|     // allocate the memory on the GPU
17|     cudaMalloc( (void**)&dev_a, N * sizeof(int) );
18|     cudaMalloc( (void**)&dev_b, N * sizeof(int) );
19|     cudaMalloc( (void**)&dev_c, N * sizeof(int) );
20|
21|     // allocate host pinned memory, used to stream
22|     cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
23|     cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
24|     cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
25|
26|     for (int i=0; i<FULL_DATA_SIZE; i++) {
27|         host_a[i] = rand();
28|         host_b[i] = rand();
29|     }
```

Stage 1

Stage 2

# The “main()” Function [Cntd.]

```
30|     cudaEventRecord( start, 0 );
31|     // now loop over full data, chunk by chunk
32|     for (int i=0; i<FULL_DATA_SIZE; i+= N) {
33|         // copy the locked memory to the device, async
34|         cudaMemcpyAsync( dev_a, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream );
35|         cudaMemcpyAsync( dev_b, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream );
36|
37|         kernel<<<(N+255)/256,256,0,stream>>>( dev_a, dev_b, dev_c );
38|
39|         // copy the data from device to locked memory
40|         cudaMemcpyAsync( host_c+i, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost, stream );
41|
42|     }
43|
44|
45|     cudaStreamSynchronize( stream );
46|
47|     cudaEventRecord( stop, 0 );
48|
49|     cudaEventSynchronize( stop );
50|     cudaEventElapsedTime( &elapsedTime, start, stop ) );
51|     printf( "Time taken: %3.1f ms\n", elapsedTime );
52|
53|     // cleanup the streams and memory
54|     cudaFreeHost( host_a );
55|     cudaFreeHost( host_b );
56|     cudaFreeHost( host_c );
57|     cudaFree( dev_a );
58|     cudaFree( dev_b );
59|     cudaFree( dev_c );
60|     cudaStreamDestroy( stream );
61|
62|     return 0;
63| }
```

Stage 3

Stage 4

Stage 5

# Example 1, Summary

- Stage 1 sets up the events needed to time the execution of the program
- Stage 2 allocates page-locked memory on the host side so that we can fall back on asynchronous memory copy operations between host and device
- Stage 3 enqueues the set of GPU operations that need to be undertaken (the “chunkification”)
- Stage 4 needed for timing reporting
- Stage 5: clean up time

# “Chunkfication”

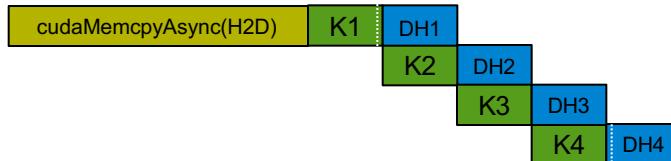
- “chunkification” similar to “tiling”
- However,
  - “tiling” is something that happens exclusively on the device (from global to shared memory).
  - “chunkification” happens on the host
- Last lecture (1D stencil operation): we pre-fetched chunks

# Gaining speed through concurrency

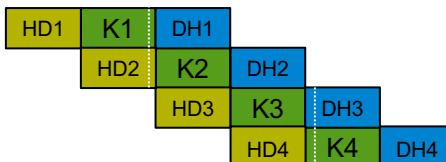
- No concurrency (1x)



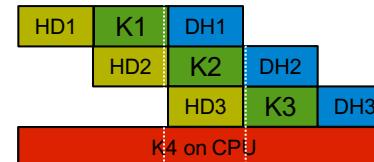
- 2-way concurrency (up to 2x speedup)



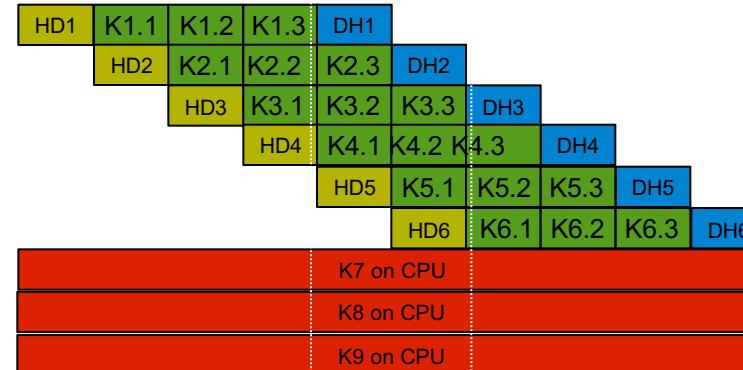
- 3-way concurrency (up to 3x speedup)



- 4-way concurrency (3x+)



- 4+ way concurrency

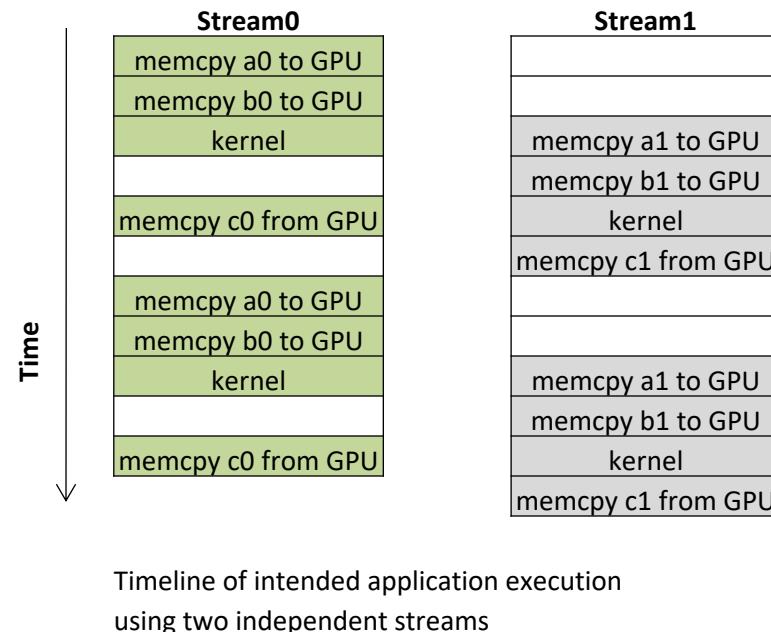


# Example 2: Using Multiple Streams

[Version 2.1]

- Implement the same example but use two streams to this end
- Why would you want to use multiple streams?
  - Overlapping GPU execution with host  $\leftrightarrow$  device data movement can improve overall performance
- Two ideas underlie the process
  - The idea of “chunkification” of the computation
    - Computation is broken into pieces that are queued up for execution on the device (we already saw this in Example 1, which uses one stream)
  - The idea of overlapping execution with PCI host  $\leftrightarrow$  device data movement

# Overlapping Execution and Data Transfer: A Desirable Scenario



- **Observations:**

- “memcpy” actually represents an asynchronous `cudaMemcpyAsync()` memory copy call
- White (empty) box: time when one stream is waiting to execute an operation that it cannot overlap with other stream’s operation
- The goal: keep both GPU engine types (execution and mem copy) busy
  - Note: recent hardware allows two copies to take place simultaneously: one from host to device, at the same time one goes on from device to host (you have two copy subengines). This example shows only one copy-engine (old CC).

# The “main()” Function, Two Streams

```
01| int main( void ) {
02|     cudaDeviceProp prop;
03|     int whichDevice;
04|     HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
05|     HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
06|     if ( !prop.deviceOverlap) {
07|         printf( "Device will not handle overlaps, so no speed up from streams\n" );
08|         return 0;
09|     }
10|
11|     cudaEvent_t      start, stop;
12|     float            elapsedTime;
13|
14|     cudaStream_t      stream0, stream1;
15|     int *host_a, *host_b, *host_c;
16|     int *dev_a0, *dev_b0, *dev_c0;
17|     int *dev_a1, *dev_b1, *dev_c1;
18|
19|     // start the timers
20|     cudaEventCreate( &start );
21|     cudaEventCreate( &stop );
22|
23|     // initialize the streams
24|     cudaStreamCreate( &stream0 );
25|     cudaStreamCreate( &stream1 );
26|
27|     // allocate the memory on the GPU
28|     cudaMalloc( (void**)&dev_a0, N * sizeof(int) );
29|     cudaMalloc( (void**)&dev_b0, N * sizeof(int) );
30|     cudaMalloc( (void**)&dev_c0, N * sizeof(int) );
31|     cudaMalloc( (void**)&dev_a1, N * sizeof(int) );
32|     cudaMalloc( (void**)&dev_b1, N * sizeof(int) );
33|     cudaMalloc( (void**)&dev_c1, N * sizeof(int) );
34|
35|     // allocate host locked memory, used to stream
36|     cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
37|     cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
38|     cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
```

Stage 1

Stage 2

Stage 3

# The “main()” Function, Two Streams

[Cntd.]

```
39|     for (int i=0; i<FULL_DATA_SIZE; i++) {           Still Stage 3
40|         host_a[i] = rand();
41|         host_b[i] = rand();
42|
43|
44|         cudaEventRecord( start, 0 );
45|         // now loop over full data, in bite-sized chunks
46|         for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
47|             // copy data from pinned memory to the device, async
48|             cudaMemcpyAsync( dev_a0, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 );
49|             cudaMemcpyAsync( dev_b0, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 );
50|
51|             kernel<<<(N+255)/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
52|
53|             // copy the data from device to locked memory
54|             cudaMemcpyAsync( host_c+i, dev_c0, N * sizeof(int), cudaMemcpyDeviceToHost, stream0 );
55|
56|
57|             // copy the locked memory to the device, async
58|             cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 );
59|             cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 );
60|
61|             kernel<<<(N+255)/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
62|
63|             // copy the data from device to locked memory
64|             cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 );
65|
66| }
```

Stage 4

# The “main()” Function, Two Streams [Cntd.]

```
67| cudaStreamSynchronize( stream0 );
68| cudaStreamSynchronize( stream1 );
69|
70| cudaEventRecord( stop, 0 );
71|
72| cudaEventSynchronize( stop );
73| cudaEventElapsedTime( &elapsedTime, start, stop ) );
74| printf( "Time taken: %3.1f ms\n", elapsedTime );
75|
76| // cleanup the streams and memory
77| cudaFreeHost( host_a );
78| cudaFreeHost( host_b );
79| cudaFreeHost( host_c );
80| cudaFree( dev_a0 );
81| cudaFree( dev_b0 );
82| cudaFree( dev_c0 );
83| cudaFree( dev_a1 );
84| cudaFree( dev_b1 );
85| cudaFree( dev_c1 );
86| cudaStreamDestroy( stream0 );
87| cudaStreamDestroy( stream1 );
88|
89| return 0;
90| }
```

Stage 5

**NOTE: the kernel doesn't actually change...**

# Example 2.1 [Version 1], Summary

- Stage 1 ensures that your device supports your attempt to overlap kernel execution with host↔device data transfer
- Stage 2 sets up the events needed to time the execution of the program
- Stage 3 allocates page-locked memory on the host side so that we can fall back on asynchronous memory copy operations between host and device and initializes data
- Stage 4 enqueues the set of GPU operations that need to be undertaken (the “chunkification”)
- Stage 5 takes care of timing reporting and clean up

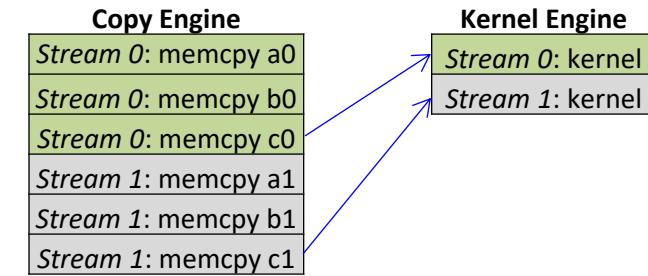
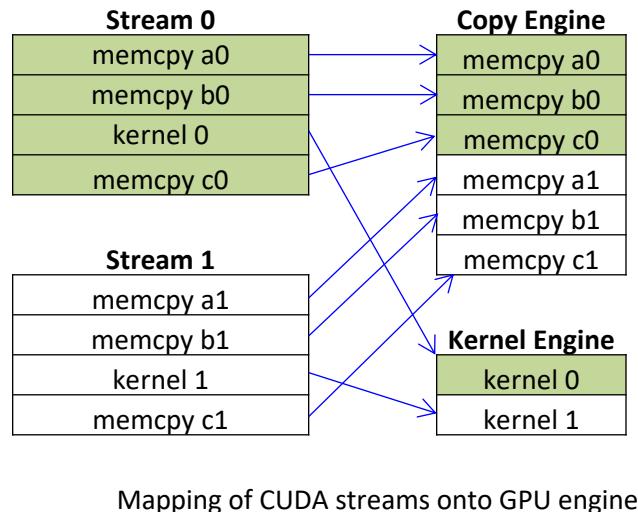
# Comments, Using Two Streams

[Version 2.1]

- Timing results provided by “CUDA by Example: An Introduction to General-Purpose GPU Programming,”
  - Sanders and Kandrot reported results on NVIDIA GTX285
- Using one stream (in Example 1): 62 ms
- Using two streams (this example, version 1): 61 ms
- Lackluster performance goes back to the way the two GPU engines (kernel execution and copy) are scheduled

# The Two Stream Example, Version 2.1

## Looking Under the Hood

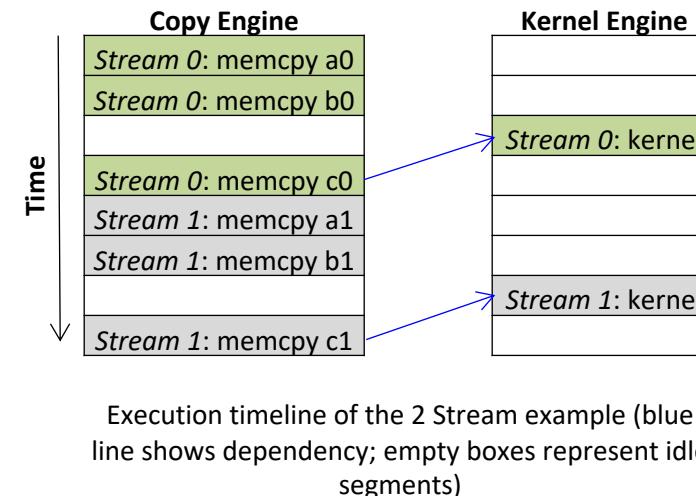


Arrows depicting the dependency of `cudaMemcpyAsync()` calls on kernel executions in the 2 Streams example

- At the left:
  - An illustration of how the work queued up in the streams ends up being assigned by the CUDA driver to the two GPU engines (copy and execution)
  - Important remark: FIFO is also observed in relation to scheduling the engines (not only the streams)
- At the right
  - Image shows dependency that is implicitly set up in the two streams given the way the streams were defined in the code
  - The queue in the Copy Engine combined with the implied stream dependencies determines the scheduling of the Copy and Kernel Engines (next slide)

# The Two Stream Example

## Looking Under the Hood



- Note that due to the *\*specific\** way in which the streams were defined (depth first), basically there is no overlap of copy & execution...
  - Explains the no net-gain in efficiency compared to the one stream example
- Remedy: go breadth first, instead of depth first
  - In the current version, execution on the two engines was inadvertently blocked by the way the streams have been set up and the existing scheduling and lack of dependency checks available in the current version of CUDA

# The Two Stream Example

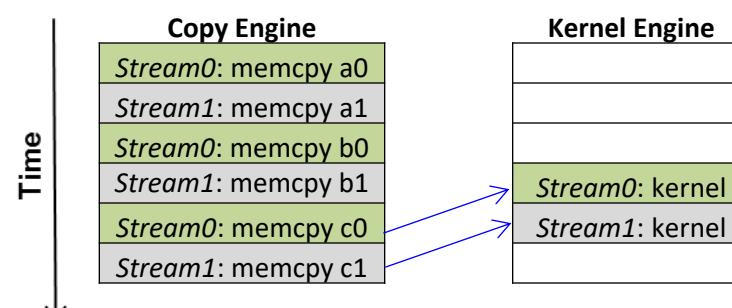
[Version 2.2: A More Effective Implementation: Breadth First]

- The way we just went about business (the depth first approach):
  - Assign the copy of **a0**, copy of **b0**, kernel execution, and copy of **c0** to stream0
  - Assign the copy of **a1**, copy of **b1**, kernel execution, and copy of **c1** to stream1
- New way (the breadth first approach):
  - Add the copy of **a0** to stream0, and then add the copy of **a1** to stream1
  - Next, add the copy of **b0** to stream0, and then add the copy of **b1** to stream1
  - Next, enqueue the kernel invocation in stream0, then enqueue one in stream1
  - Finally, enqueue the copy of **c0** back to the host in stream0 followed by the copy of **c1** in stream1

# The Two Stream Example

```
A| // loop over full data, in bite-sized chunks
B| for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
C|     // enqueue copies of a in stream0 and stream1
D|     cudaMemcpyAsync( dev_a0, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 );
E|     cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 );
F|     // enqueue copies of b in stream0 and stream1
G|     cudaMemcpyAsync( dev_b0, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 );
H|     cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 );
I|
J|     // enqueue kernels in stream0 and stream1
K|     kernel<<<(N+255)/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
L|     kernel<<<(N+255)/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
M|
N|     // enqueue copies of c from device to locked memory
O|     cudaMemcpyAsync( host_c+i , dev_c0, N * sizeof(int), cudaMemcpyDeviceToHost, stream0 );
P|     cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 );
Q| }
```

Replaces Previous Stage 4



Execution timeline of the breadth-first approach  
(blue line shows dependency)

A 20% More Effective  
Implementation (48 vs. 61 ms)

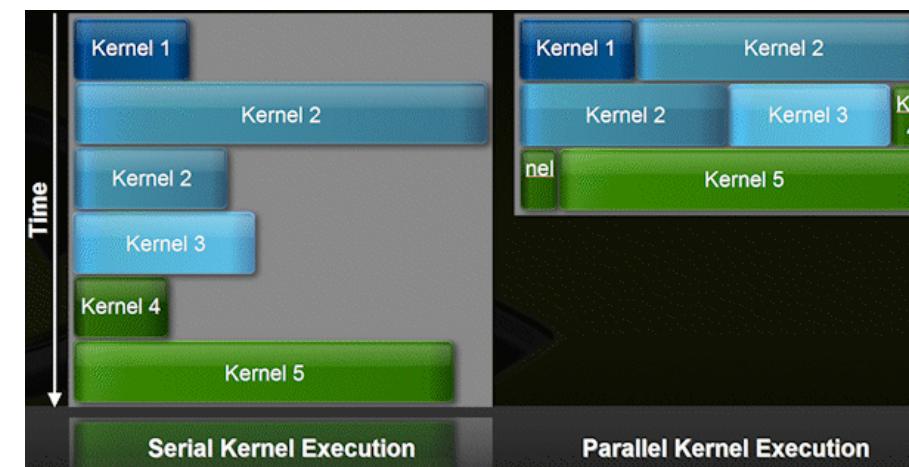
# Using Streams, Lessons Learned

- Streams provide a basic mechanism that enables task-level parallelism in CUDA C applications
- Two requirements underpin the use of streams in CUDA C
  1. `cudaHostAlloc()` should be used to allocate memory on the host so that it can be used in conjunction with a `cudaMemcpyAsync()` non-blocking copy command
    - Use of pinned host memory improves data transfer performance even when only working with one stream
  2. Effective latency hiding of kernel execution with memory copy operations requires a breadth-first approach to enqueueing operations in different streams
    - This is a consequence of the two engine setup associated with a GPU

# Concurrent Kernel Execution

[another type of concurrency through use of CUDA streams]

- As early as Fermi one can have multiple kernels run on the device at the same time
  - Example: up to 16 on Fermi
- When is this useful?
  - A kernel might be launched with an execution configuration that doesn't fully utilize entire GPU
  - Main idea: two or three independent kernels can be "squeezed" on GPU at the same time
- GPU looks like a MIMD architecture
  - Requires use of **multiple streams**



# CUDA Streams: More recent developments

- Since CUDA 5.0
  - Stream Callbacks, will call host function when stream has finished all work
  - `cudaStreamAddCallback( cudaStream_t stream, cudaStreamCallback_t callback, void* userData, unsigned int flags )`
- Since CUDA 5.5
  - You can give streams priority
    - `cudaStreamCreateWithPriority`
    - Use `cudaDeviceGetStreamPriorityRange` to get the available priorities
- Since CUDA 7:
  - nvcc –default-stream per-thread
  - Each host thread will get its own stream
  - Each stream becomes a regular non-blocking stream

# CUDA basics, wrapping up

# GPU Computing: Good in Engineering?

- What applications stand to benefit in the Engineering?
  - Image processing
  - FEA
  - Molecular Dynamics
  - Granular Dynamics
  - Finite Differences schemes
  - Quantum Chemistry
  - ...
- Generally, any application that fits the SIMD paradigm

# Topics not covered, or not covered yet

- Debugging & profiling in CUDA (next)
- Dynamic parallelism (not covered)
- Unified memory (since CUDA 6.0) `cudaMallocManaged`, `cudaDeviceSynchronize` (later)
- Using multiple GPUs (probably not covered)

# Departing Thoughts

- Hardware changes at faster pace than software
- CUDA – a bright spot in the software landscape
- For SIMD applications, GPU can deliver good speedups at small time and financial investments
- In general, investing in parallel programming skills bound to pay off

# Further Information

- CUDA tutorials video/slides at GTC
  - <http://www.gputechconf.com>
- CUDA webinars covering many introductory to advanced topics
  - <http://developer.nvidia.com/gpu-computing-webinars>
- CUDA Tools Download: <http://www.nvidia.com/getcuda>

# Further Reading

- Read, in this order:
  - NVIDIA CUDA Development Tools: Getting Started
  - NVIDIA CUDA Programming Guide
  - NVIDIA CUDA C Programming Best Practices Guide
  - NVIDIA CUDA Reference Manual
- Lots of very good examples come with the **CUDA SDK** distribution
  - More than 50 applications ready to compile/run
  - Makefiles available, ready for use
  - Lots of good code available for reuse + templates for applications
- Online material
  - NVIDIA website: code available for many application fields
  - Libraries: thrust (<http://code.google.com/p/thrust/>), cudpp (<http://gpgpu.org/developer/cudpp>)

End of CUDA basics

# Debugging GPU CUDA code

# CUDA-GDB Main Features

[see reference manual for detailed description: <https://docs.nvidia.com/pdf/cuda-gdb.pdf>]

- All the standard GDB debugging features
- Integrated CPU and GPU debugging within a single session
- Breakpoints and Conditional Breakpoints
- Inspect memory, registers, local/shared/global variables
- Supports multiple GPUs, multiple kernels
- Source and Assembly (SASS) Level Debugging
- Runtime Error Detection (stack overflow,...)

# Recommended Compilation Flags

- Compile code for your target architecture:
- Example:
  - Fermi : -gencode arch=compute\_20,code=sm\_20
  - Etc.
  - Pascal : -gencode arch=compute\_60,code=sm\_60
- Compile code with the debug flags:
  - Host code : -g
  - Device code : -G
- Example:

```
$ nvcc -g -G -gencode arch=compute_60,code=sm_60 test.cu -o test
```

# Usage [On your desktop, at home or office]

- Invoke **cuda-gdb** from the command line:

```
$ cuda-gdb my_application  
(cuda-gdb) _
```

**DO NOT use this on Euler (see next slide for Euler)**

# Usage, cuda-gdb on Euler

- Here's how things get run on Euler:

```
>> srun -p wacc -t 0-0:10:0 --gres=gpu:1 --pty -u cuda-gdb myProgramName
```

- If you don't have a visual debugger, you might use this:

```
>> cuda-gdb -tui myApp
```

# Program Execution Control

# Execution Control

- Execution Control is identical to host debugging:

- Launch the application

```
(cuda-gdb) run
```

- Resume the application (all host threads and device threads)

```
(cuda-gdb) continue
```

- Kill the application

```
(cuda-gdb) kill
```

- Interrupt the application: CTRL-C

# Execution Control

- Single-Stepping

Single-Stepping	At the source level	At the assembly level
Over function calls	<code>next</code>	<code>nexti</code>
Into function calls	<code>step</code>	<code>stepi</code>

- Behavior varies when stepping `__syncthreads()`

PC at a <i>barrier</i> ?	Single-stepping applies to	Notes
Yes	Active and divergent threads of the warp in focus and all the warps that are running the same <u>block</u> .	Required to step over the barrier.
No	<u>Active threads</u> in the warp in focus only.	

# Breakpoints

- By name

```
(cuda-gdb) break my_kernel  
(cuda-gdb) break _Z6kernelIfiEvPT_PT0
```

(cuda-gdb) break -EXCLUSIVELLAUNCH

- By file name and line number

```
(cuda-gdb) break test.cu:380
```

- By address

```
(cuda-gdb) break *0x3e840a8  
(cuda-gdb) break *$pc
```

(cuda-gdb) break \*\$bc

- At every kernel launch

```
(cuda-gdb) set cuda break_on_launch appNameHere
```

# Conditional Breakpoints

- Only reports hit breakpoint if condition is met
  - All breakpoints are still hit
  - Condition is evaluated every time for all the threads
  - Typically slows down execution
- Condition
  - C/C++ syntax
  - No function calls
  - Supports built-in variables (`blockIdx`, `threadIdx`, ...)

# Conditional Breakpoints

- Set at breakpoint creation time

```
(cuda-gdb) break my_kernel if threadIdx.x == 13
```

- Set after the breakpoint is created
  - Breakpoint 1 was previously created

```
(cuda-gdb) condition 1 blockIdx.x == 0 && n > 3
```

# Thread Focus

# Thread Focus

- There might be thousands of threads to deal with. Can't display all of them
- Thread focus dictates which thread you are looking at
- Some commands apply only to the thread in focus
  - Print local or shared variables
  - Print registers
  - Print stack contents
- Attributes of the “thread focus”
  - Kernel index : unique, assigned at kernel’s launch time
  - Block index : the application `blockIdx`
  - Thread index : the application `threadIdx`

# Devices

- To obtain the list of devices allocated to job:

```
(cuda-gdb) info cuda devices
```

Dev	Desc	Type	SMs	Wps/SM	Lns/Wp	Regs/Ln	Active	SMs	Mask
*	0	gf100	sm_20	14	48	32	64		0xffff
	1	gt200	sm_13	30	32	32	128		0x0

- The \* indicates the device of the kernel currently in focus
- Provides an overview of the hardware that supports the code

# Kernels

- To obtain the list of running kernels:

```
(cuda-gdb) info cuda kernels
```

	Kernel	Dev	Grid	SMs	Mask	GridDim	BlockDim	Name	Args
*	1	0	2	0x3fff	(240,1,1)	(128,1,1)	acos	parms=...	
	2	0	3	0x4000	(240,1,1)	(128,1,1)	asin	parms=...	

- The \* indicates the kernel currently in focus
- There is a one-to-one mapping between a kernel id (unique id across multiple GPUs) and a (dev,grid) tuple. The grid id is unique per GPU only
- The name of the kernel is displayed as are its parameters and the associated execution configuration
- Provides an overview of the code running on the hardware

# Thread Focus

- To switch focus to any currently running thread

```
(cuda-gdb) cuda kernel 2 block 1,0,0 thread 3,0,0
[Switching focus to CUDA kernel 2 block (1,0,0), thread (3,0,0)

(cuda-gdb) cuda kernel 2 block 2 thread 4
[Switching focus to CUDA kernel 2 block (2,0,0), thread (4,0,0)

(cuda-gdb) cuda thread 5
[Switching focus to CUDA kernel 2 block (2,0,0), thread (5,0,0)
```

# Thread Focus

- To obtain the current focus:

```
(cuda-gdb) cuda kernel block thread  
kernel 2 block (2,0,0), thread (5,0,0)
```

```
(cuda-gdb) cuda thread  
thread (5,0,0)
```

# Threads

- To obtain the list of running threads for kernel 2:

```
(cuda-gdb) info cuda threads kernel 2
```

Block	Thread	To	Block	Thread	Cnt	PC	Filename	Line
*	(0,0,0)	(0,0,0)	(3,0,0)	(7,0,0)	32	0x7fae70	acos.cu	380
	(4,0,0)	(0,0,0)	(7,0,0)	(7,0,0)	32	0x7fae60	acos.cu	377

- Threads are displayed as (block,thread) ranges
- Divergent threads are in separate ranges
- The \* indicates the range where the thread in focus resides
- **Cnt** indicates the number of threads within each range
  - All threads in the same range are contiguous (no hole)
  - All threads in the same range shared the same PC (and filename/line number)

# Program State Inspection

# Stack Trace

- Same (aliased) commands as in **gdb**:

```
(cuda-gdb) where  
(cuda-gdb) bt  
(cuda-gdb) info stack
```

- Applies to the thread in focus
- NOTE: backtrace command, or its alias **bt**, prints a back-trace of the entire stack

# Stack Trace

```
(cuda-gdb) info stack  
#0  fibo_aux (n=6) at fibo.cu:88  
#1  0x7bbda0 in fibo_aux (n=7) at fibo.cu:90  
#2  0x7bbda0 in fibo_aux (n=8) at fibo.cu:90  
#3  0x7bbda0 in fibo_aux (n=9) at fibo.cu:90  
#4  0x7bbda0 in fibo_aux (n=10) at fibo.cu:90  
#5  0x7cfdb8 in fibo_main<<<(1,1,1),(1,1,1)>>> (...) at fibo.cu:95
```

# Source Variables

- Source variable must be live (in the scope)
- Read a source variable

```
(cuda-gdb) print my_variable  
$1 = 3  
  
(cuda-gdb) print &my_variable  
$2 = (@global int *) 0x200200020
```

- Write a source variable

```
(cuda-gdb) print my_variable = 5  
$3 = 5
```

# Memory

- Memory read & written like source variables

```
(cuda-gdb) print *my_pointer
```

- May require storage specifier when ambiguous

@global, @shared, @local  
@generic, @texture, @parameter

```
(cuda-gdb) print * (@global int *) my_pointer
```

```
(cuda-gdb) print ((@texture float **) my_texture)[0][3]
```

# Hardware Registers

- CUDA Registers
  - Virtual PC: \$pc (read-only)
  - SASS registers: \$R0, \$R1,...
- Show a list of registers (no argument to get all of them)

```
(cuda-gdb) info registers R0 R1 R4
R0          0x6      6
R1          0xffffc68 16776296
R4          0x6      6
```

- Modify one register

```
(cuda-gdb) print $R3 = 3
```

# Code Disassembly

- Must have **cuobjdump** in \$PATH

```
(cuda-gdb) x/10i $pc
0x123830a8 <_Z9my_kernel10params+8>:    MOV R0, c [0x0] [0x8]
0x123830b0 <_Z9my_kernel10params+16>:   MOV R2, c [0x0] [0x14]
0x123830b8 <_Z9my_kernel10params+24>:   IMUL.U32.U32 R0, R0, R2
0x123830c0 <_Z9my_kernel10params+32>:   MOV R2, R0
0x123830c8 <_Z9my_kernel10params+40>:   S2R R0, SR_CTAid_X
0x123830d0 <_Z9my_kernel10params+48>:   MOV R0, R0
0x123830d8 <_Z9my_kernel10params+56>:   MOV R3, c [0x0] [0x8]
0x123830e0 <_Z9my_kernel10params+64>:   IMUL.U32.U32 R0, R0, R3
0x123830e8 <_Z9my_kernel10params+72>:   MOV R0, R0
0x123830f0 <_Z9my_kernel10params+80>:   MOV R0, R0
```

- Meaning of this command:
  - X = disassemble (eXtract)
  - 10i = 10 instructions

# Run-Time Error Detection

# cuda-memcheck

- Stand-alone run-time error checker tool
- Detects memory errors like stack overflow, illegal global address,...
- Similar to `valgrind`
- No need to recompile the application
- Not all the error reports are precise
- Can be used within `cuda-gdb`

# cuda-memcheck errors

Illegal global address

Misaligned global address

Stack memory limit exceeded

Illegal shared/local address

Misaligned shared/local address

Instruction accessed wrong memory

PC set to illegal value

Illegal instruction encountered

Illegal global address

# cuda-memcheck

- Integrated in **cuda-gdb**
  - More precise errors when used from within **cuda-gdb**
  - Must be activated before the application is launched

```
(cuda-gdb) set cuda memcheck on
```

- What does it mean “more precise”?
  - Precise
    - Exact thread idx
    - Exact PC
  - Not precise
    - A group of threads or blocks
    - The PC is several instructions after the offending load/store

# Example

```
(cuda-gdb) set cuda memcheck on  
  
(cuda-gdb) run  
[Launch of CUDA Kernel 0 (applyStencil1D) on Device 0]  
Program received signal CUDA_EXCEPTION_1, Lane Illegal Address.  
applyStencil1D<<<(32768,1,1),(512,1,1)>>> at stencil1d.cu:60  
  
(cuda-gdb) info line stencil1d.cu:60  
out[ i ] += weights[ j + RADIUS ] * in[ i + j ];
```

# Stepping: from where, and how

- Single-stepping
  - Every exception is automatically precise
- The “**autostep**” command
  - Define a window of instructions where we think the offending load/store occurs
  - **cuda-gdb** will single-step all the instructions within that window automatically and without user intervention

```
(cuda-gdb) autostep foo.cu:25 for 20 lines
```

```
(cuda-gdb) autostep *$pc for 20 instructions
```

## Tips & Miscellaneous Notes

# Best Practices

## 1. Determine scope of the bug

- Incorrect result
- Unspecified Launch Failure (ULF)
- Crash
- Hang
- Slow execution

## 2. Reproduce with a debug build

- Compile your app with `-g -G`
- Rerun, hopefully you can reproduce problem in debug model

# Best Practices

## 3. Correctness Issues

- First throw **cuda-memcheck** at it in stand-alone
- Then **cuda-gdb** and **memcheck** if needed
- **printf** is also an option, but not recommended

## 4. Performance Issues (once no bugs evident)

- Use a profiler (discussed later)
- Change the code, might have to go back to Step 1. above...

# Tips

- Always check the return code of the CUDA API routines!
- If you use **printf** from the device code...
  - Make sure to synchronize so that buffers are flushed

# Tips

- To hide devices, launch the application with

```
CUDA_VISIBLE_DEVICES=0,3
```

where the numbers are device indexes.

- To increase determinism, launch the kernels synchronously:

```
CUDA_LAUNCH_BLOCKING=1
```

# Tips

- To print multiple consecutive elements in an array, use @:

```
(cuda-gdb) print array[3]@4
```

- To find the mangled name of a function

```
(cuda-gdb) set demangle-style none  
(cuda-gdb) info function my_function_name
```

# Profiling GPU CUDA code

# Code Timing/Profiling

- **Approach 1:** command-line profiler **nvprof**
  - Collect timeline of CPU and GPU activities
  - Headless profile collection
    - Use **nvprof** on headless node to collect data
    - Visualize timeline with Visual Profiler (see below)
- **Approach 2:** use NVIDIA's **nvvvp** Visual Profiler
  - Visualize CPU and GPU activity
  - Identify optimization opportunities
  - Allows for automated analysis
  - **nvvvp** is a cross platform tool (linux, mac, windows)

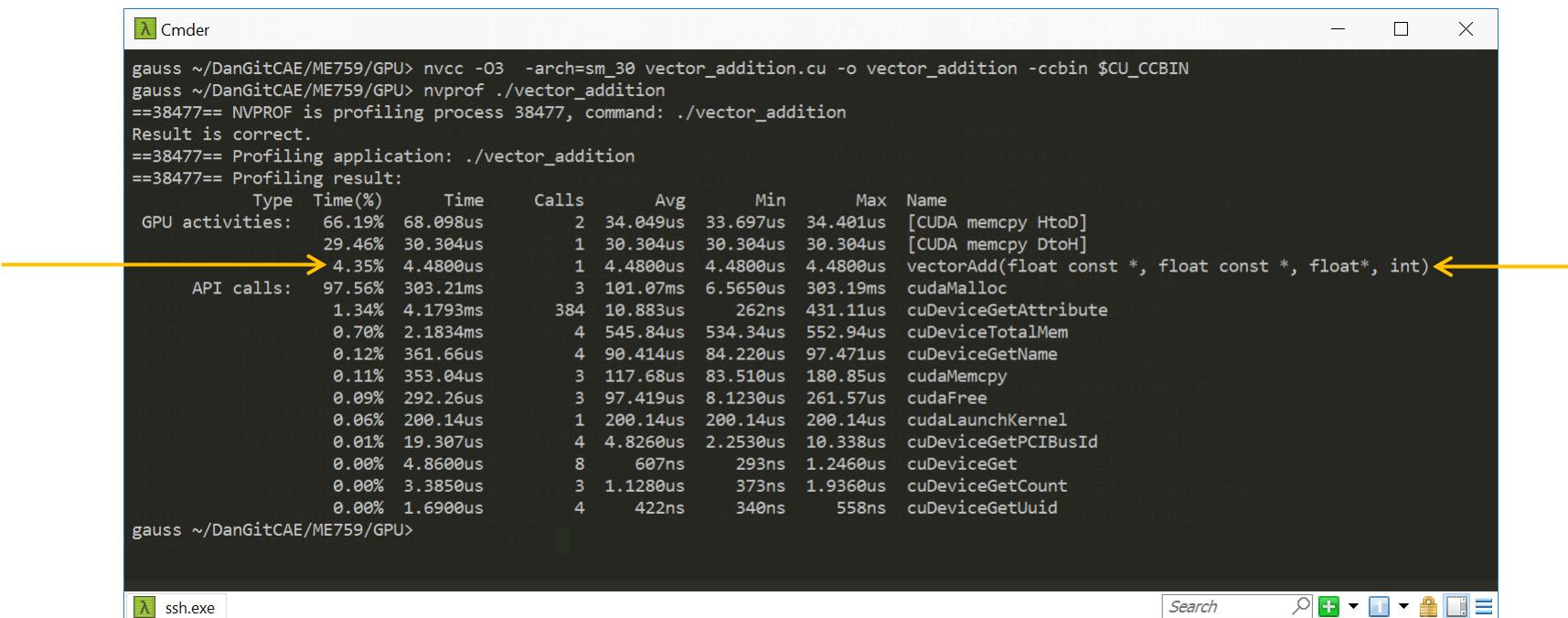
# nvprof Usage

```
$ nvprof [nvprof_args] <appName> [app_args]
```

- On Euler, don't forget to use `sbatch` (that is, rely on Slurm)
- Help on usage and arguments:  
`$ nvprof --help`
- Save profile to file:  
`$ nvprof -o profile.out <appName> [app_args]`
- Import into Visual Profiler (`nvvvp`):
  - File menu -> Import nvprof Profile
  - NOTE: Not an option on Euler
- Import into `nvprof` to generate reports:  
`$ nvprof -i profile.out`  
`$ nvprof -I profile.out --print-gpu-trace`

# nvprof – GPU Summary

```
$ nvprof vector_addition
```



```
gauss ~/DanGitCAE/ME759/GPU> nvcc -O3 -arch=sm_30 vector_addition.cu -o vector_addition -ccbin $CU_CCBIN
gauss ~/DanGitCAE/ME759/GPU> nvprof ./vector_addition
==38477== NVPROF is profiling process 38477, command: ./vector_addition
Result is correct.
==38477== Profiling application: ./vector_addition
==38477== Profiling result:
          Type  Time(%)     Time    Calls      Avg       Min       Max  Name
GPU activities:  66.19%  68.098us      2  34.049us  33.697us  34.401us  [CUDA memcpy HtoD]
                  29.46%  30.304us      1  30.304us  30.304us  30.304us  [CUDA memcpy DtoH]
                  4.35%  4.4800us      1  4.4800us  4.4800us  4.4800us  vectorAdd(float *, float const *, float*, int) ←
API calls:    97.56%  303.21ms      384  10.883us   262ns  431.11us  cuDeviceGetAttribute
              1.34%  4.1793ms      384  10.883us   262ns  431.11us  cuDeviceGetName
              0.70%  2.1834ms      4   545.84us  534.34us  552.94us  cuDeviceTotalMem
              0.12%  361.66us      4   90.414us  84.220us  97.471us  cuDeviceGetName
              0.11%  353.04us      3   117.68us  83.510us  180.85us  cudaMemcpy
              0.09%  292.26us      3   97.419us  8.1230us  261.57us  cudaFree
              0.06%  200.14us      1   200.14us  200.14us  200.14us  cudaLaunchKernel
              0.01%  19.307us      4   4.8260us  2.2530us  10.338us  cuDeviceGetPCIBusId
              0.00%  4.8600us      8   607ns   293ns  1.2460us  cuDeviceGet
              0.00%  3.3850us      3   1.1280us  373ns  1.9360us  cuDeviceGetCount
              0.00%  1.6900us      4   422ns   340ns  558ns  cuDeviceGetUuid
gauss ~/DanGitCAE/ME759/GPU>
```

- Generate CSV output

```
$ nvprof -csv vector_addition
```

# nvprof – GPU Trace

```
$ nvprof --print-gpu-trace vector_addition
```

```
A Cmder
gauss ~/DanGitCAE/ME759/GPU
gauss ~/DanGitCAE/ME759/GPU> nvprof --print-gpu-trace ./vector_addition
==38576== NVPROF is profiling process 38576, command: ./vector_addition
Result is correct.
==38576== Profiling application: ./vector_addition
==38576== Profiling result:
Start Duration Grid Size Block Size Regs* SSMem* DSMem* Size Throughput SrcMemType DstMemType Device Context Stream Name
587.08ms 34.336us - - - - 195.31KB 5.4248GB/s Pageable Device Tesla K20Xm (0) 1 7 [CUDA memcpy HtoD]
587.16ms 33.696us - - - - 195.31KB 5.5278GB/s Pageable Device Tesla K20Xm (0) 1 7 [CUDA memcpy HtoD]
587.37ms 4.5440us (196 1 1) (256 1 1) 8 0B 0B - - - Tesla K20Xm (0) 1 7 vectorAdd(float const *, float const *, float*, int) [417]
587.39ms 30.529us - - - - 195.31KB 6.1012GB/s Device Pageable Tesla K20Xm (0) 1 7 [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy
gauss ~/DanGitCAE/ME759/GPU>
gauss ~/DanGitCAE/ME759/GPU>
```

↓

```
>> nvcc -O3 -arch=sm_30 vector_addition.cu -o vector_addition -ccbin $CU_CCBIN
```

```
A ssh.exe | Search | 
A Cmder
gauss ~/DanGitCAE/ME759/GPU> nvcc -G -arch=sm_30 vector_addition.cu -o vector_addition -ccbin $CU_CCBIN
gauss ~/DanGitCAE/ME759/GPU> nvprof --print-gpu-trace ./vector_addition
==11045== NVPROF is profiling process 11045, command: ./vector_addition
Result is correct.
==11045== Profiling application: ./vector_addition
==11045== Profiling result:
Start Duration Grid Size Block Size Regs* SSMem* DSMem* Size Throughput SrcMemType DstMemType Device Context Stream Name
594.65ms 34.784us - - - - 195.31KB 5.3549GB/s Pageable Device Tesla K20Xm (0) 1 7 [CUDA memcpy HtoD]
594.73ms 33.728us - - - - 195.31KB 5.5225GB/s Pageable Device Tesla K20Xm (0) 1 7 [CUDA memcpy HtoD]
594.93ms 10.592us (196 1 1) (256 1 1) 12 0B 0B - - - Tesla K20Xm (0) 1 7 vectorAdd(float const *, float const *, float*, int) [417]
594.95ms 30.496us - - - - 195.31KB 6.1078GB/s Device Pageable Tesla K20Xm (0) 1 7 [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy
gauss ~/DanGitCAE/ME759/GPU> |
```

↓

```
>> nvcc -G -arch=sm_30 vector_addition.cu -o vector_addition -ccbin $CU_CCBIN
```

# nvprof – Powerful for getting insights into what happens on the host as well

The screenshot shows the CUDA Toolkit Documentation for nvprof. The left sidebar contains a navigation tree with sections like CUDA Toolkit v10.1.105, Profiler, Visual Profiler, Customizing the Profiler, Command Line Arguments, nvprof, and Metrics Reference. The main content area displays two tables of command-line options.

**Table 1: nvprof Options**

nvprof			
cpu-profiling			invocation. One or more of these may be specified, separated by commas: <ul style="list-style-type: none"><li>• global_access: global access</li><li>• shared_access: shared access</li><li>• branch: divergent branch</li><li>• instruction_execution: instruction execution</li><li>• pc_sampling: pc sampling, available only for GM20X+</li></ul> Note: Use <code>--export-profile</code> to specify an export file. See <a href="#">Source-Diassembly View</a> for more information.
system-profiling	on, off	off	Turn on/off power, clock, and thermal profiling. See <a href="#">System Profiling</a> for more information.
timeout (t)	{seconds}	N/A	Set an execution timeout (in seconds) for the CUDA application. Note: Timeout starts counting from the moment the CUDA driver is initialized. If the application doesn't call any CUDA APIs, timeout won't be triggered. See <a href="#">Timeout</a> and <a href="#">Flush Profile Data</a> for more information.
track-memory-allocations	on, off	off	Turn on/off tracking of memory operations, which involves recording timestamps, memory size, memory type and program counters of the memory allocations and frees. Turning this option on may incur an overhead during profiling.
unified-memory-profiling	per-process-device, off	per-process-device	Configure unified memory profiling. <ul style="list-style-type: none"><li>• per-process-device: collect counts for each process and each device</li><li>• off: turn off unified memory profiling</li></ul> See <a href="#">Unified Memory Profiling</a> for more information.

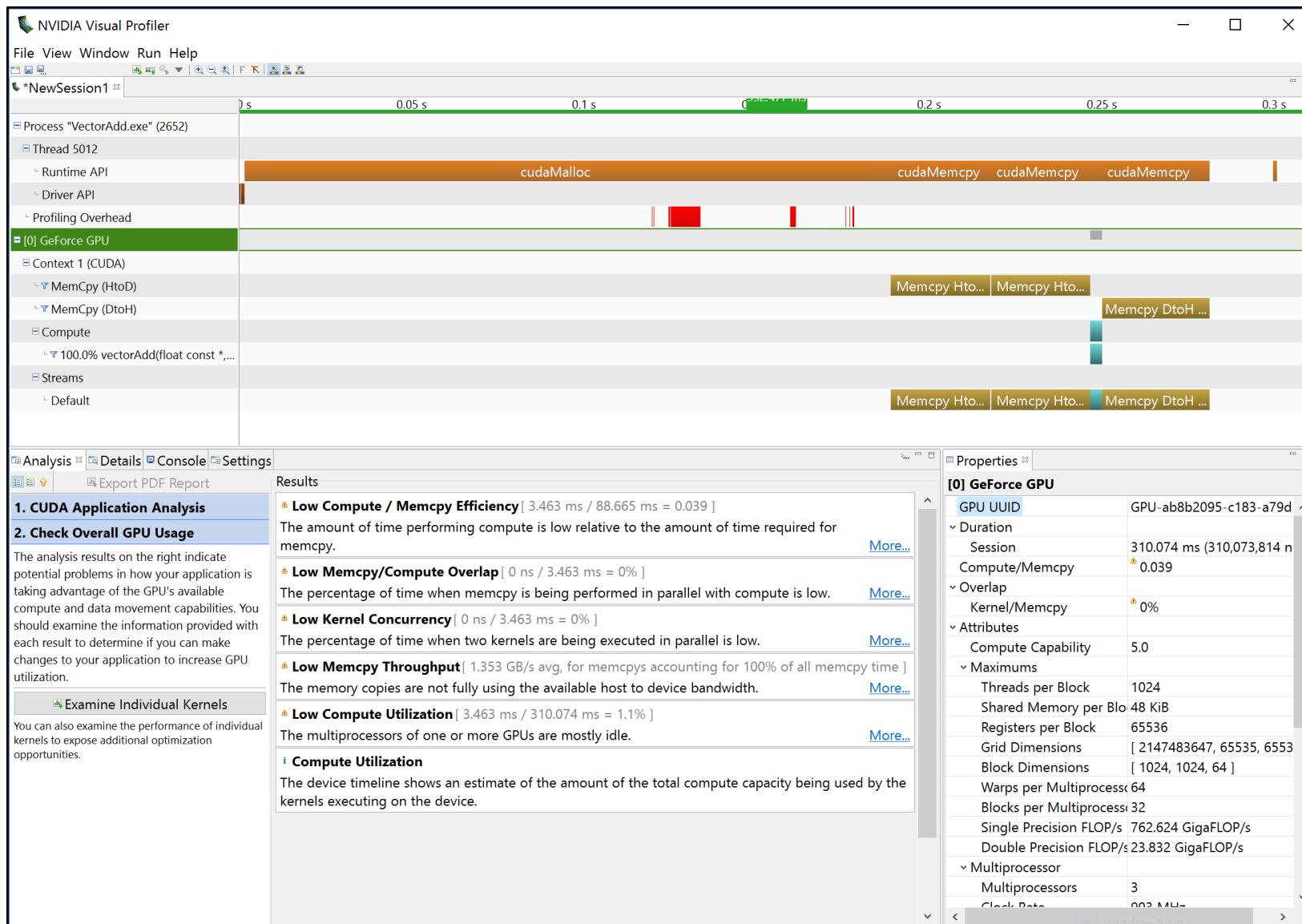
**Table 2: CPU Profiling Options**

Option	Values	Default	Description
cpu-profiling	on, off	off	Turn on CPU profiling. Note: CPU profiling is not supported in multi-process mode.
cpu-profiling-explain-ccff	{filename}	N/A	Set the path to a PGI pexplain.xml file that should be used to interpret Common Compiler Feedback Format (CCFF) messages.
cpu-profiling-frequency	{frequency}	100Hz	Set the CPU profiling frequency in samples per second. Maximum is 500Hz.
cpu-profiling-max-depth	{depth}	0 (i.e. unlimited)	Set the maximum depth of each call stack.
cpu-profiling-mode	flat, top-down, bottom-up	bottom-up	Set the output mode of CPU profiling. <ul style="list-style-type: none"><li>• flat: Show flat profile</li><li>• top-down: Show parent functions at the top</li><li>• bottom-up: Show parent functions at the bottom</li></ul>
cpu-profiling-percentage-threshold	{threshold}	0 (i.e. unlimited)	Filter out the entries that are below the set percentage threshold. The limit should be an integer between 0 and 100, inclusive.
cpu-profiling-scope	function, instruction	function	Choose the profiling scope. <ul style="list-style-type: none"><li>• function: Each level in the stack trace represents a distinct function</li><li>• instruction: Each level in the stack trace represents a distinct instruction address</li></ul>
cpu-profiling-show-ccff	on, off	off	Choose whether to print Common Compiler Feedback Format (CCFF) messages embedded in the binary. Note: this option implies <code>--cpu-profiling-scope</code> instruction.
cpu-profiling-show-library	on, off	off	Choose whether to print the library name for each sample.
cpu-profiling-thread-mode	separated, aggregated	aggregated	Set the thread mode of CPU profiling. <ul style="list-style-type: none"><li>• separated: Show separate profile for each thread</li><li>• aggregated: Aggregate data from all threads</li></ul>
cpu-profiling-unwind-stack	on, off	on	Choose whether to unwind the CPU call-stack at each sample point.
openacc-profiling	on, off	on	Enable/disable recording information from the OpenACC profiling interface. Note: if the OpenACC profiling interface is available depends on the OpenACC runtime. See <a href="#">OpenACC</a> for more information.
openmp-profiling	on, off	on	Enable/disable recording information from the OpenMP profiling interface. Note: if the OpenMP profiling interface is available depends on the OpenMP runtime. See <a href="#">OpenMP</a> for more information.

**Table 3: Print Options**

Option	Values	Default	Description
context-name	{name}	N/A	Name of the CUDA context.

# nvvp: NVIDIA Visual Profiler



# Nsight: Visual Studio Edition

VectorAdd - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM NSIGHT TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP

Local Windows Debugger Release Win32

Activity1.nvact\* kernel.cu

CUDA Launches Hierarchy Flat

Filter

Function Name	Grid Dimensions	Block Dimensions	Start Time (μs)	Duration (μs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)	Dynamic Shared Memory per Block (bytes)	Cache Configuration Executed	Global Caching Requested	Global Caching Executed	Local Memory per Thread (bytes)	Device Name
vectorAdd	{39063, 1, 1}	{256, 1, 1}	1,437,934.782	3,432.448	100.00 %	8	0	0	PREFER_SHARED	N/A	N/A	0	GeForce GPU

vectorAdd<<39063,256>> [CUDA Launch]

- Device Launches
- Call Graph
- vectorAdd [CUDA Kernel]
- Experiment Results
  - Occupancy
  - Instruction Statistics
  - Branch Statistics
  - Issue Efficiency
  - Achieved FLOPS
  - Achieved IOPS
  - Pipe Utilization
  - Memory Statistics
  - Source Profiler
    - Instruction Count
    - Divergent Branch
    - Memory Transactions

Varying Block Size

Warp per SM

Threads Per Block

Varying Register Count

Warp per SM

Registers Per Thread

Varying Shared Memory Usage

Warp per SM

Shared Memory Per Block

Achieved Occupancy

Occupancy

SM

Output Error List Find Results 1

Ready

# The CUDA Profiling Tools Interface (CUPTI)

- CUPTI: Enables the creation of profiling and tracing tools that target CUDA applications
- CUPTI provides the following APIs:
  - The Activity API
  - The Callback API
  - The Event API
  - The Metric API
  - The Profiler API
- Using APIs above, one can develop sharp & customized profiling tools
  - Can provide very detailed insights into the CPU and GPU behavior of CUDA applications
- CUPTI delivered as a dynamic library on all platforms supported by CUDA

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 17

03/02/2020

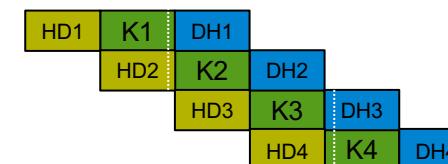
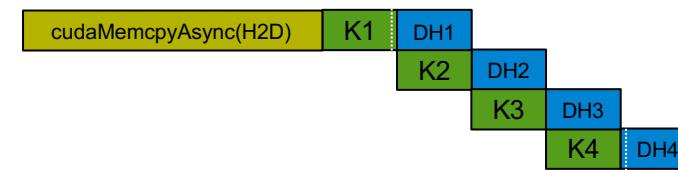
# Quote of the day

“The happiness of your life depends upon the quality of your thoughts.”

-- Marcus Aurelius, Roman Emperor [121 – 180 AD]

# Before we get going...

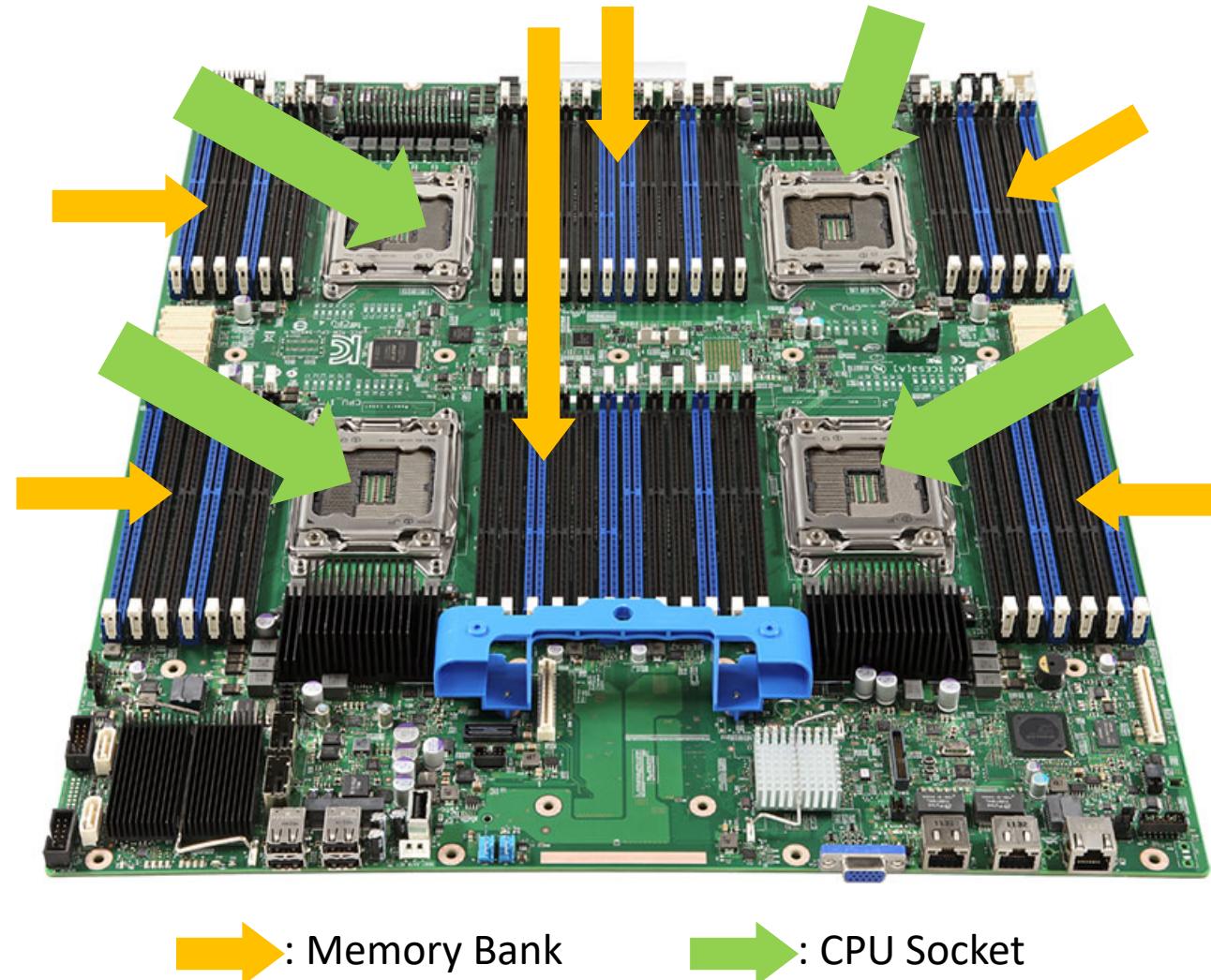
- Last time:
  - CUDA streams, wrapping up
  - Tools of the trade: GPU debugging & profiling
- Today:
  - Advanced GPU memory issues
    - Zero-copy memory
    - Unified Virtual Addressing
    - Managed memory
- Other tidbits:
  - Assignment due on Thursday at 9 pm
  - One more “GPU computing” lecture
  - Finishing teaching early – trip to Italy cancelled
    - Last lecture (the 28<sup>th</sup>), on April 3
  - April 15, at 7:15 PM



# Unified Memory (Managed Memory) in CUDA

# Multi-Socket Configurations

[four sockets: Intel S4600LT2 Xeon E5-4600 Chipset-C600-A Socket-R LGA-2011 1.46Tb DDR3-1600MHz Server Motherboard]



# The underlying theme of today's lecture

- Helpful to not think of the physical memory as one contiguous thing
- Rather, an amalgamation of bits and pieces
- Another way to look at it: regard it as a “pool of physical memory”
- Why: today the pool of physical memory includes the memory of the GPU[s]

# Finding the needle in the haystack

- If the pool includes all the memory of the system, how do you find what you need?
- Rely on the translation table & the concept of virtual memory
- Use 49 instead of 48 bits to represent addresses in this virtual memory
- NOTE: although the OS is 64 bits, modern systems typically use only 48 bits for addresses
  - 64 bits would index 18,446,744,073,709,551,615 bytes – way more than what we see in reality

# Virtual vs. Physical Memory [old slide]

- **Virtual memory**: dally and quaint fictitious space of  $2^{32}$  addresses (on 32 bit architectures) in which a process sees its **data** being placed, the **instructions** stored, etc.
  - Correction: you have 49 bits instead of 32
    - You can index into 562,949,953,421,311 bytes (about 562 Terabytes) – hundreds of times more than what in use on a system nowadays
- **Physical memory**: a busy [perhaps heterogeneous] place that hosts at the same time **data** and **instructions** associated with many applications simultaneously running on the system

# Virtual Memory: The Page Table [old slide]

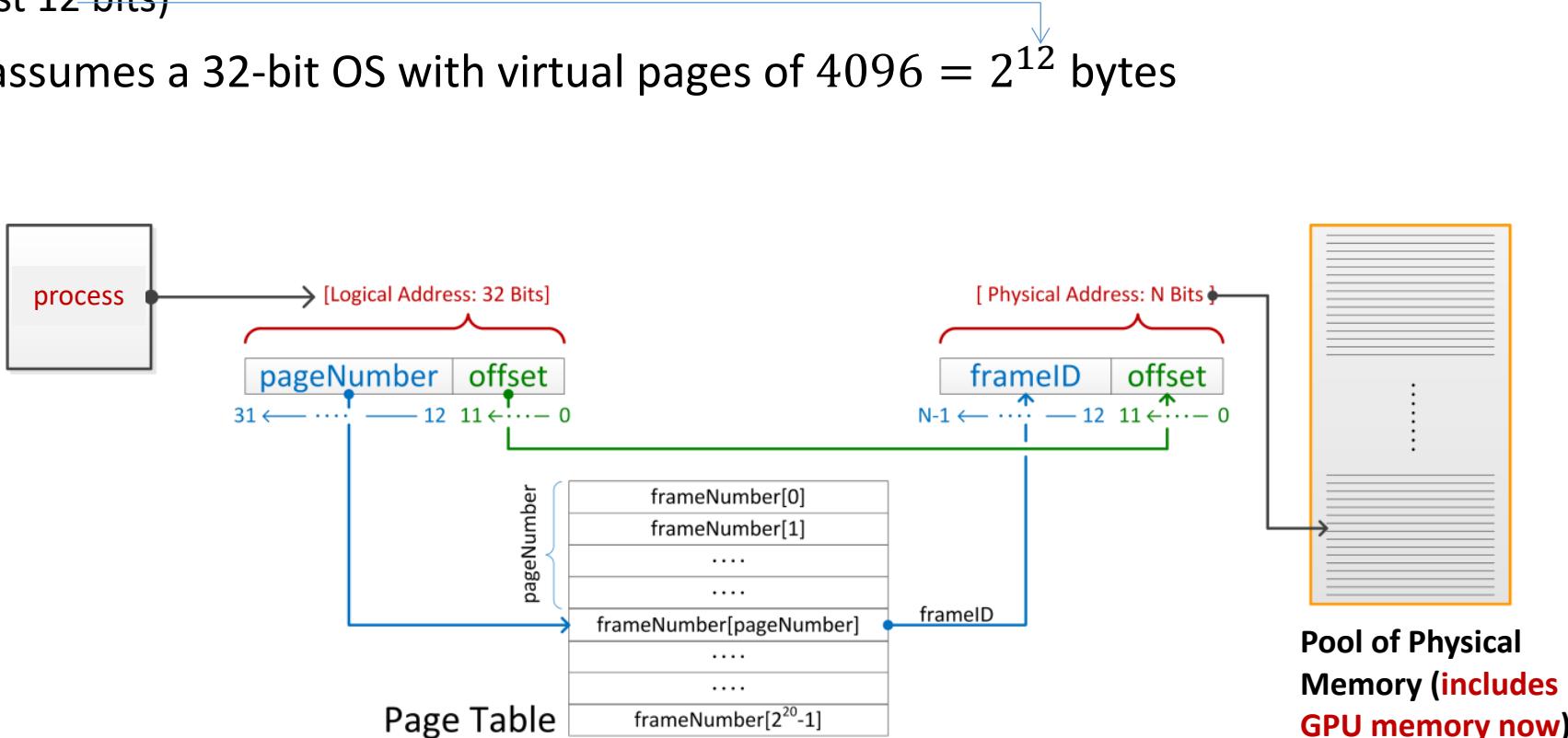
- Virtual memory allows the processor to work in a virtual world in which a process seems to have exclusive access to a very large memory space
- This virtual world is connected back to, or mapped back into, the physical memory through a Page Table
  - This is one level of indirection

# The UVA

- Key observation: relatively recently, the GPU and CPU decided to share together one virtual memory space
  - UVA: Unified Virtual Address
  - UVAS: Unified Virtual Address Space
- The trick is to translate an address from this 49-bit UVAS into the pool of physical memory

# The Page Table & The Translation Process

- A virtual address has two parts (address butchering):
  - The page number
  - The offset (last 12 bits)
- Example below assumes a 32-bit OS with virtual pages of  $4096 = 2^{12}$  bytes

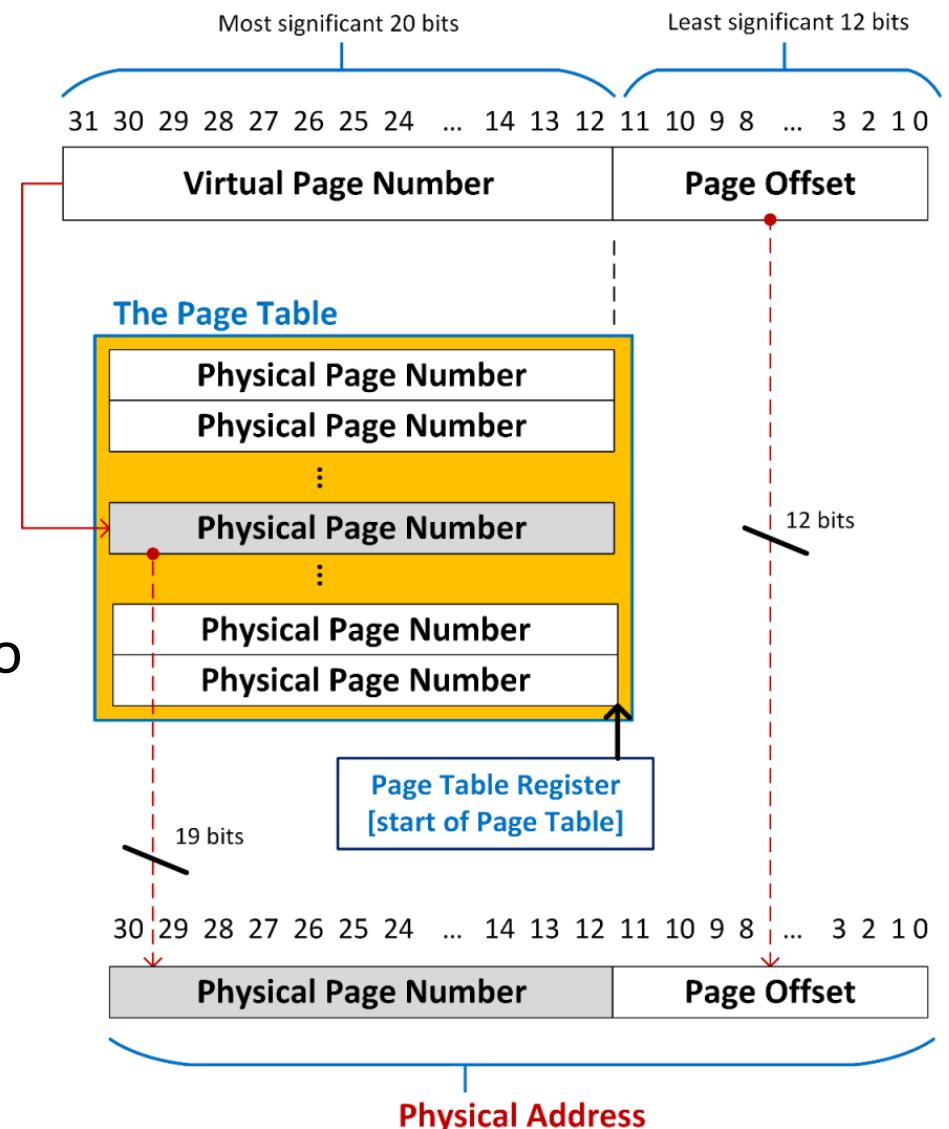


# Anatomy of a Virtual Memory Address

- A **page of virtual memory** corresponds to a **frame of physical memory**
- The size of a page (or frame, for that matter) is typically 4096 bytes
  - Compare to 64 bytes, the size of a cache block (line)
- $2^{12} = 4096$ : 12 bits of the address are sufficient to relatively position each byte in a page
  - Bits marked 0 through 11 on the previous slide

# The Translation Process

- Example: imagine that your physical memory is 2 GB
- The physical address has 31 bits:  $2^{31}=2\text{GB}$
- Then the page table converts the bits 12 through 31 or through 30 of the physical address



# Putting things in perspective, today's presentation

- Premise: Managing and optimizing host↔device data transfers a bit of a nuisance
- Punch line: Unified Memory (UM) support, available as of CUDA 6, simplifies this task
- Our goals today:
  - Briefly review history of CUDA host/device memory management
  - Explain how UM makes CUDA low-level programming more palatable

# Good to keep in mind

- In this Unified Memory discussion:
  - We are not going to learn anything that we couldn't live without
  - Discussion focus: how GPU programming became more friendly/convenient in relation to data movement
    - A matter of improved productivity
- Remark: A byproduct of this discussion is a better understanding of how CUDA is evolving
  - Happens as we speak...

# Backdrop, cudaMemcpy

- cudaMemcpy: A staple of CUDA, available in release 1.0
- The drill
  - Get some memory space in device memory (cudaMalloc call)
  - Data transferred from host memory into device memory with cudaMemcpy
  - Data processed on the device by invoking a kernel
  - Results transferred from device memory into host memory with cudaMemcpy
  - Free the cudaMalloc-ed memory on the device
  - Kick back and relax (reach for a strong one?)

# Backdrop, cudaMemcpy

- Memory allocated on the host with `malloc`
- Memory allocated on the device using the CUDA runtime function `cudaMalloc`
- Data moved back-and-forth, host to/from device, over the PCI-E pipe

# The PCI-E Pipe, Putting Things in Perspective

- PCIe x 16: peak bandwidth (what's reasonable to expect in parentheses) (per direction data)
  - V1: 4.0 (3) GB/s
  - V2: 8.0 (6) GB/s
  - V3: 15.8 (12) GB/s
  - V4: 31.5 (???) GB/s
- Bandwidths above relatively small, see for instance (values provided are middle of the road)
  - Host memory bus: about 25 – 50 GB/s per socket
  - GPU global memory bandwidth: 400 – 900 GB/s

# cudaHostAlloc: helps improve host/device transfer speeds

- Host/Device data **transfer speeds** could be improved if host memory was **not pageable**
- Therefore, rather than `malloc`-ing, allocate host memory using CUDA's **cudaHostAlloc()**
- Nothing magic, data still moves back-and-forth through same PCI-E pipe albeit at a faster clip

# cudaHostAlloc: A friend, with its pluses and **minuses**

- cudaHostAlloc cons
  - cudaHostAlloc-ing **large amounts of memory** can negatively impact overall system performance
    - Why? It reduces the amount of system memory available for paging
    - How much is too much? Not clear, dependent on the system and the applications running on the machine
  - cudaHostAlloc **memory allocation speed** is small - ballpark 5 GB/s
    - Allocating 5 GB of memory time-wise comparable to moving that much memory over the PCI-E bus

# cudaHostAlloc: A friend, with its pluses and minuses

1. Enables faster device ↔ host, back-and-forth transfers
2. Enables the use of asynchronous memory transfer and kernel execution (topic covered before)
  - Draws on the concept of CUDA stream
  - “in/out data copy” and “execution” engines working at the same time
3. Enables mapping of the host pinned memory into the memory space of the device
  - Device now capable to access data on host while executing a kernel or other device function

# cudaHostAlloc: Function Prototype

```
cudaError_t cudaHostAlloc(void** pHst, size_t sz, unsigned int flag);
```

- Last argument (“flag”) controls the magic
- “flag” values: cudaHostAllocPortable, cudaHostAllocWriteCombined, etc.
- The “flag” of most interest is “cudaHostAllocMapped”
  - Maps the memory allocated on the host in the memory space of the device for **direct access**

# Zero-Copy (Z-C) GPU-CPU Interaction

- What's gained if you pin host memory via `cudaHostAlloc`?
  - The ability to access a piece of data from pinned and mapped host memory by a thread running on the GPU without a CUDA runtime copy call to explicitly move data onto the GPU
    - This is called zero-copy (Z-C) GPU-CPU interaction, from where the name “zero-copy memory”
    - Calling `cudaMemcpy` **not a must** anymore
    - Note: data is still read/written through the PCI-E pipe; process managed by the runtime, in a transparent fashion
- The device memory ballooned, virtually, to include chunk that physically belongs to the host

# From Z-C to UVA: CUDA 2.2 to CUDA 4.0

- Pain in the rear: Z-C enabled access of data on the host from the device required one additional runtime call to `cudaHostGetDevicePointer()`
  - `cudaHostGetDevicePointer()`: given a pointer to pinned host memory produces a new pointer that can be invoked within the kernel to access data stored on the host
- The need for the `cudaHostGetDevicePointer()` call was eliminated in CUDA 4.0 with the introduction of the Unified Virtual Addressing (UVA) mechanism

# Unified Virtual Address Space: From CUDA programming guide

- All host memory allocations made via CUDA API calls and all device memory allocations on supported devices are within the UVAS. As a consequence:
  - Today, the location of any memory on any of the devices or on the host allocated through CUDA can be determined from the value of the pointer using `cudaPointerGetAttributes()`
  - When copying to or from the memory of any device the `cudaMemcpyKind` parameter of `cudaMemcpy*`() can be set to `cudaMemcpyDefault` to determine locations from the pointers
  - Allocations via `cudaHostAlloc()` are automatically portable across all the devices
  - Pointers returned by `cudaHostAlloc()` can be used directly from within kernels running on a device

# UVA – Showcasing Its Versatility...

- Set of commands below can be issued by one host thread to multiple devices
  - No need to use anything beyond cudaMemcpyDefault

```
cudaMemcpy(gpu1Dst_memPntr, host_memPntr, byteSize1, cudaMemcpyDefault)
cudaMemcpy(gpu2Dst_memPntr, host_memPntr, byteSize2, cudaMemcpyDefault)
cudaMemcpy(host_memPntr, gpu1Dst_memPntr, byteSize1, cudaMemcpyDefault)
cudaMemcpy(host_memPntr, gpu2Dst_memPntr, byteSize2, cudaMemcpyDefault)
```
- UVA support is the enabler for the peer-to-peer (P2P), inter-GPU, data transfer
  - P2P not discussed here
  - UVA is the underpinning technology for P2P

# UVA is a Step Forward Relative to Z-C

- Z-C Key Accomplishment: use pointer within **device function** access **host data**
  - Z-C focused on a data access, an issue relevant in the context of functions executed on the device
- UVA had a data access component but also a data transfer component:
  - Data access: A GPU could access data on a different GPU, a novelty back in CUDA 4.0
  - Data transfer: copy data in between GPUs
    - `cudaMemcpy` is the main character in this play, data transfer initiated on the host side

# UVA & Unified Memory (UM)

- The Unified Virtual Address idea had ok impact, perhaps good enough to write home about
- **Unified Memory (UM)** took this idea to town by allowing the CPU to tap into GPU memory
  - UM works in conjunction with a “managed memory pool”
- Fallout: improved significantly CUDA ease of use, particularly for multi-GPU systems

# Unified Memory, a first example

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 8

__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    int *ret;
    cudaMalloc(&ret, SZ * sizeof(int));
    AplusB<<<1, SZ>>>(ret, 10, 100);
    int *host_ret = (int *)malloc(SZ * sizeof(int));
    cudaMemcpy(host_ret, ret, SZ * sizeof(int), cudaMemcpyDefault);
    for (int i = 0; i < SZ; i++)
        printf("%d: A+B = %d\n", i, host_ret[i]);
    free(host_ret);
    cudaFree(ret);
    return 0;
}
```

12 lines of code

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 8

__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    int *ret;
    cudaMallocManaged(&ret, SZ * sizeof(int));
    AplusB<<<1, SZ>>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for (int i = 0; i < SZ; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return 0;
}
```

10 lines of code

```
TRUMAN Debug> ./unifMemCUDAexPart1.exe
0: A+B = 110
1: A+B = 111
2: A+B = 112
3: A+B = 113
4: A+B = 114
5: A+B = 115
6: A+B = 116
7: A+B = 117
```

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 8

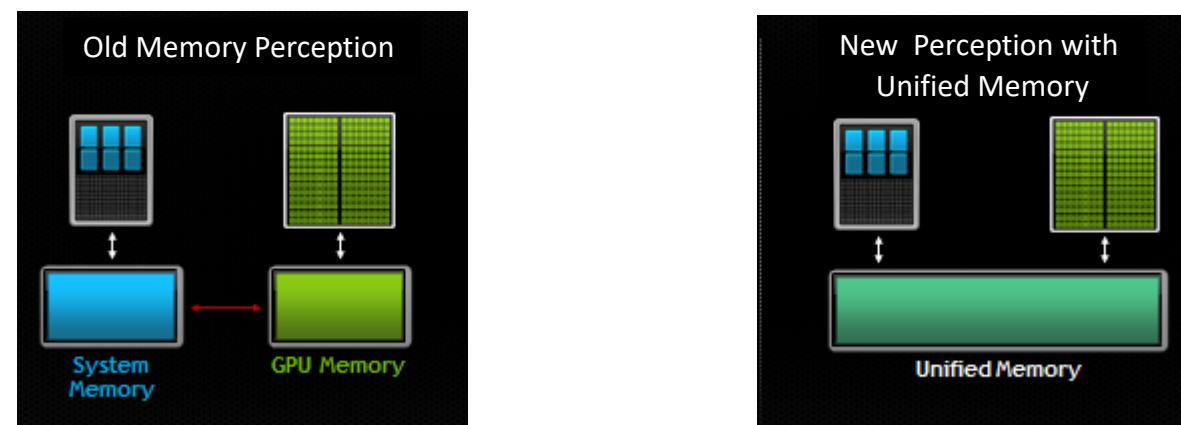
__device__ __managed__ int ret[SZ];
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    AplusB<<<1, SZ>>>(10, 100);
    cudaDeviceSynchronize();
    for (int i = 0; i < SZ; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

8 lines of code  
(can make it 7)

# Unified Memory: further thoughts

- One memory allocation call takes care of memory setup at both ends; i.e., device and host
  - The main actor: the CUDA runtime function `cudaMallocManaged()`
- New way of perceiving the memory interplay in GPGPU computing
  - No distinction is made between memory on the host and memory on the device
  - It's just **a pool of memory**, albeit with different access times when accessed by different processors
    - "processor": any independent execution unit with a dedicated memory management unit (MMU)
      - Includes both CPUs and GPUs of any type and architecture



# The UM innovation

- No longer any need for explicit memory transfers between host and device
  - Any allocation created in **the managed memory space** automatically migrated to where it is needed
- Also, UM eliminates the need to call the `cudaMalloc/cudaHostAlloc` duo
  - It takes a different perspective on handling memory in the GPU/CPU interplay

# UM - Other interesting tidbits

- Reliance on a managed memory space in which all processors see a single coherent memory image with a common address space
- The runtime manages data access and locality within a CUDA program without need for explicit memory copy calls
- Memory is migrated where it is used, the page table of the processors is simply updated
- One can now oversubscribe the amount of memory on the device
  - Allocate more than available; might spill into host memory

# Then and Now: no-UM vs. UM

```
#include "math.h"
#include <iostream>

const int ARRAY_SIZE = 1000;
using namespace std;

__global__ void increment(double *aArray, double val, unsigned int sz) {
    unsigned int indx = blockIdx.x * blockDim.x + threadIdx.x;
    if (indx < sz)
        aArray[indx] += val;
}

int main(int argc, char **argv) {
    double *hA;
    double *dA;
    hA = (double *)malloc(ARRAY_SIZE * sizeof(double));
    cudaMalloc(&dA, ARRAY_SIZE * sizeof(double));

    for (int i = 0; i < ARRAY_SIZE; i++)
        hA[i] = 1. * i;

    double inc_val = 2.0;
    cudaMemcpy(dA, hA, sizeof(double) * ARRAY_SIZE, cudaMemcpyHostToDevice);
    increment<<<2, 512>>>(dA, inc_val, ARRAY_SIZE);
    cudaMemcpy(hA, dA, sizeof(double) * ARRAY_SIZE, cudaMemcpyDeviceToHost);

    double error = 0.;
    for (int i = 0; i < ARRAY_SIZE; i++)
        error += fabs(hA[i] - (i + inc_val));

    cout << "Test: " << (error < 1.E-9 ? "Passed" : "Failed") << endl;

    cudaFree(dA);
    free(hA);
    return 0;
}
```

```
#include "math.h"
#include <iostream>

const int ARRAY_SIZE = 1000;
using namespace std;

__global__ void increment(double *aArray, double val, unsigned int sz) {
    unsigned int indx = blockIdx.x * blockDim.x + threadIdx.x;
    if (indx < sz)
        aArray[indx] += val;
}

int main(int argc, char **argv) {
    double *mA;
    cudaMallocManaged(&mA, ARRAY_SIZE * sizeof(double));

    for (int i = 0; i < ARRAY_SIZE; i++)
        mA[i] = 1. * i;

    double inc_val = 2.0;
    increment<<<2, 512>>>(mA, inc_val, ARRAY_SIZE);
    cudaDeviceSynchronize();

    double error = 0.;
    for (int i = 0; i < ARRAY_SIZE; i++)
        error += fabs(mA[i] - (i + inc_val));

    cout << "Test: " << (error < 1.E-9 ? "Passed" : "Failed") << endl;

    cudaFree(mA);
    return 0;
}
```

# Requirements, UM

- GPU with SM architecture 3.0 or higher (Kepler class or newer)
- A 64-bit host application and non-embedded operating system (Linux, Windows, macOS)
- CUDA 6.0 or higher
- NOTE: UM is very much work in progress, not all features supported on Windows or MacOS
  - Linux kernel has good support for UM

# UM vs. Z-C

- Recall that with Z-C, data is always on the host in pinned CPU system memory
  - The device can update data on the host (if host memory pinned), but not vice-versa
- UM: data stored on the device but migrated where needed
  - Data access and locality managed by CUDA runtime, handling transparent to the user
  - UM provides “single-pointer-to-data” model
  - The host can finally **modify** data that is stored on the device
    - Before, the host could drop data in device memory, but not directly modify it

# UM: Required API Changes

- As far as the CUDA API is concerned, UM led to **\*three\*** new family members :
  - `cudaMallocManaged`
  - `__managed__`
  - `cudaStreamAttachMemAsync()`
- There is a lot more that doesn't meet the eye. Very tough nut to crack, managed memory

# cudaMallocManaged

- Function Prototype:

```
cudaError_t cudaMallocManaged(void** devPtr, size_t size, unsigned int flag)
```

- A drop-in replacement for `cudaMalloc()` – they are semantically identical
- Allocates managed memory on the device
- First two arguments have the expected meaning
- Caveats:
  - `devPtr`: pointer accessible from both Host and Device
  - `flag` controls the default stream association for this allocation
    - `cudaMemAttachGlobal` - memory is accessible from any stream on any device
    - `cudaMemAttachHost` – memory on this device accessible by host only
- Free memory with the same `cudaFree()`

# managed

- Global/file-scope variable annotation combines with \_\_device\_\_
- Declares global-scope migrate-able device variable
- Symbol accessible from both GPU and CPU code

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 8

__device__ __managed__ int ret[SZ];
__global__ void AplusB(int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    AplusB<<<1, SZ>>>(10, 100);
    cudaDeviceSynchronize();
    for (int i = 0; i < SZ; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

# cudaStreamAttachMemAsync()

- Manages concurrency in multi-threaded CPU applications
- Not relevant here

# UM, other quick points

- Managed memory is **interoperable and interchangeable** with device-specific allocations, such as those created using the `cudaMalloc()` routine
  - Translation:
    - `cudaMallocManaged()` – allocated memory behaves in all situations like memory allocated with `cudaMalloc()`
    - But not vice-versa
- All CUDA operations that are valid on device memory are also valid on managed memory

# UM, other quick points

- On Pascal and later GPUs, managed memory might not be physically allocated when `cudaMallocManaged()` returns
  - It may only be populated on access (or prefetching)
    - Page table entries and memory frames might not be created until they are accessed by the GPU or the CPU
  - The pages can migrate to any processor's memory at any time, and the CUDA runtime employs heuristics to maintain data locality and prevent excessive page faults

# UM, Nuts and Bolts

- Managed memory migration is at the page level
  - The default page size is currently the same as the OS page size today (typically 4 KiB)
- The runtime intercepts CPU dirty pages and detects page faults
  - Moves from device over PCI-E only the dirty pages
  - Transparently, pages touched by the CPU (GPU) are moved back to the device (host) when needed

# UM, Nuts and Bolts

- Simultaneous access to managed memory from the CPU & GPUs with CC<6.0 not possible
  - Pre-Pascal GPUs lack hardware page faulting, so coherence can't be guaranteed
  - Therefore, if CC<6.0, an access from the CPU while a kernel is running will cause a segmentation fault
- On Pascal and later GPUs, the CPU & the GPU can simultaneously access managed memory, since they can both handle page faults
  - NOTE: up to application developer to ensure there are no race conditions caused by simultaneous accesses

# UM: Access to managed memory pool

- Playing it safe: use `cudaDeviceSynchronize()` before having the host work on memory that the GPU is also working on
- Note: Any function that logically guarantees the GPU finished execution is actually acceptable
  - Examples: `cudaStreamSynchronize()`, `cudaMemcpy()`, `cudaMemset()`, etc.

# UM: Access to managed pool – Example, for CC<6.0

Bad: Execution seg. faults

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
    x = 10;
}

int main() {
    kernel << < 1, 1 >> >();
    y = 20; // ERROR: CPU access concurrent with GPU
    cudaDeviceSynchronize();
    return 0;
}
```

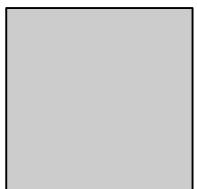
Good: Code runs ok

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
    x = 10;
}

int main() {
    kernel << < 1, 1 >> >();
    cudaDeviceSynchronize();
    y = 20; // GPU is idle so access is OK
    return 0;
}
```

# Unified Memory, under the hood

GPU A  
(better yet, processor A)

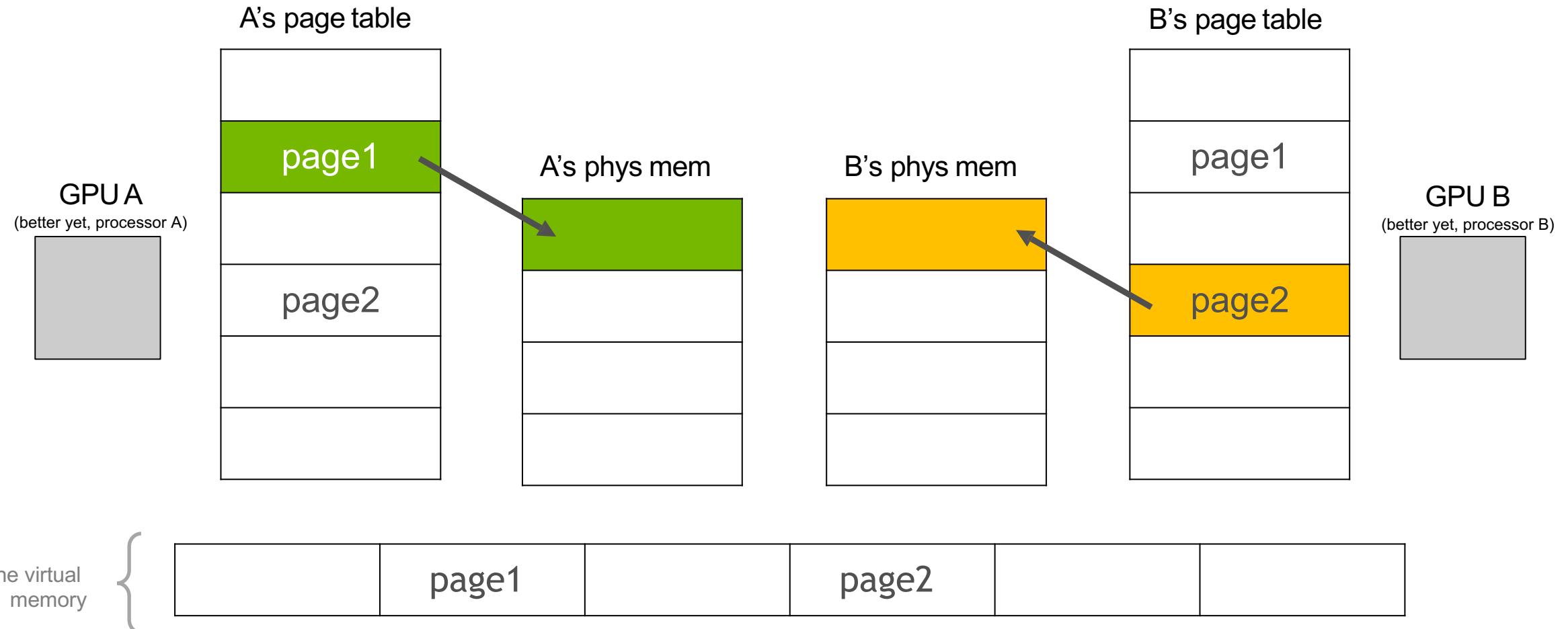


GPU B  
(better yet, processor B)



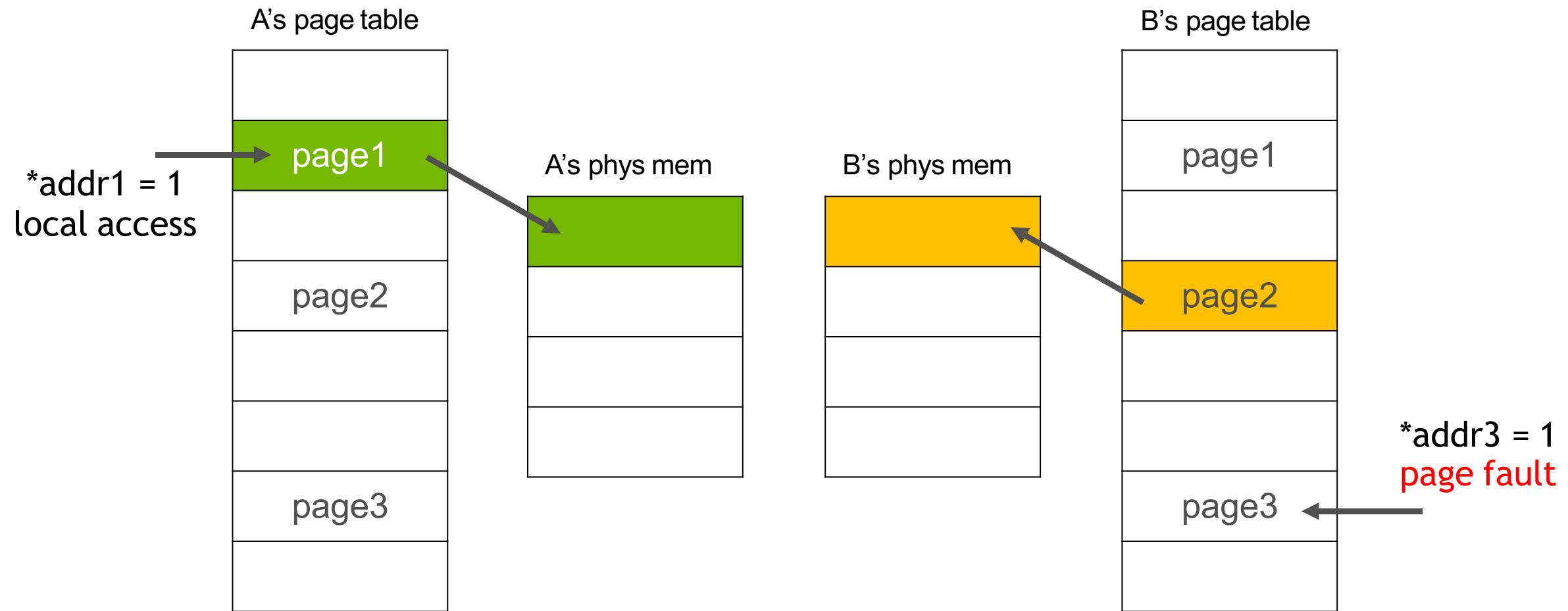
**Single virtual memory shared between processors**

# Unified Memory, under the hood

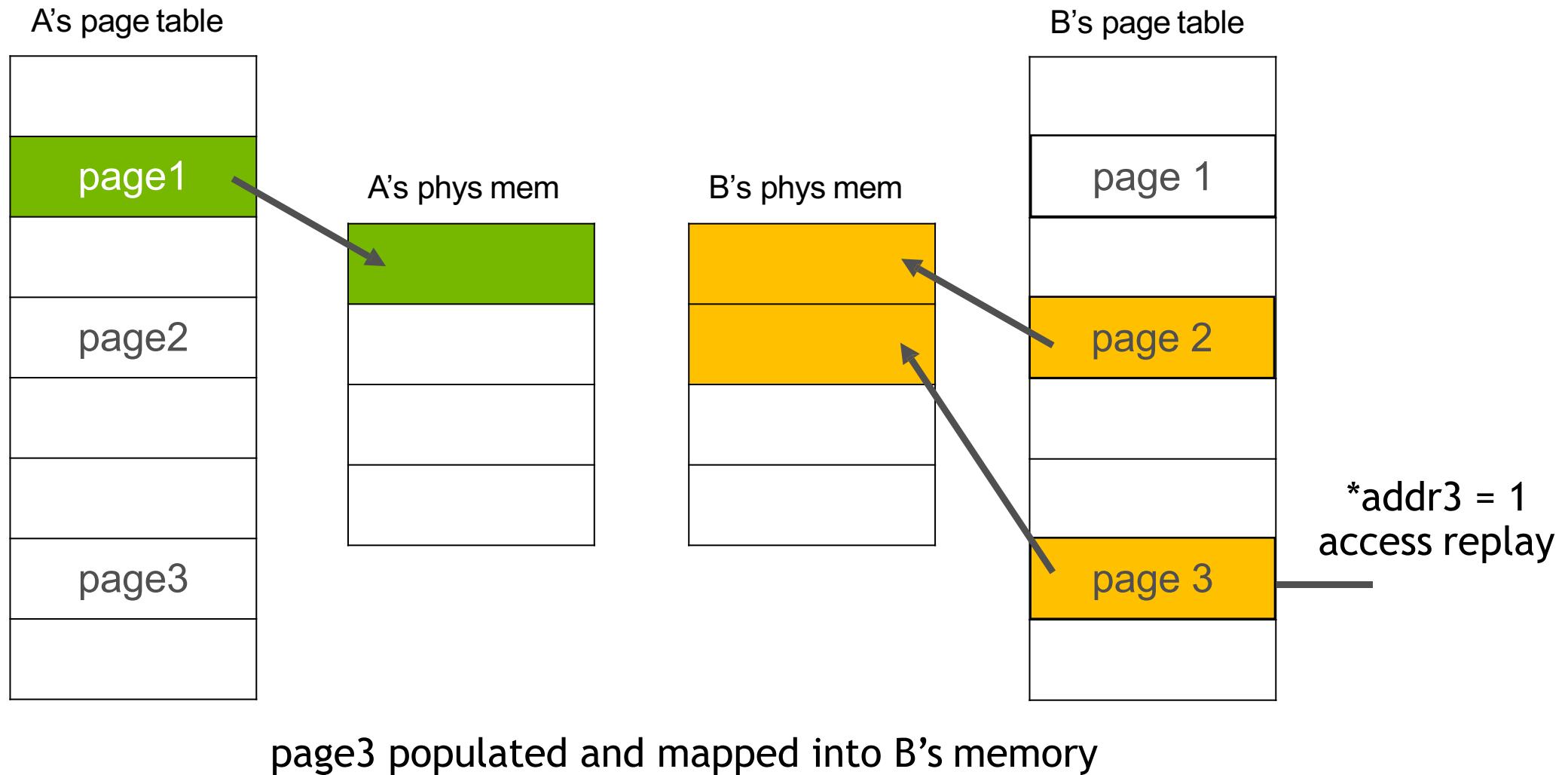


All processors operate in/perceive one **single** virtual memory. The physical memory is a odd thing

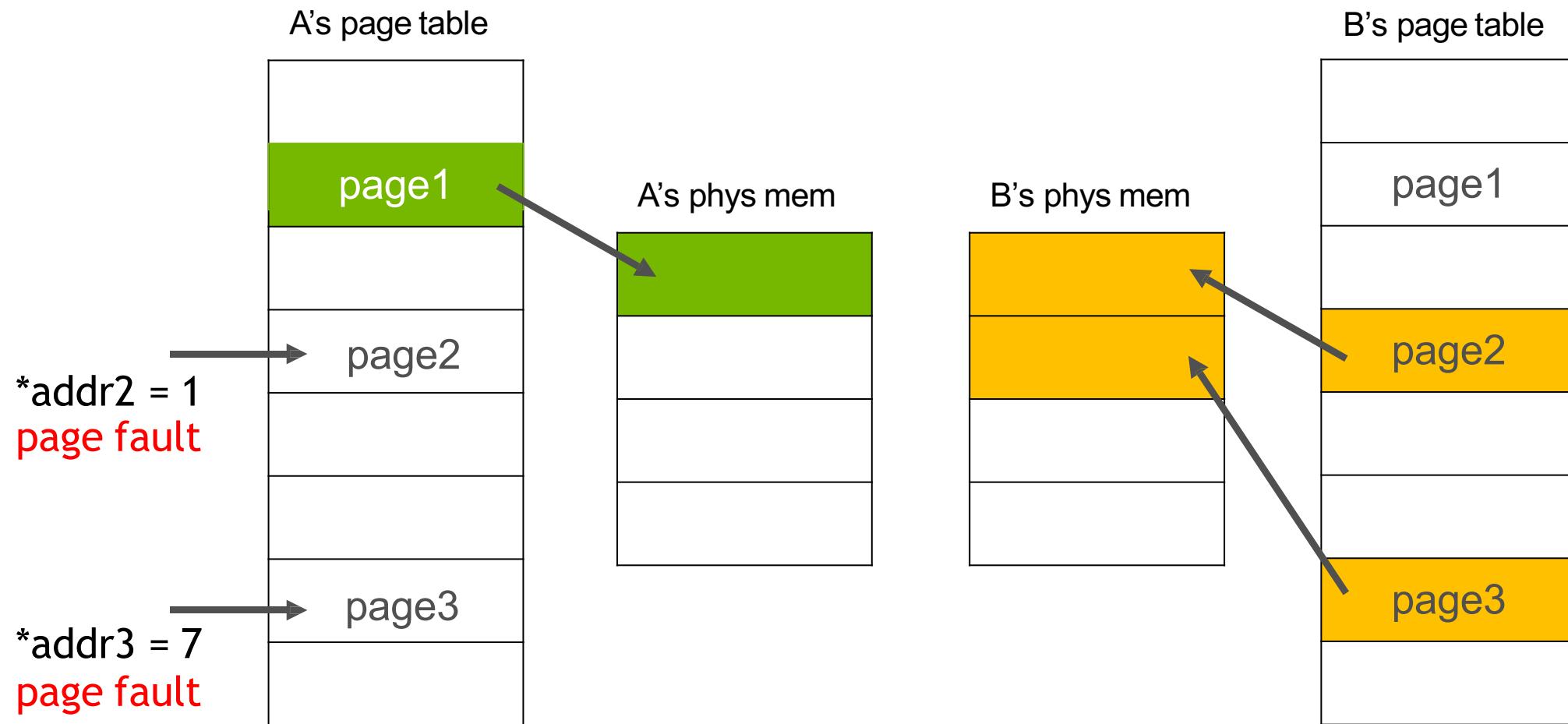
# Unified Memory, under the hood



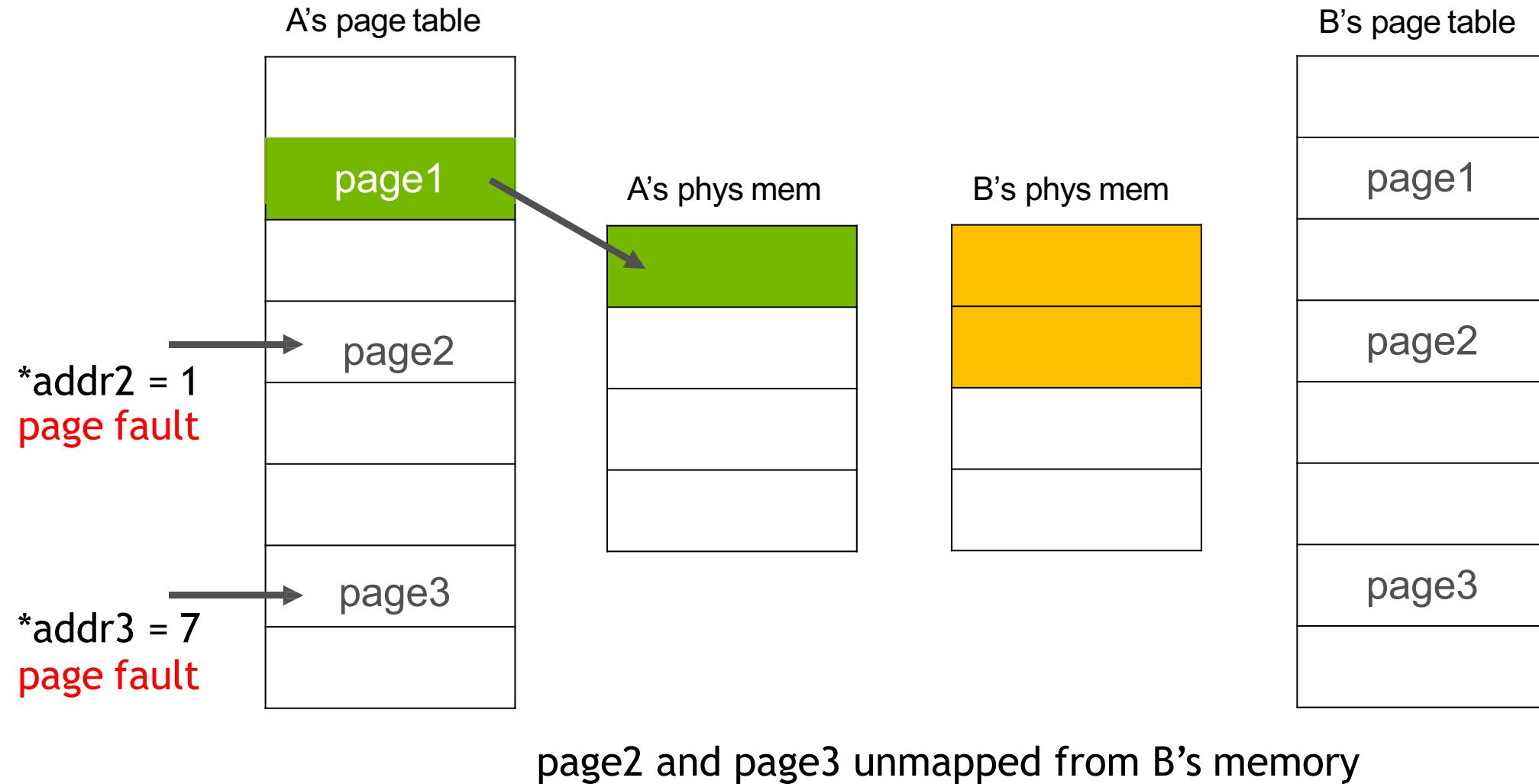
# Unified Memory, under the hood



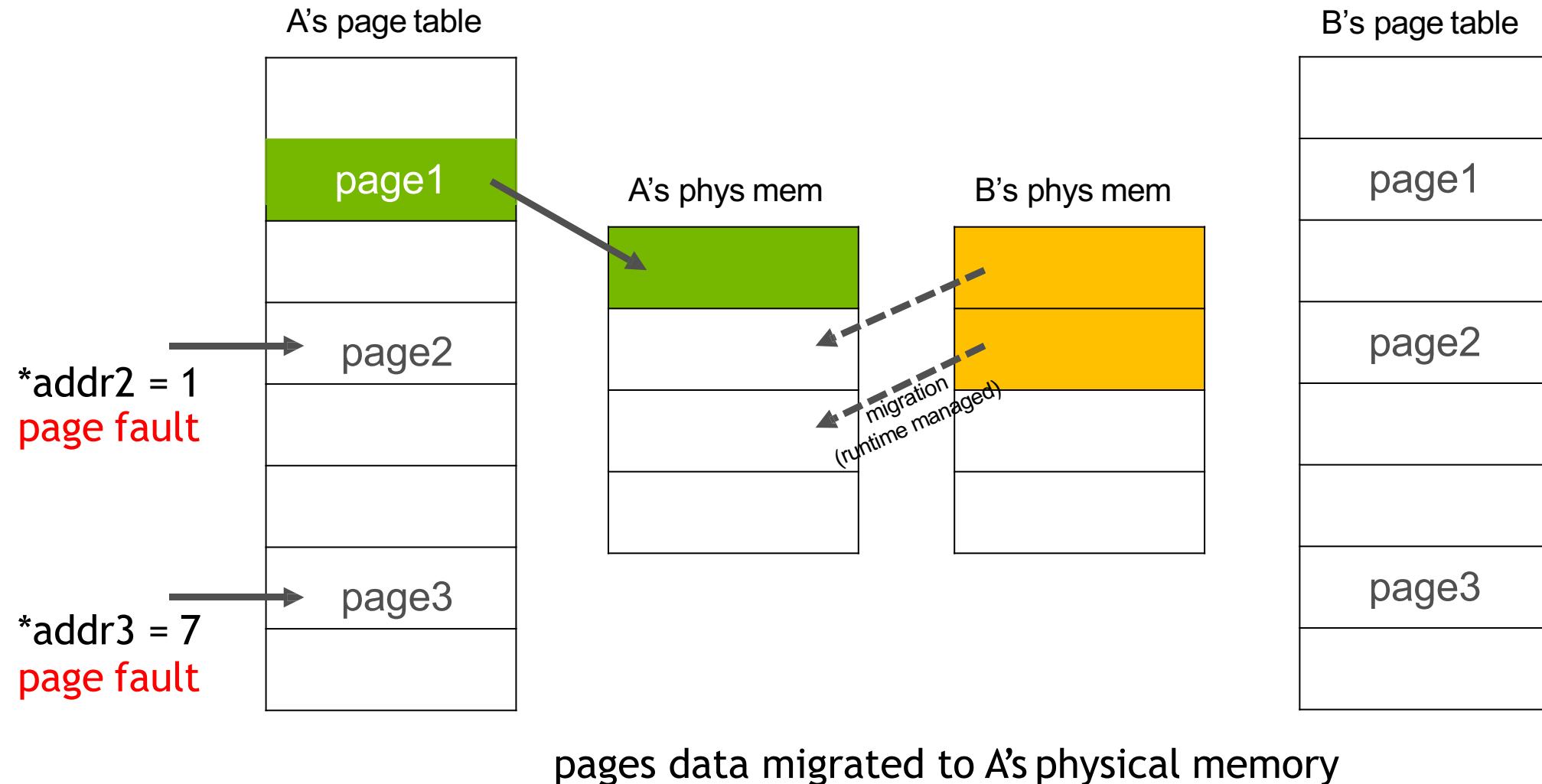
# Unified Memory, under the hood



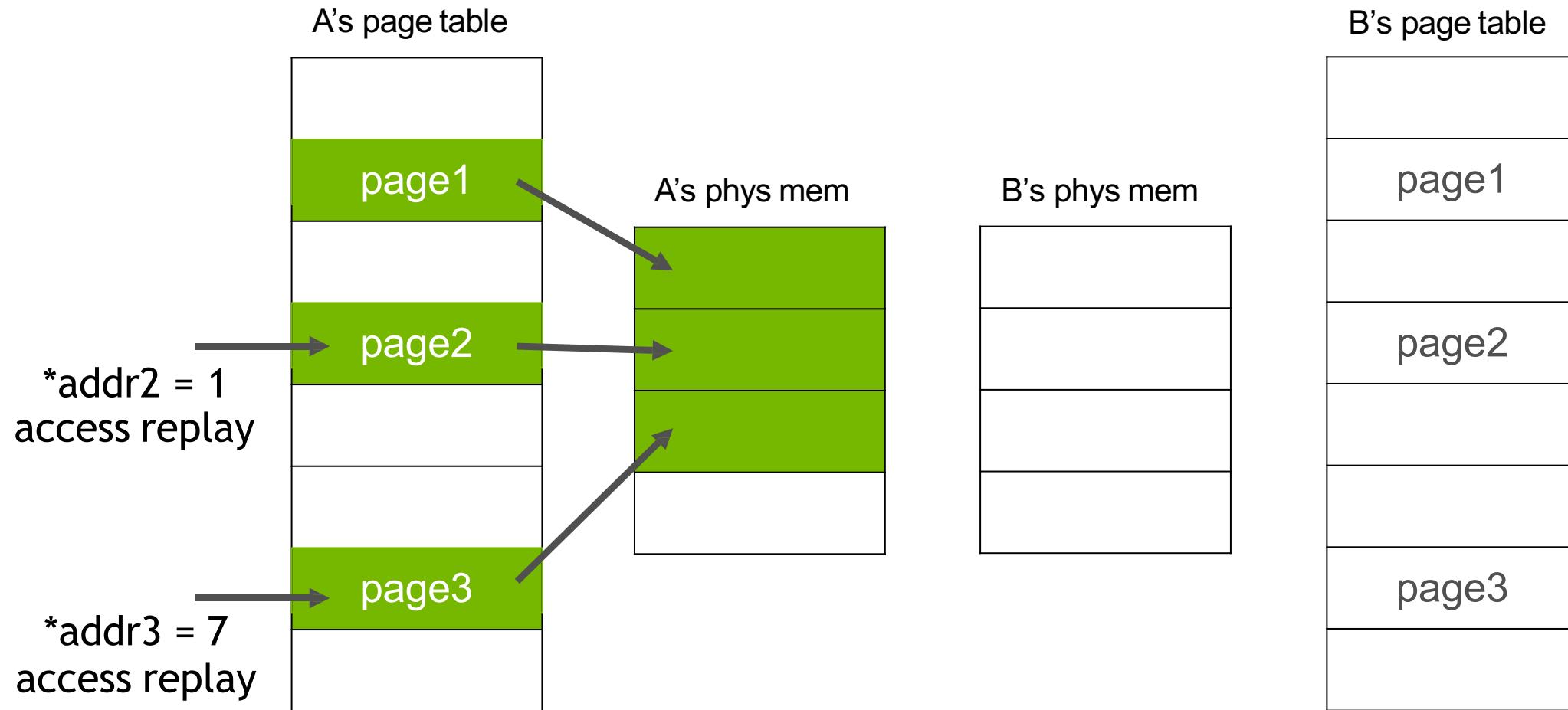
# Unified Memory, under the hood



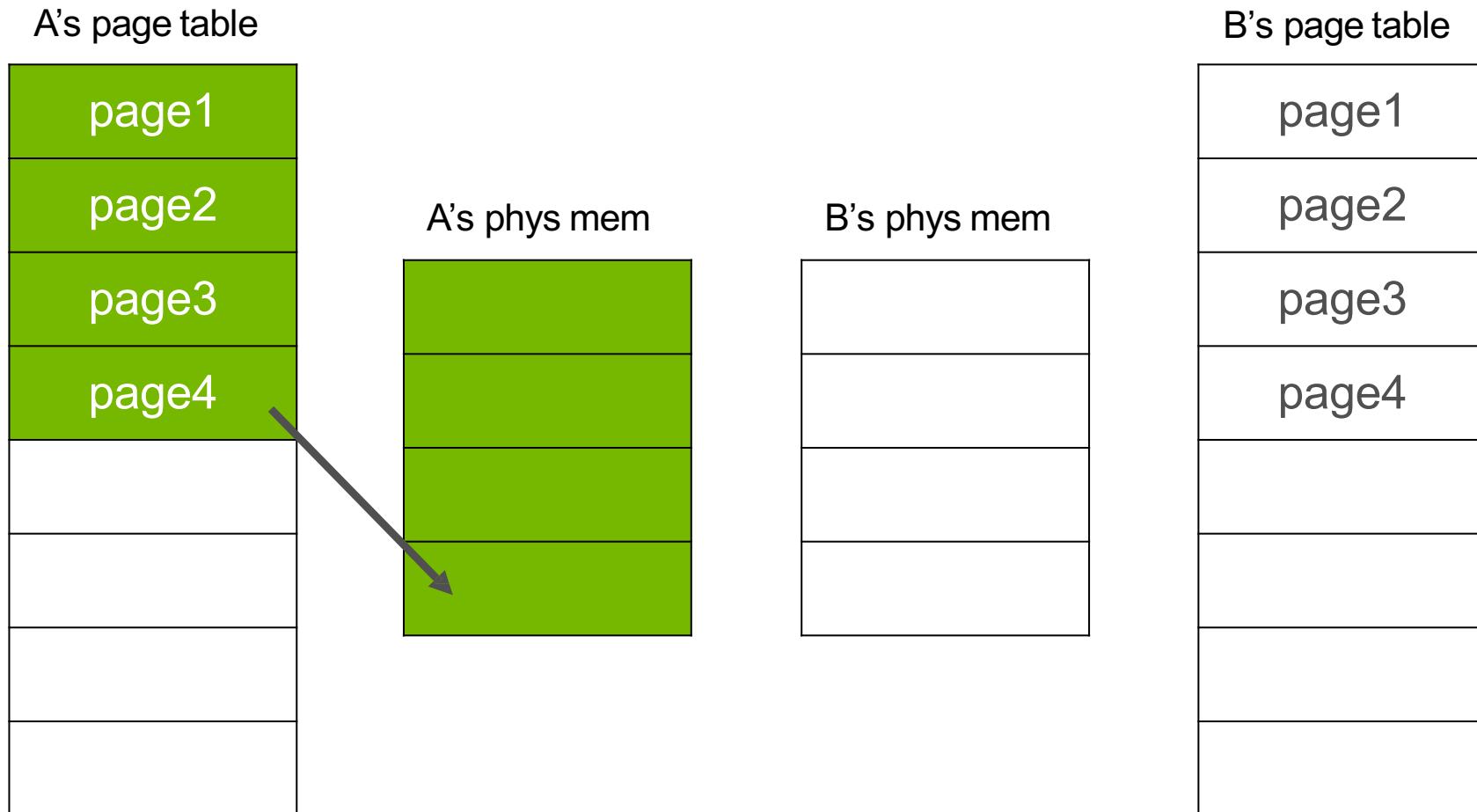
# Unified Memory, under the hood



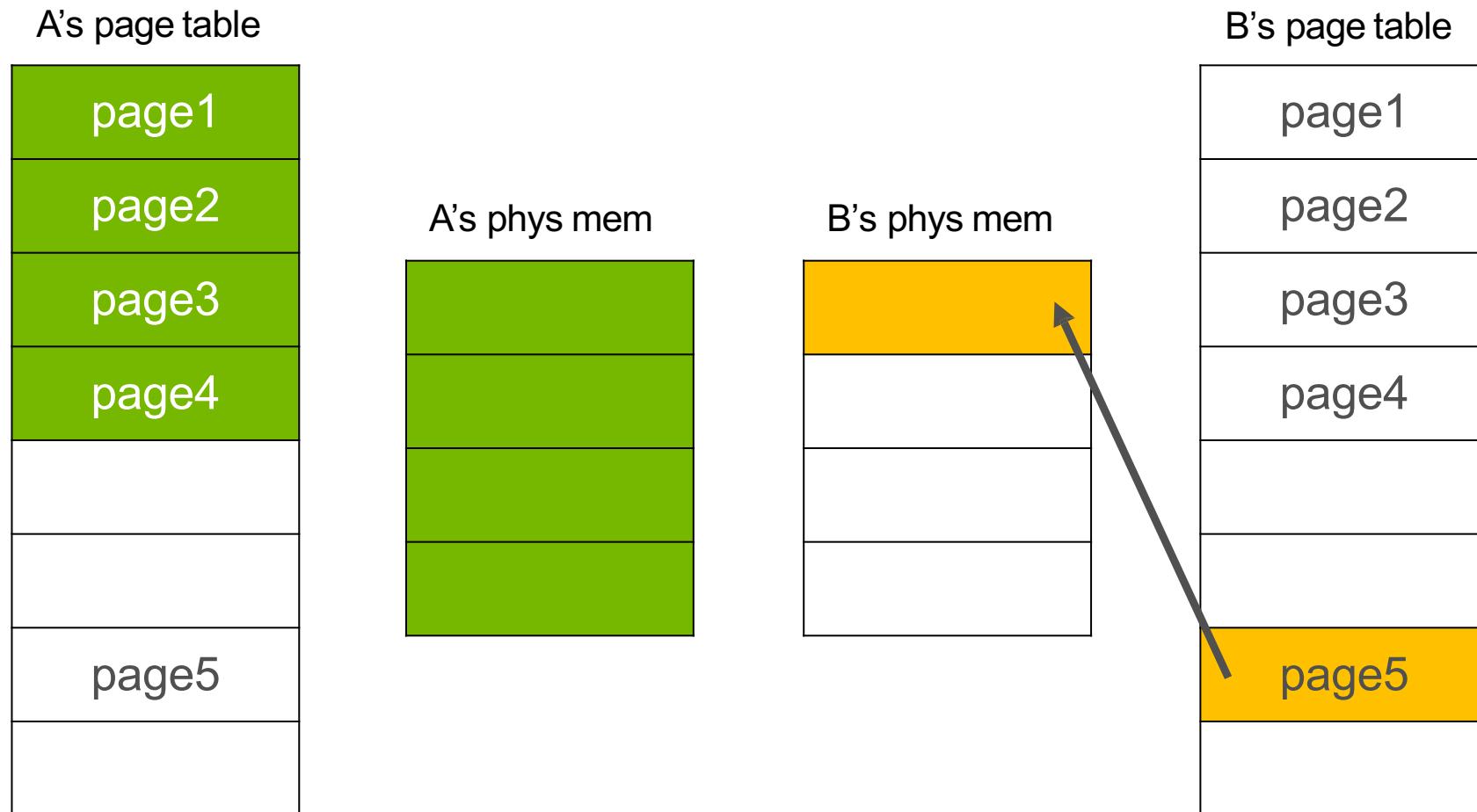
# Unified Memory, under the hood



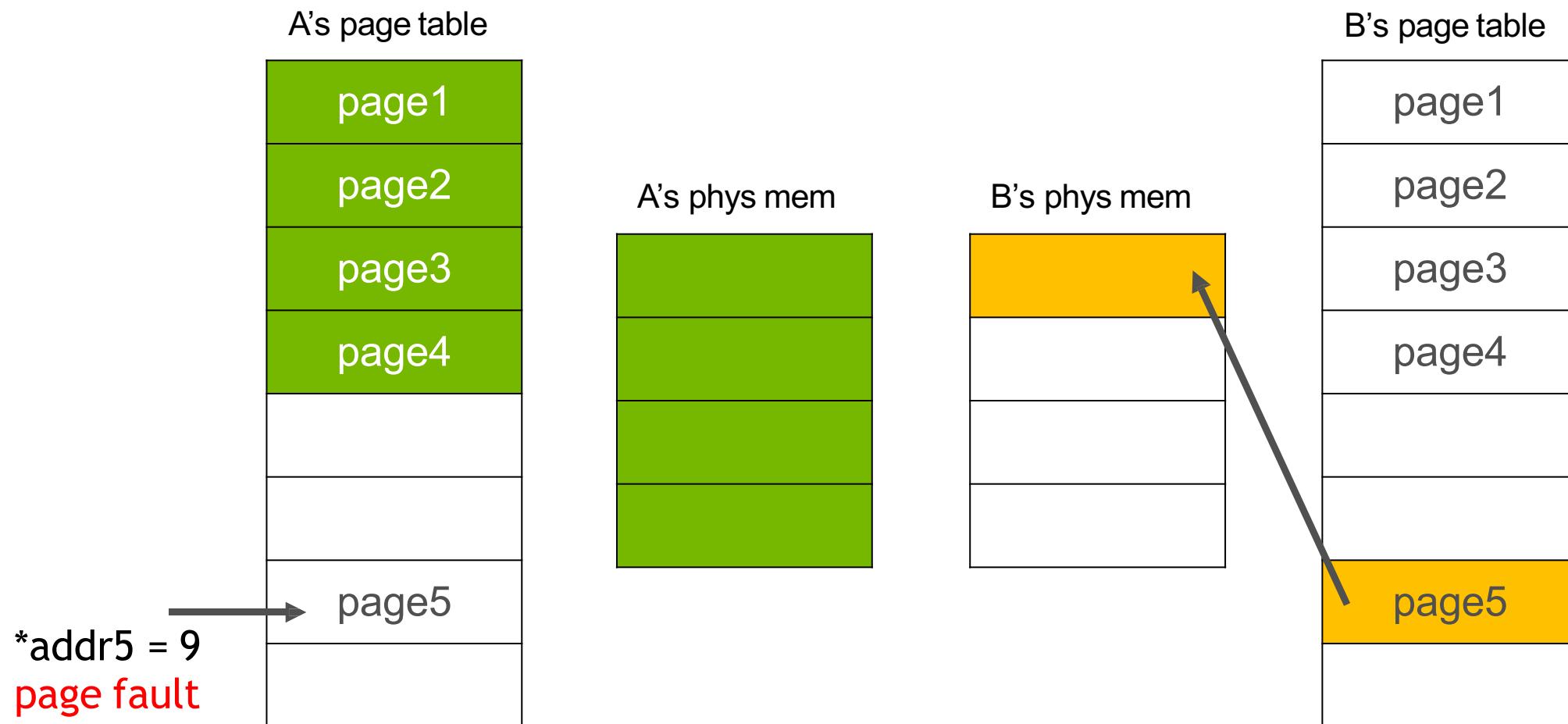
# Unified Memory, under the hood



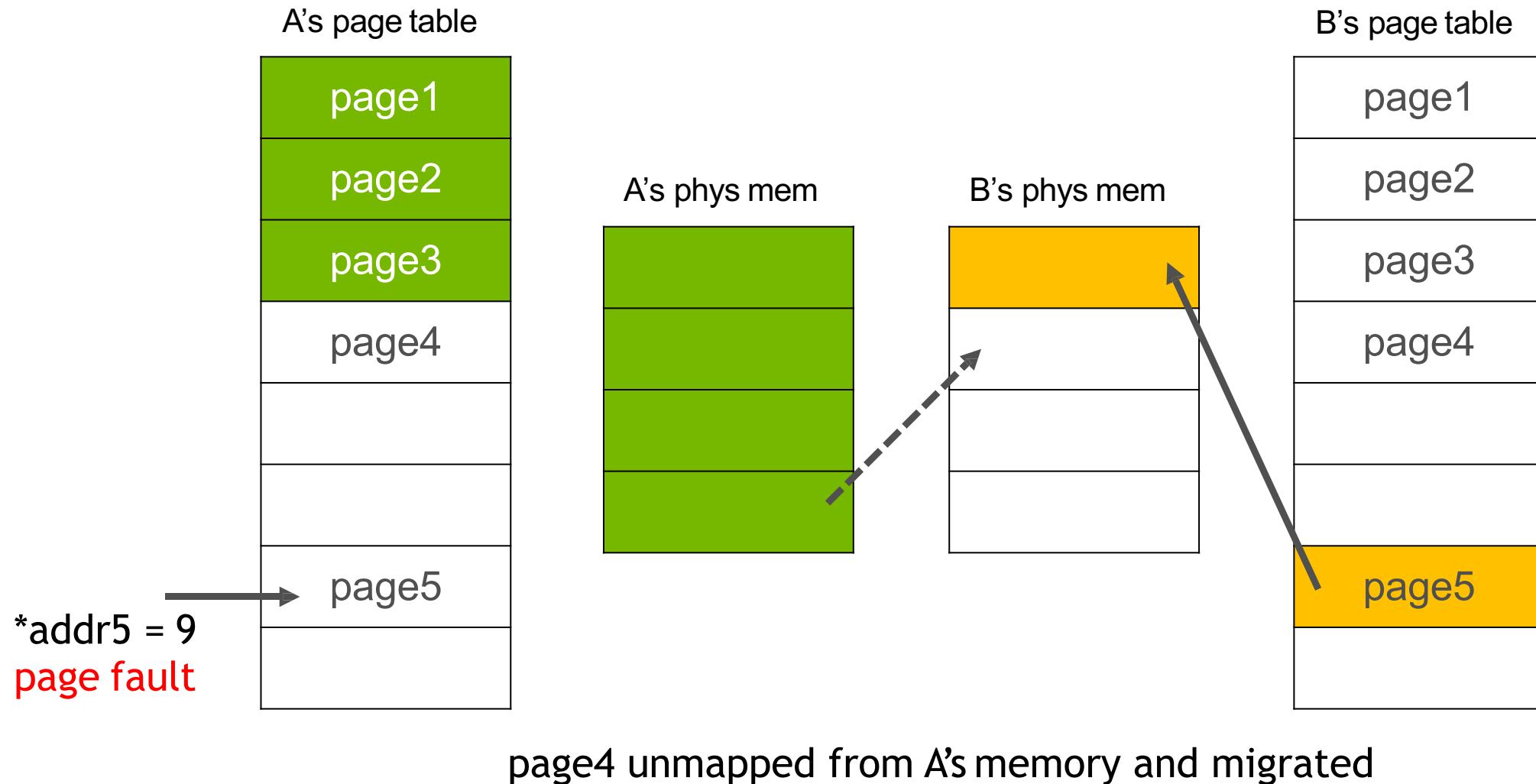
# Unified Memory, under the hood



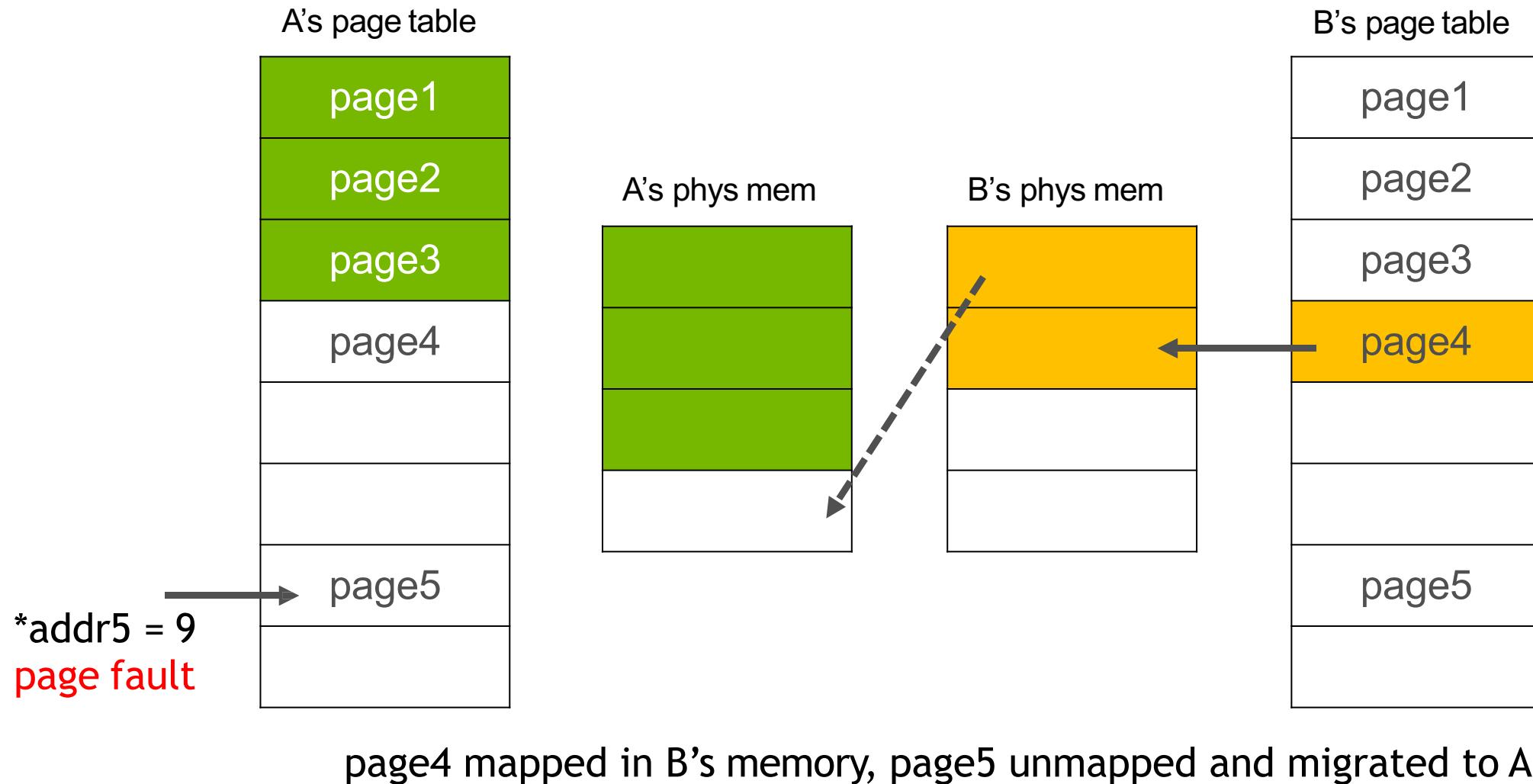
# Unified Memory, under the hood



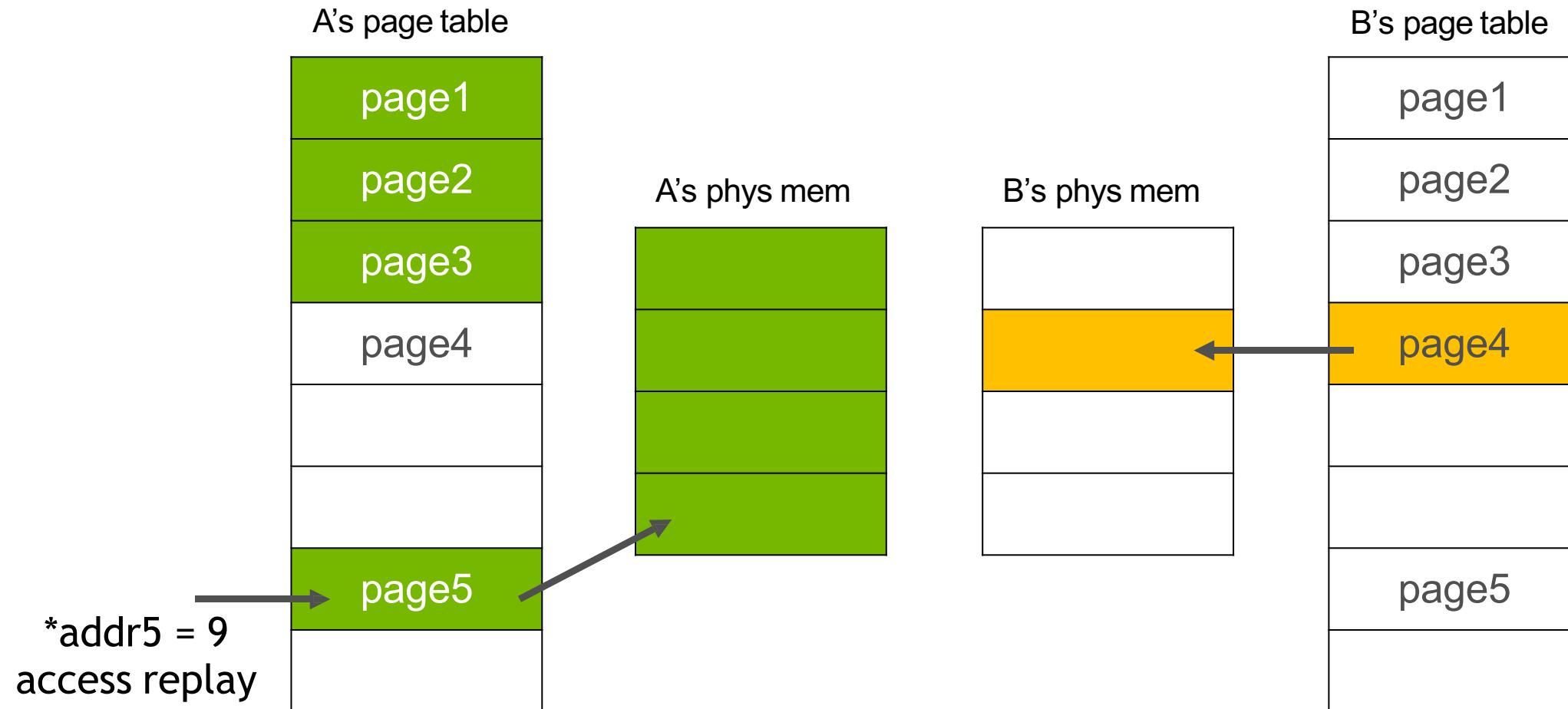
# Unified Memory, under the hood



# Unified Memory, under the hood



# Unified Memory, under the hood



# UM level of support

- Linux is most mature in UM support
  - E.g.: additional Unified Memory features on Linux (only)
    - On-demand page migration
    - GPU memory oversubscription
- Windows and MacOS support UM but not have the two features above

# Unified Memory platforms

	KEPLER	PASCAL	VOLTA
<b>Linux + x86</b>	No GPU fault support No concurrent access	On-demand migration	On-demand migration
<b>Linux + Power 9</b>		On-demand migration ≈80GB/s CPU-GPU BW	On-demand migration ≈150GB/s CPU-GPU BW Access counters HW coherency ATS support
<b>Windows</b>		No GPU fault support No concurrent access	
<b>MacOS</b>		No GPU fault support No concurrent access	
<b>Tegra</b>		Cached on CPU and iGPU No concurrent access	

# CUDA + Power 9 on Euler

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 8

__global__ void AplusB(int a, int b, int* result){
    result[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    int ret[SZ];
    AplusB <<<1, SZ >>>(10, 100, ret);
    cudaDeviceSynchronize();
    for (int i = 0; i < SZ; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

```
me759@euler ~ /t/cuda> nvprof ./ats_static
==19313== NVPYPROF is profiling process 19313, command: ./ats_static
0: A+B = 110
1: A+B = 111
2: A+B = 112
3: A+B = 113
4: A+B = 114
5: A+B = 115
6: A+B = 116
7: A+B = 117
==19313== Profiling application: ./ats_static
==19313== Profiling result:
      Type  Time(%)       Time     Calls      Avg       Min      Max  Name
GPU activities: 100.00%  5.5040us      1  5.5040us  5.5040us  5.5040us  AplusB(int, int, int*)
      API calls:   98.68% 196.71ms      1  196.71ms  196.71ms  196.71ms  cudaLaunchKernel
                  0.83% 1.6498ms      1  1.6498ms  1.6498ms  1.6498ms  cuDeviceTotalMem
                  0.44% 884.81us     96  9.2160us   199ns  357.96us  cuDeviceGetAttribute
                  0.04% 76.012us      1  76.012us  76.012us  76.012us  cuDeviceGetName
                  0.00% 8.8440us      1  8.8440us  8.8440us  8.8440us  cudaDeviceSynchronize
                  0.00% 2.6110us      1  2.6110us  2.6110us  2.6110us  cuDeviceGetPCIBusId
                  0.00% 1.9410us      3    647ns   363ns  1.0430us  cuDeviceGetCount
                  0.00% 1.1720us      2    586ns   326ns   846ns  cuDeviceGet
                  0.00% 316ns         1   316ns   316ns   316ns  cuDeviceGetUuid
```

## CPU vs GPU

CPU code

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
cpu_func2(data, N);  
  
cpu_func3(data, N);  
  
free(data);
```

GPU code w/ Unified Memory

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
gpu_func2<<<...>>>(data, N);  
cudaDeviceSynchronize();  
  
cpu_func3(data, N);  
  
free(data);
```

This function call not needed  
if synchronous kernel calls

## Explicit mem. Management vs. Unified Memory

### Explicit Memory Management

```
void *data, *d_data;  
data = malloc(N);  
cudaMalloc(&d_data, N);  
cpu_func1(data, N);  
cudaMemcpy(d_data, data, N, ...)  
gpu_func2<<<...>>>(d_data, N);  
cudaMemcpy(data, d_data, N, ...)  
cudaFree(d_data);  
cpu_func3(data, N);  
  
free(data);
```

### GPU code w/ Unified Memory

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
gpu_func2<<<...>>>(data, N);  
cudaDeviceSynchronize();  
  
cpu_func3(data, N);  
  
free(data);
```

## Full Control with Prefetching

### Explicit Memory Management

```
void *data, *d_data;  
data = malloc(N);  
cudaMalloc(&d_data, N);  
cpu_func1(data, N);  
cudaMemcpy(d_data, data, N, ...)  
gpu_func2<<<...>>>(d_data, N);  
cudaMemcpy(data, d_data, N, ...)  
cudaFree(d_data);  
cpu_func3(data, N);  
  
free(data);
```

### Unified Memory + Prefetching

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
cudaMemPrefetchAsync(data, N, GPU)  
gpu_func2<<<...>>>(data, N);  
cudaMemPrefetchAsync(data, N, CPU)  
cudaDeviceSynchronize();  
cpu_func3(data, N);  
  
free(data);
```

## Deep Copy

### Explicit Memory Management

```
char **data;  
// allocate and initialize data on the CPU  
  
char **d_data;  
char **h_data = (char**)malloc(N*sizeof(char*));  
for (int i = 0; i < N; i++) {  
    cudaMalloc(&h_data[i], N);  
    cudaMemcpy(h_data[i], data[i], N, ...);  
}  
cudaMalloc(&d_data, N*sizeof(char*));  
cudaMemcpy(d_data, h_data, N*sizeof(char*), ...);  
  
gpu_func<<<...>>>(d_data, N);
```

### GPU code w/ Unified Memory

```
char **data;  
// allocate and initialize data on the CPU  
  
gpu_func<<<...>>>(data, N);
```

# UM – Why Bother?

## 1. A matter of convenience

- Much simpler to write code using this memory model
- For the casual programmer, the code will probably run faster due to data locality
  - The CUDA runtime will take care of moving the data where it ought to be

## 2. Looking ahead, we'll see more of this in the next 18 – 24 months

- Already the case for integrated GPUs that are part of the system chipset (Tegra Xavier)
- Better support coming for Windows & MacOS

# High vantage point, the big picture

- In the beginning the host and device memory spaces were totally separated
- Then, the device could access memory on the host (Z-C memory access)
- Then, the idea of a unified virtual space took off and we had UVA
  - P2P communication, no need for pointer de-mangling
- Finally, the UM (managed memory) came along; processors can access each other's memory
  - The field is in flux

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 18

03/04/2020

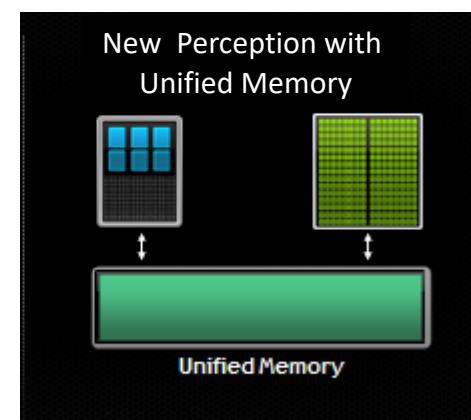
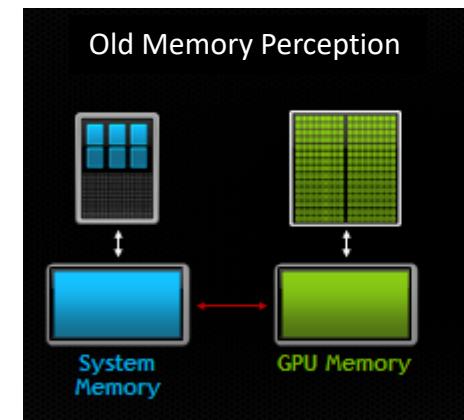
# Factoid of the day

**Chuck Norris has never blinked in his entire life. Never.**

-- Chuck Norris, American martial artist, actor, film producer and screenwriter. Inventor of Chun Kuk Do. [1940 – ]

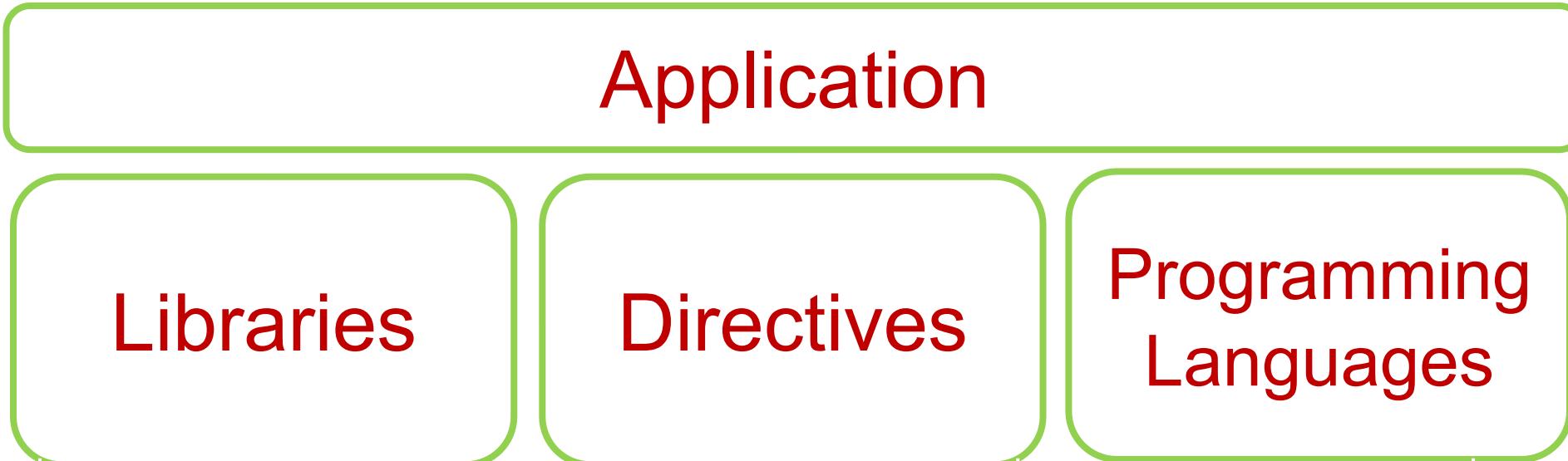
# Before we get going...

- Last time:
  - Advanced GPU memory issues
    - Zero-copy memory
    - Unified Virtual Addressing
    - Managed memory
- Today:
  - Off-the-shelf GPU computing
    - thrust
    - CUB
  - Further discussion, Final Project topics
- Other tidbits:
  - Assignment due on Thursday at 9 pm
  - This is the last “GPU computing” lecture
  - ME759 Exam: April 15, at 7:15 PM, in 1106ME
    - Review: Tu, April 14, at 7:00 PM, in Canvas



# GPU Computing with `thrust`

# Three Ways to Accelerate on GPU



Easiest Approach

Maximum Performance

Direction of increased performance  
(and effort)

# Acknowledgments

- The **thrust** slides include material provided by Nathan Bell previously of NVIDIA (at Google now)
- Modified here and there, responsible for inaccuracies

# Design Philosophy, thrust

- Increase programmer productivity
  - Build applications quickly via generic programming
    - Leverage a template-based approach
- Should be running fast
  - Efficient leveraging of GPU hardware

# What is thrust?

- A template library for parallel computing on GPU **and** CPU\*
- Mimics the C++ STL
- Heavy use of C++ containers
  - The one lecture where C++ comes into play heavily
- Provides ready to use generic algorithms
  - Sorting, reduction, scan, etc.

# What is thrust?

- **thrust** is a header library
  - All its functionality accessed by `#include`-ing the appropriate **thrust** header file
- Program is compiled with **nvcc** as per usual, no special tools are required
- C++ syntax. Related to high-level, **host-side** code that you have to produce
  - The concepts of execution configuration, shared memory, occupancy, blah, blah, blah: all but gone

# Work Plan, **thrust**

- Namespaces, containers, iterators
- Algorithms
- General transformations. Zipping & fusing
- Thrust example: Processing rainfall data

# Namespaces

- Avoid name collisions: thrust vs. std namespaces

```
// allocate host memory
thrust::host_vector<int> h_vec(10);

// call STL sort
std::sort(h_vec.begin(), h_vec.end());

// call Thrust sort
thrust::sort(h_vec.begin(), h_vec.end());

// for brevity
using namespace thrust;

// without namespace
int sum = reduce(h_vec.begin(), h_vec.end());
```

# thrust containers: `host_vector` & `device_vector`

- thrust provides two vector containers: `host_vector` and `device_vector`
  - `host_vector` is stored in host memory
  - `device_vector` lives in GPU device memory
- thrust's vector containers are just like `std::vector` in the C++ STL
  - Like `std::vector`, `host_vector` & `device_vector` are generic containers – they store whatever

# thrust containers: host\_vector & device\_vector

- Common operations become concise & readable through use of containers

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec;

// manipulate device values from the host
d_vec[0] = 13;
d_vec[1] = 27;
std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;

// copy host vector to device
h_vec = d_vec;

// vector memory automatically released w/ free() or cudaFree()
```

# thrust containers

- Compatible with STL containers (thrust & std containers play well together)
  - Below, a std **list** plays well with a thrust device **vector**

```
// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);

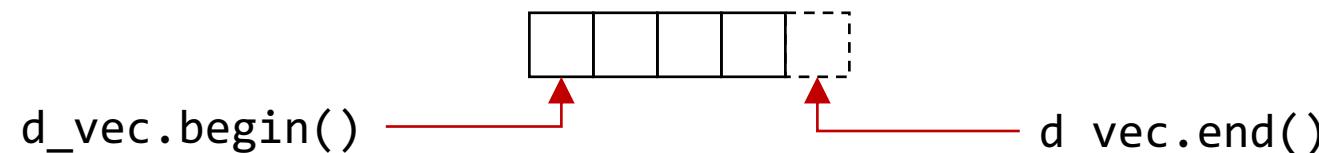
// copy list to device vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// alternative method using vector constructor
thrust::device_vector<int> d_vec2(h_list.begin(), h_list.end());
```

# Iterators: think of them as pointers

- Sequences are defined by pairs of iterators
- For vector containers, iterators act like pointers
- They can be used like pointers (e.g. incremented)
- Can be converted to raw pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);
d_vec.begin(); // returns iterator at first element of d_vec
d_vec.end();   // returns iterator one past the last element of d_vec
```



# Interoperability: iterators to pointers

- Convert iterators to raw pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector's memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<< (N+255) / 256, 256 >>>(N, ptr);

// use ptr in a CUDA API function
cudaMemcpy(ptr, ... );
```

# Interoperability: pointers to iterators

- Wrap raw pointers with `device_ptr`

```
// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```

# Namespaces, containers, iterators: closing remarks

- Namespaces
  - Used to avoid collisions and increase functionality
- Containers
  - Manage both host & device memory
  - Automatic allocation and deallocation
  - Simplify data transfers
- Iterators in thrust
  - Behave like pointers
  - Can move back-and-forth between iterators and raw pointers

# Work Plan, **thrust**

- Namespaces, containers, iterators
- Algorithms
- General transformations. Zipping & fusing
- Thrust example: Processing rainfall data

# Algorithms

- Element-wise operations
  - `for_each, transform, gather, scatter ...`
- Reductions
  - `reduce, inner_product, reduce_by_key ...`
- Prefix Sums [scans]
  - `inclusive_scan, inclusive_scan_by_key ...`
- Sorting
  - `sort, stable_sort, sort_by_key ...`

# Thrust Example: Sort

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void) {
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

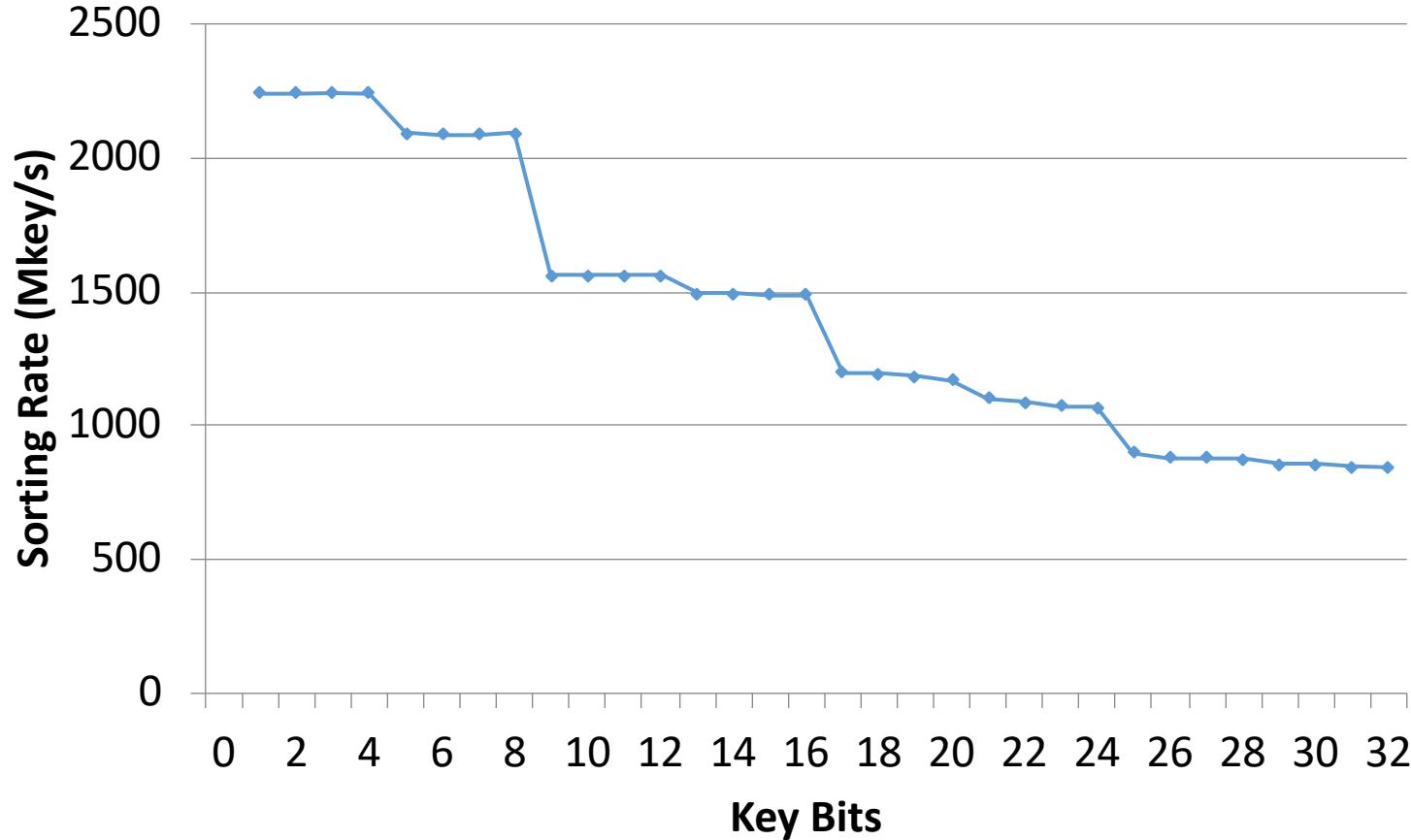
    return 0;
}
```

# Leveraging Parallel Primitives, benchmark test

- Test: sort 32M keys on each platform
  - Performance measured in millions of keys per second [higher is better]
- Conclusion: **sort** is highly optimized

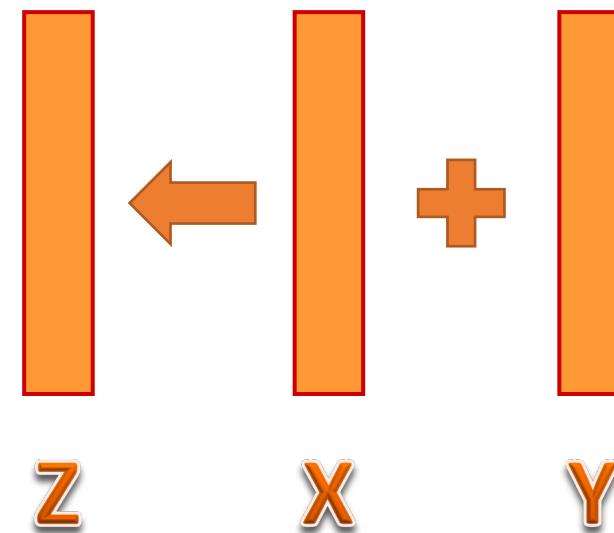
data type	std::sort [per second]	tbb::parallel_sort [per second]	thrust::sort [per second]
char	25.1	68.3	3532.2
short	15.1	46.8	1741.6
int	10.6	35.1	804.8
long	10.3	34.5	291.4
float	8.7	28.4	819.8
double	8.5	28.2	358.9

# Input-Sensitive Optimizations



# Example: Vector Addition

```
for (int i = 0; i < N; i++)  
    Z[i] = X[i] + Y[i];
```



# Example, Vector Addition

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {
    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    thrust::transform(X.begin(), X.end(),
                     Y.begin(),
                     Z.begin(),
                     thrust::plus<float>());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
```

# Maximum Value – after all, it's just a reduce operation

```
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {
    thrust::device_vector<float> X(3);

    X[0] = 10.f; X[1] = 30.f; X[2] = 20.f;

    float init = X[0];

    float result = thrust::reduce(X.begin(), X.end(),
                                 init,
                                 thrust::maximum<float>());

    std::cout << "maximum is " << result << "\n";

    return 0;
}
```

# Algorithms

- Standard operators

```
// allocate memory
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

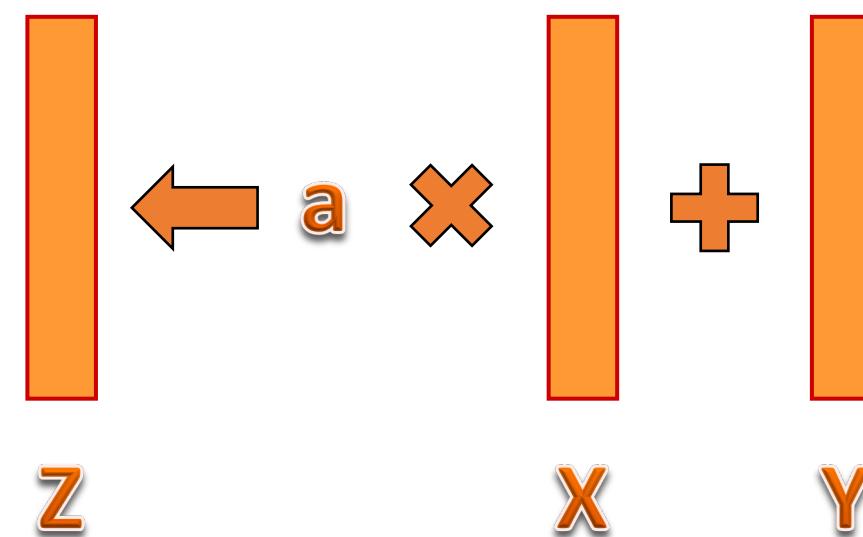
// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());

// transform A - B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), minus<int>());

// multiply reduction
int product = reduce(A.begin(), A.end(), 1, multiplies<int>());
```

# Example: SAXPY (of sorts)

```
for (int i = 0; i < N; i++)  
    Z[i] = a * X[i] + Y[i];
```



# SAXPY

functor

```
struct saxpy
{
    float m_a;

    saxpy(float a) : m_a(a) {}

    __host__ __device__
    float operator()(float x, float y)
    {
        return m_a * x + y;
    }
};
```

} state  
} constructor  
} call operator

```
void main(void){
    thrust::device_vector<float> X(3), Y(3), Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    float aVal = 2.0f;
    thrust::transform(X.begin(), X.end(),
                     Y.begin(),
                     Z.begin(),
                     saxpy(aVal));

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";
}
```

# Custom Types & Operators

```
struct negate_float2
{
    __host__ __device__
    float2 operator()(float2 a)
    {
        return make_float2(-a.x, -a.y);
    }
};

// declare storage
device_vector<float2> input = ...
device_vector<float2> output = ...

// create function object or ‘functor’
negate_float2 fnctr;

// negate vectors
transform(input.begin(), input.end(), output.begin(), fnctr);
```

# Custom Types & Operators

```
// compare x component of two float2 structures
struct compare_float2
{
    __host__ __device__
    bool operator()(float2 a, float2 b)
    {
        return a.x < b.x;
    }
};

// declare storage
device_vector<float2> vec = ...

// create comparison functor
compare_float2 comp;

// sort elements by x component
sort(vec.begin(), vec.end(), comp);
```

# Custom Types & Operators

```
// return true if x is greater than threshold
struct is_greater_than
{
    int threshold;

    is_greater_than(int t) { threshold = t; }

    __host__ __device__
    bool operator()(int x) { return x > threshold; }
};

device_vector<int> vec = ...

// create predicate functor (returns true for x > 10)
is_greater_than pred(10);

// count number of values > 10
int result = count_if(vec.begin(), vec.end(), pred);
```

# Algorithms, More Examples...

Algorithm	Description
<code>reduce</code>	Sum of a sequence
<code>find</code>	First position of a value in a sequence
<code>mismatch</code>	First position where two sequences differ
<code>inner_product</code>	Dot product of two sequences
<code>equal</code>	Whether two sequences are equal
<code>min_element</code>	Position of the smallest value
<code>count</code>	Number of instances of a value
<code>is_sorted</code>	Whether sequence is in sorted order
<code>transform_reduce</code>	Sum of transformed sequence

# Work Plan, **thrust**

- Namespaces, containers, iterators
- Algorithms
- General transformations. Zipping & fusing
- Thrust example: Processing rainfall data

# General Transformations

Unary Transformation

```
for (int i = 0; i < N; i++)
    X[i] = f(A[i]);
```

Binary Transformation

```
for (int i = 0; i < N; i++)
    X[i] = f(A[i], B[i]);
```

Ternary Transformation

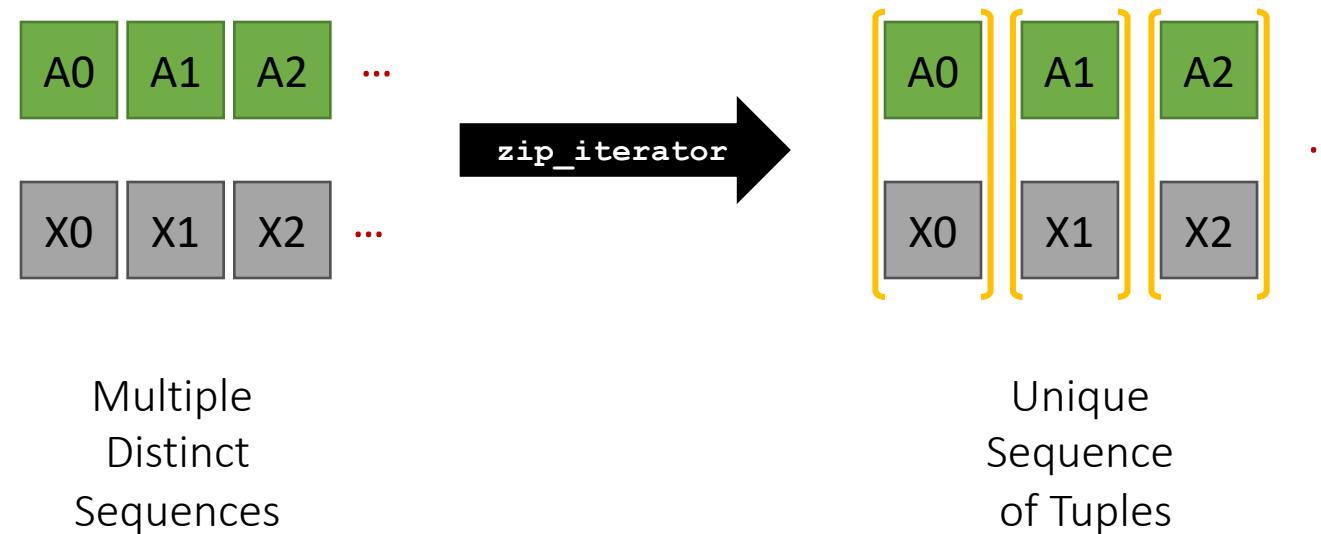
```
for (int i = 0; i < N; i++)
    X[i] = f(A[i], B[i], C[i]);
```

General Transformation

```
for (int i = 0; i < N; i++)
    X[i] = f(A[i], B[i], C[i], ...);
```

- Like C++ STL, **thrust** provides built-in support for unary and binary transformations
- Transformations involving 3 or more input ranges must use a different approach

# The Zip Operation (“zipping”)



# Example: General Transformations

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include<iostream>

struct linear_combo {
    __host__ __device__
    float operator()(thrust::tuple<float, float, float> t) {
        float x, y, z;
        thrust::tie(x,y,z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};
```

Functor  
Definition

```
void main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3); //  $U = 2X + 3Y + 4Z$ 

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;
    Z[0] = 20; Z[1] = 30; Z[2] = 25;

    thrust::transform
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         U.begin(),
         linear_combo());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "U[" << i << "] = " << U[i] << "\n";
}
```

These are the important parts:  
three different entities are  
zipped together into a big one

# Example: thrust::transform\_reduce

The main point here:

Previous slide → transform

This slide → transform\_reduce

Punch line: “fusing” operations together whenever possible is a good idea

```
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/iterator/zip_iterator.h>
#include<iostream>

struct linear_combo {
    __host__ __device__
    float operator()(thrust::tuple<float,float,float> t) {
        float x, y, z;
        thrust::tie(x,y,z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};

void main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3), U(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;
    Z[0] = 20; Z[1] = 30; Z[2] = 25;

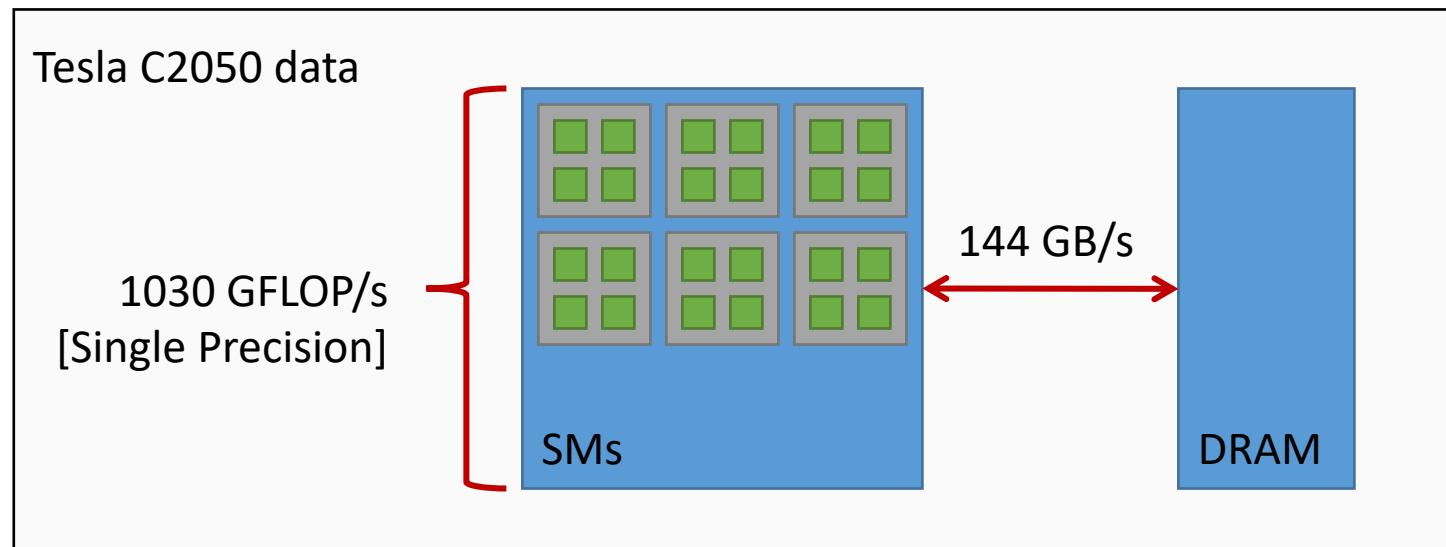
    thrust::plus<float> binary_op;
    float init = 0.f;

    float myResult = thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         linear_combo(),
         init,
         binary_op);

    std::cout << myResult << std::endl;
}
```

# Performance Considerations [discussion tided to “fusing”]

- Picture below shows the two key parameters in any computation
  - Peak flop rate
  - Max bandwidth



- Punch line (for Tesla C2050):
  - In one second I do 1030 GFLOPs. In one second, I can bring over 144 GB of data
  - Therefore, on average, I need to do  $1030/144$  operations per byte of data; i.e., an average of 7.1 ops per byte

# Arithmetic Intensity [discussion tided to “fusing”]

Kernel	FLOP/Byte*
Vector Addition	1 : 12
SAXPY	2 : 12
Max Index	1 : 12
Reduce	1 : 4
Ternary Transformation	5 : 16

\* excludes indexing overhead

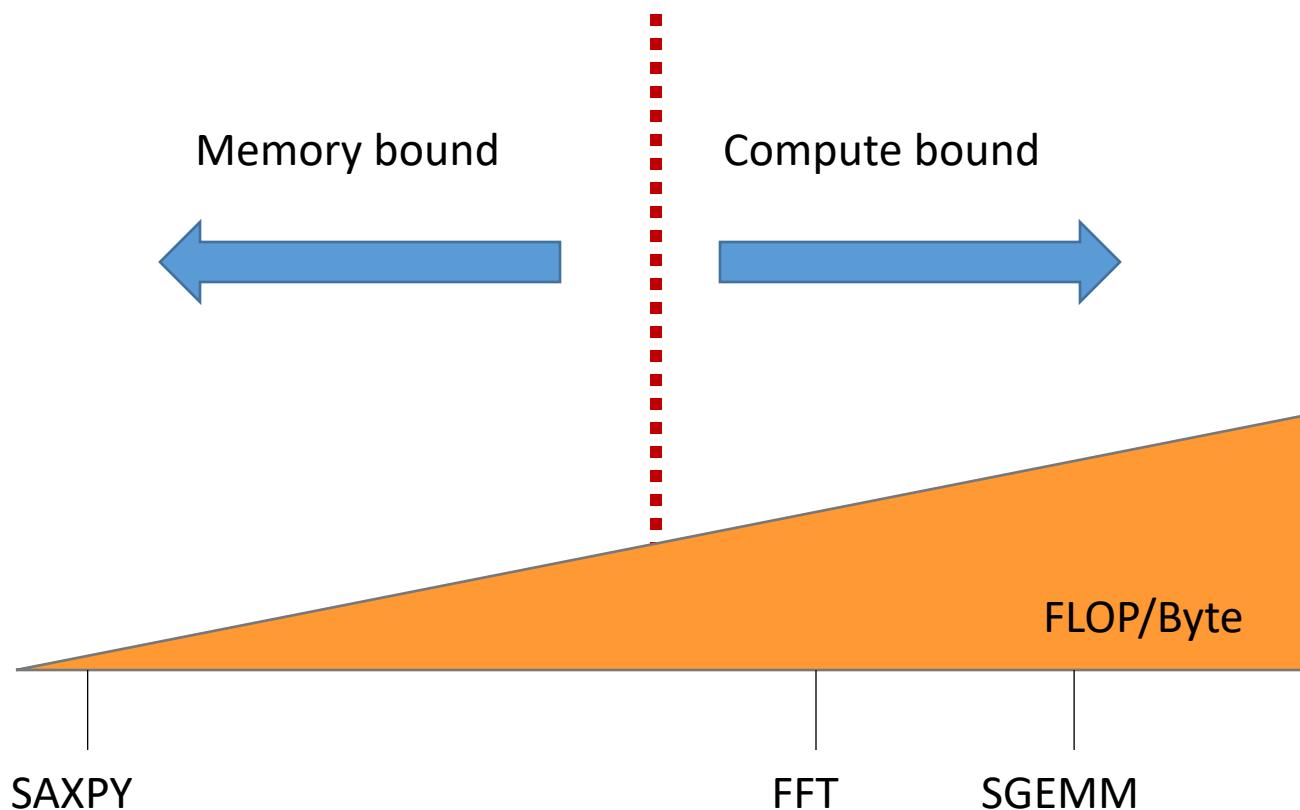
“Byte” refers to a Global Memory byte

Ternary Transformation:  $w[i] = \alpha x[i] + \beta y[i] + \gamma z[i]$

Hardware**	FLOP/Byte
GeForce GTX 280	~7.0 : 1
GeForce GTX 480	~7.6 : 1
Tesla C870	~6.7 : 1
Tesla C1060	~9.1 : 1
Tesla C2050	~7.1 : 1

\*\* lists the number of flop per byte of data to reach peak Flop/s rate

# Arithmetic Intensity [discussion tided to “fusing”]



# Fusing, as a strategy to increase the arithmetic intensity

```
for (int i = 0; i < N; i++)  
    U[i] = F(X[i], Y[i], Z[i]);  
  
for (int i = 0; i < N; i++)  
    V[i] = G(X[i], Y[i], Z[i]);
```

```
for (int i = 0; i < N; i++)  
{  
    U[i] = F(X[i], Y[i], Z[i]);  
    V[i] = G(X[i], Y[i], Z[i]);  
}
```

## Loop Fusion

- One way to look at things...
  - Zipping: reorganizes data for **thrust** processing
  - Fusing: reorganizes computation for efficient **thrust** processing

# Fusing Transformations

```
typedef thrust::tuple<float,float> Tuple2;
typedef thrust::tuple<float,float,float> Tuple3;

struct linear_combo {
    __host__ __device__
    Tuple2 operator()(Tuple3 t) {
        float x, y, z; thrust::tie(x,y,z) = t;
        float u = 2.0f * x + 3.0f * y + 4.0f * z;
        float v = 1.0f * x + 2.0f * y + 3.0f * z;
        return Tuple2(u,v);
    }
};

void main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3), V(3);

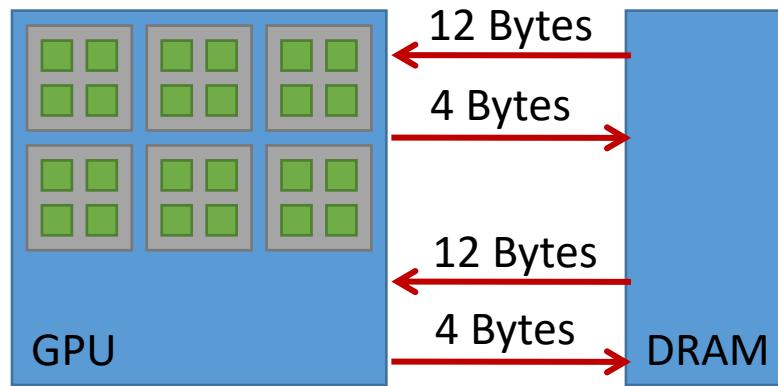
    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;
    Z[0] = 20; Z[1] = 30; Z[2] = 25;

    thrust::transform
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         thrust::make_zip_iterator(thrust::make_tuple(U.begin(), V.begin())),
         linear_combo());
}
```

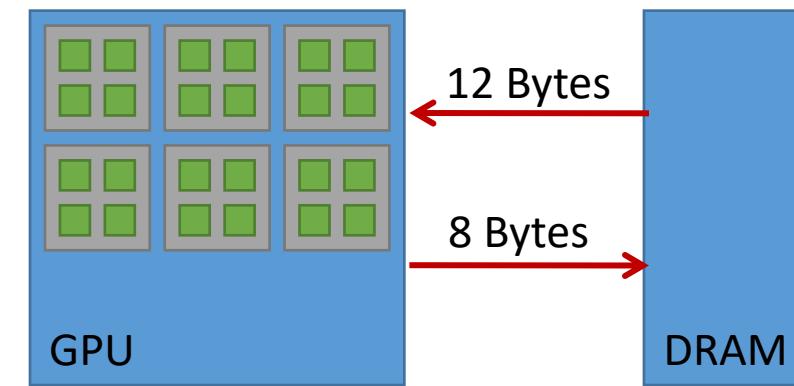
Functor  
Definition

# Fusing Transformations in Previous Example

Original Implementation



Optimized Implementation



- Since the operation is completely memory bound the expected speedup is  $\sim 1.6x$  ( $= 32/20$ )

# Fusing Transformations

```
for (int i = 0; i < N; i++)
    Y[i] = F(X[i]);
```

```
for (int i = 0; i < N; i++)
    sum += Y[i];
```

```
for (int i = 0; i < N; i++)
    sum += F(X[i]);
```

Loop Fusion

# Fusing Transformations

```
#include <thrust/device_vector.h>
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

int main(void) {
    thrust::device_vector<float> X(3);

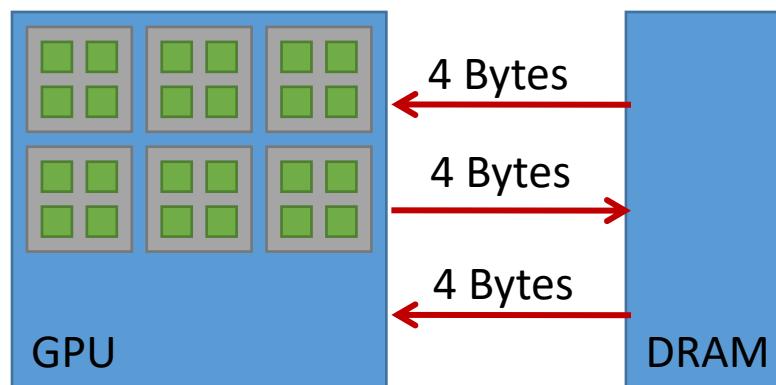
    X[0] = 10; X[1] = 30; X[2] = 20;

    float result = thrust::transform_reduce
        (X.begin(), X.end(),
         _1 * _1,
         0.0f,
         thrust::plus<float>());

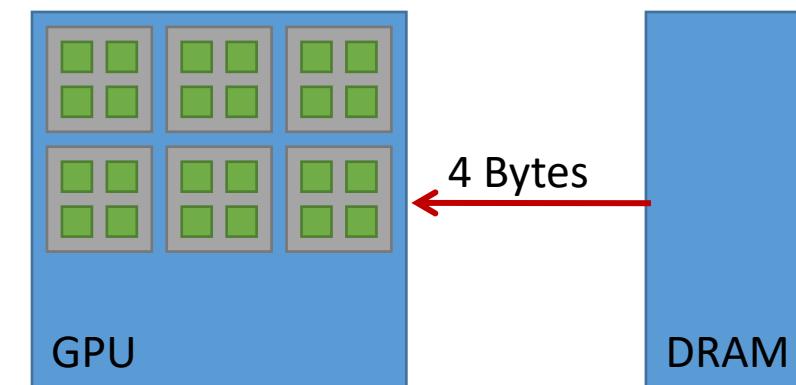
    std::cout << "sum of squares is " << result << "\n";
    return 0;
}
```

# Fusing Transformations in Previous Example

Original Implementation



Optimized Implementation



- Try to answer this: how many times will we be able to run faster if we fuse?

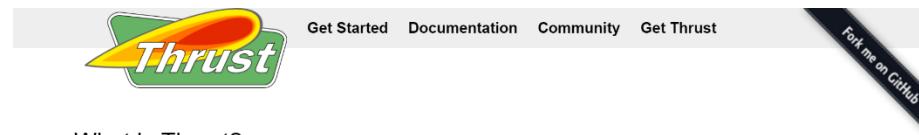
# thrust Wrap-Up

- Good productivity and execution boost at the price of having to deal with C++ syntax
  - No need to be aware of execution configuration, shared memory, etc.
- Key concepts
  - Functor
  - Zipping data
  - Fusing operations
- Why not always use `thrust`?
  - There is no “perform finite element analysis” support in `thrust`
  - Thrust provides support for primitives – up to us to use them as needed

# thrust on GitHub

- Quick Start Guide
- Examples
- News
- Documentation
- Mailing List (thrust-users)

<http://thrust.github.io/>



## What is Thrust?

Thrust is a parallel algorithms library which resembles the C++ Standard Template Library (STL). Thrust's **high-level** interface greatly enhances programmer **productivity** while enabling performance portability between GPUs and multicore CPUs. **Interoperability** with established technologies (such as CUDA, TBB, and OpenMP) facilitates integration with existing software. Develop **high-performance** applications rapidly with Thrust!

## Recent News

- [Thrust v1.8.1 release](#) (18 Mar 2015)
- [Thrust v1.8.0 release](#) (12 Jan 2015)
- [Thrust v1.7.0 release](#) (02 Jul 2013)
- [Thrust Content from GTC 2012](#) (12 May 2012)
- [Thrust v1.6.0 release](#) (07 Mar 2012)
- [Thrust v1.5.1 release](#) (30 Jan 2012)
- [Thrust v1.5.0 release](#) (28 Nov 2011)
- [Thrust v1.3.0 release](#) (05 Oct 2010)

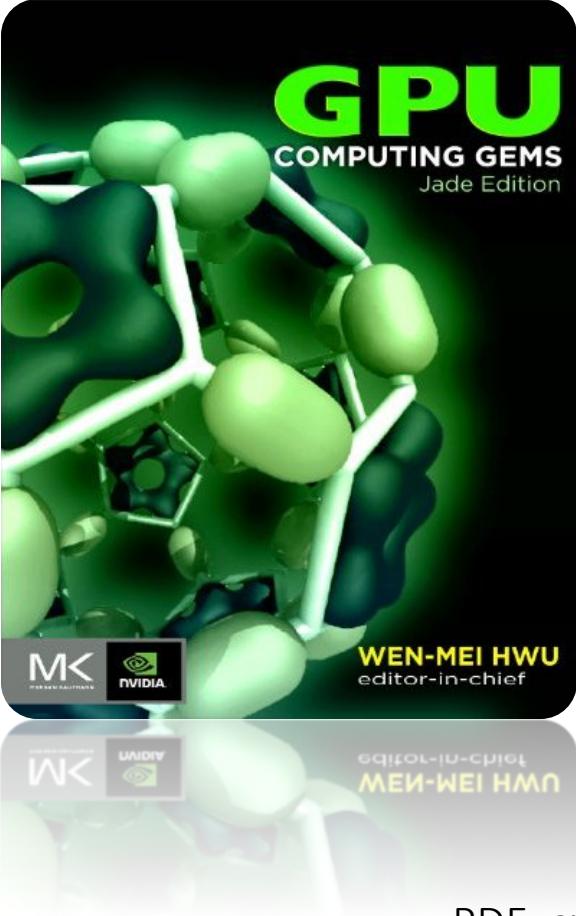
[View all news »](#)

## Examples

Thrust is best explained through examples. The following source code generates random numbers serially and then transfers them to a parallel device where they are sorted.

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/count.h>
```

# thrust in “GPU Computing Gems”



CHAPTER

# 26

Nathan Bell and Jared Hoberock

This chapter demonstrates how to leverage the Thrust parallel template library to implement high-performance applications with minimal programming effort. Based on the C++ Standard Template Library (STL), Thrust brings a familiar high-level interface to the realm of GPU Computing while remaining fully interoperable with the rest of the CUDA software ecosystem. Applications written with Thrust are concise, readable, and efficient.

## 26.1 MOTIVATION

With the introduction of CUDA C/C++, developers can harness the massive parallelism of the GPU through a standard programming language. CUDA allows developers to make fine-grained decisions about how computations are decomposed into parallel threads and executed on the device. The level of control offered by CUDA C/C++ (henceforth CUDA C) is an important feature: it facilitates the development of high-performance algorithms for a variety of computationally demanding tasks which (1) merit significant optimization and (2) profit from low-level control of the mapping onto hardware. For this class of computational tasks CUDA C is an excellent solution.

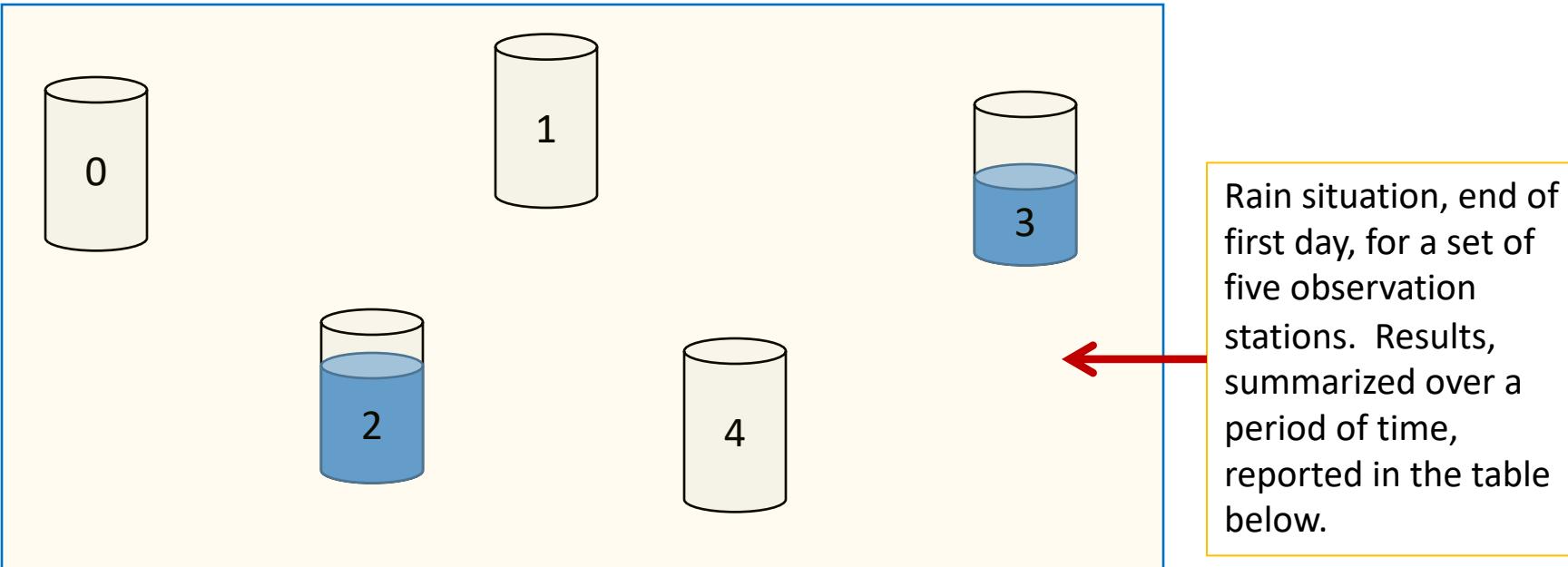
Thrust [1] solves a complementary set of problems, namely those that are (1) implemented efficiently without a detailed mapping of work onto the target architecture or those that (2) do not merit or simply will not receive significant optimization effort by the user. With Thrust, developers describe their computation using a collection of *high-level* algorithms and completely *delegate* the decision of how to implement the computation to the library. This abstract interface allows programmers to describe *what to compute* without placing any additional restrictions on how to carry out the computation. By capturing the programmer’s intent at a high level, Thrust has the discretion to make informed decisions about how to map the computation onto the target architecture. The library automatically handles the details of memory management, thread synchronization, and other low-level concerns, allowing the developer to focus on the computation itself. This abstraction layer makes it easier to port code between different hardware platforms and to reuse code across multiple applications. It also makes it easier to experiment with different parallel execution models, such as SIMD or vectorized execution, without having to rewrite the underlying computation. This is particularly useful for research and prototyping, where the focus is on exploring different algorithmic approaches rather than optimizing them for a specific hardware configuration.

PDF available at <http://goo.gl/adj9S>

# Work Plan, **thrust**

- Namespaces, containers, iterators
- Algorithms
- General transformations. Zipping & fusing
- Thrust example: Processing rainfall data

# Example, **thrust**: Processing Rainfall Data



day	[0	0	1	2	5	5	6	6	7	8	... ]
site	[2	3	0	1	1	2	0	1	2	1	... ]
measurement	[9	5	6	3	3	8	2	6	5	9	... ]

Remarks:

- 1) Time series sorted by day
- 2) Measurements of zero are excluded from the time series

# Example: Processing Rainfall Data

```
day      [0  0  1  2  5  5  6  6  7  8  ... ]  
site     [2  3  0  1  1  2  0  1  2  1  ... ]  
measurement [9  5  6  3  3  8  2  6  5  9  ... ]
```

- Given the data above, here're some questions you might ask:
  - Total rainfall at a given site
  - Total rainfall between given days
  - Total rainfall on each day
  - Number of days with any rainfall

# Total Rainfall at a Given Site

```
struct one_site_measurement
{
    int siteOfInterest;

    one_site_measurement(int site) : siteOfInterest(site) {}

    __host__ __device__
    int operator()(thrust::tuple<int,int> t)
    {
        if (thrust::get<0>(t) == siteOfInterest)
            return thrust::get<1>(t);
        else
            return 0;
    }
};

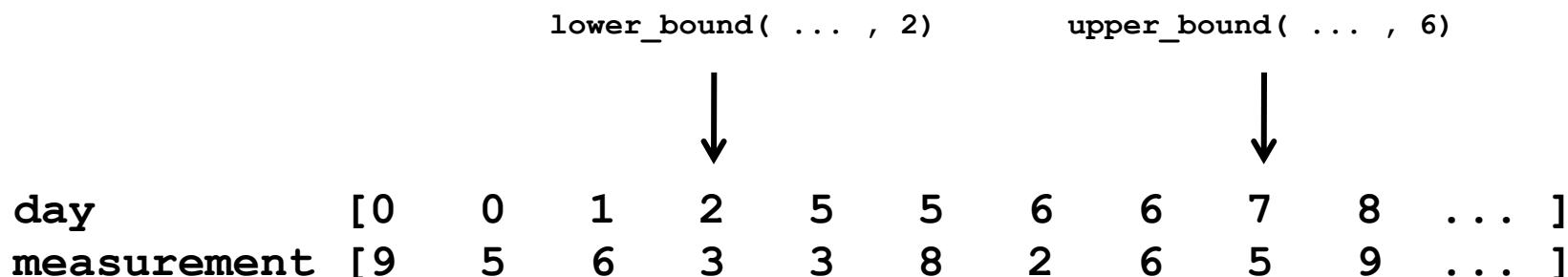
template <typename Vector>
int compute_total_rainfall_at_one_site(int siteID, const Vector& site, const Vector& measurement)
{
    return thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(site.begin(), measurement.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(site.end(), measurement.end())),
         one_site_measurement(siteID),
         0,
         thrust::plus<int>());
}
```

Functor  
Definition

# Total Rainfall Between Given Days

```
template <typename Vector>
int compute_total_rainfall_between_days(int first_day, int last_day,
                                         const Vector& day, const Vector& measurement)
{
    int first = thrust::lower_bound(day.begin(), day.end(), first_day) - day.begin();
    int last = thrust::upper_bound(day.begin(), day.end(), last_day) - day.begin();

    return thrust::reduce(measurement.begin() + first, measurement.begin() + last);
}
```



You will need to include several header files  
(not all for the code snippet above)

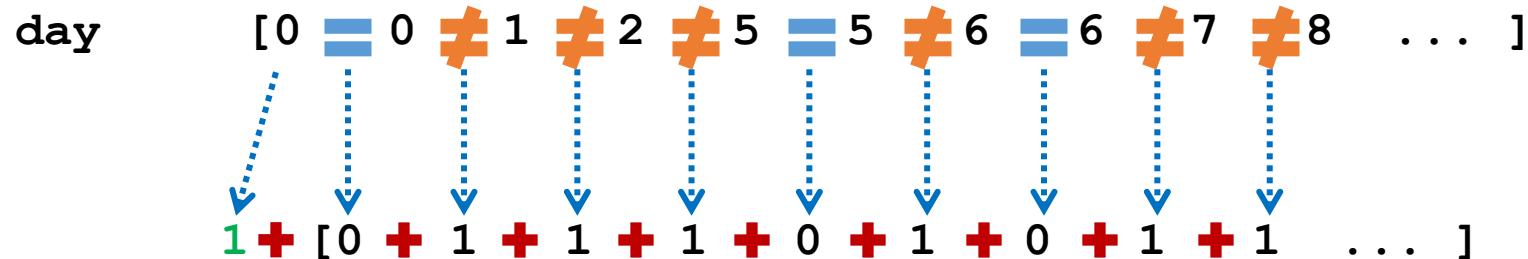
```
#include <thrust/device_vector.h>
#include <thrust/binary_search.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>
```

# Number of Days with Any Rainfall

```
template <typename Vector>
int compute_number_of_days_with_rainfall(const Vector& day)
{
    return thrust::inner_product(day.begin(), day.end() - 1,
                                day.begin() + 1,
                                0,
                                thrust::plus<int>(),
                                thrust::not_equal_to<int>()) + 1;
}
```

$\oplus$

$\otimes$



$$\text{inner\_product}(x, y) = (x_0 \otimes y_0) \oplus (x_1 \otimes y_1) \oplus (x_2 \otimes y_2) \oplus \dots$$

# Total Rainfall on Each Day

```
template <typename Vector>
void compute_total_rainfall_per_day(const Vector& day, const Vector& measurement,
                                    Vector& day_output, Vector& measurement_output)
{
    size_t N = compute_number_of_days_with_rainfall(day); //see previous slide

    day_output.resize(N);
    measurement_output.resize(N);

    thrust::reduce_by_key(day.begin(), day.end(),
                         measurement.begin(),
                         day_output.begin(),
                         measurement_output.begin());
}
```

<b>day</b>	[0	= 0	1	2	5	= 5	6	= 6	7	8	...	]
<b>measurement</b>	[9	+ 5	6	3	3	+ 8	2	+ 6	5	9	...	]



<b>day_output</b>	[0	1	2	5	6	7	8	...	]
<b>measurement_output</b>	[14	6	3	11	8	5	10	...	]

# Final Project Issues

# Default Final Project topic suggested by Dan

- Work on a parallel code used in Europe and Japan to deploy rover on moon of Mars
- As is, the code is too slow
- Tasks:
  - Profiling the code
  - Suggest speed improvements
  - Implement some of your suggestions for speed improvements

# Intel-suggested Final Project topics

- Coming from group at Intel made up of 13 compiler engineers working on optimizing machine learning workloads on various hardware platforms
- Work w/ specialized hardware groups to ensure fast execution of inference and training workloads through hardware-software codesign
  - See <https://github.com/plaidml/plaidml>
- Currently working closely with the LLVM community and the MLIR group at google to contribute to the affine dialect in MLIR
  - See <https://mlir.llvm.org/docs/Dialects/AffineOps>
- Researching repurposing machine learning optimization techniques for scientific workloads

# Default Final Project topics suggested by Intel group

Topic	Description	Helpful References
LLVM Optimization Analysis	Analyze LLVM Optimization Passes using established benchmarks	<ol style="list-style-type: none"><li>1. <a href="https://rd.springer.com/content/pdf/10.1007%2F978-3-319-91479-4_23.pdf">https://rd.springer.com/content/pdf/10.1007%2F978-3-319-91479-4_23.pdf</a></li></ol>
Survey and Analysis of Tensor Compilers	Analyze and benchmark tensor compilers for ML across various HW OR Investigate the results produced in 2.	<ol style="list-style-type: none"><li>1. <a href="https://arxiv.org/abs/2002.03794">https://arxiv.org/abs/2002.03794</a></li><li>2. <a href="http://www.eecs.harvard.edu/~htk/publication/2019-mapl-tillet-kung-cox.pdf">http://www.eecs.harvard.edu/~htk/publication/2019-mapl-tillet-kung-cox.pdf</a></li></ol>
High performance GEMM library for a GPU	Utilizing the techniques described in papers 1 and 2 on the right And the MLIR's new support for affine.parallel: (see 3 on the right) Implement a high performance GEMM library for a GPU	<ol style="list-style-type: none"><li>1. <a href="https://arxiv.org/abs/2003.00532">https://arxiv.org/abs/2003.00532</a></li><li>2. <a href="https://paperswithcode.com/paper/stripe-tensor-compilation-via-the-nested">https://paperswithcode.com/paper/stripe-tensor-compilation-via-the-nested</a></li><li>3. <a href="https://mlir.llvm.org/docs/Dialects/AffineOps/#affineparallel-affineparallellop">https://mlir.llvm.org/docs/Dialects/AffineOps/#affineparallel-affineparallellop</a></li></ol>
Distributed SLIDE	Reproduce the results presented for SLIDE in 1. Using the code provided on github (2) Implement a distributed version of SLIDE.	<ol style="list-style-type: none"><li>1. <a href="https://arxiv.org/pdf/1903.03129.pdf">https://arxiv.org/pdf/1903.03129.pdf</a></li><li>2. <a href="https://github.com/keroro824/HashingDeepLearning/tree/master/SLIDE">https://github.com/keroro824/HashingDeepLearning/tree/master/SLIDE</a></li></ol>

# Intel related Final Projects

- The Intel folks will read your Final Project
- Participate only if you are OK with this. Maybe you don't want to share your findings/work
- You'll have to explicitly indicate that you are ok with me sharing your Final Project with them
  - There will be an option in the Final Project template
    - Note: there will be two templates
      - Final Project Proposal
      - Final Project Report

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 19

03/06/2020

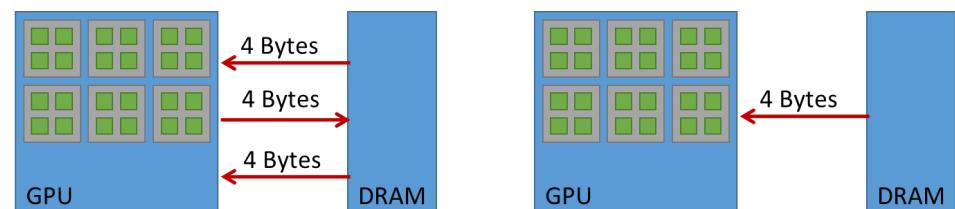
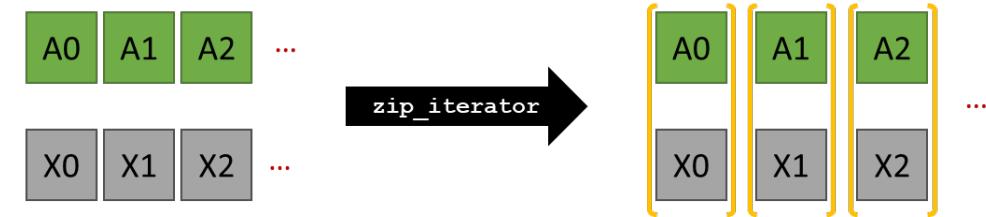
# Quote of the day

“The hardest thing of all is to find a black cat in a dark room, especially if there is no cat.”

-- Confucius, Chinese philosopher and politician [551 – 479 BC]

# Before we get going...

- Last time:
  - Off-the-shelf GPU computing: thrust
  - Further discussion, Final Project topics
- Today:
  - CUB
  - Multi-core parallel computing
    - Hardware consideration
    - Getting started with OpenMP
- Other tidbits:
  - ME759 Exam: April 15, at 7:15 PM, in 1106ME
    - Review: Tu, April 14, at 7:00 PM, in Canvas



# GPU Computing with CUB

# Hi, my name is CUB!

- People know me as CUB, but my real name is “CUDA UnBound”
- Currently, I’m at version V1.8.0 (DOB: 02/16/2018)
- I’m developed as open-source project by good folks at NVIDIA Research
  - Primary contributor: Duane Merrill
- You can fork me if you want (on GitHub: <https://github.com/NVlabs/cub>)
- Something about me: I can run really, really fast...

# CUB, factoids

- CUB provides software components at each layer of the CUDA programming model
- Like `thrust`, it's a headers library, not one binary file that you link against
- Implications, given that it's a header-file library:
  - When you need to call a function, you need to include the header file that includes the \*source\* code of the CUB function you want to call
  - Rationale, header-file library: if the compiler sees the code, it can engage in all sort of optimizations
    - Inlining, heavy use of templating, compiles for the exact architecture you have, etc.
  - Fallout: it takes forever to compile

# Straight from CUB webpage

- Device-wide primitives
  - Sort, prefix scan, reduction, histogram, etc.
- Block-wide “collective” primitives
  - I/O, sort, prefix scan, reduction, histogram, etc.
- Warp-wide “collective” primitives
  - Warp-wide prefix scan, reduction, etc.
- Thread and resource utilities
  - PTX intrinsics, device reflection, texture-caching iterators, caching memory allocators, etc.

# CUB ruminations

- The device-wide support is not a novelty, `thrust` does it too...
  - `thrust` built on top of CUB, by the way
- The novelty:
  - You can call CUB from a kernel function that uses the threads in one block to accomplish something
  - You can call CUB from a kernel function that uses the threads in a warp to accomplish something
    - The threads in a warp start behaving like the 32 threads of a chip with 32 cores running 32 threads

# Reiterating what CUB does

[slide hyperlinks should work...]

## 1. Parallel primitives

- [Warp-wide "collective" primitives](#)
  - Cooperative warp-wide prefix scan, reduction, etc.
  - Safely specialized for each underlying CUDA architecture
- [Block-wide "collective" primitives](#)
  - Cooperative I/O, sort, scan, reduction, histogram, etc.
  - Compatible with arbitrary thread block sizes and types
- [Device-wide primitives](#)
  - Parallel sort, prefix scan, reduction, histogram, etc.
  - Compatible with CUDA dynamic parallelism

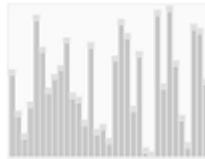
## 2. Utilities

- [Fancy iterators](#)
- [Thread and thread block I/O](#)
- [PTX intrinsics](#)
- [Device, kernel, and storage management](#)

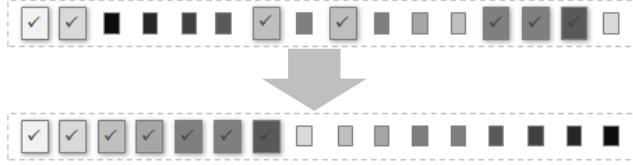
# CUB, device-wide operations

- You can call these from your C host code:

- Histogram



- Partition



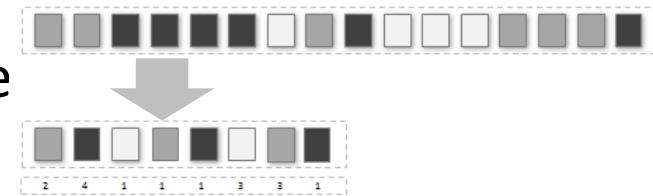
- Radix sort



- Reduce



- Run length encode



- Scan



- Select

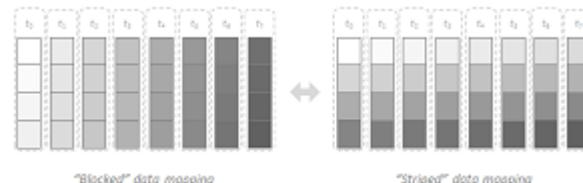
- Sparse matrix-vector multiplications

# CUB, block-wide operations [callable from a kernel function]

- BlockDiscontinuity class provides collective methods for flagging discontinuities within an ordered set of items partitioned across a CUDA thread block



- BlockExchange class provides collective methods for rearranging data partitioned across a CUDA thread block



- BlockHistogram

- BlockLoad class provides collective data movement methods for loading a linear segment of items from memory into a blocked arrangement across a CUDA thread block



- BlockRadixSort class provides collective methods for sorting items partitioned across a CUDA thread block using a radix sorting method

- BlockReduce

- BlockScan

- BlockStore class provides collective data movement methods for writing a blocked arrangement of items partitioned across a CUDA thread block to a linear segment of memory



# Example, back of the envelope, for BlockDiscontinuity

```
#include <cub/cub.cuh>

__global__ void ExampleKernel(...)

{
    // Specialize BlockDiscontinuity for a 1D block of 128 threads on type int
    typedef cub::BlockDiscontinuity<int, 128> BlockDiscontinuity;
    // Allocate shared memory for BlockDiscontinuity
    __shared__ typename BlockDiscontinuity::TempStorage temp_storage;
    // Obtain a segment of consecutive items that are blocked across threads
    int thread_data[4];
    ...
    // Collectively compute head flags for discontinuities in the segment
    int head_flags[4];
    BlockDiscontinuity(temp_storage).FlagHeads(head_flags, thread_data, cub::Inequality());
}
```

First important actor



Second important actor



# CUB, warp-level primitives

- Idea: each warp of threads will work on a chunk of an array and do a scan or reduce on that chunk of the array
- WarpScan class provides collective methods for computing a parallel prefix scan of items partitioned across a CUDA thread warp.
- WarpReduce class provides collective methods for computing a parallel reduction of items partitioned across a CUDA thread warp.

# Example, CUB: sort by key, device-level operation

```
#define CUB_STDERR // print CUDA runtime errors to console
#include <cub/util_allocator.cuh>
#include <cub/device/device_radix_sort.cuh>
#include "test/test_util.h"

using namespace cub;

// Caching allocator for device memory
CachingDeviceAllocator g_allocator(true);

struct pairsOfBodies {
    unsigned int body_I;
    unsigned int body_J;
};

int main() {
    const unsigned int num_items = 8;

    // host side setup
    pairsOfBodies h_vals[num_items];
    unsigned int h_keys[num_items] = { 2, 0, 7, 3, 5, 4, 1, 6 };
    pairsOfBodies h_vals_out[num_items];
    unsigned int h_keys_out[num_items];
    h_vals[0].body_I = 3; h_vals[0].body_J = 0;
    h_vals[1].body_I = 9; h_vals[1].body_J = 2;
    h_vals[2].body_I = 0; h_vals[2].body_J = 9;
    h_vals[3].body_I = 2; h_vals[3].body_J = 4;
    h_vals[4].body_I = 1; h_vals[4].body_J = 5;
    h_vals[5].body_I = 1; h_vals[5].body_J = 7;
    h_vals[6].body_I = 2; h_vals[6].body_J = 9;
    h_vals[7].body_I = 6; h_vals[7].body_J = 8;
```

```
// device side setup
unsigned int* d_keysIN = NULL;
unsigned int* d_keysOUT = NULL;
pairsOfBodies* d_valsIN = NULL;
pairsOfBodies* d_valsOUT = NULL;

CubDebugExit(g_allocator.DeviceAllocate((void**)& d_keysIN, sizeof(unsigned int) * num_items));
CubDebugExit(g_allocator.DeviceAllocate((void**)& d_keysOUT, sizeof(unsigned int) * num_items));
CubDebugExit(g_allocator.DeviceAllocate((void**)& d_valsIN, sizeof(pairsOfBodies) * num_items));
CubDebugExit(g_allocator.DeviceAllocate((void**)& d_valsOUT, sizeof(pairsOfBodies) * num_items));

// get memory set aside on the device
size_t temp_storage_bytes = 0;
void* d_temp_storage = NULL;
CubDebugExit(DeviceRadixSort::SortPairs(d_temp_storage, temp_storage_bytes, d_keysIN, d_keysOUT, d_valsIN, d_valsOUT, num_items));

// Caching allocator for device memory

CubDebugExit(g_allocator.DeviceAllocate(&d_temp_storage, temp_storage_bytes));
// initialize data on the device
CubDebugExit(cudaMemcpy(d_keysIN, h_keys, sizeof(unsigned int) * num_items, cudaMemcpyHostToDevice));
CubDebugExit(cudaMemcpy(d_valsIN, h_vals, sizeof(pairsOfBodies) * num_items, cudaMemcpyHostToDevice));
// do the actual sort
CubDebugExit(DeviceRadixSort::SortPairs(d_temp_storage, temp_storage_bytes, d_keysIN, d_keysOUT, d_valsIN, d_valsOUT, num_items));

// get data back
CubDebugExit(cudaMemcpy(h_keys_out, d_keysOUT, sizeof(unsigned int) * num_items, cudaMemcpyDeviceToHost));
CubDebugExit(cudaMemcpy(h_vals_out, d_valsOUT, sizeof(pairsOfBodies) * num_items, cudaMemcpyDeviceToHost));

// clean up
if (d_keysIN) CubDebugExit(g_allocator.DeviceFree(d_keysIN));
if (d_keysOUT) CubDebugExit(g_allocator.DeviceFree(d_keysOUT));
if (d_valsIN) CubDebugExit(g_allocator.DeviceFree(d_valsIN));
if (d_valsOUT) CubDebugExit(g_allocator.DeviceFree(d_valsOUT));
if (d_temp_storage) CubDebugExit(g_allocator.DeviceFree(d_temp_storage));

return 0;
}
```

# Example, CUB: reduce, device-level operation

```
#define CUB_STDERR // print CUDA runtime errors to console
#include <stdio.h>
#include <cub/util_allocator.cuh>
#include <cub/device/device_reduce.cuh>
#include "test/test_util.h"
using namespace cub;
CachingDeviceAllocator g_allocator(true); // Caching allocator for device memory

int main() {
    const size_t num_items = 10;
    // Set up host arrays
    int h_in[num_items] = { 2, 3, -1, 0, 3, 6, 7, 2, -2, 0 };
    int sum = 0;
    for (unsigned int i = 0; i < num_items; i++)
        sum += h_in[i];

    // Set up device arrays
    int* d_in = NULL;
    CubDebugExit(g_allocator.DeviceAllocate((void**)& d_in, sizeof(int) * num_items));
    // Initialize device input
    CubDebugExit(cudaMemcpy(d_in, h_in, sizeof(int) * num_items, cudaMemcpyHostToDevice));
    // Setup device output array
    int* d_sum = NULL;
    CubDebugExit(g_allocator.DeviceAllocate((void**)& d_sum, sizeof(int) * 1));

    // Request and allocate temporary storage
    void* d_temp_storage = NULL;
    size_t temp_storage_bytes = 0;
    CubDebugExit(DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in, d_sum, num_items));
    CubDebugExit(g_allocator.DeviceAllocate(&d_temp_storage, temp_storage_bytes));

    // Do the actual reduce operation
    CubDebugExit(DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in, d_sum, num_items));
    int gpu_sum;
    CubDebugExit(cudaMemcpy(&gpu_sum, d_sum, sizeof(int) * 1, cudaMemcpyDeviceToHost));
    // Check for correctness
    printf("\t%s\n", (gpu_sum == sum ? "Test passed." : "Test failed."));
    printf("\tSum is: %d\n", gpu_sum);

    // Cleanup
    if (d_in) CubDebugExit(g_allocator.DeviceFree(d_in));
    if (d_sum) CubDebugExit(g_allocator.DeviceFree(d_sum));
    if (d_temp_storage) CubDebugExit(g_allocator.DeviceFree(d_temp_storage));

    return 0;
}
```

# CUB, departing thoughts

- CUB is not the friendliest library to use, and the documentation is both dense and limited
- CUB provides amazing performance
- Use it when you start processing big data
  - It's worth spending the time to make friends with CUB
  - We use CUB in the lab

# Multi-core parallel computing with OpenMP

# Opportunities for Efficiency Gains

Cluster (distributed mem.)	Group of nodes communicate through fast interconnect	MPI, Charm++, Chapel
Multi-Socket Node	Group of CPUs on the same node, talk through main mem.	OpenACC, OpenMP, MPI
Acceleration (GPU/Phi)	Compute devices accelerating parallel computation on one node	CUDA, OpenCL, OpenMP, OpenACC
Multi-Core Processor	Communication through shared caches and main mem.	OpenMP, TBB, pthreads, OpenACC
Vectorization	Higher operation throughput via special/fat registers	AVX, SSE, OpenMP
Pipelining	Sequence of instruction sharing functional units	Assembly
Superscalar	Non-sequence instructions sharing functional units	Assembly

We have full control → 

We have little to no control → 

- Any pipeline issues that we can address?
- Do the threads work together harmoniously? Any NUMA issues to be aware of?
- Hitting the cache?
- Any opportunities for SIMD? Use GPU or AVX?
- Communication/synchronization reduced as much as possible?

# OpenMP: Step #1 – Know your hardware

# Feature Length on a Chip: Moore's Law at Work

- 2013 – 22 nm
- 2015 – 14 nm
- 2017 – 10 nm
- 2019 – 7 nm
- 2021 – 5 nm
- 2023 – 3 nm (Samsung?)

# Implications

- One can take this in two directions
  - Keep the number of transistors constant on a chip, but decrease its size (by a factor of  $\sqrt{2}$ ), or
  - Keep size constant, but increase computational power and/or smarts of the chip by adding more transistors

# Keeping the Transistor Count Constant

- A 12 cores chip, which was top of line in 2013, how much space will it take in the future?
- Back of the envelope numbers
  - Size of chip – assume a square of length “L”
    - 2013: L about 20 mm
    - 2015:  $L \approx 14$  mm
    - 2017:  $L \approx 10$  mm
    - 2019:  $L \approx 7$  mm
    - 2021:  $L \approx 5$  mm → a fifth of an inch fits on your phone

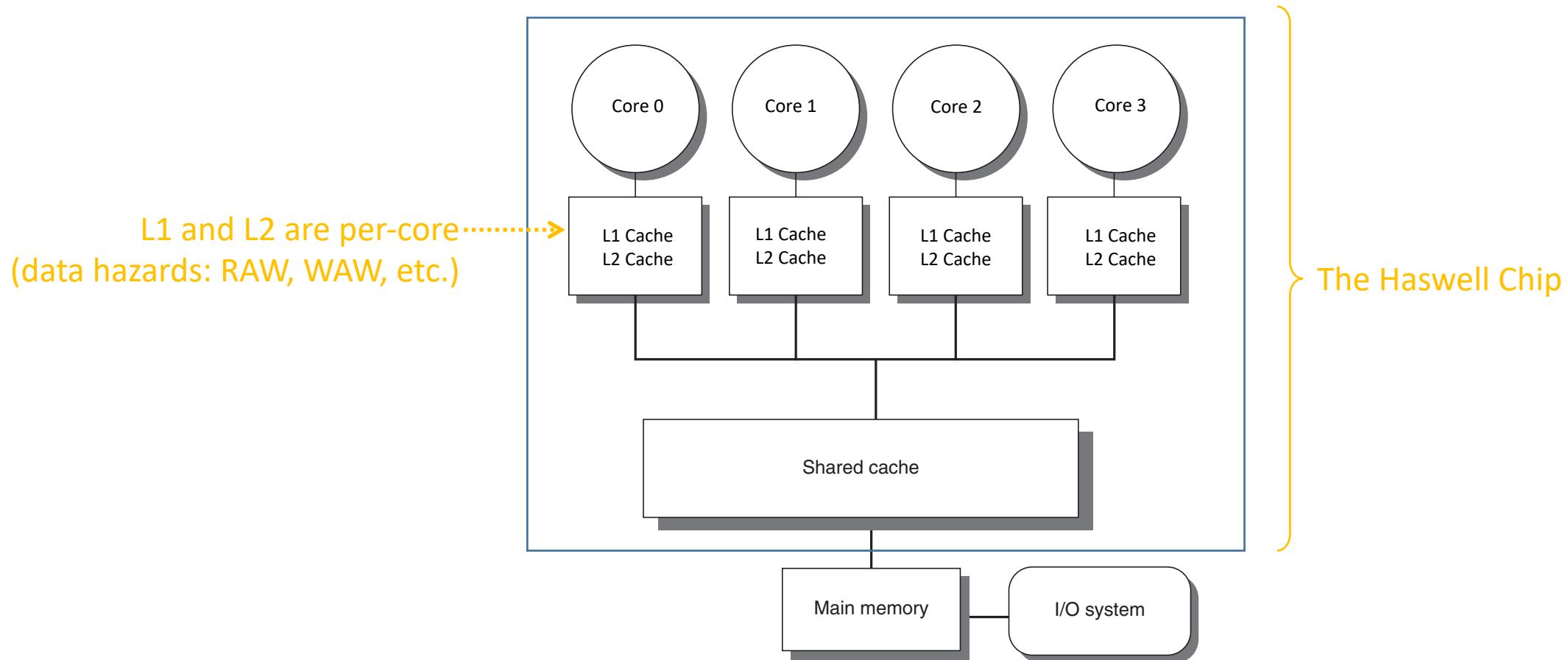
# Keeping the Area Constant

- October 2015:
  - Intel Xeon w/ 18 cores – 5.7 billion transistors (E7-8890 v3, \$7200)
- June 2017:
  - AMD EPYC – 32 cores, 19.2 billion transistors (EPYC 7601, \$4,200)
- February 2020:
  - AMD Ryzen™ Threadripper™ 3990X – 64 cores, 39.54 billion transistors, 288MB of cache, 7nm, \$3990

# Illustrating Hardware, OpenMP

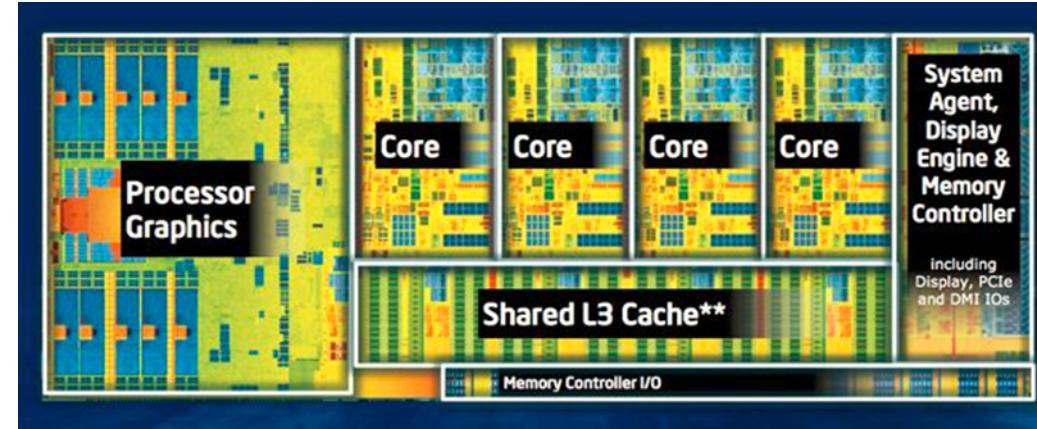
- Intel Haswell
  - Released in June 2013
  - 22 nm technology
  - Transistor budget: 1.4 billions
    - Tri-gate, 3D transistors
  - Typically comes in four cores
  - Has an integrated GPU
  - Deep pipeline – 16 stages
  - Sophisticated infrastructure for ILP acceleration
  - Superscalar
  - Supports HTT

# General Schematic, Haswell



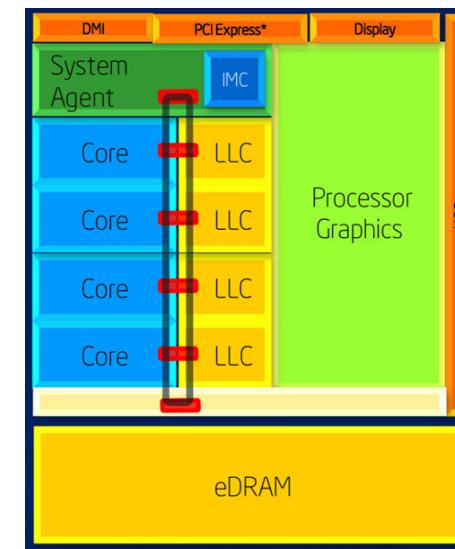
# Illustrating Hardware, Haswell

- Actual layout of the chip:



- Schematic of the chip organization

- LLC: last level cache (L3)
- Three clocks:
  - A core's clock ticks at 2.7 to 3.0 GHz but adjustable up to 3.7-3.9 GHz
  - Graphics processor ticking at 400 MHz but adjustable up to 1.3 GHz
  - Ring bus and shared L3 cache - a frequency that is close to but not necessarily identical to that of the cores



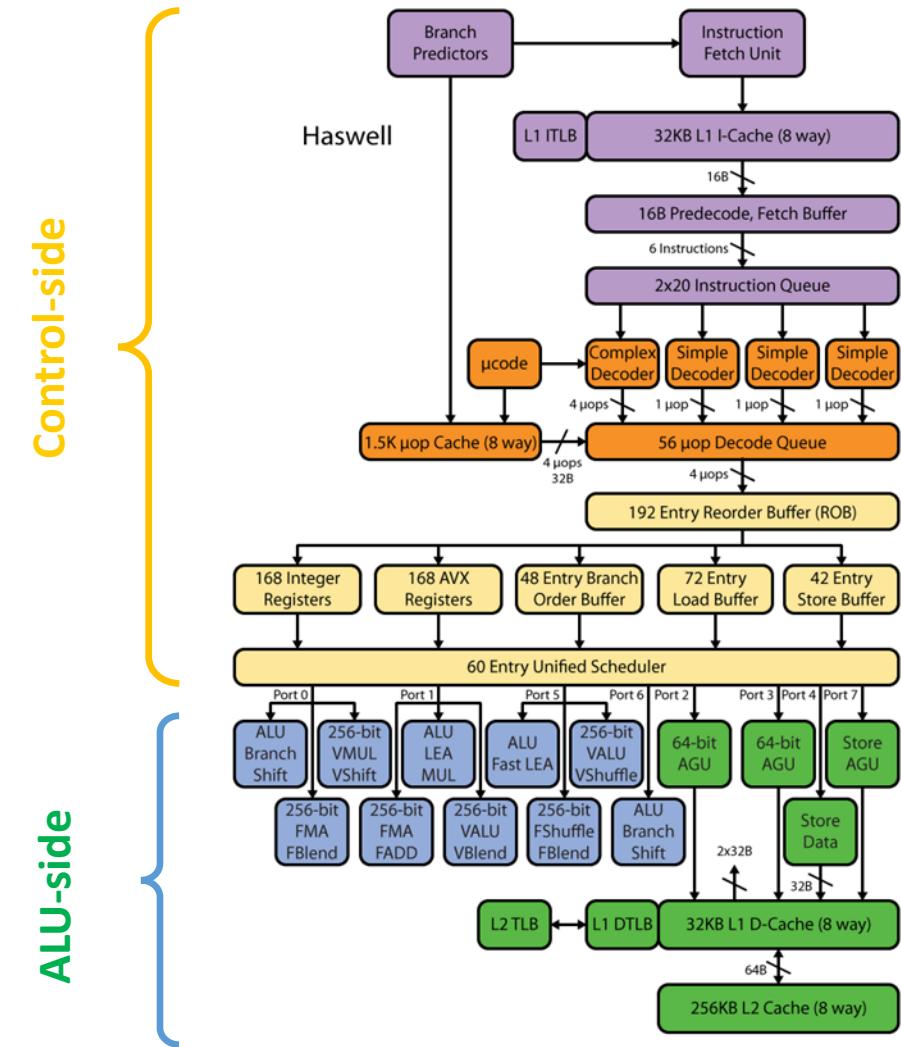
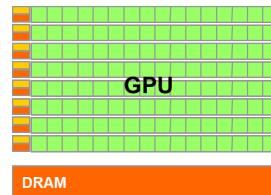
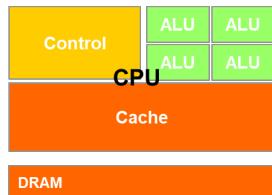
# Caches

- Data:
  - L1 – 32 KB per core
  - L2 – 512 KB or 1024 KB per core
  - L3 – 8 MB per CPU
    - This is LLC – last level cache for this chip
- Instruction:
  - L0 – room for about 1500 microoperations (uops) per core
  - L1 – 32 KB per core
- Cache is a black hole for transistors
  - Example: 8 MB of L3 translates into:
    - $8 * 1024 * 1024 * 8$  (bits) \* 6 (transistors per bit, SRAM) = 402 million transistors out of 1.4 billions

# Haswell Microarchitecture

[30,000 Feet]

- Microarchitecture components:
  - Instruction pre-fetch support (purple)
  - Instruction decoding support (orange)
    - CISC into uops
    - Can be regarded as CISC to RISC step
  - Instruction Scheduling support (yellowish)
  - Instruction execution
    - Arithmetic (blueish)
    - Memory related (green)



# Moving from HW to SW

# Acknowledgements

- Many OpenMP slides are from Intel's library of OpenMP presentations
  - Slides used with permission
  - Credit given where due: IOMPP
    - IOMPP stands for "Intel OpenMP Presentation"
- Several code examples originate in the Microsoft's Developer Network (MSDN) collection of OpenMP code snippets
- Other slides contain input from various sites on the internet
  - I tried whenever possible to give credit where due
  - Apologies for omitting any source

# OpenMP: When Used

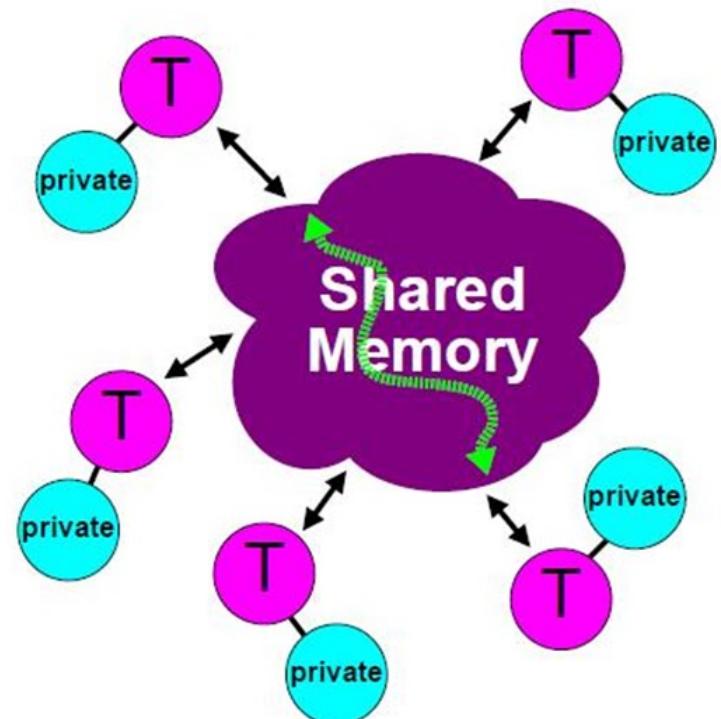
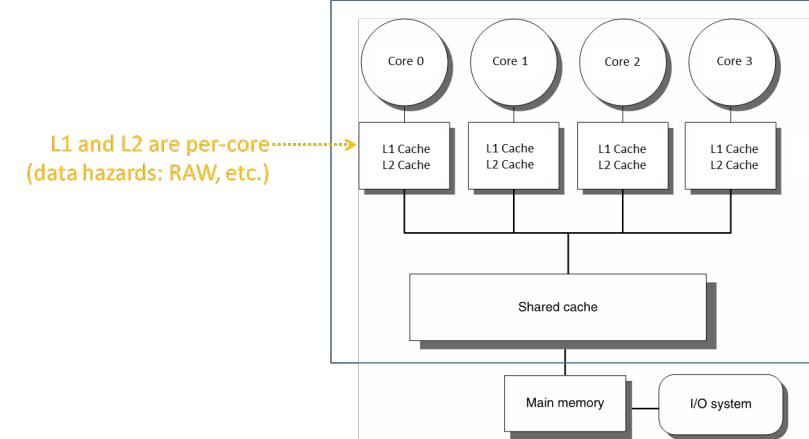
- CUDA: targeted parallelism on the GPU
- OpenMP: targets parallelism on SMP architectures
  - Handy when
    - You have a multi-core processor, say 16 cores/socket
    - Might have multiple sockets, say 2
    - You have a good amount of system memory, say 64 GB
- MPI: targeted parallelism on a cluster (distributed computing)
  - Note that MPI implementation can handle transparently an SMP architecture such as a workstation with two hexcore CPUs that draw on a good amount of shared memory

# process vs. thread

- How similar: Both processes and threads are independent sequences of execution
- How different:
  - Threads (of the same process) run in a **shared memory space**
    - However, each thread has its own PC
    - Who draws on it: OpenMP
  - Processes run in **separate memory spaces**
    - Each process has its
    - Who draws on it: MPI

# OpenMP Attributes

- Parallel execution of multiple tasks via multiple threads
- Threads have access to a pool of memory that is shared
- Threads can have private data
  - Not accessible by other threads
- When/How data is written/read from memory is transparent to programmer
  - However, programmer has some control (atomic, flush, etc.)
- Synchronization in thread execution is implicit but can be made explicit as well



# OpenMP: What's Reasonable to Expect

- If you have more than 16 cores or so in an SMP setup, it's unlikely that you can get a speed-up on that scale. All sorts of overheads kick in to slow you down
  - Beyond 16: law of diminishing return
- Some of the reasons for lack of scaling:
  - Cache coherence (more in a couple of lectures)
  - False cache sharing (more in a couple of lectures)
  - No over-commitment of hardware (recall CUDA/GPU computing and over-subscription of the SM hardware assets)

# Data vs. Task Parallelism

- **Data parallelism**
  - You have a large amount of data elements and each data element needs to be processed to produce a result
  - When this processing can be done in parallel, we have data parallelism
  - Example:
    - Adding two long arrays of doubles to produce yet another array of doubles
- **Task parallelism**
  - You have a collection of tasks that need to be completed
  - Task parallelism: these tasks can be performed in parallel
  - Example, task parallelism:
    - Prepare a soup, make a salad, bake a cake, microwave popcorn

# Objectives, ME759

- Understand OpenMP at the level where you can
  - Implement data parallelism
  - Implement task parallelism

# OpenMP: What Is It?

- Portable **programming environment** that invokes at run-time a shared-memory multi-threading execution of the code
  - Language bindings: Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows
  - Alternative to using `posix threads`
- Standardizes task & loop-level parallelism
- Combines serial and parallel code in single source

# OpenMP Mission Statement

“Standardize **directive-based** multi-language **high-level** parallelism that is performant, productive and portable.”

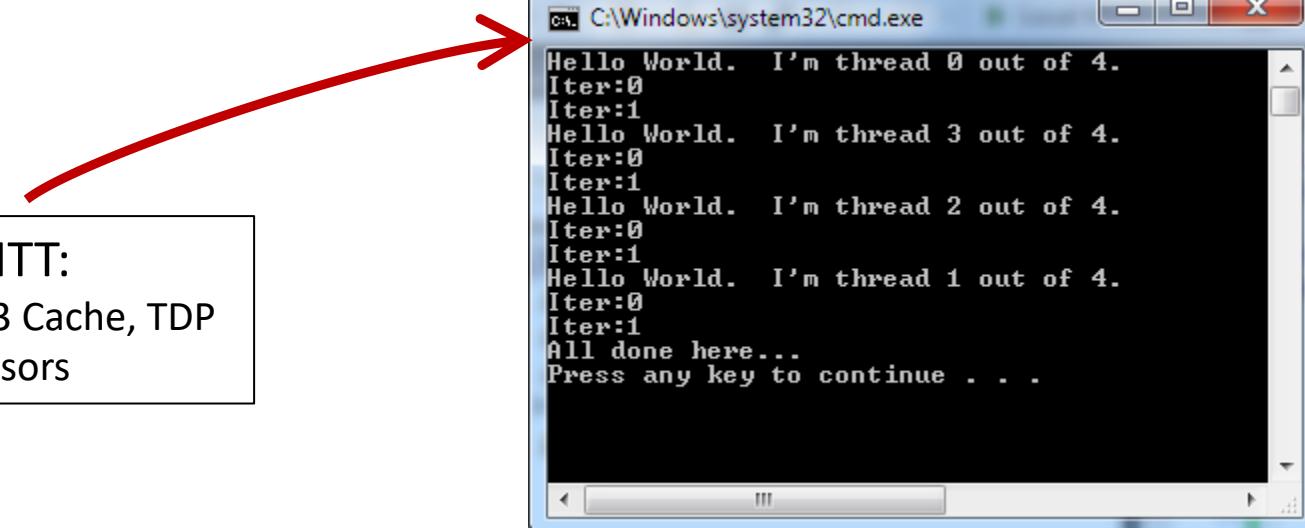
```
#include <stdio.h>
#include <omp.h>
```

```
int main() {
#pragma omp parallel
{
    int myId = omp_get_thread_num();
    int nThreads = omp_get_num_threads();

    printf("Hello World. I'm thread %d out of %d.\n", myId, nThreads);
    for( int i=0; i<2 ;i++ )
        printf("Iter:%d\n",i);
}
printf("All done here...\n");
}
```

## Example: Hello World

Here's a two core laptop, with HTT:  
Intel Core i5-3210M @ 2.50GHz 3 MB L3 Cache, TDP  
35 Watts, Two-Core Four-Thread Processors



# Comments, “Hello World!” Example

- OMP parallel region on previous slide just like a CUDA kernel
  - Each CUDA thread executes the kernel
  - In OpenMP, each thread executes the parallel region
- Important difference:
  - Variables inside GPU kernel are truly local variables, stored in registers
  - OMP variables in a parallel region may or may not be visible to other threads executing the code of the parallel region
    - Scoping of a variable in OpenMP is tricky
      - Discussed later
- An OpenMP structured block that marks a “parallel region” is rarely used like a kernel is in GPU computing
  - Most often, used in a different context (such as for work sharing, more to come)

# OpenMP: Historical Perspective

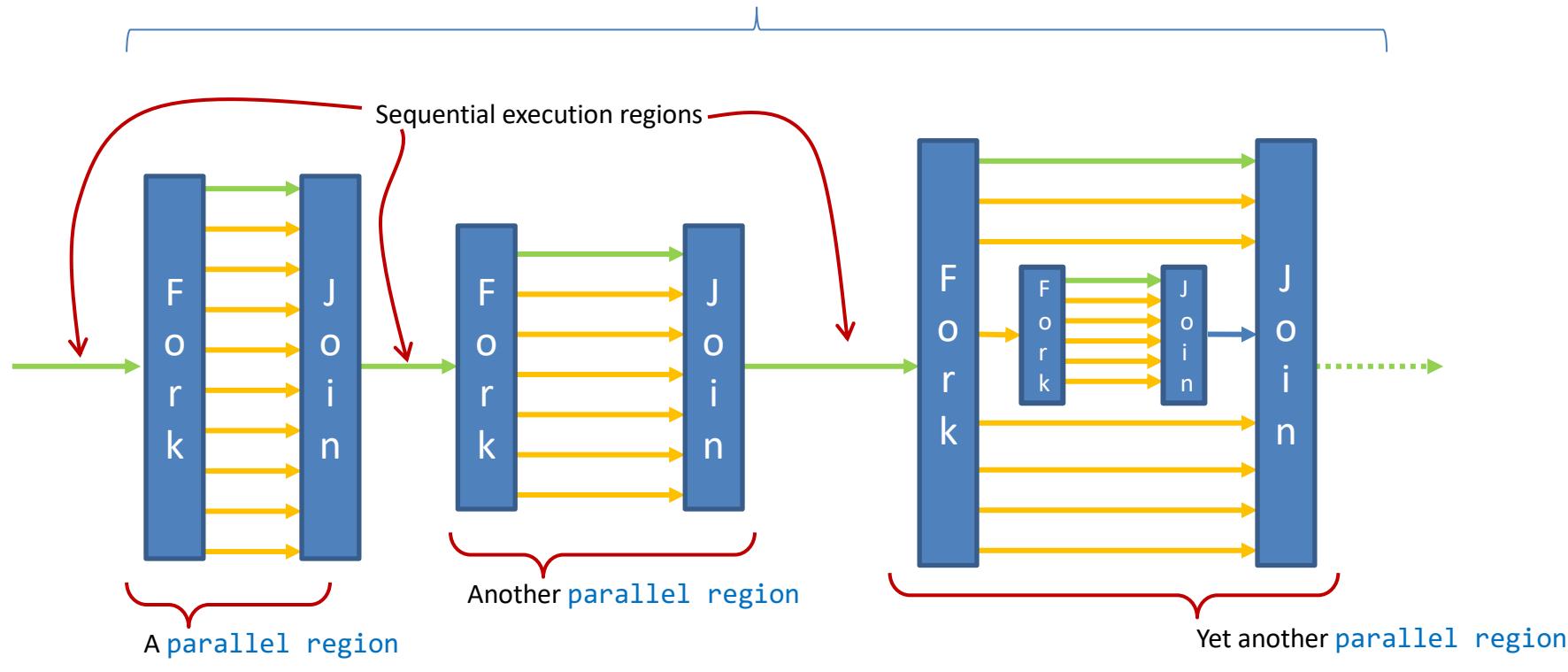
- Very good at coarse-grained, task parallelism
- Current spec is OpenMP 5.0
  - Released in November 2018, <http://www.openmp.org>
  - More than 600 Pages
  - Not yet implemented by any compiler
- Not all compilers are equally up-to-date w/ the standard
  - GNU compiler supports up to OpenMP 4.5 , but does not fully support GPU offloading
  - IBM XL compiler up to speed
  - llvm compiler supports 3.1 with some 4.0 and 4.5 features
  - Microsoft compiler lagging behind (Version 2.0 as of March 2020)

# OpenMP Programming Model

- **Master thread** spawns a **team of threads** as needed
  - Managed transparently on your behalf
  - Relies on low-level thread `fork/join` methodology to implement parallelism
    - The developer is spared the details
- Straightforward to ease into the use of OpenMP
  - Parallelism is added incrementally
  - The sequential program evolves into a parallel program

# OpenMP Execution Model

A small window into the execution flow of an OpenMP program



# OpenMP, Compiling Using the Command Line

- Method depends on compiler

- GCC:

```
$ gcc -o integrate_omp integrate_omp.c -fopenmp
```

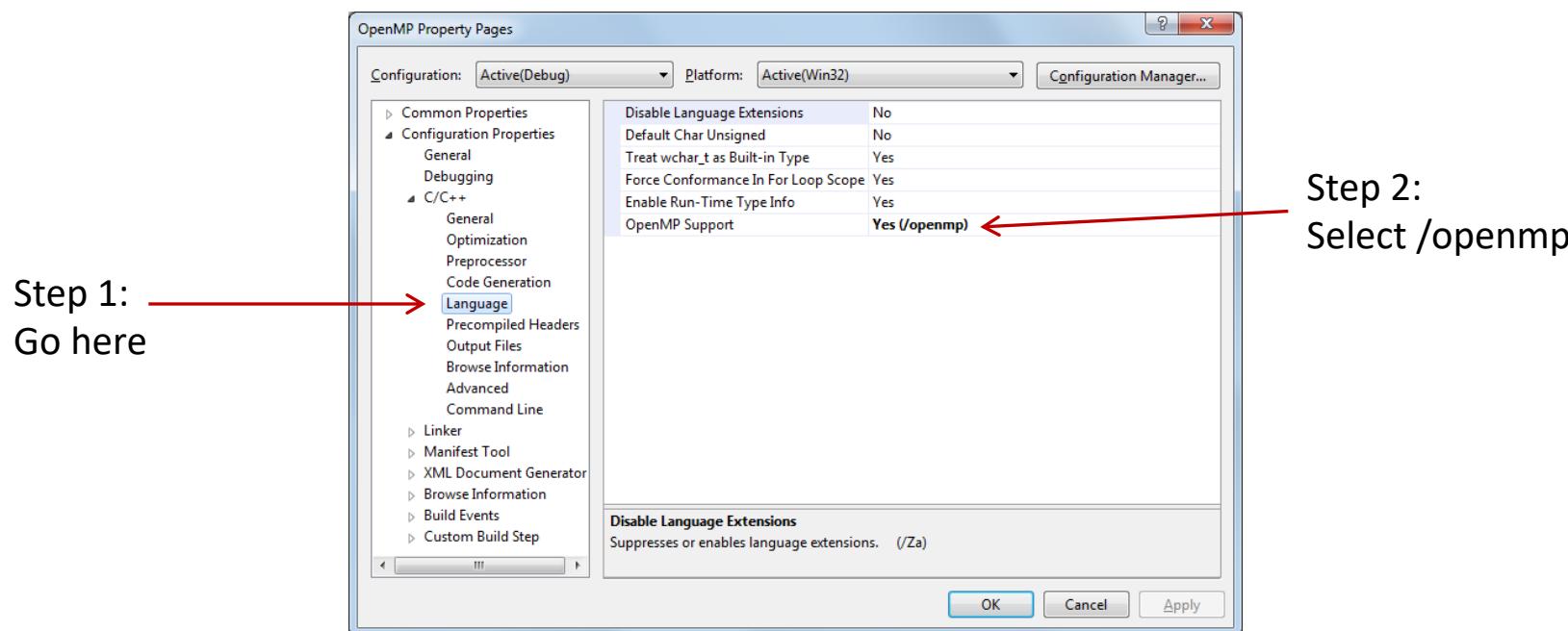
- ICC:

```
$ icc -o integrate_omp integrate_omp.c -openmp
```

- MSVC:

```
$ cl /openmp integrate_omp.c
```

# Visual Studio Specific



# Example: Calculate Entries in a Table

- Version 0: Fill up the table, use a for loop to visit each entry

```
#include <math.h>

int main()
{
    const int size = 256;
    double sinTable[size]; // sin table to be initialized

    for (int n = 0; n<size; ++n)
        sinTable[n] = sin(2 * M_PI * n / size);

    // the table is now initialized
}
```

# Example: Calculate Entries in a Table in Parallel w/ OpenMP

- Version 1:

- What gets used most of the time

```
#include "omp.h"
#include <math.h>

int main()
{
    const int size = 256;
    double sinTable[size]; // sin table to be initialized

#pragma omp parallel for
    for (int n = 0; n<size; ++n)
        sinTable[n] = sin(2 * M_PI * n / size);

    // the table is now initialized
}
```

# Example: Calculate Entries in a Table in Parallel w/ OpenMP

- Version 2:

- Uses wide registers & vector operations (CPU needs to have special chops to support this)
- Not supported by MS compiler

```
#include "omp.h"
#include <math.h>

int main()
{
    const int size = 256;
    double sinTable[size]; // sin table to be initialized

#pragma omp simd
    for (int n = 0; n<size; ++n)
        sinTable[n] = sin(2 * M_PI * n / size);

    // the table is now initialized
}
```

# Example: Calculate Entries in a Table in Parallel w/ OpenMP

- Version 3:

- Offloading code to different devices, such as a GPU
- Not supported by MS compiler

```
#include "omp.h"
#include <math.h>

int main()
{
    const int size = 256;
    double sinTable[size]; // sin table to be initialized

#pragma omp target teams distribute parallel for map(from:sinTable[0:256])
    for (int n = 0; n<size; ++n)
        sinTable[n] = sin(2 * M_PI * n / size);

    // the table is now initialized
}
```

# A Few Syntax Details to Get Started

- Picking up the OpenMP header file

```
#include "omp.h" (for C/C++)
use omp_lib      (for FORTRAN)
```

- Most OpenMP constructs are **compiler directives**

- C and C++: the **directives** take the form of **pragmas**:

```
#pragma omp construct [clause [clause]...]
```

- Fortran: the **directives** take one of the forms:

```
C$OMP construct [clause [clause]...]
```

```
!$OMP construct [clause [clause]...]
```

```
*$OMP construct [clause [clause]...]
```

# Why Compiler Directives?

- A code design principle nurtured by OpenMP:
  - The same code, with no modifications, can run with or without OpenMP
  - Accounts for the case when the OpenMP runtime is not available or not used
- Implications: you have to “hide” all the compiler directives behind pragmas (comments, in Fortran)
- Directives are picked up by the compiler only if instructed to do so
  - Example: Visual Studio – you have to have the /openmp flag on

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 20

03/09/2020

# Quote of the day

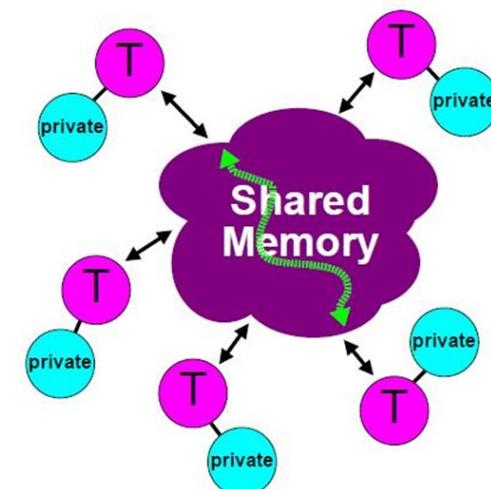
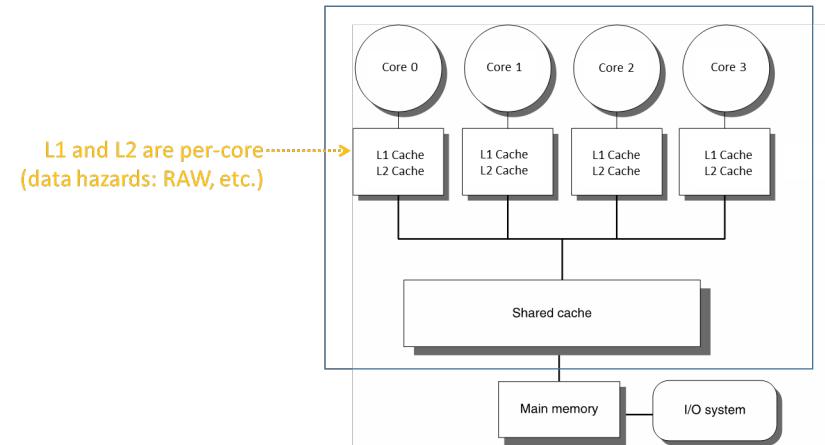
[on the day when the stock market dropped 1884 points on opening and triggered a trading halt]

“I lost our life savings in the stock market. Now let's move on to the real issue: Lisa's hogging of the maple syrup.”

-- Homer Simpson, Safety Inspector[1989 -].

# Before we get going...

- Last time:
  - CUB
  - Multi-core parallel computing
    - Hardware consideration
    - Getting started with OpenMP
- Today:
  - OpenMP generalities
  - Work-sharing constructs in OpenMP
- Other tidbits:
  - ME759 Exam: April 15, at 7:15 PM, in 1106ME
    - Review: Tu, April 14, at 7:00 PM, in Canvas
  - Next time: further discussion of default projects



# Compiler Directives, Examples

Directive	Description
<a href="#"><u>atomic</u></a>	Specifies that a memory location will be updated atomically
<a href="#"><u>barrier</u></a>	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier
<a href="#"><u>critical</u></a>	Specifies that code is only executed on one thread at a time
<a href="#"><u>flush</u></a>	Specifies that all threads have the same view of memory for all shared objects
<a href="#"><u>for</u></a>	Causes the work done in a for loop inside a parallel region to be divided among threads
<a href="#"><u>master</u></a>	Specifies that only the master thread should execute a section of the program
<a href="#"><u>ordered</u></a>	Specifies that code under a parallelized for loop should be executed like a sequential loop
<a href="#"><u>parallel</u></a>	Defines a parallel region, which is code that will be executed by multiple threads in parallel
<a href="#"><u>sections</u></a>	Identifies code sections to be divided among all threads
<a href="#"><u>single</u></a>	Indicates that a section of code should be executed on a single thread, not necessarily the master thread
<a href="#"><u>threadprivate</u></a>	Specifies that a variable is private to a thread

# OpenMP Beyond Directives: User-Level Runtime Routines

- Examples of function calls supported by OpenMP API:

- Modify/check the number of threads

```
omp_[set|get]_num_threads()  
omp_get_thread_num()  
omp_get_max_threads()
```

- Are we in a parallel region?

```
omp_in_parallel()
```

- How many processors in the system?

```
omp_get_num_procs()
```

- Explicit locks

```
omp_[set|unset]_lock()
```

- Many more...

# Example: Set/Get Number of Threads

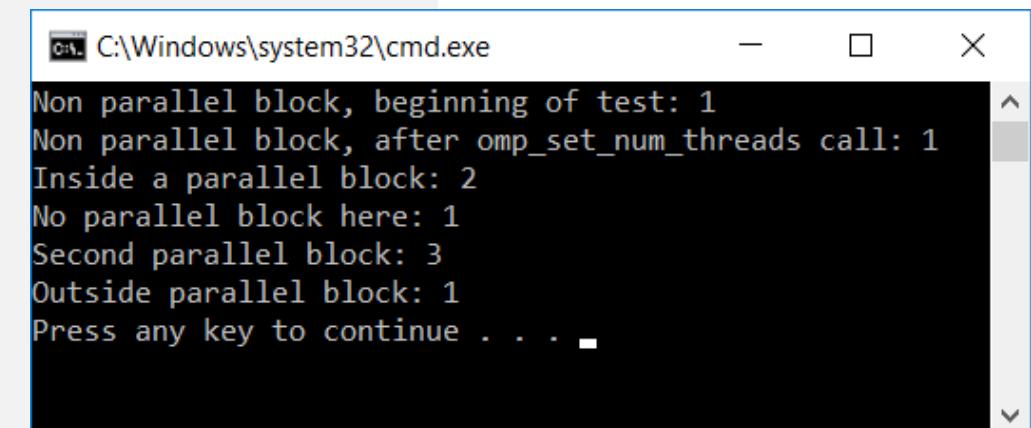
```
// omp_get_num_threads.c
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main() {
    printf_s("Non parallel block, beginning of test: %d\n", omp_get_num_threads());
    omp_set_num_threads(2); // NB: run-time action, sets OpenMP behavior
    printf_s("Non parallel block, after omp_set_num_threads call: %d\n", omp_get_num_threads());

#pragma omp parallel
#pragma omp master
{
    printf_s("Inside a parallel block: %d\n", omp_get_num_threads());

    printf_s("No parallel block here: %d\n", omp_get_num_threads());
    // changed the number of threads to be used inside parallel block;
    // to that end, use a compiler directive...
#pragma omp parallel num_threads(3) // NB: compile-time clause
#pragma omp master
{
    printf_s("Second parallel block: %d\n", omp_get_num_threads());
}

    printf_s("Outside parallel block: %d\n", omp_get_num_threads());
}
```



```
C:\Windows\system32\cmd.exe
Non parallel block, beginning of test: 1
Non parallel block, after omp_set_num_threads call: 1
Inside a parallel block: 2
No parallel block here: 1
Second parallel block: 3
Outside parallel block: 1
Press any key to continue . . .
```

# Example: Get Max Number of Threads

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

int main() {
    //omp_set_num_threads(8);
    printf_s("I can go w/ this many threads:%d\n", omp_get_max_threads());
#pragma omp parallel
#pragma omp master
{
    printf_s("Here's how many threads I use in this parallel region: %d\n", omp_get_num_threads());
}

#pragma omp parallel num_threads(3)
#pragma omp master
{
    printf_s("Max. number of threads: %d\n", omp_get_max_threads());
    printf_s("Actual number of threads used in this other parallel region: %d\n", omp_get_num_threads());
}

printf_s("Here's the max number of threads at end:%d\n", omp_get_max_threads());

return 0;
}
```

the maximum number of threads that can be used to form a new team if a parallel region without a num\_threads clause is encountered

```
C:\Windows\system32\cmd.exe
I can go w/ this many threads:12
Here's how many threads I use in this parallel region: 12
Max. number of threads: 12
Actual number of threads used in this other parallel region: 3
Here's the max number of threads at end:12
Press any key to continue . . .
```

Got this after uncommenting  
the first line in `main()` function

```
C:\Windows\system32\cmd.exe
I can go w/ this many threads:8
Here's how many threads I use in this parallel region: 8
Max. number of threads: 8
Actual number of threads used in this other parallel region: 3
Here's the max number of threads at end:8
Press any key to continue . . .
```

# OpenMP: Anchored by Three Pillars

- Pillar 1: Compiler directives [introduced already]
  - Allows for incremental parallelization of the code
  - A compiler that doesn't speak OpenMP simply ignores pragmas
    - Code runs just like before; i.e., sequentially
  - Directives have *clauses*, to further qualify a directive's behavior
- Pillar 2: User-level runtime function calls [introduced already]
  - Your compiler needs to link against the OpenMP library
  - You really need access to OpenMP, directives won't do
- Pillar 3: Environment variables [discussed now]
  - Another way of controlling OpenMP behavior
  - Provides some of the support that the OpenMP functions provide
  - Addresses portability issue – helps you bypass use of run-time function calls

# Example, two ways to accomplish the same thing

- Example 1: controlling the number of threads – the following are equivalent

- `omp_set_num_threads(8)`

- For the C shell:

- `setenv OMP_NUM_THREADS 8`

- For the Bash shell:

- `export OMP_NUM_THREADS=8`

- For the Windows shell:

- `set OMP_NUM_THREADS=8`

- Example 2: controlling nested parallelism – the following are equivalent

- `omp_set_nested(1);`

- `export OMP_NESTED=1`

# OpenMP: Environment Variables, two quick remarks

- Environment Variables team up with compiler directives so that you don't need to call any OpenMP function
  - Then, it is truly that if you don't have an OpenMP-aware compiler you are totally fine
    - The code will work sequentially, oblivious to any OpenMP directives you pepper in the code
- Good to know: A function call setting trumps an Environment Variable setting

# Function calls or Env Vars? Which one is better?

- Not all function calls in OpenMP API have an Environment Variables counterpart
  - Therefore, you can tune better behavior at run-time via function calls
    - There's a lot of function calls, not that many Environment Variables
- Limitation of Env Vars: once execution started, you can't dynamically change an Env Var setting
  - Can't change env variable `OMP_NUM_THREADS` while code runs

# OpenMP Env Variables, some examples...

<a href="#"><u>OMP_CANCELLATION</u></a> :	Set whether cancellation is activated
<a href="#"><u>OMP_DISPLAY_ENV</u></a> :	Show OpenMP version and environment variables
<a href="#"><u>OMP_DEFAULT_DEVICE</u></a> :	Set the device used in target regions
<a href="#"><u>OMP_DYNAMIC</u></a> :	Dynamic adjustment of threads
<a href="#"><u>OMP_MAX_ACTIVE_LEVELS</u></a> :	Set the maximum number of nested parallel regions
<a href="#"><u>OMP_MAX_TASK_PRIORITY</u></a> :	Set the maximum task priority value
<a href="#"><u>OMP_NESTED</u></a> :	Nested parallel regions
<a href="#"><u>OMP_NUM_THREADS</u></a> :	Specifies the number of threads to use
<a href="#"><u>OMP_PROC_BIND</u></a> :	Whether threads may be moved between CPUs
<a href="#"><u>OMP_PLACES</u></a> :	Specifies on which CPUs the threads should be placed
<a href="#"><u>OMP_STACKSIZE</u></a> :	Set default thread stack size
<a href="#"><u>OMP_SCHEDULE</u></a> :	How threads are scheduled
<a href="#"><u>OMP_THREAD_LIMIT</u></a> :	Set the maximum number of threads
<a href="#"><u>OMP_WAIT_POLICY</u></a> :	How waiting threads are handled

# OpenMP: Putting Things in Perspective

- OpenMP: portable and scalable model for shared memory parallel applications
  - No need to dive deep and work with POSIX `pthread`s
  - Code becomes portable
- Under the hood
  - Compiler translates OpenMP functions and `directives` to `pthread` calls
  - Program begins with a `master thread`
  - Master thread[s] forked when hitting a `parallel region`

# What's the Scope of a Directive?

- Directives: What does the OpenMP standard say?
  - “An OpenMP executable directive applies to the succeeding **structured block** or an **OpenMP construct**.”
- **structured block**: talk about it next
- **OpenMP construct**: defer discussion till “Work sharing” segment of lecture
- **structured block** and **OpenMP construct** are the two sides of the “parallel region” coin

# Structured Blocks (C/C++)

- **structured blocks**

- A block with one point of entry at the top and one point of exit at the bottom
- The only “branches” allowed are `exit()` function calls

## A structured block

```
#pragma omp parallel
{
    int id = omp_get_thread_num();

more: res[id] = do_big_job (id);

    if( not_conv(res[id]) )goto more;
}
printf ("I'm outside par. region!\n");
```

## Not a structured block

```
if (go_now()) goto more;
#pragma omp parallel
{
    int id = omp_get_thread_num();
more: res[id] = do_big_job(id);
    if ( conv (res[id]) ) goto done;
    goto more;
}
done: if (!really_done()) goto more;
```



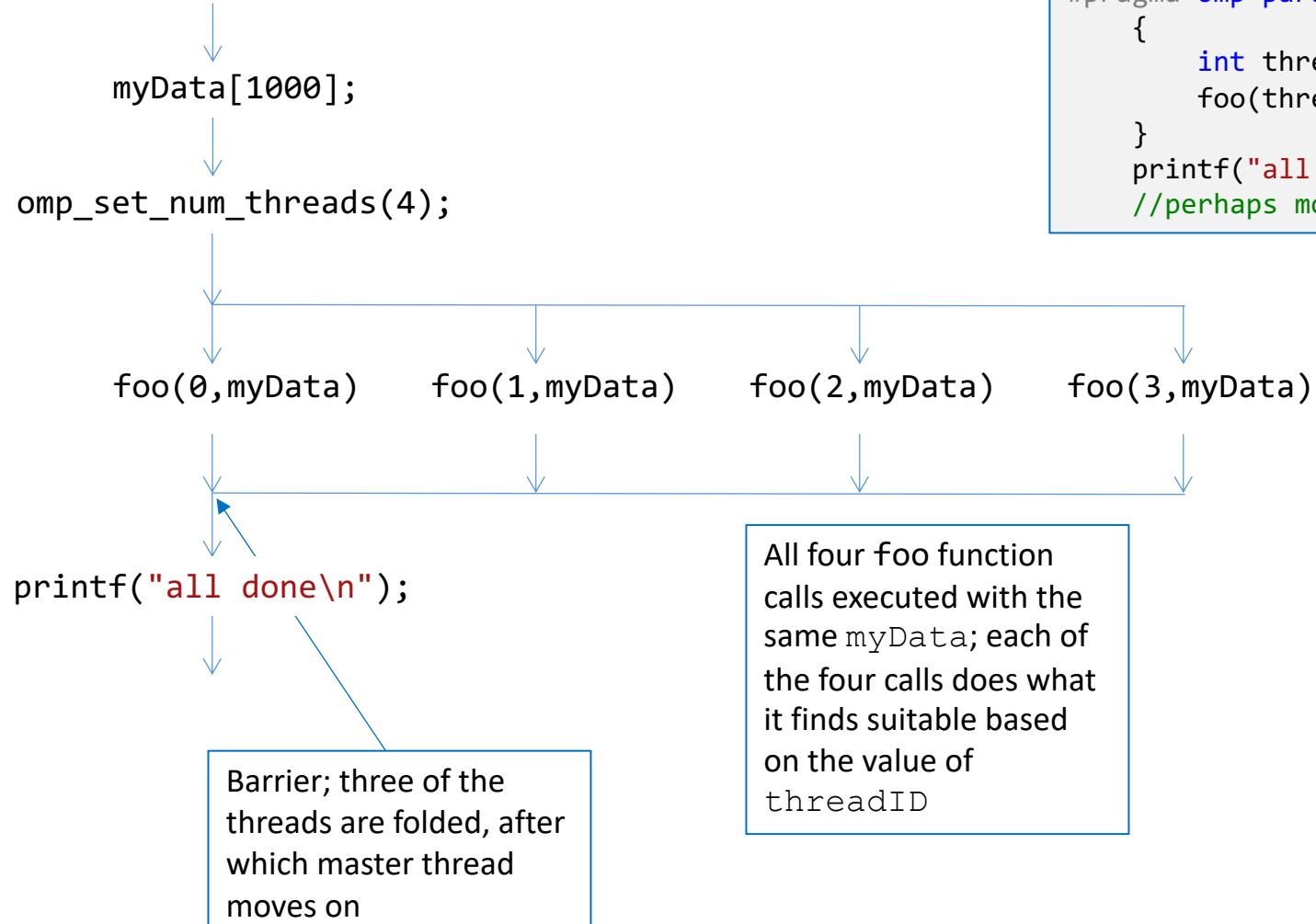
**IMPORTANT FACT:** There is an implicit barrier at the right “}” curly brace where threads wait on each other. This is the point at which the spawned worker threads complete execution and either go to sleep or spin idle.

# Example, Parallel Regions

- You ask the runtime to set up four threads
- Each of the four threads executes in parallel the code in between the curly brackets
- The threads wait for each other right before the `cout` statement, at which point all threads except master are folded back into pool
  - The right curly bracket “}”; i.e., end of parallel region, acts as a barrier
  - `printf` statement only executed by master
- Important: each thread has own “`threadID`”-type variable
  - Like `threadIdx.x` in CUDA – allows to identify what work you do

```
//perhaps more code here (defining foo, etc.)  
double myData[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int threadID = omp_get_thread_num();  
    foo(threadID, myData);  
}  
printf("all done\n");  
//perhaps more code here
```

# Another Way to Look at It



```
// perhaps more code here (defining foo, etc.)  
double myData[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int threadID = omp_get_thread_num();  
    foo(threadID, myData);  
}  
printf("all done\\n");  
//perhaps more code here
```

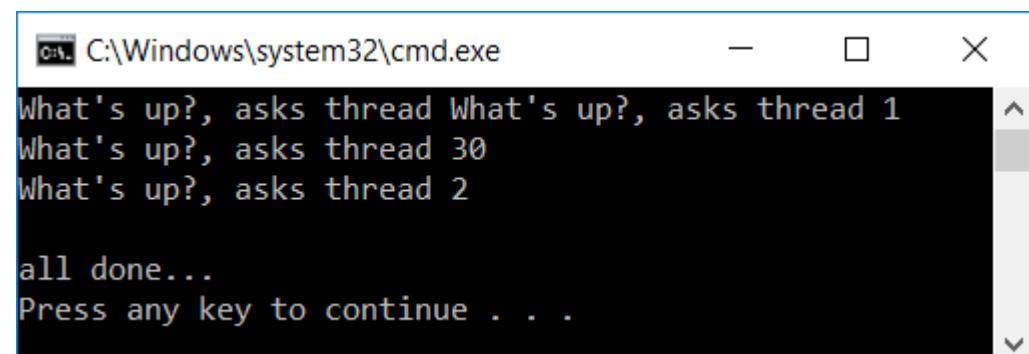
# Parallel Regions, Another Example

- What do you get when you execute this code?
- The code in the parallel region is like a GPU kernel (a “structured block”), which gets called by the number of parallel threads you work with

```
#include "omp.h"
#include <iostream>

void whatsUpQuestionMark()
{
    int myThreadID = omp_get_thread_num();
    std::cout << "What's up?, asks thread " << myThreadID << std::endl;
}

int main()
{
#pragma omp parallel num_threads(4)
    {
        whatsUpQuestionMark();
    }
    std::cout << "all done...\n";
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
What's up?, asks thread 1
What's up?, asks thread 30
What's up?, asks thread 2
all done...
Press any key to continue . . .
```

# whatsUpQuestionmark Example: What Happens Under the Hood

```
#pragma omp parallel num_threads(4)
{
    whatsUpQuestionMark();
}
```

- The OMP compiler generates code logically (qualitatively) analogous to that on the right of this slide, given an OMP pragma such as the one above
- The OMP runtime uses a thread pool so full cost of thread creation and retiring is not incurred with each parallel region
- Only *three* threads are created because one parallel-section invocation associated with master thread

```
void thunk() // wrapper, useful when having a function w/ arguments
{
    whatsUpQuestionMark();
}

pthread_t tid[4];

for (int i = 1; i < 4; ++i) //note that i starts at 1
    pthread_create(&tid[i], 0, thunk, 0); //create 3 threads

thunk(); // this is the call done by master thread

for (int i = 1; i < 4; ++i) //three threads folded; see Remarks below
    pthread_join(tid[i]);
```

- **Remarks**

- Thread creation happens perhaps before, don't need to create threads over and over again
- The threads might not be joined, they're folded back into a pool to avoid overhead when used in the next parallel region

# Timing an OpenMP Application

- What to pay attention to in example below:

- `double omp_get_wtime()` – returns a value in *seconds* of the time elapsed from some arbitrary, but consistent point in the past guaranteed not to change during execution of program
- `double omp_get_wtick()` – returns the number of seconds between clock ticks
  - Dictates the resolution that you can be expecting out of a timing call

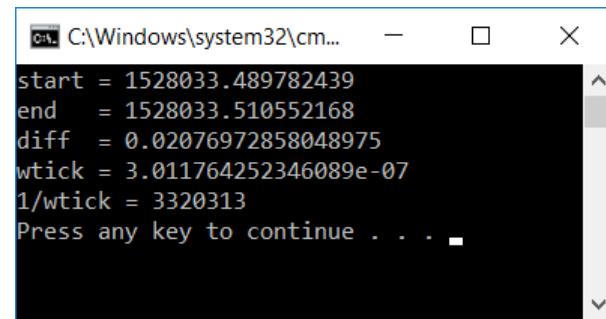
```
#include "omp.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main() {
    int const N = 100000;
    double dummy[N];

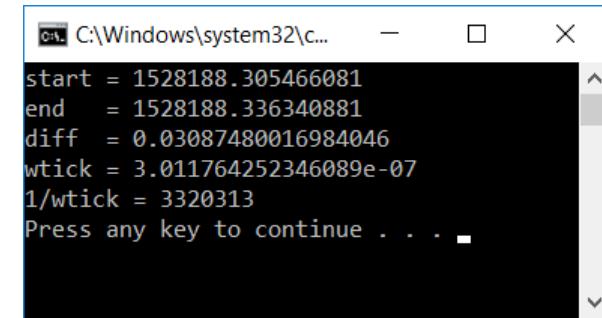
    double start = omp_get_wtime();
    for (int i = 0; i < N; i++) {
        int temp = rand();
        dummy[i] = 2.*temp / (pow(temp*temp, 1.5) + 0.2);
    }
    double end = omp_get_wtime();

    printf_s("start = %.16g\n", start);
    printf_s("end   = %.16g\n", end);
    printf_s("diff  = %.16g\n", end - start);

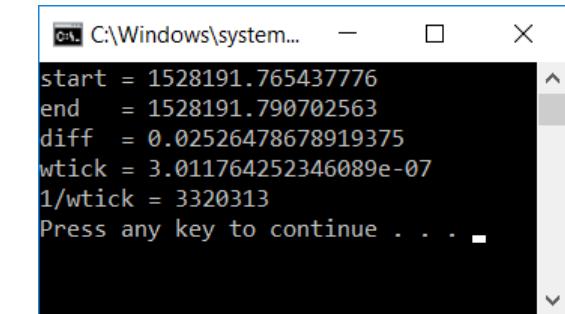
    double wtick = omp_get_wtick();
    printf_s("wtick  = %.16g\n", wtick);
    printf_s("1/wtick = %.16g\n", 1.0 / wtick);
}
```



```
start = 1528033.489782439
end   = 1528033.510552168
diff  = 0.02076972858048975
wtick = 3.011764252346089e-07
1/wtick = 3320313
Press any key to continue . . .
```



```
start = 1528188.305466081
end   = 1528188.336340881
diff  = 0.03087480016984046
wtick = 3.011764252346089e-07
1/wtick = 3320313
Press any key to continue . . .
```



```
start = 1528191.765437776
end   = 1528191.790702563
diff  = 0.02526478678919375
wtick = 3.011764252346089e-07
1/wtick = 3320313
Press any key to continue . . .
```

# [new topic] Nested Parallelism in OpenMP

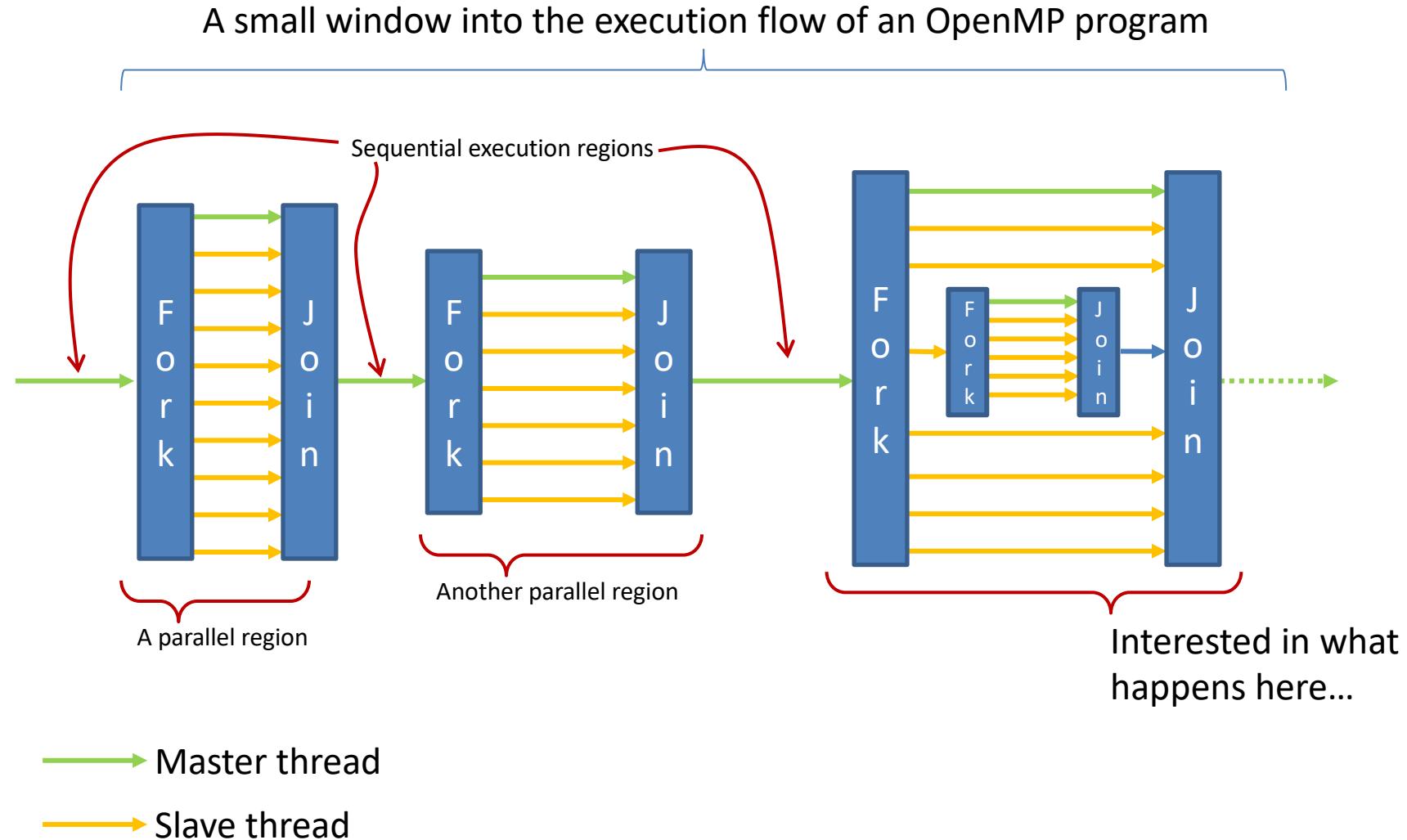
- **Backdrop:**

- You have a block of code that is executed in parallel
- Inside that block, there is a “for” loop that is executed in parallel

- **Questions:**

- What happens in this case?
- How can you control what happens?
- Is what happens desirable or undesirable?

# Nested Parallelism: A closer look



# Nested Parallelism: What Happens & How to Control

- When a thread, call it Joe, on a team of threads hits a parallel region, it spawns a collection of threads of which Joe is the master
- Is it always like this?
  - No, you can actually control behavior
    - If you prohibit the existence of nested parallel regions in your code, a parallel thread encountering a new parallel region will not be able to spawn new parallel threads
      - That embedded parallel region newly encountered will be executed by thread “Joe” only

# Nested Parallelism: What Happens & How to Control

- API to control nested parallelism behavior
  - Invoke library routine `omp_set_nested()`
  - `OMP_NESTED` environment variable
- Function call provides fine level of control
  - In some parts of the code nested parallelism is ok
  - Some other parts are better off without nested parallelism

# Relevant Actors, Nested Parallelism

- Relevant API elements
    - `omp_set_dynamic(...)`
    - `omp_get_dynamic(...)`
    - `omp_set_nested(...)`
    - `omp_get_nested(...)`
  - Note: There are equivalent clauses and/or environment variables that accomplish the same thing
- 
- Provides mechanism to control whether the number of threads available in subsequent parallel region can be adjusted by the runtime library. If argument is nonzero, the runtime library can adjust the number of threads. If argument is zero, the runtime library cannot dynamically adjust the number of threads.

# The OMP **single** directive

- **single** directive identifies a section of code that must be run by a single thread
- Example: see next slide,
  - Example highlights several features
    - **single** directive
    - **omp\_set\_dynamics (...)**
    - **omp\_set\_nested(...)**

# Example [wicked, think about what happens in the code below]

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

void report_num_threads(int whichLevel)
{
#pragma omp single
{
    printf("Level %d: number of threads in the team - %d\n", whichLevel, omp_get_num_threads());
}
}

int main() {
    omp_set_dynamic(1);
    omp_set_nested(1);

#pragma omp parallel num_threads(2)
{
    report_num_threads(1);
#pragma omp parallel num_threads(3)
{
    report_num_threads(2);
#pragma omp parallel num_threads(4)
{
    report_num_threads(3);
}
}
}
return(0);
}
```

Please reflect on this: thread maintains context even when alone inside of function call

```
C:\Windows\system32\cmd.exe
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 3
Level 2: number of threads in the team - 3
Level 3: number of threads in the team - 4
Level 3: number of threads in the team - 4
Level 3: number of threads in the team - 4
Level 3: number of threads in the team - 4
Level 3: number of threads in the team - 4
Press any key to continue . . .
```

# “setting” and “getting: their scope

- **set** methods affect only parallel regions at **same or inner nesting levels** encountered by the **calling thread**
  - The mechanics of it
    - Upon creation of a team of threads, slave threads inherit values from master thread
- **get** methods return values associated with the **calling thread**

# Departing Thoughts – 1

- Nested parallel regions: an immediate way to engage more threads in a computation
- Nesting parallel regions prone to create large number of threads
  - Number of threads created: the product of the number of threads forked at each level of nested parallelism
    - Example: N threads at outer level, each creating M threads at inner level –  $N \times M$  threads
- Looming danger: oversubscribe the system. Might slow you down...
  - Function calls and environment variables available to manage tightly the spawning of threads

# Departing Thoughts – 2

- Examples of good reasons to employ nested parallelism
  - Insufficient parallelism at outer level
    - Many-core processors today can handle many threads
      - IBM POWER9 chip: handles 96 threads
    - Example: have outer loop w/ 10 trips on a 64 core workstation
  - Load balance problems: more threads in flight can help balance the load

# Nested Parallelism: Food for Thought

- Backdrop: Creating any parallel region entails some overhead
- If you want to avoid this overhead and also avoid oversubscription
  - Should you parallelize the inner loop?
  - Should you parallelize at the outer loop?

# Work Plan, OpenMP

- What is OpenMP?

Parallel regions

## **Work sharing**

Variable Scoping Issues

Synchronization

Performance issues

- Loose ends

# Work Sharing

- **Work sharing:** general term used in OpenMP to describe distribution of work across threads
- Three primary avenues for work sharing in OpenMP:
  - `omp for` construct
  - `omp sections` construct
  - `omp task` construct

Each of them **automatically** divides work among threads

# Work sharing, an alternative to GPU-style parallel computing

- Recall previous discussion:
  - **structured block**: we already defined what this is
    - Enables GPU-style parallel computing – each thread executes the same structured block
  - **OpenMP construct**: talk about this next

# Work Plan, OpenMP

- What is OpenMP?

Parallel regions

## **Work sharing – Parallel for loops**

Variable Scoping Issues

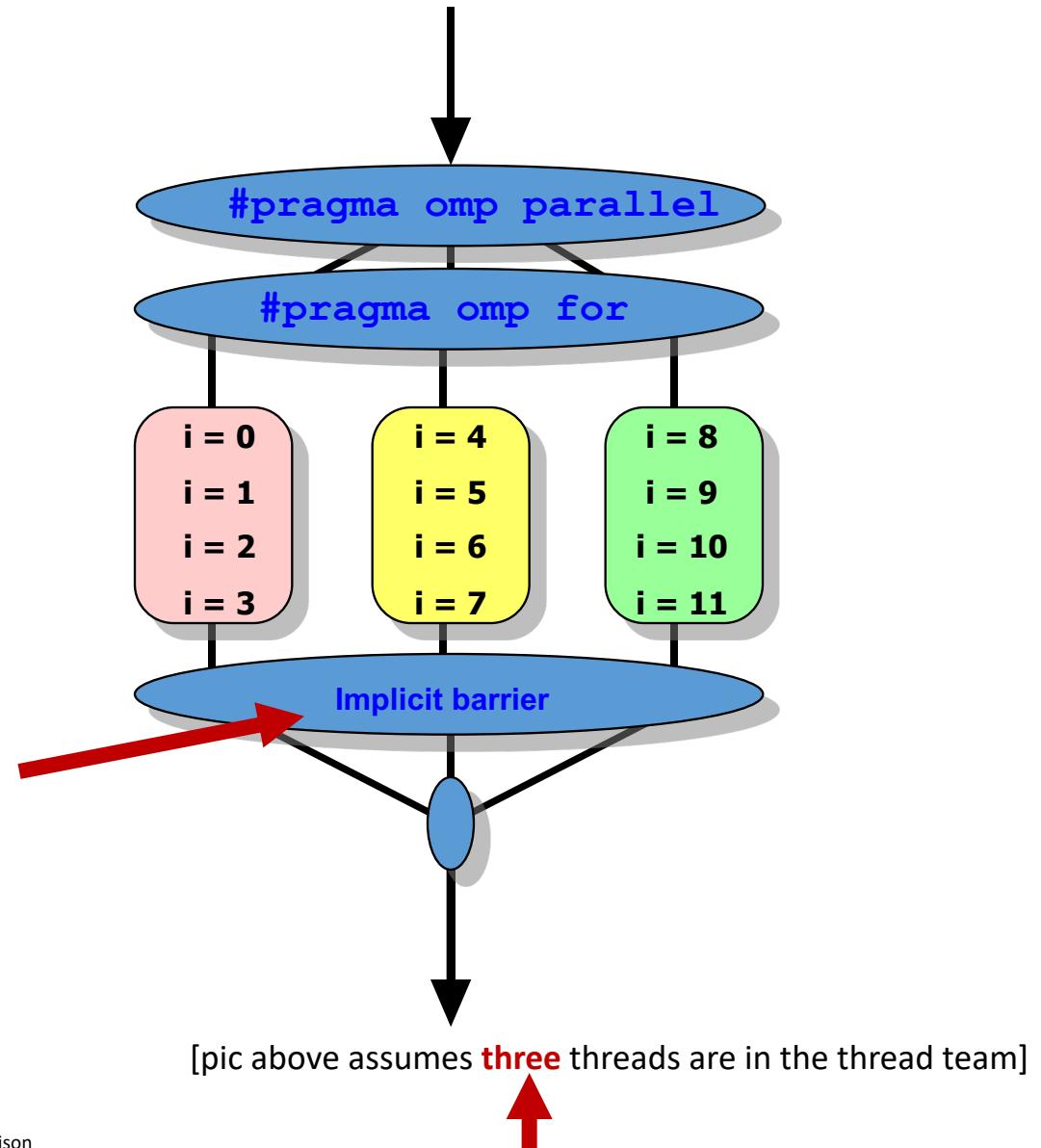
Synchronization

- Loose ends

# The **omp for** Directive

```
// assume N=12 <-- !!!  
#pragma omp parallel  
#pragma omp for  
  for(i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct



# Combining Constructs

- These two code segments are equivalent

```
#pragma omp parallel
{
    #pragma omp for
    for ( int i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
#pragma omp parallel for
for (int i=0;i< MAX; i++) {
    res[i] = huge();
}
```

There is an implicit barrier here.  
(at the end of the parallel region,  
that is)

# Problems with default partitioning

- Most OpenMP implementations use as default block partitioning
  - Each thread is assigned roughly  $n/\text{thread\_count}$  iterations
- This may lead to load imbalance when the work per iteration varies

```
sum = 0;  
for(i = 0; i <= n; i++)  
    sum += f(i);
```

- (assume the time required by a call to  $f(i)$  is **proportional** to  $i$ )

What would be a good partitioning of iterations over threads in this case?

# The **schedule** Clause

- The **schedule** clause affects how loop iterations are mapped onto threads

## **schedule(static [,chunk])**

- Blocks of iterations of size `chunk` assigned to each thread
- Round robin distribution
- Low overhead, may cause load imbalance

## **schedule(dynamic[,chunk])**

- Threads grab `chunk` iterations
- When done with iterations, thread requests next set of `chunk` iterations
- Higher threading overhead, can reduce load imbalance

## **schedule(guided[,chunk])**

- Dynamic schedule starting with large block
- Size of the blocks shrinks; no smaller than `chunk` though

# schedule Clause Example

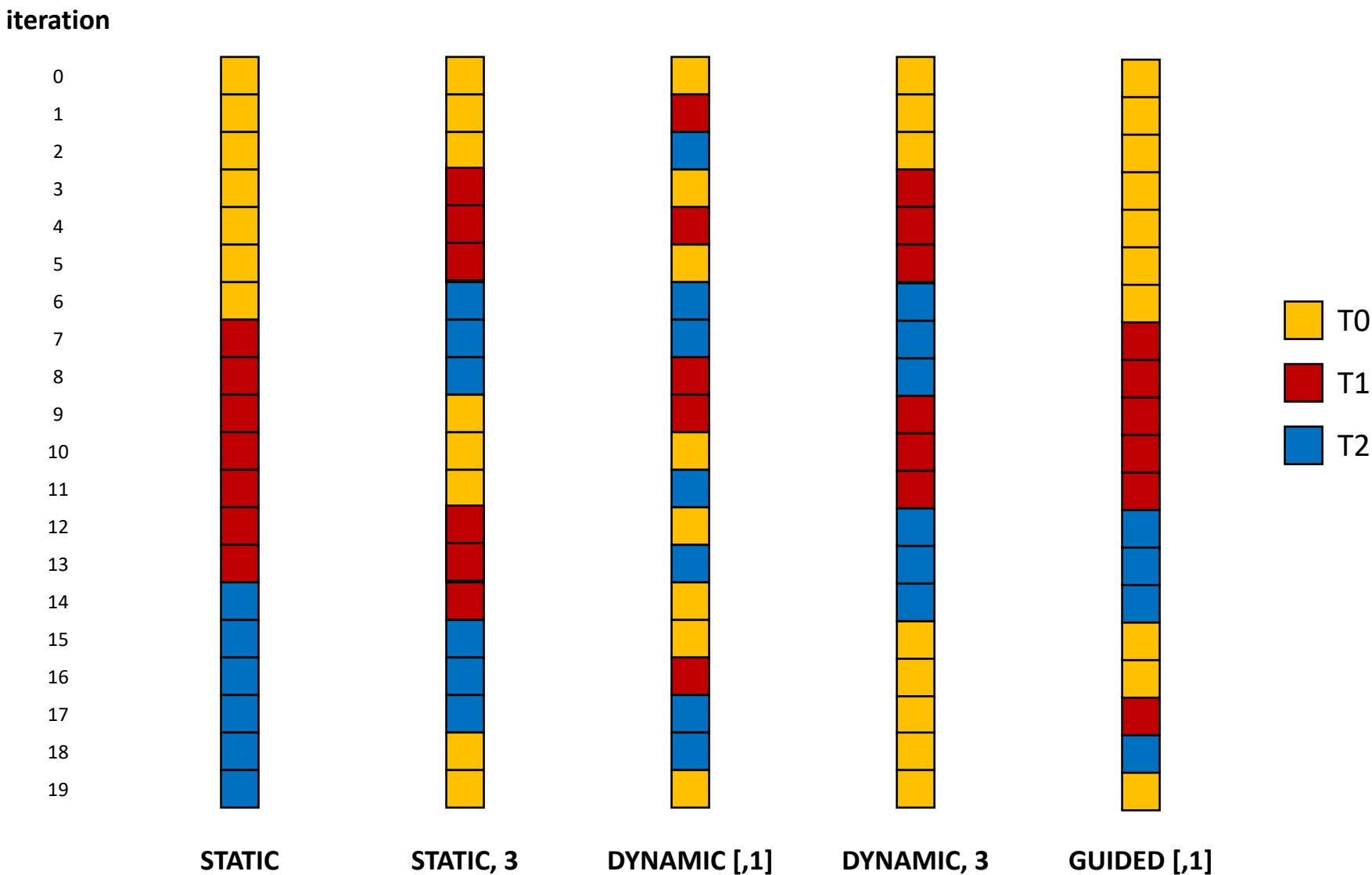
- Iterations are divided into chunks of 8

- If start = 3, then first chunk is

i={3,5,7,9,11,13,15,17}

```
#pragma omp parallel for schedule(static, 8)
    for( int i = start; i <= end; i += 2 )
    {
        if ( TestForPrime(i) )  gPrimesFound++;
    }
```

# Loop Scheduling (Assume 3 Threads)



# Choosing a Schedule

- STATIC **best** if you know you have balanced loops – least overhead
- STATIC, n good for loops with mild or smooth load imbalance
  - Prone to introduce “false sharing” (discussed later)
- DYNAMIC useful if iterations have widely varying loads
  - Prone to adversely impact data locality (cache misses)
- GUIDED often less expensive than DYNAMIC
  - Beware of loops where first iterations are the most expensive

# What for loops can be parallelized?

- OpenMP will only parallelize for loops that are in **canonical form**



```
for (index = start ; index < end  
      || index ≤ end  
      || index ≥ end  
      || index > end ; index += incr  
                     index -= incr  
                     index = index + incr  
                     index = incr + index  
                     index = index - incr)
```

# Work Plan, OpenMP

- What is OpenMP?

Parallel regions

## **Work sharing – Parallel sections**

Variable Scoping Issues

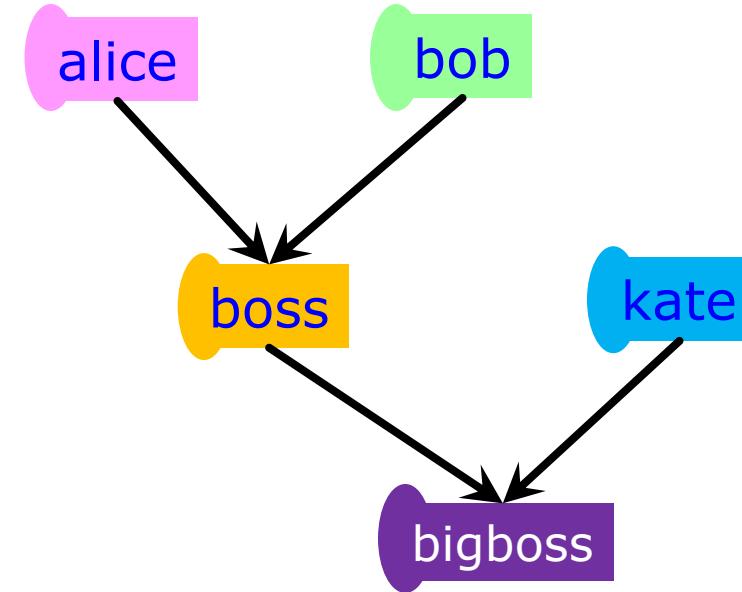
Synchronization

Performance issues

- Loose ends

# Function Level Parallelism

```
a = alice();
b = bob();
s = boss(a, b);
k = kate();
printf ("%6.2f\n", bigboss(s,k));
```



- `alice`, `bob`, and `kate` can be invoked for execution in parallel
- Example above: good illustration of the concept of task parallelism

# Parallel **omp sections** Directive

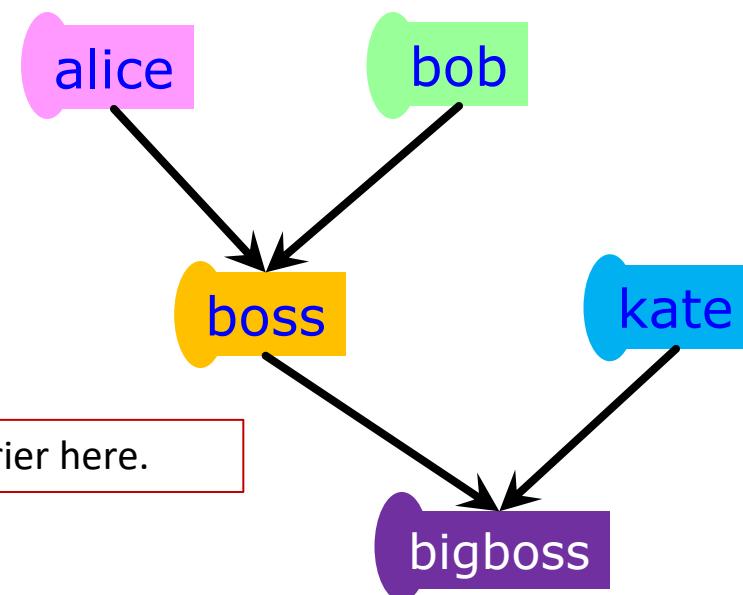
- **#pragma omp sections** 
- Must be inside a parallel region
- Precedes a code block containing  $N$  sub-blocks of code that may be executed concurrently by  $N$  threads
- Encompasses all **omp section** blocks, see below

- **#pragma omp section** 
- Precedes each sub-block of code within the encompassing block described above
- Enclosed program segments are distributed for parallel execution among available threads

# Functional Level Parallelism Using `omp sections`

```
#pragma omp parallel sections
{
    #pragma omp section
        a = alice();
    #pragma omp section
        b = bob();
    #pragma omp section
        k = kate();
}
double s = boss(a, b);
printf ("%6.2f\n", bigboss(s,k));
```

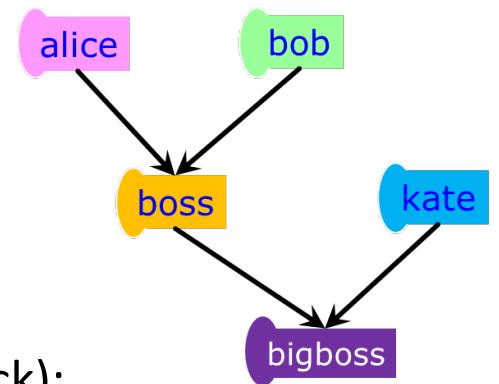
There is an implicit barrier here.



Is there another way to parallelize this?

# Different Way to Go at It [Assume only two threads available]

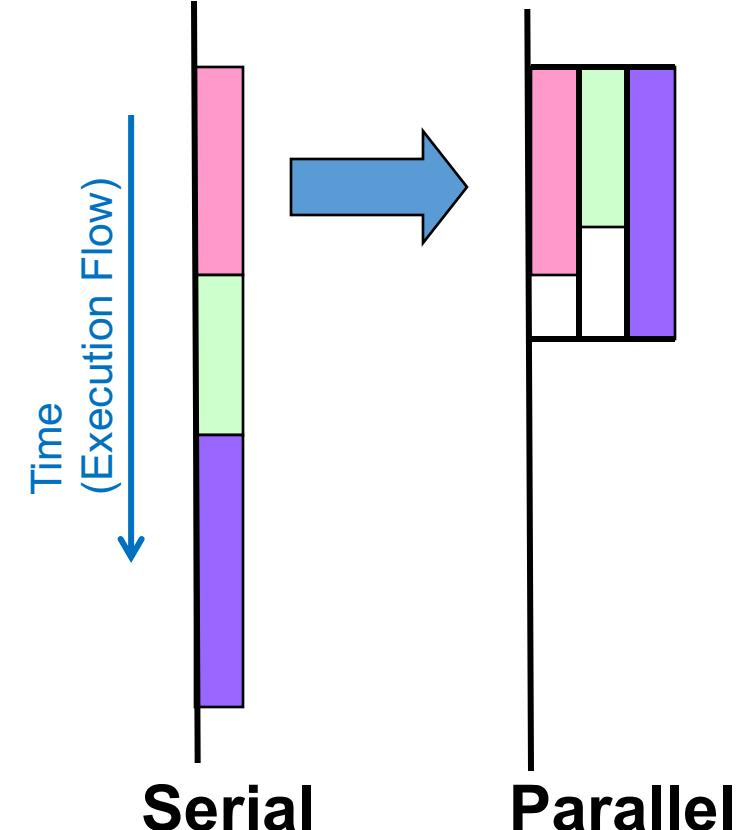
- Approach on previous slide might be improved if only two OpenMP threads
  - Also depends how balanced the “alice”, “bob” and “kate” jobs are
- Better use of resources (consequence of *implicit synchronization* at end of parallel block):
  - Step 1: use one sections to handle alice and bob
  - Step 2: use another sections to handle kate and boss
  - Step 3: bigboss executed by master threads



# Advantage of Parallel Sections

- Independent sections of code can execute concurrently → reduces execution time

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



The pink and green tasks are executed at no additional time-penalty in the shadow of the blue task

# Example, **sections**: using 2 threads

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

void spin_in_place(double duration) {
    double start_time = omp_get_wtime();
    while (omp_get_wtime() - start_time < duration) {}
}

int main() {
    printf("\nUsing 2 threads on 3 sections\n");
    double start_time = omp_get_wtime();

#pragma omp parallel sections num_threads(2)
{
#pragma omp section
{
    printf("Start work 1\n"); spin_in_place(2); printf("End work 1\n");
}
#pragma omp section
{
    printf("Start work 2\n"); spin_in_place(2); printf("End work 2\n");
}
#pragma omp section
{
    printf("Start work 3\n"); spin_in_place(2); printf("End work 3\n");
}

printf("Wall clock time: %.2g\n", omp_get_wtime() - start_time);
return 0;
}
```

Cmder

```
gauss ~/CodeBits759> g++ -fopenmp sectionsExample1.cpp
gauss ~/CodeBits759> ./a.out

Using 2 threads on 3 sections
Start work 1
Start work 2
End work 1
Start work 3
End work 2
End work 3
Wall clock time: 4
gauss ~/CodeBits759> |
```

Note: The host machine (gauss) has tons of cores.

# Example, sections: using 4 threads

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

void spin_in_place(double duration) {
    double start_time = omp_get_wtime();
    while (omp_get_wtime() - start_time < duration) {}
}

int main() {
    printf("\nUsing 4 threads on 3 sections\n");
    double start_time = omp_get_wtime();

#pragma omp parallel sections num_threads(4) ← !
    {
#pragma omp section
    {
        printf("Start work 1\n"); spin_in_place(2); printf("End work 1\n");
    }
#pragma omp section
    {
        printf("Start work 2\n"); spin_in_place(6); printf("End work 2\n"); ← !
    }
#pragma omp section
    {
        printf("Start work 3\n"); spin_in_place(2); printf("End work 3\n");
    }
}

printf("Wall clock time: %.2g\n", omp_get_wtime() - start_time);
return 0;
}
```

λ Cmder

```
gauss ~/CodeBits759> g++ -fopenmp sectionsExample2.cpp
gauss ~/CodeBits759> ./a.out

Using 4 threads on 3 sections
Start work 2
Start work 3
Start Work 1
End work 1
End work 2
Wall clock time: 6
gauss ~/CodeBits759> |
```

Note: The host machine (gauss) has tons of cores.

# Work Plan, OpenMP

- What is OpenMP?

Parallel regions

## **Work sharing – Parallel tasks**

Variable Scoping Issues

Synchronization

Performance issues

- Loose ends

# OpenMP tasks

- Task – Most important feature added as of OpenMP 3.0 version
- Allows parallelization of irregular problems
  - Unbounded loops
  - Recursive algorithms
  - Producer/consumer
- Start with brief discussion why the task OpenMP construct useful

# OpenMP Tasks – Motivation [1/3]

- Parallelization of a dynamic list traversal: it couldn't be done in OpenMP for a long while

```
p = listhead;
while (p) {
    process(p);
    p = next(p);
}
```

OpenMP was initially “somewhat tailored for large array-based applications.”

OpenMP Application Program Interface, version 2.5 – OpenMP ARB, May 2005

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 21

03/11/2020

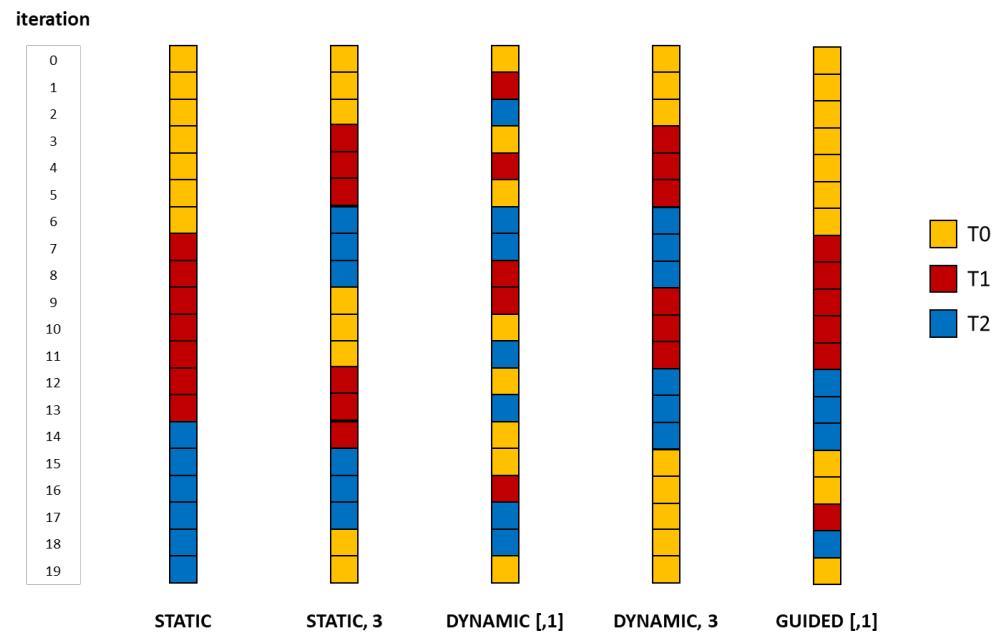
# The quote the day

“If with a pure mind a person speaks or acts, happiness follows them like a never-departing shadow.”

-- Siddhartha Gautama, The Buddha [563 BC -- 483 BC]

# Before we get going...

- Last time:
  - OpenMP generalities
  - Work-sharing constructs in OpenMP
- Today:
  - Further discussion of default projects
  - Tasks, wrap up
  - Variable scoping issues
- Other tidbits:
  - ME759 Exam: April 15, at 7:15 PM, in 1106ME
    - Review: Tu, April 14, at 7:00 PM, in Canvas



# ME759 Final Project aspects

# Final Project Proposals

- **Proposal** Issues:
  - Two pages long (shorter, if it makes sense).
  - PDF file to be uploaded in folder FinalProjectProposal
  - Due on 03/27 at 9 pm
- Proposal doc template available [here](#). You must use this template document
  - Simply populate the fields in there
  - Template use: Everybody stays on the same page (actually, two pages)
- I will try to provide feedback by April 3
  - This will give you a month to work on the Final Project

# Final Project, good to know

- Project can be individual or team-based
  - Teams can have up to three students
    - Multi-student proposals: need to spell out who does what
    - Each student submits project proposal, intermediate progress, and final project docs
      - The docs can be clones, it's ok

# Things to touch on in your Project **Proposal** document

- **Project Title:** state the title
- **Link to git repo for project:** this is what will be cloned to check your work upon delivery
- **Problem statement:** explain in clear terms what you want to do.
- **Motivation/Rationale:** explain why you chose to work on this project
- **Explain how you contemplate going about it:** indicate if you'll use GPU/OpenMP/MPI parallel computing, what libraries, etc. Indicate what algorithms/approaches you are considering
- **ME759 aspects the proposed work draws on:** bulleted list, be brief
- **How you will demonstrate what you accomplished:** particularly important if what you do is a small piece of a bigger project that you will continue to pursue after wrapping up ME759.
- **Team member[s]:** Name + email + home department + advisor (if one exists).
  - If more students, also indicate each student's anticipated role.
- **Deliverables:** what you expect to deliver on May 6 at 7:45 am
- **Milestones:** indicate what will be accomplished by April 22 milestone
- **Other remarks:** say here anything else that you think Dan should be aware of

# Final Project Tied to Your Research

- OK to propose a small part of a big undertaking that would be too large to accomplish within one month
  - This is fine as long as
    - Project proposed represents a step towards something that is ambitious
    - You structure the Final Project so that progress can be measured/demonstrated
- IMPORTANT: Your score will reflect how well you accomplish what's spelled out in Project Proposal, not the big task that can't be done in one month

# 10/23/2014 Email – From Todd T. [former 759 Student, Spring 2012]

"Hi Dan and Andrew,

Hope time finds you well! Recall my ME964 project to add first-class CUDA discovery, scheduling, management into HTCondor? Well, at long last, earlier this year it all got released into the mainstream stable version of HTCondor, announced at HTCondor Week 2014 back in May, and pushed into all the various distros (Debian, Fedora, etc). It is being used on big clusters in industry and academia. Long ago we talked about telling Nvidia about this, and giving credit to this happening as a 'success story' for nvidia's support of SBEL as a center of excellence..."

# Example, 2012 Final Project: GPU Scheduling on a Cluster

- Spring 2012 Final Project
  - Tied to the Condor Project
    - Condor: started at UW-Madison by Professor Miron Livny (in 1985)
    - Two students working for Professor Livny took 759 in back in the day
- Places where GPU Scheduler used for science today
  - Large Hadron Collider (LHC) – CERN, Switzerland
    - Hadron collider: 2013 Nobel Prize in Physics for Higgs boson
  - Laser Interferometer Gravitational-wave Observatory (Ligo) – WA / LA, USA
    - Space/Time wrinkle, 2017 Nobel Prize in Physics
  - Ice Cube, Antarctica
    - South Pole Neutrino Observer, run by UW-Madison
    - Last year: \$35 million NSF to UW-Madison

# Words of wisdom

- If you are stress-adverse, propose something that is clearly doable
- Keep in mind that all you have about a month
  - Factor in exams, final reports, etc. for **other** classes. You have less time than you think you do
- My advice (doesn't apply to all):
  - Tie the Final Project to your research
  - If no research topic yet, tie it to something that you are curious about
  - Be ambitious, yet propose something that demonstrates quantifiable progress
- Failing is totally ok, if it comes despite hard work

# Milestones and deliverables

- Final Project proposal (PDF doc) – March 27, 9 pm
  - Canvas folder: FinalProjectProposal
- Final Project progress report, a one pager (PDF doc) – April 23, 9 pm
  - Canvas folder: FinalProjectIntermediate
- Final Project report (PDF doc) – May 6, 7:45 am
  - Canvas folder: FinalProject
- Code in your git repo – May 6, 7:45 am
  - Used to verify results reported in Final Project document

# Final Projects suggestions – in case you can't come up with a topic

- Two tiers
  - Tier one: UW-Madison brewed, they are easier (four of them)
  - Tier two: Intel brewed, they are tough (four of them)

# Tier 1, Option 1: Large code for physics-based simulation

- For more details, see [here](#).
  - Topic related to code developed by former ME759 student (became his PhD thesis)
  - Use case scenario put together by person working with the European Space Agency (Cecily Sunday)
  - Tasks:
    - Profile a large code
    - Look at how's parallelized
    - Suggest avenues for improving speed
    - Implement some of these changes to improve execution speed (which is really low, right now)

# Tier 1, Option 2: Image processing

- For more details, see [here](#)

- Issues of interest:

- Finding edges in a pic
  - Image stitching

# Tier 1, Option 3 : Solving $\mathbf{Ax} = \mathbf{b}$

- For more details, see [here](#)
  - Task: Implement a parallel solution to solving  $\mathbf{Ax} = \mathbf{b}$
  - There are two flavors that you might consider
    - Iterative solvers
    - Direct solvers
  - To make it even more interesting:
    - Dense problems
    - Sparse problems

# Tier 1, Option 4: Break password through brute force

- For more details, see [here](#).
  - Task: write a parallel code that attempts to break a password through brute force
  - Important: your implementation should only be used for academic purposes and should never be used to engage in any illegal or otherwise dubious activity of a nefarious nature

# Tier 2: Intel-suggested Final Project topics

- Coming from group at Intel made up of 13 compiler engineers working on optimizing machine learning workloads on various hardware platforms
- Working w/ specialized hardware groups to ensure fast execution of inference and training workloads through hardware-software co-design
  - See <https://github.com/plaidml/plaidml>
- Currently working closely with the LLVM community and the MLIR group at Google to contribute to the affine dialect in MLIR
  - See <https://mlir.llvm.org/docs/Dialects/AffineOps>
- Researching repurposing machine learning optimization techniques for scientific workloads

# Default Final Project topics suggested by Intel group

Topic	Description	Helpful References
LLVM Optimization Analysis	Analyze LLVM Optimization Passes using established benchmarks	<ol style="list-style-type: none"><li>1. <a href="https://rd.springer.com/content/pdf/10.1007%2F978-3-319-91479-4_23.pdf">https://rd.springer.com/content/pdf/10.1007%2F978-3-319-91479-4_23.pdf</a></li></ol>
Survey and Analysis of Tensor Compilers	Analyze and benchmark tensor compilers for ML across various HW OR Investigate the results produced in 2.	<ol style="list-style-type: none"><li>1. <a href="https://arxiv.org/abs/2002.03794">https://arxiv.org/abs/2002.03794</a></li><li>2. <a href="http://www.eecs.harvard.edu/~htk/publication/2019-mapl-tillet-kung-cox.pdf">http://www.eecs.harvard.edu/~htk/publication/2019-mapl-tillet-kung-cox.pdf</a></li></ol>
High performance GEMM library for a GPU	Utilizing the techniques described in papers 1 and 2 on the right And the MLIR's new support for affine.parallel: (see 3 on the right) Implement a high performance GEMM library for a GPU	<ol style="list-style-type: none"><li>1. <a href="https://arxiv.org/abs/2003.00532">https://arxiv.org/abs/2003.00532</a></li><li>2. <a href="https://paperswithcode.com/paper/stripe-tensor-compilation-via-the-nested">https://paperswithcode.com/paper/stripe-tensor-compilation-via-the-nested</a></li><li>3. <a href="https://mlir.llvm.org/docs/Dialects/AffineOps/#affineparallel-affineparallellop">https://mlir.llvm.org/docs/Dialects/AffineOps/#affineparallel-affineparallellop</a></li></ol>
Distributed SLIDE	Reproduce the results presented for SLIDE in 1. Using the code provided on github (2) Implement a distributed version of SLIDE.	<ol style="list-style-type: none"><li>1. <a href="https://arxiv.org/pdf/1903.03129.pdf">https://arxiv.org/pdf/1903.03129.pdf</a></li><li>2. <a href="https://github.com/keroro824/HashingDeepLearning/tree/master/SLIDE">https://github.com/keroro824/HashingDeepLearning/tree/master/SLIDE</a></li></ol>

# Intel related Final Projects

- If you want to, the Intel folks will read your Final Project
- You'll have to explicitly indicate that you are ok with me sharing your Final Project with them
  - There will be a box to check in both the proposal and final docs
- This is hard stuff, very challenging
  - Work on it if you want to get a conversation going with this Intel group

- Back to OpenMP program

# OpenMP Tasks – Motivation [3/3]

- Use `single nowait` inside a `parallel` region

```
#pragma omp parallel private(p)
{
    p = listhead;
    while (p != listend) {
        #pragma omp single nowait
        process(p);
        p = next(p);
    }
}
```

- Why we shouldn't settle for this:
  - Relatively **high cost** of the `single` construct
  - Each thread needs to traverse the **entire list** and determine if another thread has already processed that element
    - Ensuring that no other thread has processed a list element is not trivial at all
      - The code ensuring this is not shown here and would have to be inside the `process (p)` function call

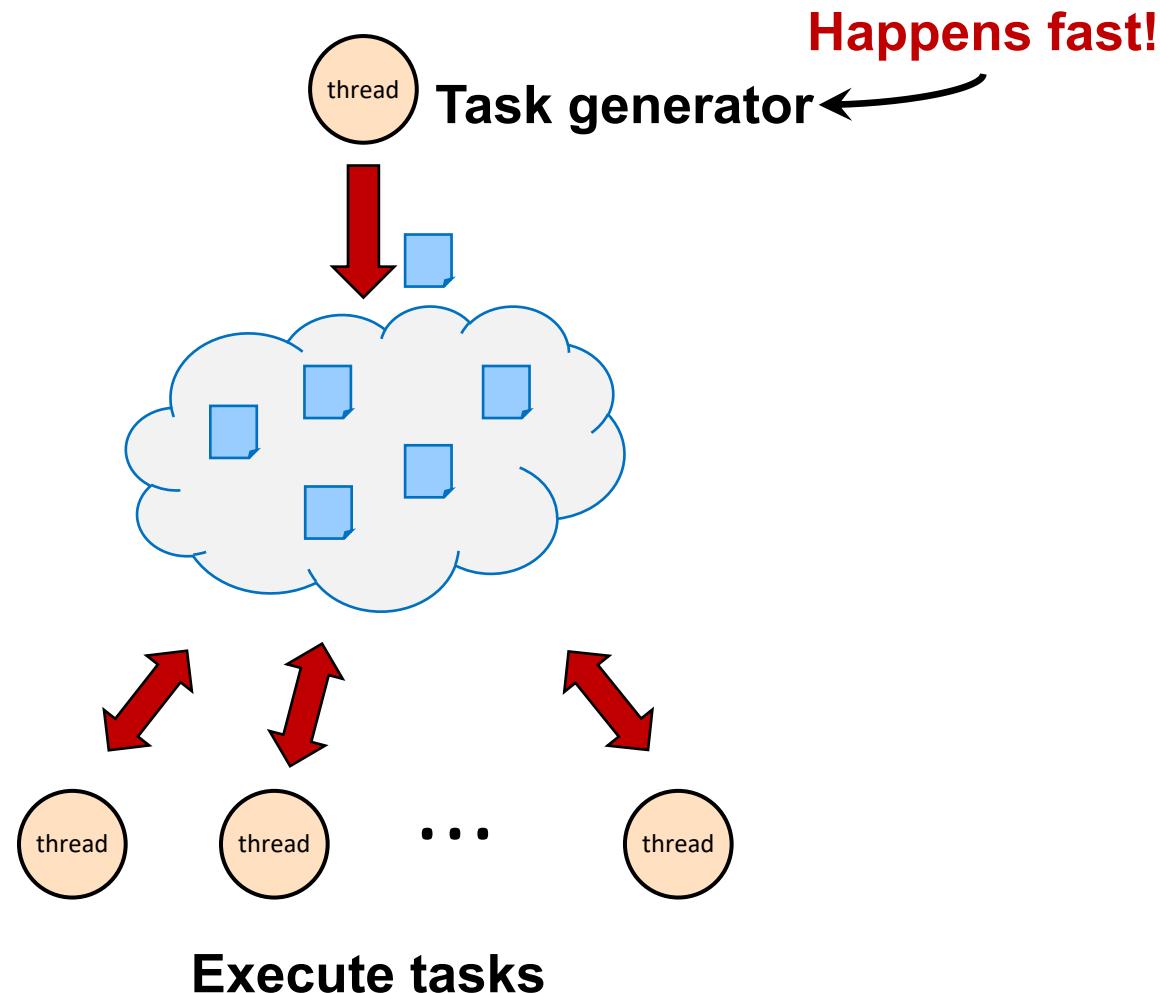
# One-Slide Side-Trip: When Compiler Doesn't Speak OpenMP

- Imagine your compiler doesn't speak OpenMP
  - Previous piece of code: is like ignoring all the pragmas

```
#pragma omp parallel private(p)
{
    p = listhead;
    while (p != listend) {
        #pragma omp single nowait
        process(p);
        p = next(p);
    }
}
```

- Ignoring pragmas: You'd get your sequential implementation that you used to have before you decorated the code w/ OpenMP directives

# The **task** Concept in OpenMP

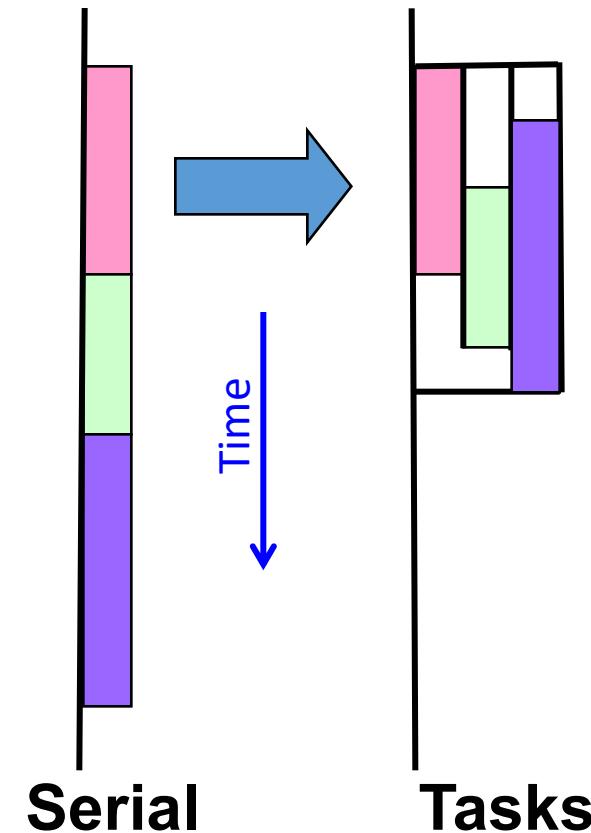


# Tasks: Who Does What and When?

- The developer (you, that is), does this:
  - Use a pragma to specify where/what the tasks are
  - Ensure that there are **no dependencies** (that is, tasks can be executed independently)
- The OpenMP runtime system does this:
  - Generates a new task whenever a thread encounters a `task` construct
  - Decide the moment of execution (can be immediate or delayed)

# What Are Tasks? How Do They Work?

- Tasks are independent units of work
- One or more threads assigned to generate tasks
- The OpenMP threads pick up & execute the “posted” tasks
- Tasks might be executed immediately or might be deferred
  - The OS & runtime decide which of the above
    - Decision transparent to the user



# What Are Tasks?

[Preamble]

- Three concepts important in the context of tasks
  - Code to execute
    - The literal code in your program enclosed by the task directive
  - Data environment
    - The shared & private data manipulated by the task
  - Internal control variables (ICV)
    - Thread scheduling and environment variables

# What Are Tasks?

[The specifics...]

- More formal definition:
  - A task is a specific instance/combo of executable code along w/ its data environment and ICV
    - This “instance/combo” is generated when a thread encounters a `task` construct
- In other words, there are three activities: (1) packaging, (2) posting the job, and (3) execution of the jobs
  - A thread packages a new instance of a task; i.e., its code & data & ICVs
  - This “combo” is placed in a queue of “things to do”
  - Some thread in the team executes the task at some later time

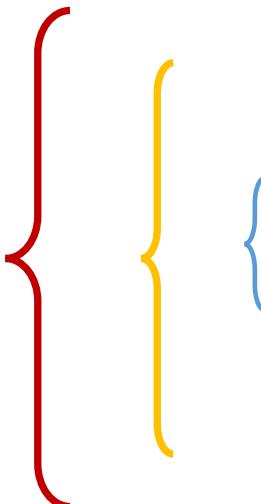
# The task Directive

- A team of threads is created at the `omp parallel` construct
- A single thread is chosen to execute the while loop – let's call this thread “L”
- Thread L runs the `while` loop, creates tasks, and fetches next pointers
- Each time L crosses the `omp task` construct it generates a new task and has a thread assigned to it
- Each task run by one thread

```
#pragma omp parallel //threads are ready to go now
{
    #pragma omp single nowait
    {
        node *p = head_of_list;
        while(p != end_of_list) {
            #pragma omp task firstprivate(p)
            process(p);
            p = p->next;
        }
    }
}
```

- All tasks complete at the barrier at the end of the parallel region's construct
- Each task has its `own stack space` that will be destroyed when the task is completed

```
#include <omp.h>
#include <list>
#include <iostream>
#include <math.h>
```



```
typedef std::list<double> LISTDBL;

void doSomething(LISTDBL::iterator& itr) { *itr *= 2; }

int main() {
    LISTDBL test;
    LISTDBL::iterator it;

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 8; j++)
            test.push_back(pow(10.0, i + 1) + j);
    for (it = test.begin(); it != test.end(); ++it) printf("%g\n", *it);

    it = test.begin();

#pragma omp parallel num_threads(8)
{
#pragma omp single nowait
{
    while (it != test.end()) {
#pragma omp task firstprivate(it)
    {
        doSomething(it);
    } // end omp task
    it++;
}
printf("...master done\n");
} // end omp single
} // end omp parallel

for (it = test.begin(); it != test.end(); ++it) printf("%g\n", *it);

return 0;
}
```

Initial values...

Final values...

Final values...

```
gauss ~/CodeBits759>
gauss ~/CodeBits759> g++ -fopenmp tasksExample1.cpp
gauss ~/CodeBits759> ./a.out
10
11
12
13
14
15
16
17
100
101
102
103
104
105
106
107
1000
1001
1002
1003
1004
1005
1006
1007
10000
10001
10002
10003
10004
10005
10006
10007
...master done
20
22
24
26
28
30
32
34
200
202
204
206
208
210
212
214
2000
2002
2004
2006
2008
2010
2012
2014
20000
20002
20004
20006
20008
20010
20012
20014
gauss ~/CodeBits759> |
```

# Tasks: Synchronization Issues

- Setup:
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guarantee completion of Task A before invoking the execution of Task B
- Tasks are guaranteed to be complete at thread or task barriers:
  - At the directive: **#pragma omp barrier**
  - At the directive: **#pragma omp taskwait**

# Task Completion Example

```
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

# Comments: **section** vs. **task**

- **sections** have a “*static*” attribute
  - Things are mostly settled at compile time
- The **task** OpenMP construct more recent and more sophisticated
  - Has a “*dynamic*” attribute: things figured out at run time
  - A task can encapsulate any block of code
  - Can handle nested loops and scenarios when the number of jobs is not clear
  - Big hammer, watch your fingers...

# Work Plan, OpenMP

- What is OpenMP?

Parallel regions

Work sharing – Parallel tasks

## **Variable Scoping Issues**

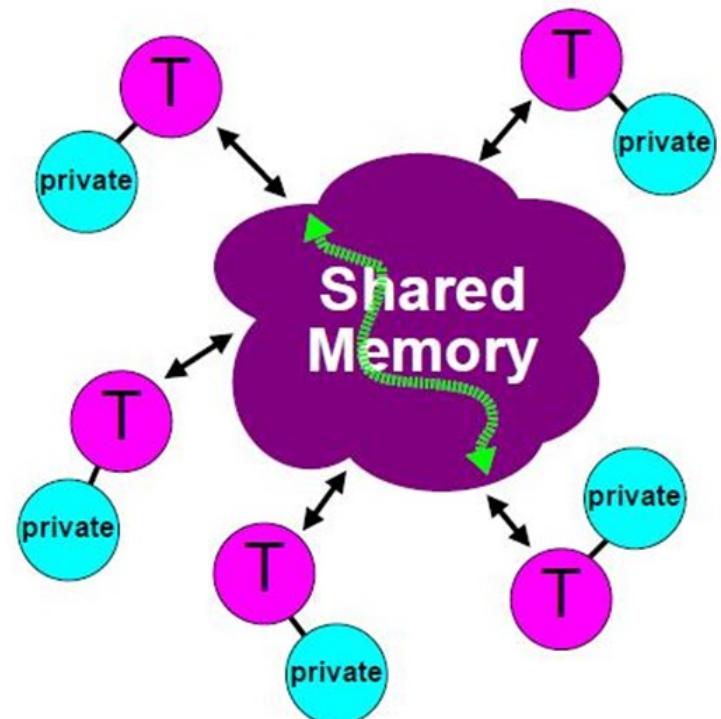
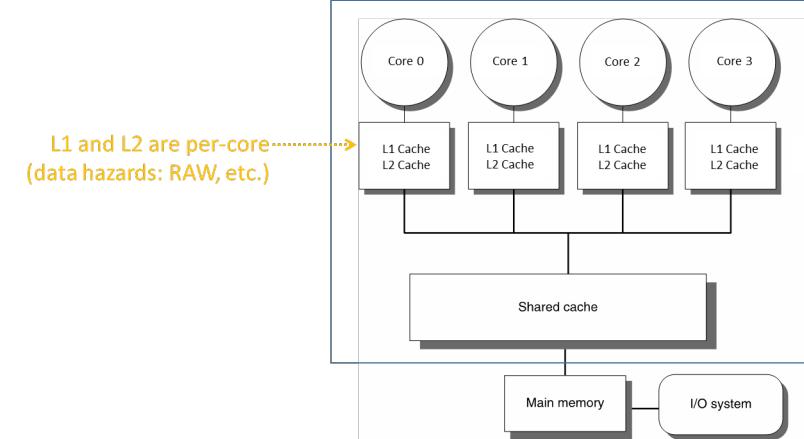
Synchronization

Performance issues

- Loose ends

# OpenMP Attributes

- Parallel execution of multiple tasks via multiple threads
- Threads have access to a pool of memory that is shared
- Threads can have private data
  - Not accessible by other threads
- When/How data is written/read from memory is transparent to programmer
  - However, programmer has some control (atomic, flush, etc.)
- Synchronization in thread execution is implicit but can be made explicit as well



# Variable Scoping Issues in OpenMP

- Important question: what is shared and what is not?
  - That is, what can a thread access, and what cannot it access?
- Important why?
  - If a variable is shared, then you better be aware of that
    - Parallel computing data hazards, in relation to WAR, RAW, WAW
- “Variable scoping in OpenMP”:
  - Understanding whether a variable that a thread works with is private or shared

# Variable Scoping Issues in OpenMP – What is shared?

- OpenMP uses a shared-memory programming model
- **Shared variable** - a variable that can be read or written by multiple threads
- **Private variable** – a variable whose value is only visible to a thread (for read/write)

# Variable Scoping Issues in OpenMP – What is **shared**?

- Basic rule:
  - Any variable declared prior to a parallel region is shared in that parallel region
    - That is, it is available for read/write to all parallel threads of the parallel region
- Other examples of variables being shared among threads
  - Global variables
  - File scope variables
  - Namespace scope variables
  - Variables with `const`-qualified type having no mutable member
  - `static` variables which are declared in a scope inside the construct
- Note: there's a **shared** clause that you can use if you feel like explicitly indicating that a variable is shared

# The **private** Clause

- Reproduces for each thread variable[s] declared private in the pragma (see example below)
  - Consequence: each thread will have a private copy of that variable in the associated parallel region
    - The value that Thread\_1 stores in x is different than value that Thread\_2 stores in variable x
  - NOTE: Private variables are **un-initialized**; C++ object is default-constructed

```
void* work(float* c, int N) {
    float x, y;
    int i;
#pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
    }
}
```

# Data Scoping – What is Private?

- Examples of OpenMP variables treated as **private** by default:
  - Stack (local) variables in functions called from parallel regions
    - C/C++: call by copy vs. call by reference
  - Loop iteration variables
  - Automatic variables within a statement block

# Example [more dwelling on the **private** concept]: Is “i” shared or private?

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

int main() {
    int i = 10;

#pragma omp parallel num_threads(4)
{
    int threadID = omp_get_thread_num();
    printf("threadID = %d  i = %d\n", threadID, i);
    i = 1000 + threadID;
}

printf("i = %d\n", i);

return 0;
}
```

```
curie ~/ME759/OpenMP> g++ firstPrivateP0.cpp -fopenmp
curie ~/ME759/OpenMP> ./a.out
threadID = 0  i = 10
threadID = 2  i = 10
threadID = 1  i = 10
threadID = 3  i = 10
i = 1003
curie ~/ME759/OpenMP> ./a.out
threadID = 3  i = 10
threadID = 0  i = 10
threadID = 2  i = 10
threadID = 1  i = 10
i = 1001
curie ~/ME759/OpenMP> ./a.out
threadID = 0  i = 10
threadID = 3  i = 10
threadID = 1  i = 10
threadID = 2  i = 10
i = 1002
curie ~/ME759/OpenMP> ./a.out
threadID = 0  i = 10
threadID = 3  i = 10
threadID = 1  i = 10
threadID = 2  i = 10
i = 1002
curie ~/ME759/OpenMP> ./a.out
threadID = 0  i = 10
threadID = 3  i = 10
threadID = 2  i = 10
threadID = 1  i = 10
i = 1001
curie ~/ME759/OpenMP>
```

# private: somewhat sneaky

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

int main(void) {
    int i = 10;

#pragma omp parallel num_threads(4) private(i)
{
    int threadID = omp_get_thread_num();
    printf("threadID = %d: i = %d\n", threadID, i);
    i = 1000 + threadID;
}

printf("i = %d\n", i);

return 0;
}
```

- What do we expect each thread to print?
- What do we expect the master thread to print at the end?

```
curie ~/ME759/OpenMP> g++ firstPrivateP1.cpp -fopenmp
curie ~/ME759/OpenMP> ./a.out
threadID = 0  i = 0
threadID = 2  i = 0
threadID = 3  i = 0
threadID = 1  i = 0
i = 10
```

```
curie ~/ME759/OpenMP> g++ firstPrivateP1.cpp -Wall -fopenmp
... some stuff here...
firstPrivateP1.cpp:10:11: warning: 'i' is used uninitialized in this function [-Wuninitialized]
    printf("threadID = %d  i = %d\n", threadID, i);
    ~~~~~^~~~~~
firstPrivateP1.cpp:5:7: note: 'i' was declared here
    int i = 10;
        ^
```

# private vs. firstprivate

- **firstprivate**

- Goes one step beyond where `private` stops
- Specifies that each thread should have its own instance of a variable
- Moreover, the variable is initialized using the value of the variable of the same name from the master thread

# private vs. firstprivate

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

int main(void) {
    int i = 10;

#pragma omp parallel num_threads(4) firstprivate(i)
{
    int threadID = omp_get_thread_num();
    printf("threadID = %d: i = %d\n", threadID, i);
    i = 1000 + threadID;
}

printf("i = %d\n", i);

return 0;
}
```

- What do we expect each thread to print?
- What do we expect the master thread to print at the end?

```
gauss ~/CodeBits> g++ -fopenmp firstPrivateExample1.cpp
gauss ~/CodeBits> ./a.out
threadID = 3: i = 10
threadID = 0: i = 10
threadID = 1: i = 10
threadID = 2: i = 10
i = 10
gauss ~/CodeBits>
```

# The `lastprivate` OpenMP directive

- The enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration of the work-sharing construct (`for` or `section`)

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

int main(void) {
    int i;

#pragma omp parallel for num_threads(4) schedule(static, 2) lastprivate(i)
    for (int j = 0; j < 11; j++) {
        i = omp_get_thread_num();
    }

    printf("i = %d\n", i);
    return 0;
}
```

- What do we expect the master thread to print at the end?

```
gauss ~/CodeBits759>
gauss ~/CodeBits759>
gauss ~/CodeBits759>
gauss ~/CodeBits759> g++ -fopenmp lastPrivateVarExample.cpp -Wall
gauss ~/CodeBits759> ./a.out
i = 1
gauss ~/CodeBits759>
```

# Harping On Data Scoping

- A key tenet of OpenMP: that of **data independence** between threads
- When parallel threads execute parallel regions it is assumed
  - Data dependencies are nonexistent across parallel regions, OR
  - If data dependencies exist, there will be no race conditions
- Since you placed the `omp parallel` directive in your code, it is **your responsibility** to make sure that data dependencies do not lead to race conditions
  - Compilers not smart enough and sometimes can't identify data dependency between what might look like independent parallel tasks

# Data Dependency Example [1/5]

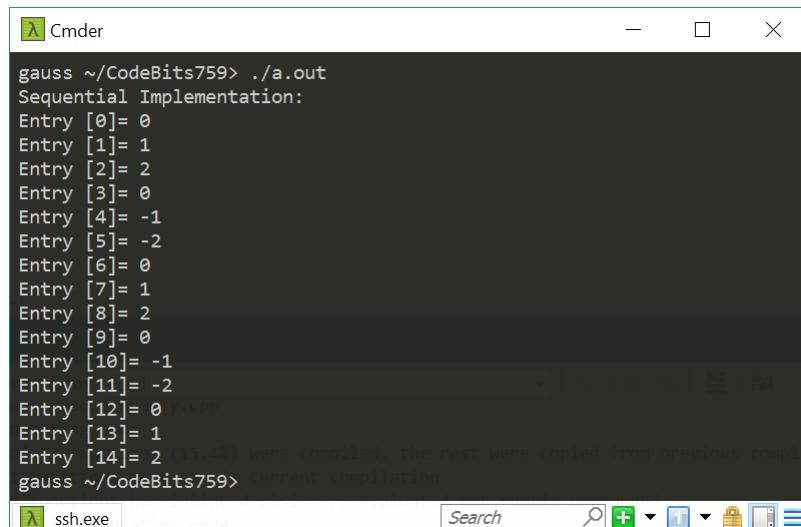
```
#include <omp.h>
#include <stdlib.h>
#include<iostream>

int main()
{
    const int N = 15;
    int* a = (int*)calloc(N, sizeof(int)); //array a has a bunch of zeroes

    //Basic idea: initialize the first "offset" entries of a and then
    //populate all the other entries of a based on these first entries
    const int offset = 3;
    for (int i = 0; i < offset; i++)
        a[i] = i;

    for (int i = offset; i < N; i++)
        a[i] -= a[i - offset];
     //print out the content of a
    std::cout << "Sequential Implementation:\n";
    for (int i = 0; i < N; i++)
        std::cout << "Entry [" << i << "] = " << a[i] << std::endl;

    return 0;
}
```



```
gauss ~/CodeBits759> ./a.out
Sequential Implementation:
Entry [0]= 0
Entry [1]= 1
Entry [2]= 2
Entry [3]= 0
Entry [4]= -1
Entry [5]= -2
Entry [6]= 0
Entry [7]= 1
Entry [8]= 2
Entry [9]= 0
Entry [10]= -1
Entry [11]= -2
Entry [12]= 0
Entry [13]= 1
Entry [14]= 2
```

# Data Dependency Example [2/5]

```
#include <omp.h>
#include <stdlib.h>
#include<iostream>

int main()
{
    const int N = 15;
    int* a = (int*)calloc(N, sizeof(int)); //array a has a bunch of zeroes; N of them.

    //Basic idea: initialize the first "offset" entries of a and then
    //populate all the other entries of a based on these first entries
    const int offset = 3;
    for (int i = 0; i < offset; i++)
        a[i] = i;

    #pragma omp parallel num_threads(2)
    {
        #pragma omp single
        std::cout << "Running w/ this many threads: " << omp_get_num_threads() << std::endl;

        #pragma omp for
        for (int i = offset; i < N; i++)
            a[i] -= a[i - offset];
    }

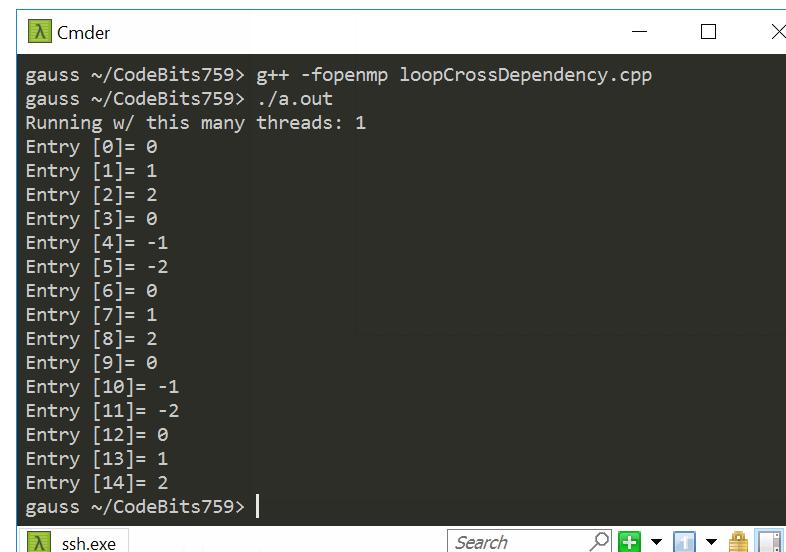
    //print out the content of a
    for (int i = 0; i < N; i++)
        std::cout << "Entry [" << i << "] = " << a[i] << std::endl;

    return 0;
}
```

We're going to keep changing this value here

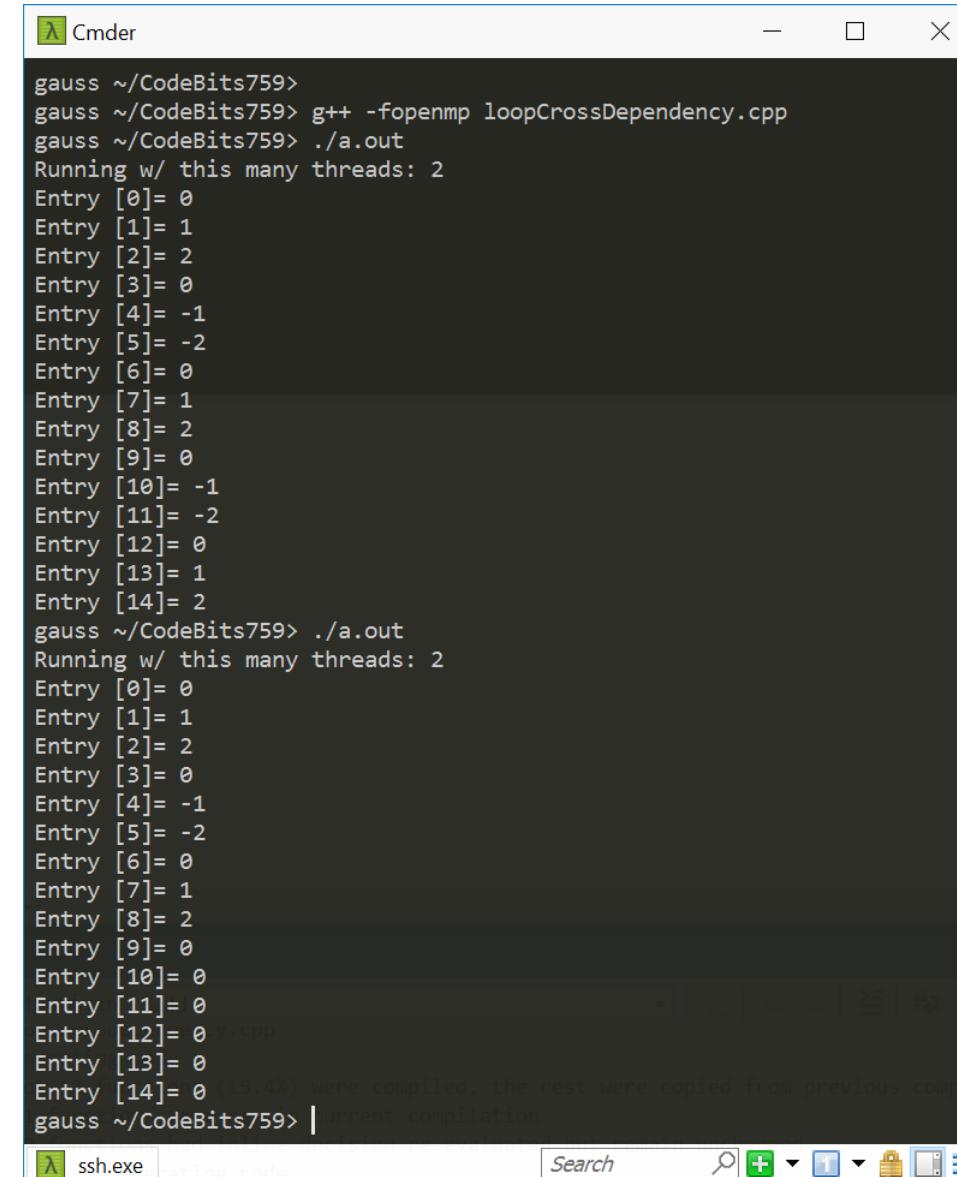
# Data Dependency Example: 1 Thread [3/5]

- Running OpenMP with one thread is essentially running sequentially



```
gauss ~/CodeBits759> g++ -fopenmp loopCrossDependency.cpp
gauss ~/CodeBits759> ./a.out
Running w/ this many threads: 1
Entry [0]= 0
Entry [1]= 1
Entry [2]= 2
Entry [3]= 0
Entry [4]= -1
Entry [5]= -2
Entry [6]= 0
Entry [7]= 1
Entry [8]= 2
Entry [9]= 0
Entry [10]= -1
Entry [11]= -2
Entry [12]= 0
Entry [13]= 1
Entry [14]= 2
gauss ~/CodeBits759>
```

# Data Dependency Example: 2 Threads [4/5]



```
gauss ~/CodeBits759>
gauss ~/CodeBits759> g++ -fopenmp loopCrossDependency.cpp
gauss ~/CodeBits759> ./a.out
Running w/ this many threads: 2
Entry [0]= 0
Entry [1]= 1
Entry [2]= 2
Entry [3]= 0
Entry [4]= -1
Entry [5]= -2
Entry [6]= 0
Entry [7]= 1
Entry [8]= 2
Entry [9]= 0
Entry [10]= -1
Entry [11]= -2
Entry [12]= 0
Entry [13]= 1
Entry [14]= 2
gauss ~/CodeBits759> ./a.out
Running w/ this many threads: 2
Entry [0]= 0
Entry [1]= 1
Entry [2]= 2
Entry [3]= 0
Entry [4]= -1
Entry [5]= -2
Entry [6]= 0
Entry [7]= 1
Entry [8]= 2
Entry [9]= 0
Entry [10]= 0
Entry [11]= 0
Entry [12]= 0
Entry [13]= 0
Entry [14]= 0
gauss ~/CodeBits759> current compilation
```

# Data Dependency Example: 3 Threads [5/5]

```
gauss ~/CodeBits759> g++ -fopenmp loopCrossDependency.cpp
gauss ~/CodeBits759> ./a.out
Running w/ this many threads: 3
Entry [0]= 0
Entry [1]= 1
Entry [2]= 2
Entry [3]= 0
Entry [4]= -1
Entry [5]= -2
Entry [6]= 0
Entry [7]= 0
Entry [8]= 0
Entry [9]= 0
Entry [10]= 0
Entry [11]= 0
Entry [12]= 0
Entry [13]= 0
Entry [14]= 0
gauss ~/CodeBits759>
```

# Quiz: what's shared and what's not?

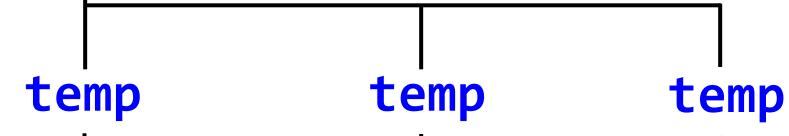
```
float A[10];
main() {
    int index[10];
#pragma omp parallel
    {
        Work(index);
    }
    printf("%d\n", index[1]);
}
```

A, index, and count are shared by all threads, but temp is local to each thread

```
extern float A[10];
void Work(int* index)
{
    float temp[10];
    static int count;
    <...>
}
```

Assumed to be in another translation unit

A, index, count



A, index, count

# Data Scoping – Words of Wisdom

- When in doubt,
  - Explicitly indicate who is what
  - Write a toy example to experiment
- OpenMP provides the clause **default(none)**
  - In this case, the compiler requires that we specify scope for each variable declared outside the block
  - Variable declared inside the block are private
- Data scoping: a common sources of errors in OpenMP

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 22

03/13/2020

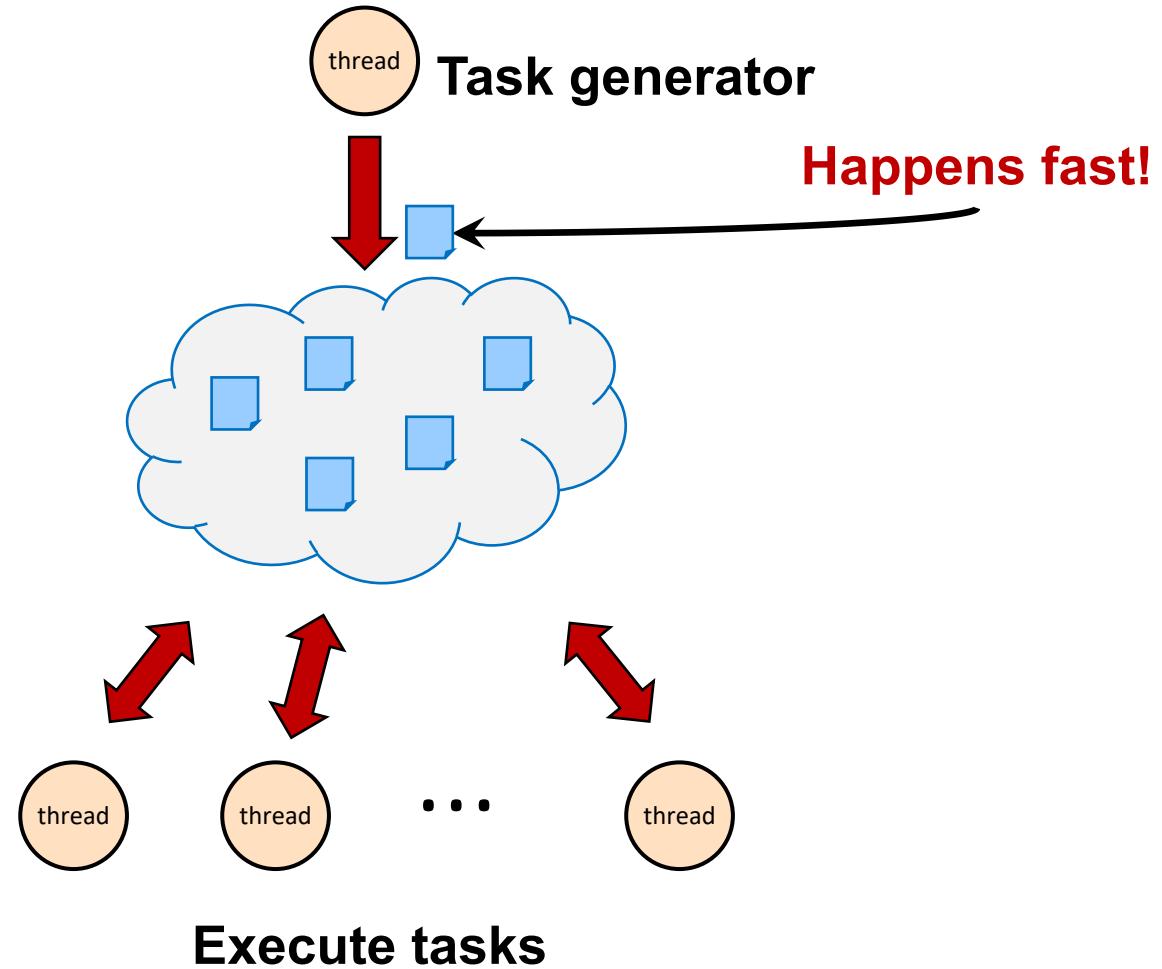
# The quote the day

“Det er vanskeligt at spaa, især naar det gælder fremtiden.”  
“It is difficult to make predictions, especially about the future.”

-- Danish folklore

# Before we get going...

- Last time:
  - Discussion of default projects
  - Tasks, wrap up
  - Variable scoping issues
- Today:
  - Synchronization issues
  - Performance issues
  - NUMA & Cache issues
- Other tidbits:
  - Last lecture with folks in the audience
    - Not clear yet how I'll deliver the last 6 lectures
    - Ideally, I'd keep recording in this same room
  - ME759 Exam: April 15, at 7:15 PM, in 1106ME
    - Review: Tu, April 14, at 7:00 PM, in Canvas



# Work Plan, OpenMP

- What is OpenMP?

- Parallel regions

- Work sharing – Parallel tasks

- Variable Scoping Issues

- Synchronization**

- Performance issues

- Loose ends

# The **barrier** Construct

- This is an **explicit** barrier synchronization
- Each thread waits until all threads arrive
- Example below assumes that in `DoSomeWork(X,Y)` X is an input and Y is the output:

```
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A,B); // input is A, output is B
    #pragma omp barrier
    DoSomeWork(B,C); // input is B, output is C
}
```

Quick question: how about breaking this into two parallel regions?

# Implicit Barriers

- Several OpenMP constructs have *implicit* barriers
  - `parallel` – necessary barrier – cannot be removed
  - `for`
  - `single`
  - `sections`
- Unnecessary barriers hurt performance and can be removed with the `nowait` clause
  - The `nowait` clause is applicable to:
    - `for` directive
    - `single` directive
    - `sections` directive

# The **nowait** Clause

```
#pragma omp for nowait  
for(...)  
{ [...] }
```

```
#pragma single nowait  
{ [...] }
```

- Use when threads unnecessarily wait between independent computations

```
#pragma omp for schedule(static) nowait  
for(int i=0; i<n; i++)  
    a[i] = bigFunc1(i);  
  
#pragma omp for schedule(dynamic,1)  
for(int j=0; j<m; j++)  
    b[j] = bigFunc2(j);
```

# Example, Synchronization Issue: Computing the Dot Product

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
#pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

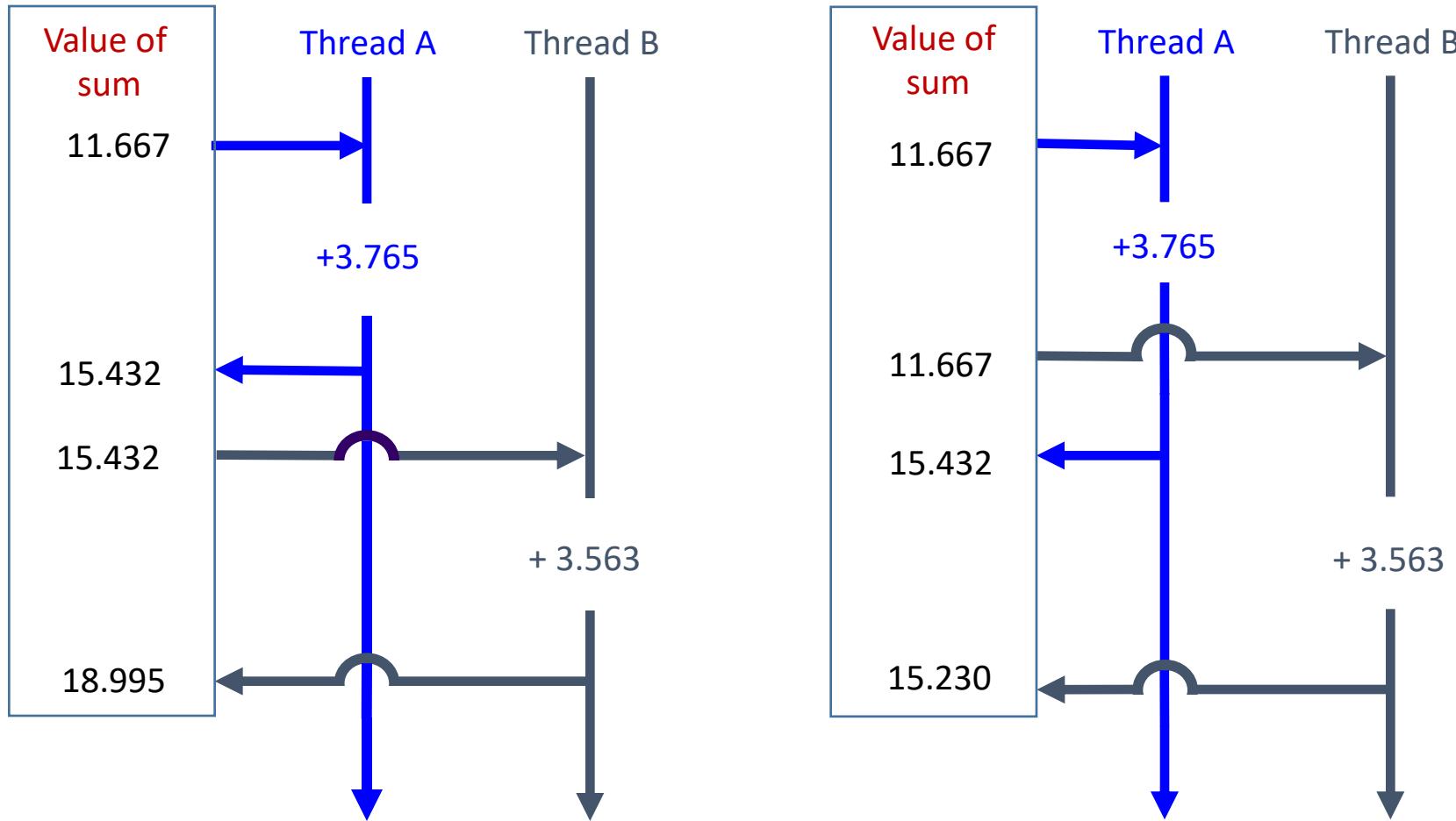
**What is wrong here?**

# Race Condition

- **Race condition:** two or more threads access a shared variable at the same time
  - Leads to nondeterministic behavior
- Both Thread A and Thread B are executing the statement

```
sum += a[i] * b[i];
```

# Race conditions (data hazards) – two possible scenarios



**Order of thread execution causes non-deterministic behavior in a data race**

# Example: Computing the Dot Product, second attempt

- The **critical** construct: protects access to shared, modifiable data
- The **critical** section allows only one thread to enter it at a given time

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
#pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
#pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

# The **critical** Directive

**#pragma omp critical(*(name)*)**: defines a critical region on a structured block

Threads wait their turn – only one at a time  
calls `consum()` thereby preventing race  
conditions

Naming the critical construct `RES_lock` is  
optional but highly **recommended**

```
float* RES;
//... more stuff happens here
#pragma omp parallel num_threads(4)
{
    #pragma omp for
        for(int i=0; i<8; i++) {
            float B = job1(i);

# pragma omp critical(RES_lock)
            consum(B, RES);

            job2(B);
        }
}
```

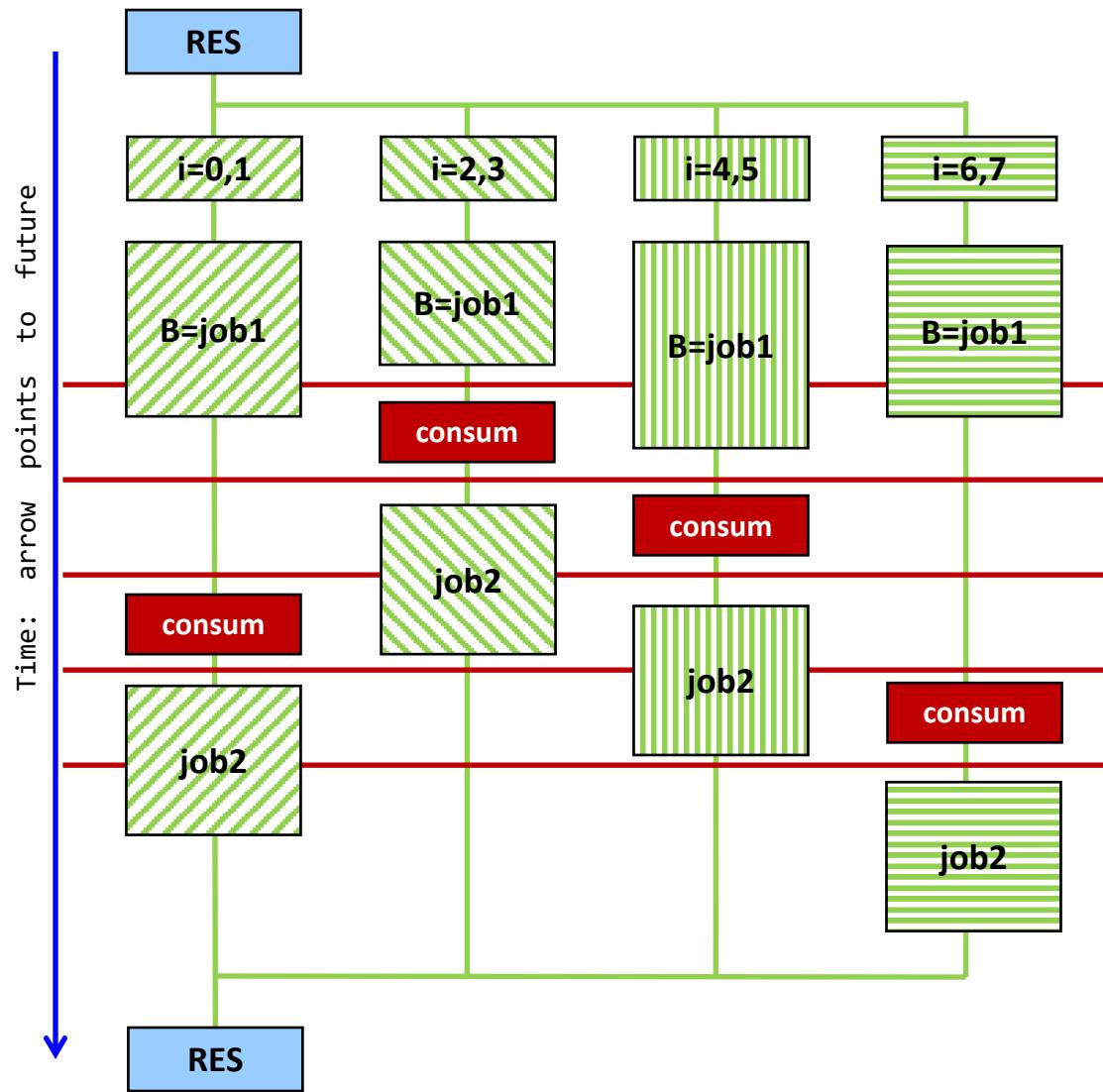
# OpenMP **critical** Construct

```
float* RES;

#pragma omp parallel
{
#pragma omp for num_threads(4)
for(int i=0; i<8; i++) {
    float B = job1(i);

#pragma omp critical(RES_lock)
    consum(B, RES);

    job2(B);
}
}
```



# The **atomic** Directive

- Applies only to simple update of memory location
  - It's a **guarded memory access operation**
  - Also seen in GPU computing
- Special case of a **critical** section
  - Atomic introduces less overhead than **critical**

```
index[0] = 2;  
index[1] = 3;  
index[2] = 4;  
index[3] = 5;  
index[4] = 4;  
index[5] = 0;  
index[6] = 5;  
index[7] = 1;
```

```
#pragma omp parallel for shared(x, y, index)  
    for (i = 0; i < n; i++) {  
#pragma omp atomic  
    x[index[i]] += work1(i);  
    y[i] += work2(i);  
}
```

# OpenMP **atomic** Directive

- Unlike a **critical** directive, the **atomic** directive:
  - Can only protect a single assignment
  - Applies only to simple **update of memory**
- The assignment that can be protected by **atomic** must be one of:
  - x <op>= <expression>;
  - x++;
  - ++x;
  - x--;
  - x;
- <op> can be one of: +, \*, -, /, &, ^, |, <<, >>
- <expression> must not reference x
- Only the load and store of x is protected (not <expression>)

# Synchronisation is costly. Avoid if possible.

- Barriers can be very expensive
  - Typically 1000s cycles to synchronise
- Recall that you have explicit and implicit barriers
- Avoid barriers via:
  - **Careful** use of the `nowait` clause
  - Parallelize at the outermost level possible
    - May require re-ordering of loops / indexes
  - Use `critical` or `atomic`. Yet this may impact performance too

# The reduction Construct

```
#pragma omp parallel for reduction(+:sum)
    for(i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
```

- Local copy of **sum** for each thread engaged in the reduction is private
  - Each local sum initialized to the identity operand associated with the operator that comes into play
    - Here we have “+”, so it’s a zero (0)
- All local copies of **sum** added together and stored in “global” variable

# The reduction Clause

```
#pragma omp for reduction(op:list)
```

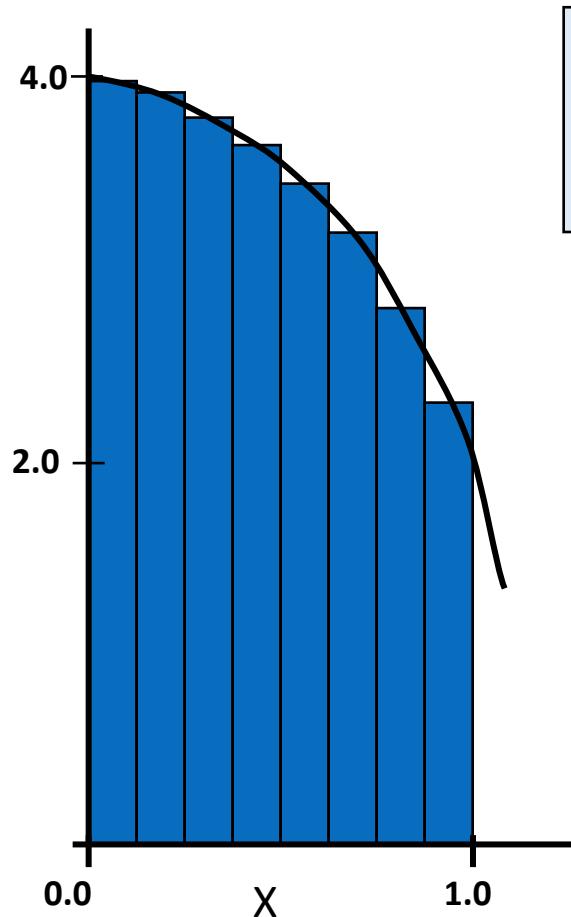
- The variables in `list` will be shared in the enclosing parallel region
- Here's what happens inside the parallel or work-sharing construct:
  - A private copy of each list variable is created and initialized depending on the “`op`”
  - These copies are updated locally by threads
- At end of construct, local copies are combined through “`op`” into a single value

# C/C++ reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

Operand	Initial value	Notes
+	0	
*	1	
-	0	
&	$\sim 0$	Bitwise AND
	0	Bitwise OR
$\wedge$	0	Bitwise XOR
$\&\&$	1	Logical AND
$\ $	0	Logical OR

# reduction Example: Numerical Integration



$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

```
static long num_steps=100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;

    step = 1.0/num_steps;
    for (i=0; i< num_steps; i++) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

# Scaling Analysis: Computation of $\pi$

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

void main() {
    static long num_steps = 1000000;
    const double step = 1.0 / (double)num_steps;
    double sum = 0.0;
    double startT = omp_get_wtime();
    const int nThreadsUsed = 1;
#pragma omp parallel num_threads(nThreadsUsed)
    {
#pragma omp for reduction(+:sum)
        for (int i = 0; i < num_steps; i++) {
            double x = (i + 0.5)*step;
            double dummy = 1. / (1. + x*x);
            sum += dummy;
        }
    }
    double myPi = 4.*step * sum;
    double timeElapsed = omp_get_wtime() - startT;

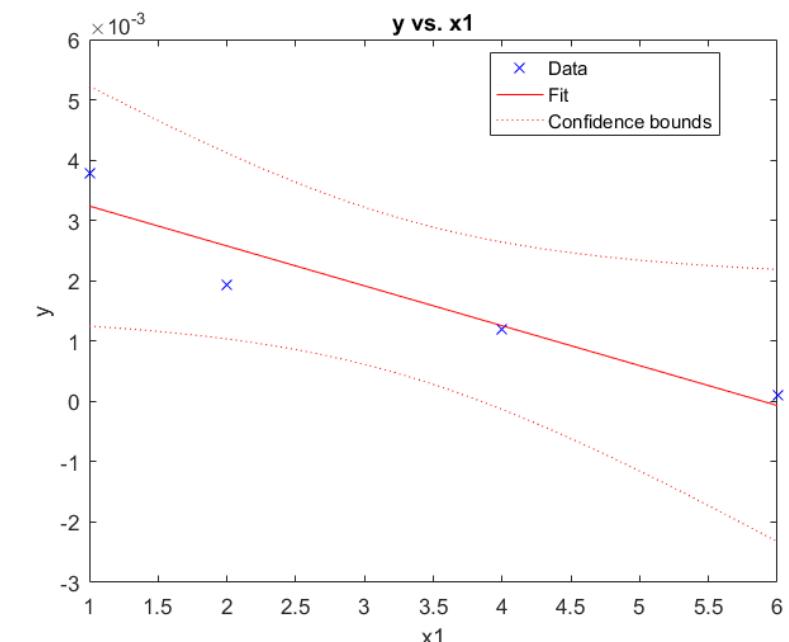
    printf("Pi is: %f\n", myPi);
    printf("It took this long [s]: %f\n", timeElapsed);
    printf("Used this many threads: %d\n", nThreadsUsed);
}
```

```
C:\Windows\system32\cmd.exe
Pi is: 3.14159
It took this long [s]: 0.00377254
Used this many threads: 1
Press any key to continue . . .
```

```
C:\Windows\system32\cmd.exe
Pi is: 3.14159
It took this long [s]: 0.00119085
Used this many threads: 4
Press any key to continue . . .
```

```
C:\Windows\system32\cmd.exe
Pi is: 3.14159
It took this long [s]: 0.00194108
Used this many threads: 2
Press any key to continue . . .
```

```
C:\Windows\system32\cmd.exe
Pi is: 3.14159
It took this long [s]: 0.0009728
Used this many threads: 6
Press any key to continue . . .
```



# Wait, it can get even better: the **simd** directive

```
#include <omp.h>
#include <stdio.h>
#include <sys/time.h>

int main() {
    int num_steps = 1000000;
    const double step = 1.0 / (double)num_steps;
    double sum = 0.0;
    const int nThreadsUsed = 4;

    timeval start, end;
    gettimeofday(&start, NULL);
#pragma omp parallel num_threads(nThreadsUsed)
    {
#pragma omp for simd reduction(+:sum)
        for (int i = 0; i < num_steps; i++) {
            double x = (i + 0.5) * step;
            double dummy = 1. / (1. + x * x);
            sum += dummy;
        }
    }
    double myPi = 4. * step * sum;
    gettimeofday(&end, NULL);
    int time_used = 1000000 * (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec);

    printf("Pi is: %f\n", myPi);
    printf("It took this long [s]: %d\n", time_used);
    printf("Used this many threads: %d\n", nThreadsUsed);

    return 0;
}
```

Might be bonus problem on next assignment  
(if we figure out compiler issues)

# OpenMP reduction Works Beyond **for** Loops [1/2]

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

int processNumber(int someNumber) {
    return 2 * someNumber;
}

int main() {
    int sum = 0;

#pragma omp parallel sections num_threads(3) reduction(+:sum)
{
#pragma omp section
{
    int someVal = processNumber(omp_get_thread_num());
    printf("First section: %d\n", someVal);
    sum += someVal;
}
#pragma omp section
{
    int someOtherVal = processNumber(omp_get_thread_num());
    printf("Second section: %d\n", someOtherVal);
    sum += someOtherVal;
}
#pragma omp section
{
    int dummy = processNumber(omp_get_thread_num());
    printf("Last section: %d\n", dummy);
    sum += dummy;
}
printf("Sum is: %d\n", sum);
return 0;
}
```



```
gauss ~/CodeBits> g++ -fopenmp reductionSectionsExample.cpp *  
gauss ~/CodeBits> ./a.out  
Last section: 2  
Second section: 4  
First section: 0  
Sum is: 6  
gauss ~/CodeBits> ./a.out  
First section: 2  
Last section: 0  
Second section: 4  
Sum is: 6  
gauss ~/CodeBits> ./a.out  
First section: 2  
Last section: 4  
Second section: 0  
Sum is: 6  
gauss ~/CodeBits> ./a.out  
Second section: 2  
First section: 0  
Last section: 4  
Sum is: 6  
gauss ~/CodeBits> ./a.out  
Last section: 2  
Second section: 4  
First section: 0  
Sum is: 6  
gauss ~/CodeBits> ./a.out  
First section: 4  
Second section: 2  
Last section: 0  
Sum is: 6  
gauss ~/CodeBits> ./a.out  
First section: 4  
Last section: 2  
Second section: 0  
Sum is: 6  
gauss ~/CodeBits> ./a.out  
Second section: 4  
Last section: 0  
First section: 2  
Sum is: 6  
gauss ~/CodeBits> ./a.out  
Last section: 0  
Second section: 2  
First section: 4  
Sum is: 6  
gauss ~/CodeBits> |
```

**NOTE:** sum is always the same. What's called "First section" is not necessarily executed by thread with smallest id.

# OpenMP reduction Works Beyond for Loops [2/2]

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

int processNumber(int someNumber) {
    return 2 * someNumber;
}

int main() {
    int sum = 0;

#pragma omp parallel sections num_threads(4) reduction(+:sum)
    {
#pragma omp section
    {
        int someVal = processNumber(omp_get_thread_num());
        printf("First section: %d\n", someVal);
        sum += someVal;
    }
#pragma omp section
    {
        int someOtherVal = processNumber(omp_get_thread_num());
        printf("Second section: %d\n", someOtherVal);
        sum += someOtherVal;
    }
#pragma omp section
    {
        int dummy = processNumber(omp_get_thread_num());
        printf("Last section: %d\n", dummy);
        sum += dummy;
    }

    printf("Sum is: %d\n", sum);
    return 0;
}
```



Quiz: Why different sum values when we run this code?

```
cmder
gauss ~/CodeBits> g++ -fopenmp reductionSectionsExample.cpp
gauss ~/CodeBits> ./a.out
First section: 2
Last section: 6
Second section: 4
Sum is: 12
gauss ~/CodeBits> ./a.out = processNumber(omp_get_thread_num())
Last section: 6  printf("Last section: %d\n", dummy);
First section: 2 sum += dummy;
Second section: 0
Sum is: 8
gauss ~/CodeBits> ./a.out
First section: 2 sum is: %d\n", sum);
Second section: 0
Last section: 4
Sum is: 6
gauss ~/CodeBits> ./a.out
Second section: 6
First section: 2
Last section: 4
Sum is: 12
gauss ~/CodeBits> ./a.out
First section: 2
Second section: 0
Last section: 4
Sum is: 6
gauss ~/CodeBits> ./a.out
First section: 2
Second section: 4
Last section: 0
Sum is: 6
gauss ~/CodeBits> ./a.out
Last section: 2
Second section: 4
First section: 6
Sum is: 12
gauss ~/CodeBits> ./a.out
Second section: 4
First section: 0
Last section: 6
Sum is: 10
gauss ~/CodeBits> in current compilation
```

ssh.exe

Search

200

# A Histogram Example

[more advanced, uses reduction w/ C++ STL]

```
#include <omp.h>
#include <algorithm>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <vector>

// Read in vector of pixel values
std::vector<int> *read_image(FILE *fp, int d1, int d2) {
    int xVal;
    int len = d1 * d2;
    std::vector<int> *x = new std::vector<int>(len);
    for (int i = 0; i < len; i++) {
        fscanf(fp, "%d ", &xVal);
        (*x)[i] = xVal;
    }
    return x;
}
```

```
int main(int argc, char *argv[]) {
    // Set number of threads
    const auto num_threads = std::atoi(argv[1]);
    omp_set_num_threads(num_threads);

    // Read in Image
    FILE *fid = std::fopen("picture.inp", "r");
    int d1 = 1800;
    int d2 = 1200;
    int len = d1 * d2;
    std::vector<int> *image = read_image(fid, d1, d2);

    // Histogram Counter
    std::vector<int> histCounter = std::vector<int>(7);
    std::fill(histCounter.begin(), histCounter.end(), 0);

    // Declaration of the std::vector addition
#pragma omp declare reduction(vec_int_add : std::vector<int> : \
    std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
    initializer(omp_priv = omp_orig))

    double startTime = omp_get_wtime(); // Start Clock
#pragma omp parallel for reduction(vec_int_add : histCounter)
    // Perform the histogram; shared(histCounter)
    for (int jj = 0; jj < len; jj++)
        (histCounter)[(*image)[jj]]++;

    double endTime = omp_get_wtime(); // Stop Clock

    // Output information
    for (int ii = 0; ii < 7; ii++)
        std::cout << (histCounter)[ii] << std::endl;
    std::cout << num_threads << std::endl;
    std::cout << std::setprecision(16) << (endTime - startTime) * 1000 << std::endl;

    // delete(histCounter);
    delete (image);
    return 0;
}
```

# Work Plan, OpenMP

- What is OpenMP?

- Parallel regions

- Work sharing – Parallel tasks

- Variable Scoping Issues

- Synchronization

- Performance issues**

- Loose ends

# Performance

- Easy to write OpenMP code. Not necessarily easy to write efficient OpenMP code
- More common causes for performance not being what you hoped it could be
  - Too much sequential code in your app
  - Too much communication
  - Load imbalance
  - Synchronization
  - Compiler (non-)optimization

# Too much sequential code in your app

- Beware of Amdahl's law
  - If your algorithm/solution is sequential in nature, maybe it's time to change it
- Thinking within the context of OpenMP
  - All code outside of parallel regions is **sequential**
    - In fact, even if in parallel region, code might not run in parallel – recall the `master`, `single` and `critical` directives)
- Bottom line:
  - Seek to reduce amount of execution time where only one thread executes code

# Too much communication [that you fail to notice]

- On shared memory machines, OpenMP can lead to heavy duty memory operations
- Memory accesses are expensive (unless cached)
  - ~100 cycles for a main memory access compared to 1-3 cycles for a flop
- Unlike the message passing model (discussed later, for MPI), communication is spread throughout the program
  - Hard to analyse and monitor; transparent to user, **hard to pin down**
- When you look at your code you should understand where the data that you manipulate comes from

# Load Imbalance

- Load imbalance: one thread gets too much work, while the others idle waiting for it
  - Can arise from both communication and computation
- Happen most often in conjunction with `for` loops; worth experimenting with different scheduling options
  - Can use `schedule(runtime)`
    - Helps with experimentation, no need to recompile
    - Example: `setenv OMP_SCHEDULE "dynamic,5"`
- If no available scheduling appropriate, contemplate taking the load balancing problem in your hands

# Synchronisation

- Barriers can be very expensive
  - Typically 1000s cycles to synchronise
- Avoid barriers via:
  - Careful use of the `nowait` clause
  - Parallelize at the outermost level possible
    - May require re-ordering of loops +/ indexes
  - Use of `critical` / `atomic` routines may impact performance

# Quiz: Parallelization of the outer loop & synchronization???

- How come parallelization of outer loop (instead of inner loop) will be more advantageous given OpenMP implicit synchronization at end of parallel region?
- NOTE: code at right goes one step further and collapse two loops into one
  - There is no “outer loop” anymore
  - You must make sure there is no data dependencies when you ask for this collapse

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < M; j++)
    {
        //do something here, based on i and j.
    }
}
```

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < M; j++)
    {
        //do something here, based on i and j.
        //must have no data dependency
    }
}
```

# Compiler (non)-optimization

- Sometimes the addition of parallel directives can prevent the compiler from performing sequential optimization
- Symptoms:
  - Parallel code running with 1 thread has longer execution and higher instruction count than sequential code
- Sometimes, it really helps to make shared data private and local to a function

# Attractive Features of OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can implement complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, code is easy to read

# Further Reading, OpenMP

- Michael Quinn (2003) Parallel Programming in C with MPI and OpenMP
- Chapman, Barbara, Gabrielle Jost, and Ruud van der Pas. (2008) Using OpenMP, Cambridge, MA: MIT Press.
- Kendall, Ricky A. (2007) Threads R Us, [http://www.nccs.gov/wp-content/training/scaling\\_workshop\\_pdfs/threadsRus.pdf](http://www.nccs.gov/wp-content/training/scaling_workshop_pdfs/threadsRus.pdf)
- Mattson, Tim, and Larry Meadows (2008) SC08 OpenMP “Hands-On” Tutorial, <http://openmp.org/mp-documents/omp-hands-on-SC08.pdf>
- LLNL OpenMP Tutorial, <https://computing.llnl.gov/tutorials/openMP/>
- OpenMP.org, <http://openmp.org/>
- OpenMP 5.0 API Syntax Reference Guide – very nice one-stop shopping:
  - <https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-111802-print.pdf>

# Work Plan, OpenMP

- What is OpenMP?

- Parallel regions

- Work sharing – Parallel tasks

- Variable Scoping Issues

- Synchronization

- Performance issues

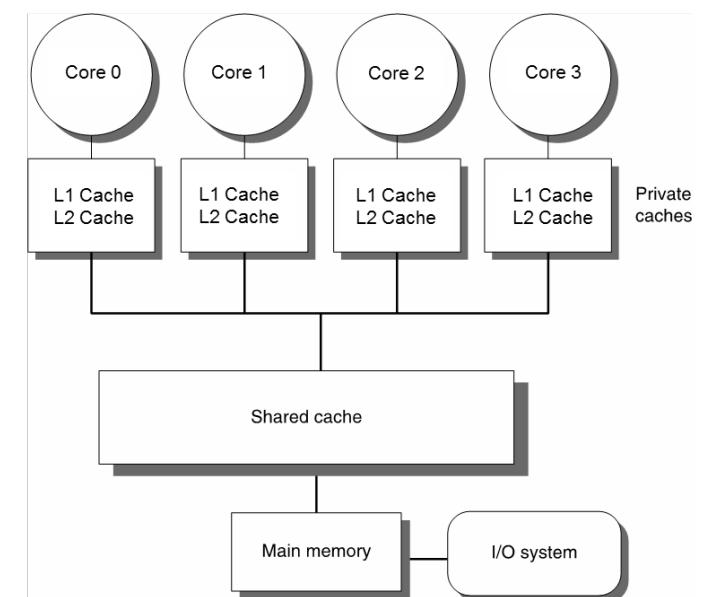
- **Loose ends**

# Loose ends: topics discussed

- Non-uniform memory access (NUMA) issues
- Cache issues

# Preamble: the SMP model

- Up to this point, not concerned with the mechanics of a shared memory access
- Model embraced when discussing OpenMP thus far: **SMP**
  - Symmetric Multi-Processor model
  - All threads/cores have similar overhead to accessing the main memory
- SMP model works well for one-chip, four core systems



# SMP: not a good model, for today's high-end hardware

- Today's server/cluster power nodes have many CPUs, each with many cores
  - "multi-socket configurations"
- SMP model doesn't quite apply
- Not all memory accesses are equal (loss of "symmetry")
  - A certain core has access time to certain system memory smaller than or equal to any other core on the node

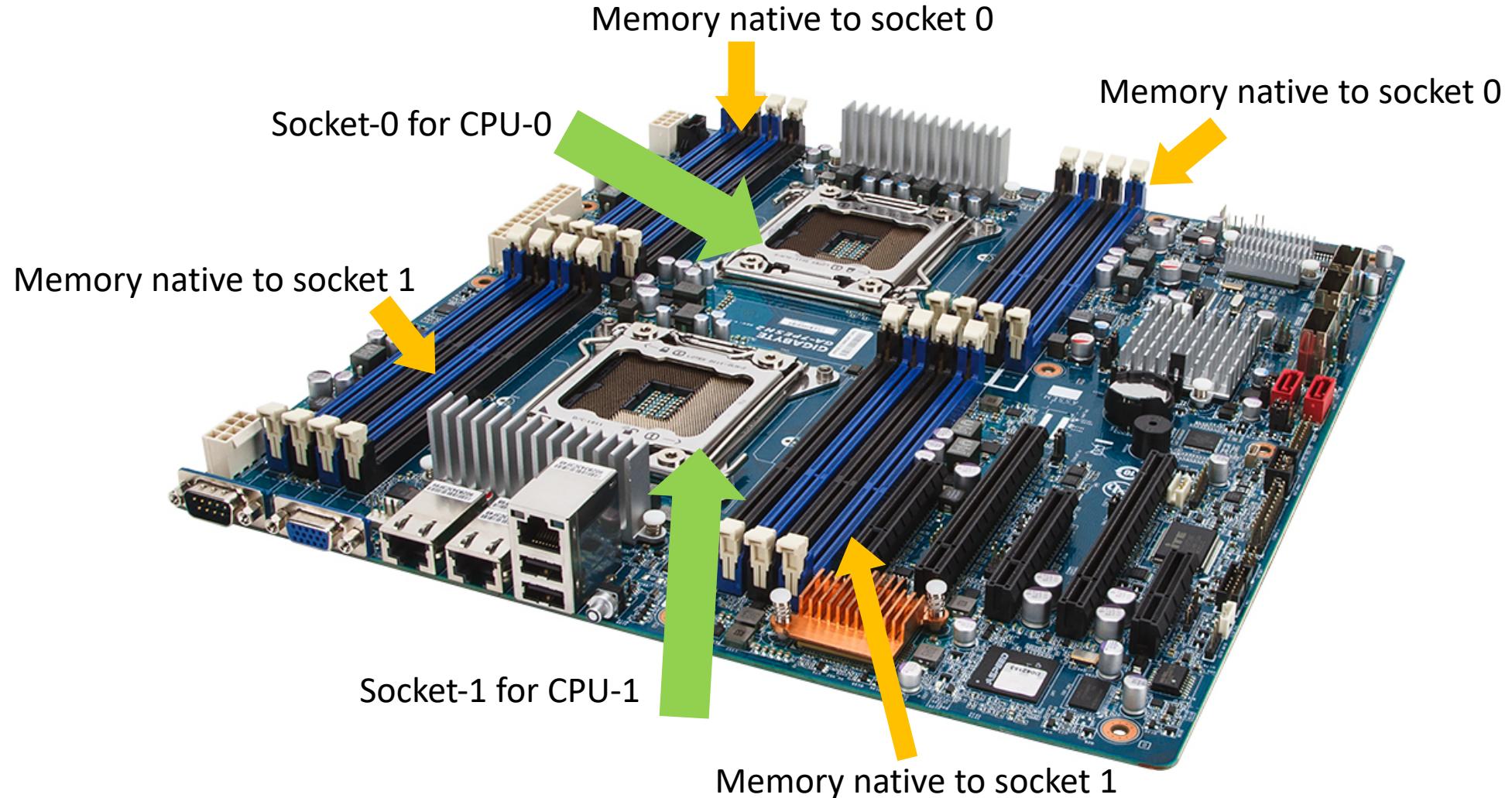
Side note (nomenclature issue):

- **server node**: used a lot by Amazon and Facebook for database manipulations
- **cluster node**: mostly used by Oak Ridge or Argonne National Lab in their supercomputers

# Multi-Socket Configurations

[1/2]

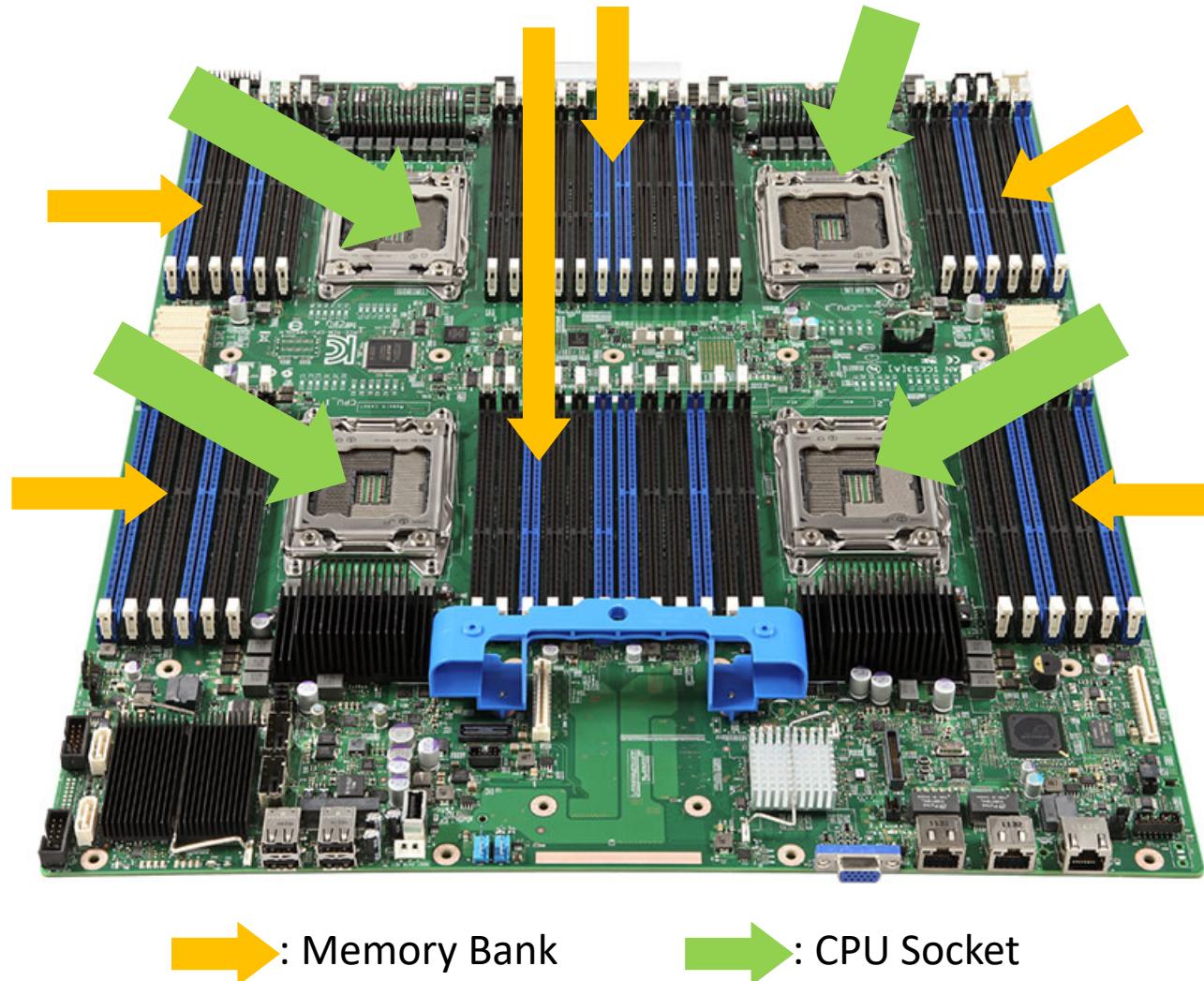
[two sockets, in this case]



# Multi-Socket Configurations

[2/2]

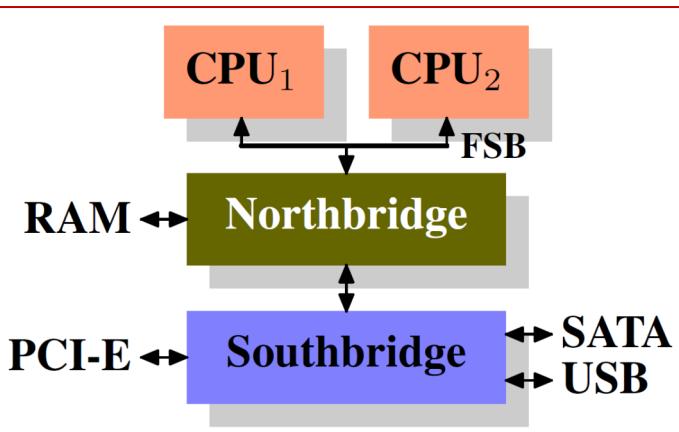
[four sockets: Intel S4600LT2 Xeon E5-4600 Chipset-C600-A Socket-R LGA-2011 1.46Tb DDR3-1600MHz Server Motherboard]



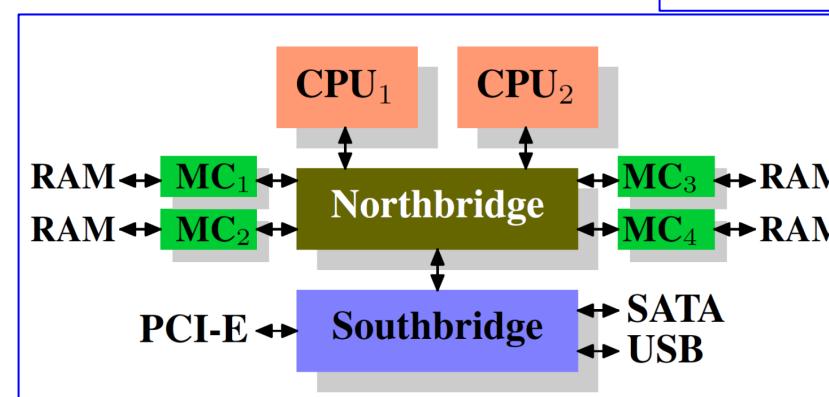
# Examples, Recent Layouts

- AMD Zen (EPYC 7601): Two socket system
  - 32 cores per processor  $\Rightarrow$  128 threads per socket
  - 8 channel memory access per socket
  - CPUs communicate with each other via high-speed bus
  - Each processor:  $\approx \$4,400$
- Intel Xeon Scalable (Platinum 8180): Eight socket system
  - 28 cores per processor  $\Rightarrow$  56 threads per socket
  - 6 channel memory access per socket
  - Each socket can only link directly to 3 neighboring sockets
  - Each processor  $\approx \$5,000$

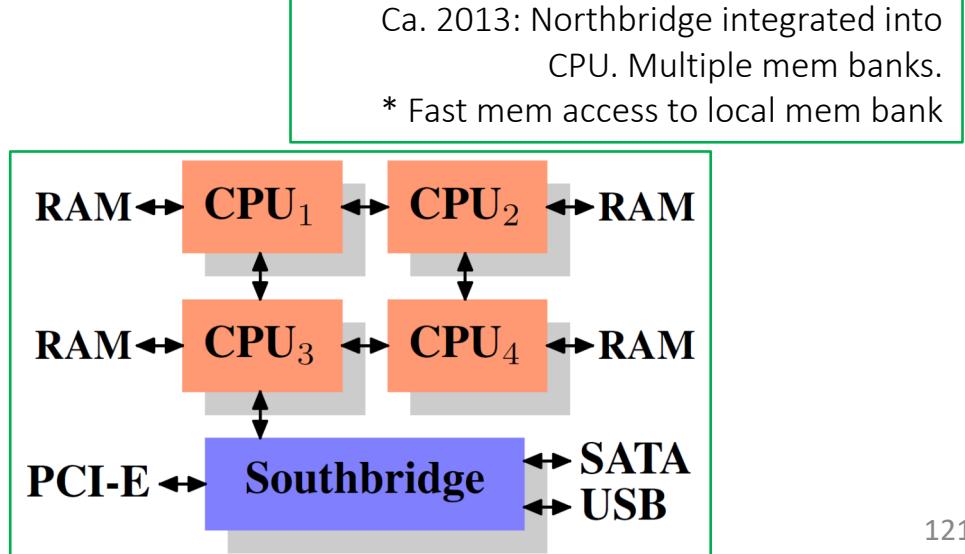
# Hardware Evolution: CPU count & Memory Access Solutions [Intel Specific]



Ca. 2005: RAM access streamed through Northbridge, unique Memory Controller.  
\* Poor scalability



Ca. 2008: RAM access streamed through Northbridge but multiple mem banks.  
\* Poor yet faster mem access



Ca. 2013: Northbridge integrated into CPU. Multiple mem banks.  
\* Fast mem access to local mem bank

# Non-uniform memory access (NUMA)

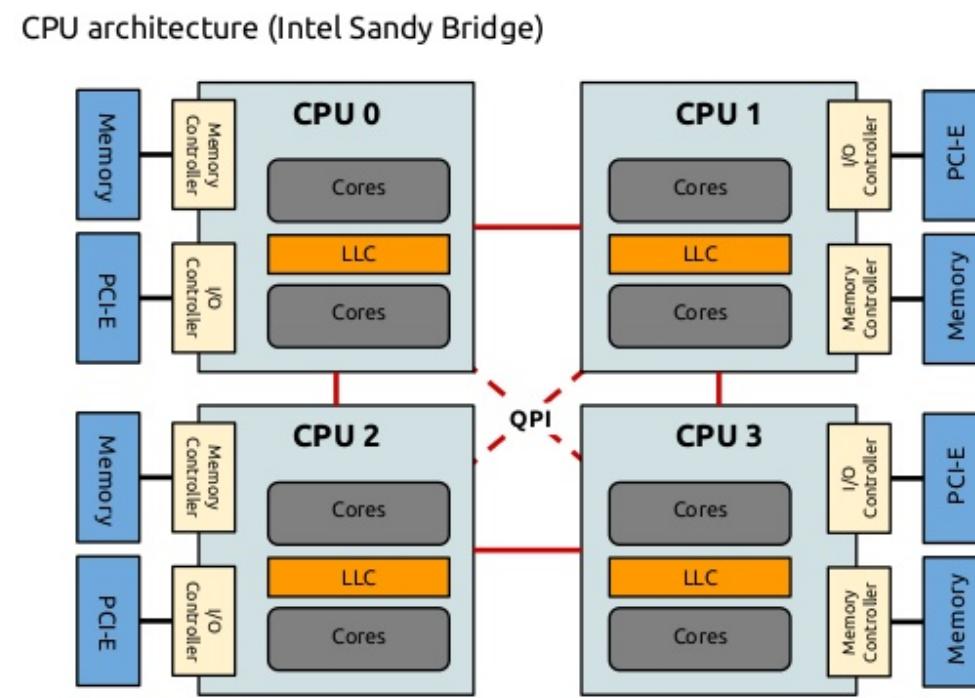
- NUMA punchline: cost of memory access depends on which memory bank stores your data
- Example:
  - Assume four memory banks: RAM\_1, RAM\_2, RAM\_3, RAM\_4
  - Assume four CPU sockets: CPU\_A, CPU\_B, CPU\_C, CPU\_D
  - RAM\_1 bank acts as “native memory” for CPU\_A, ..., RAM\_4 bank acts as “native memory” for CPU\_D
- Remarks:
  - Certainly, any CPU can access any memory bank
  - Thread running on a core of CPU\_A will take less time to access data stored in bank RAM\_1 than in RAM\_3

# The NUMA Factor

- The ratio between the largest and shortest \*average\* amount of time for a thread running on a particular core to reach data in memory
- Facts, related to the **NUMA factor**
  - A low NUMA factor is desirable, it suggests not much of a difference which bank data is stored
  - A high NUMA encountered in expensive hardware solutions that have hundreds to thousands of CPUs yet they have a \*shared\* memory layout
    - SGI's Altix back in the day
    - Less common today
  - NOTE: A system w/ NUMA factor = 1 is an SMP system

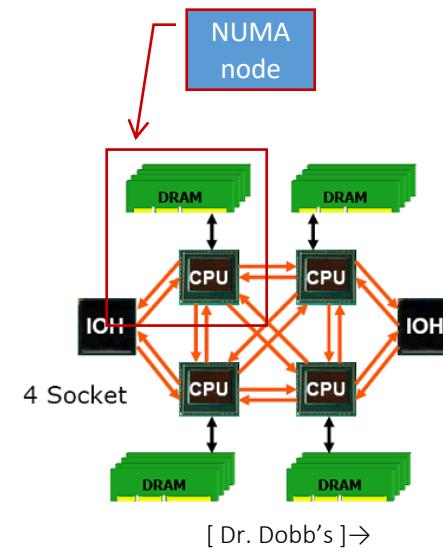
# NUMA – A Schematic [Intel specific]

- Image below shows a motherboard w/ four sockets
- LLC: last level cache (these days, most often it is L3 cache)
- QPI: QuickPath Interconnect (more later; also, replaced by Ultra Path Interconnect more recently (Xeon Skylake, 2017))



# “NUMA Node”

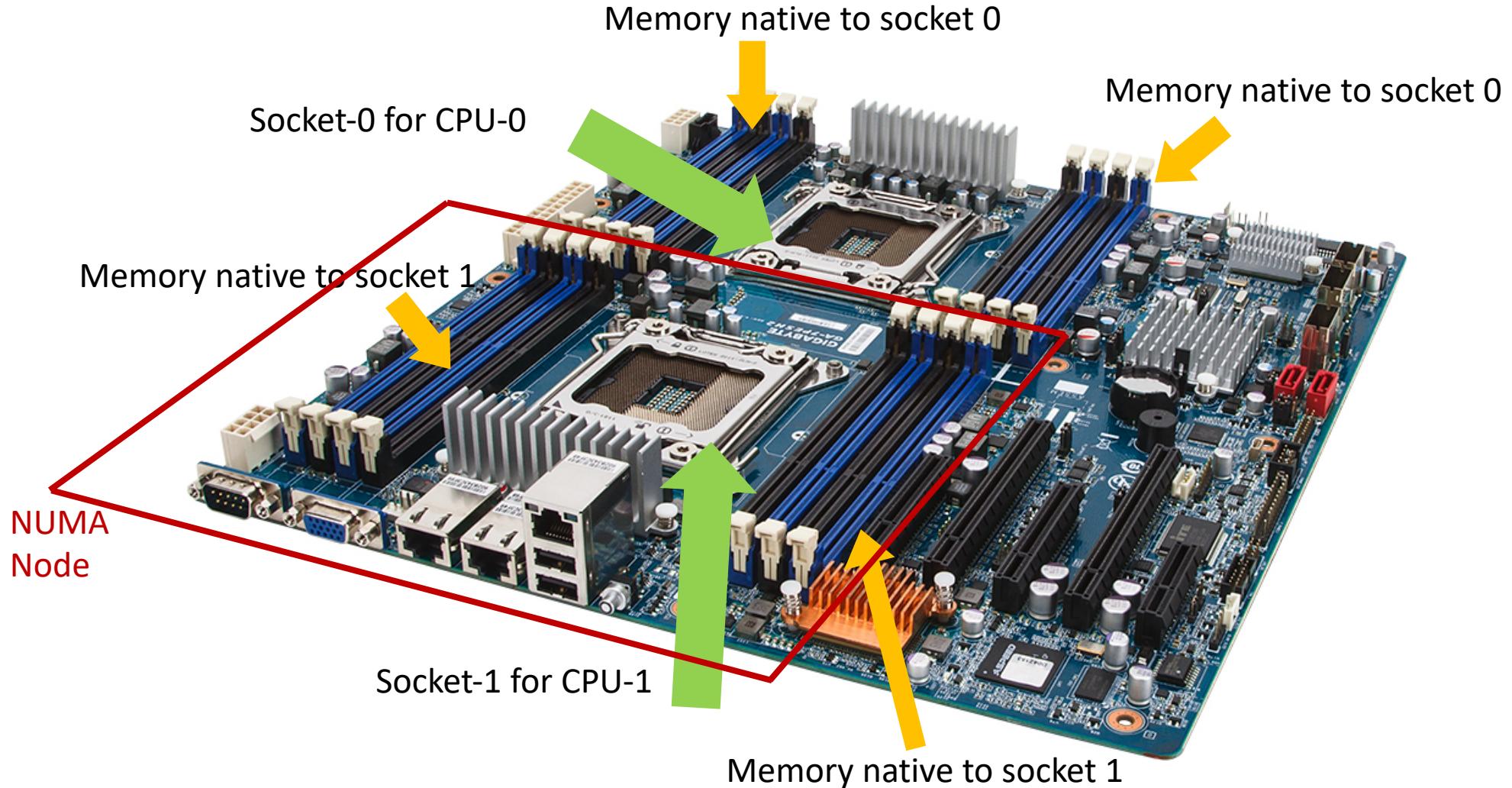
- Framing the discussion:
  - Focus on the case of **\*one\*** motherboard that
    - Hosts multiple sockets/multiple cores
    - Hosts multiple mem banks
  - This is the commodity hardware of today
- NUMA node
  - Processors **and** memory bank next to it
    - Combines to look much like a small SMP system in its own right
- Notation: IOH – Input/Output Hub



# Multi-Socket Configurations

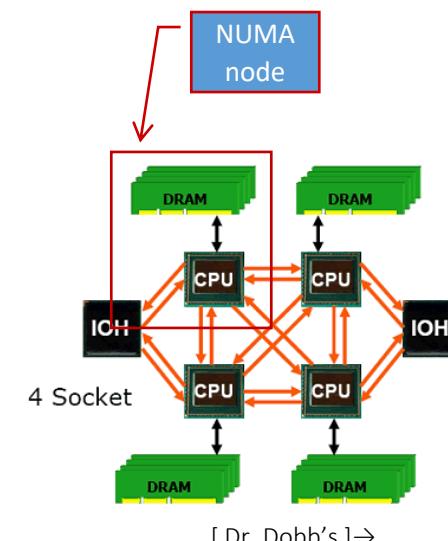
[1/2]

[two sockets, in this case]



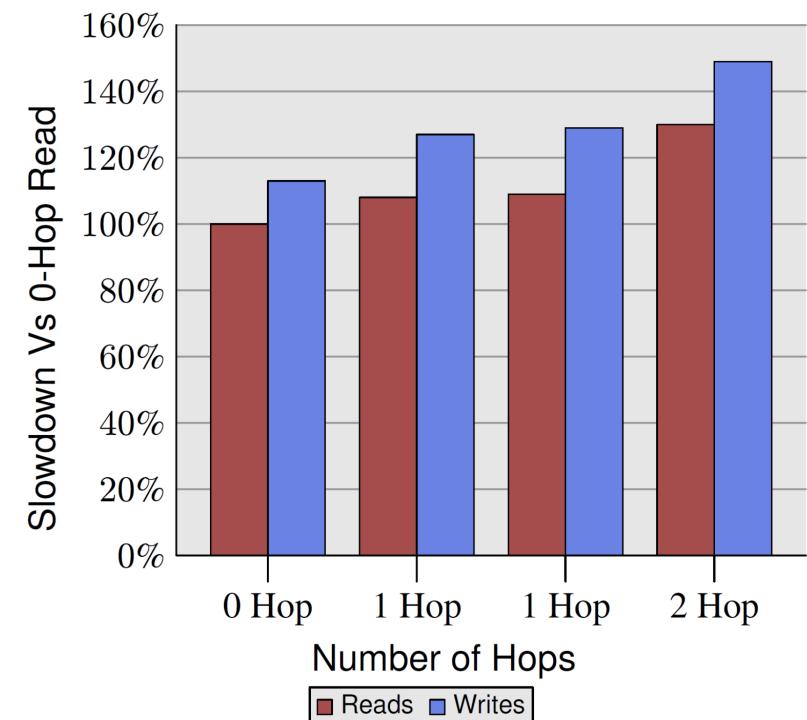
# Point-to-Point Technologies: a “local network” between nodes

- Each CPU can access data stored in a different NUMA node
  - Happens over a very fast interconnect, which is like a local network
    - AMD’s solution name: [Hyper Transport \(HT\)](#)
    - Intel’s solution name: [QuickPath Interconnect \(QPI\)](#) (more recently: Ultra Path Interconnect ([UPI](#)))
  - The fast interconnect (local network): shown in orange in picture
  - Possible topologies associated with this “local network”:
    - 2D-torus
    - 2D-lattice (only moves in x or y direction – like 2D grid)
    - 3D-Hypercube (3D version of lattice)
    - Etc.



# NUMA Performance Hit, order of magnitude

- What sort of slowdown should one expect if accessing memory outside a NUMA node?
- Approximate values:
  - 20% slowdown for reads
  - 30% slowdown for writes
- Values above depend on the number of hops required to reach memory bank
  - Closer NUMA nodes lead to lower values



# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 23

03/23/2020

# The quote the day

**“The best way to predict the future is to create it.”**

-- Peter Drucker, Austrian-born American management consultant. Recipient of the Presidential Medal of Freedom [1909, Vienna – 2005, California]

# Things to do

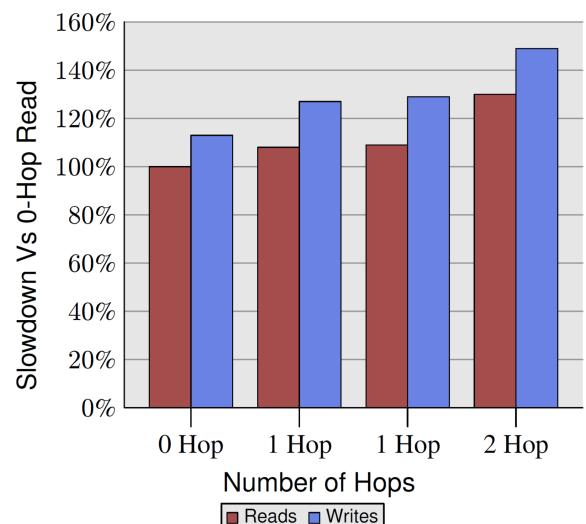
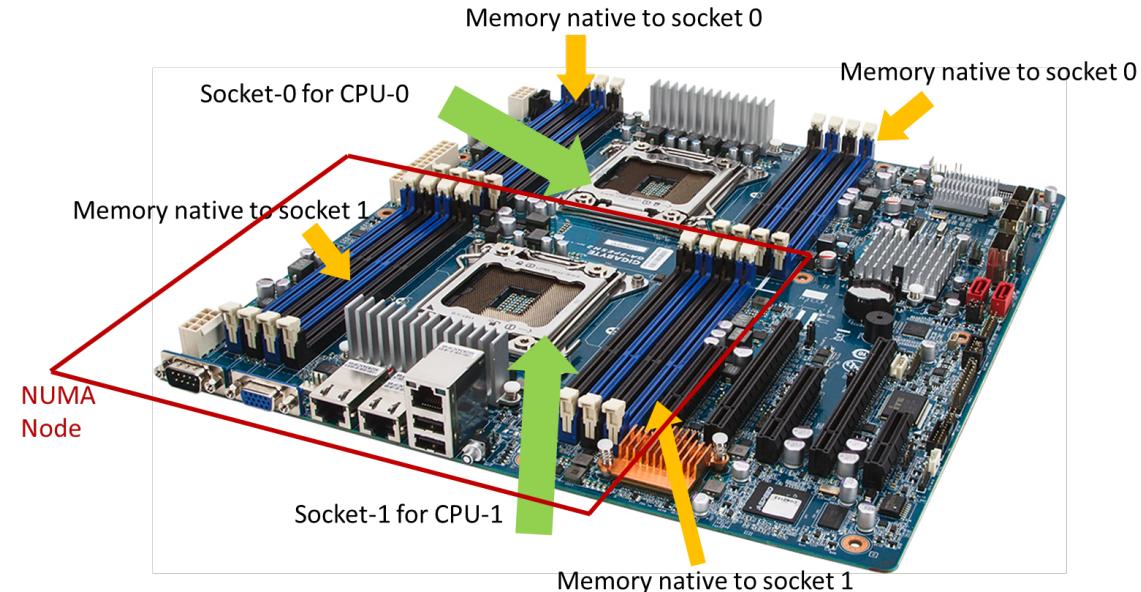
- Dan: start the Canvas recording
- Everybody else:
  - Please mute your microphone
  - Please ask questions (limit: two per person per lecture; after that, pose on Piazza)
  - When you ask questions, first unmute your mic

# joke of the day – worked on it for 40 mins or so

- ME759 wasn't good enough for prime-time television so as of today they downgraded me to radio only. How about my good looks? Don't they count for anything anymore???

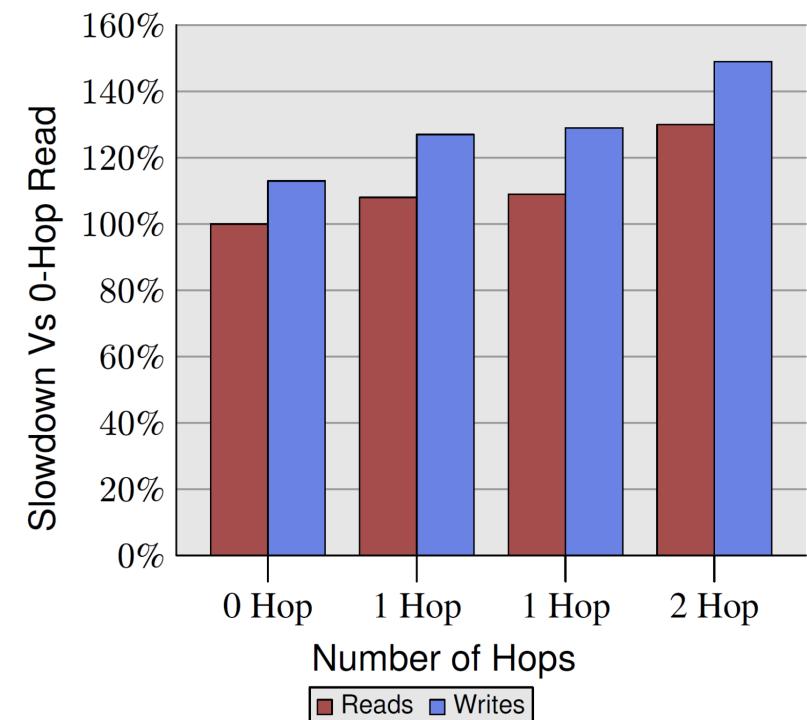
# Before we get going...

- Last time:
  - NUMA in the context of OpenMP
- Today:
  - NUMA aspects, wrap up
  - Cache issues
  - Optimization aspects (?)
- Other tidbits:
  - Lectures held in Canvas Blackboard Ultra
    - Recording of the lecture available in Canvas
  - ME759 Exam: April 15, at 7:15 PM, in Canvas
    - Review: Tu, April 14, at 7:00 PM, in Canvas
  - Office hours moved to Canvas (same time/date)
  - Assignment due on Th, 9 PM
  - Final Project Proposal due on Friday, 9 PM



# NUMA Performance Hit, order of magnitude

- What sort of slowdown should one expect if accessing memory outside a NUMA node?
- Approximate values:
  - 20% slowdown for reads
  - 30% slowdown for writes
- Values above depend on the number of hops required to reach memory bank
  - Closer NUMA nodes lead to lower values



# NUMA aspects: How the OS comes into play

- NUMA aspects where Operating System (OS) comes into play

- 1) When a thread `mallocs` memory, how should this memory be allocated?

- Examples:
  - All memory allocated in one NUMA node
  - Memory split in chunks, each NUMA node being associated with one such chunk (“striping the memory”)
- See next slide for implications

- 2) The process of having a thread assigned to execute on a certain core

- Example: A thread w/ thread id 3 used to execute on Core #X, but then it runs on Core #Y, on a different NUMA node
- Related concept: [affinity](#) – how you can prod the runtime/OS to assign a thread to a certain core

# 1) The `malloc` issue, and the “first touch” story

- Memory allocated with `malloc` not actually set aside when `malloc` gets hit
  - That only happens when data is written to it
- **x** first touch is only by Master thread
  - Memory might be allocated in its NUMA node
- **a** touched by all threads at the same time
  - Reflect on how chunks of the array are going to be spread all over the memory, very likely in different banks
- Reflect on differences between Case 1 & 2

```
int main() {
    const int N = 4000000;
    double *x = (double*)malloc(N * sizeof(double));
    double *a = (double*)malloc(N * sizeof(double));

    for (i = 0; i < N; i++)
        x[i] = simpleFunc(i);

    omp_set_num_threads(64);
#pragma omp parallel for schedule(static,5)
    for (i = 0; i < N; i++)
        a[i] = someFunc(x[i]);

    return 0;
}
```

Case 1

```
int main() {
    const int N = 4000000;
    double *x = (double*)malloc(N * sizeof(double));
    double *a = (double*)malloc(N * sizeof(double));

    for (i = 0; i < N; i++)
        x[i] = simpleFunc(i);

    omp_set_num_threads(64);
#pragma omp parallel for schedule(dynamic,1)
    for (i = 0; i < N; i++)
        a[i] = someFunc(x[i]);

    return 0;
}
```

Case 2

## 2) Affinity: Preamble/Backdrop

- Imagine the following OpenMP scenario, when having 16 threads, dual socket, 8 cores/socket
  - The execution hits `omp parallel for`, with static schedule
    - Assume Master gets Core 0, Thread 1 gets Core 1, ..., Thread 15 gets Core 15
    - Note: Worker threads folded back at end of parallel for
  - Next, after worker threads folded back, Master thread goes through a sequential segment of the code
    - Meanwhile, Core 2 has data in its cache that just sits there
  - Next yet, the execution hits a second `omp parallel for`, which this time has a schedule “dynamic”
    - A thread will be associated with Core 2 and it gets assigned to work on data that is in Core 13’s NUMA node
      - Can be worse: thread on Core 2 touches data that Core 13 keeps touching too – caches are not happy either

# Affinity

- **Affinity:** how threads executing your code get assigned/pinned to CPU cores
  - These cores can be physical or virtual
- Why “virtual cores”?
  - Recall that one physical core shows up as two or sometimes four virtual cores
    - Example: Intel’s HTT. IBM has one physical core associated w/ 4 virtual cores
  - From back in the day: “virtual core” is what enables what we called TLP (thread-level parallelism)

# Affinity

- Aspects that you might be curious about and/or may want to control
  - Where is my master thread running ?
  - Can I control how my threads are assigned in the first place?
  - Can I pin my threads to prevent migration?

# Controlling affinity in OpenMP

- Two actors come into play:
  - `OMP_PROC_BIND` – allows you to dictate a distribution policy
  - `OMP_PLACES` – allows you to control locations
- NOTE: there are no OpenMP runtime functions for binding/placing.
  - The only way to do it at this point is through environment variables, as explained next

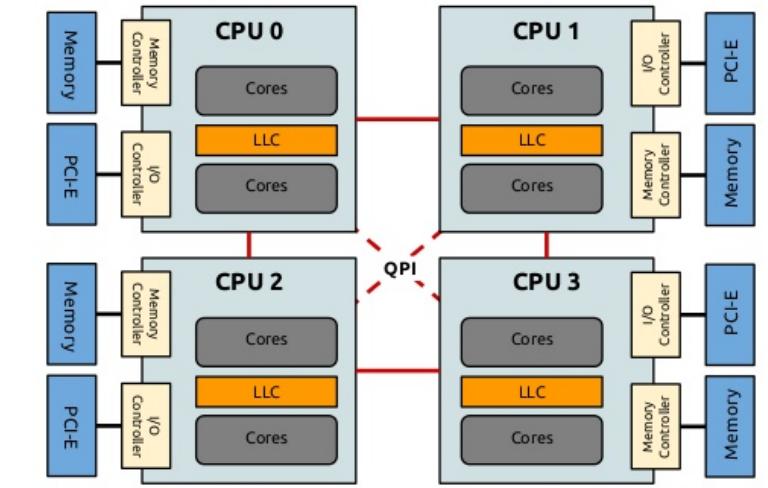
# The distribution policy as dictated by **OMP\_PROC\_BIND**

- `export OMP_PROC_BIND = master|close|spread|false|true`
  - **master**: collocate threads with the master thread
  - **close**: place threads close to the master in the places list
  - **spread** (default): spread out threads as much as possible
  - **false**: set no binding
  - **true**: lock threads to a core
- NOTE: above, we used an environment variable. Can accomplish via runtime function calls

# How things come into play

- **spread** (spreads out threads as much as possible)
  - This is useful if your code is memory bound
    - Likely to improve aggregate system memory bandwidth
    - You see more cache (less cache contention)
  - Example: two sockets, 8 cores per socket, you ask for 8 OpenMP threads
    - Have first four OpenMP threads on one socket and use its bandwidth for talking w/ main memory
    - Have last four OpenMP threads on the other socket and use its bandwidth for talking w/ main memory

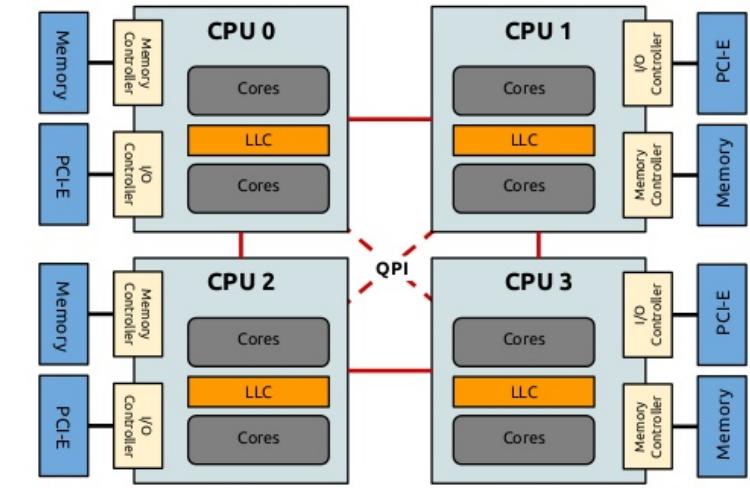
CPU architecture (Intel Sandy Bridge)



# How things come into play

- **close** (places the threads close to the master)
  - This is useful if you don't do many trips to main memory since you keep hitting the cache
    - Typically useful if you are compute bound
    - Likely to reduce synchronization costs (single, barrier, etc.)

CPU architecture (Intel Sandy Bridge)



# Control locations through OMP\_PLACES

- Nomenclature is pretty darn confusing; using Linux parlance below:
  - Place (the confusing part): smallest entity able to run an OpenMP thread (“the CPU”, in Linux parlance)
  - Place list: one or more CPUs → thread pinning is done place by place
- The take home:
  - Migration (if any) of the threads allowed to occur within a place (between the CPUs of that place, that is)
- Against this backdrop, OMP\_PLACES can assume one of these values
  - **threads**: Hardware thread (hyper threading assumed “ON” – more later)
  - **cores**: core
  - **sockets**: node (socket)
  - A place list defined by user, explicitly referencing the underlying hardware of the machine

# Control locations through OMP\_PLACES

- **threads, cores, sockets**: are OpenMP abstractions, there is some liberty in how the runtime interprets them
  - Example: is hyper-threading on or off?  
Because it leads to different outcomes...
- A **place list** is the real thing – you explicitly state the places
  - The numactl command shows how your hardware is organized

(Linux nomenclature, confusing...)

```
ME759node ~> numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 16 17 18 19 20 21 22 23
node 0 size: 64064 MB
node 0 free: 53060 MB
node 1 cpus: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
node 1 size: 64482 MB
node 1 free: 59450 MB
node distances:
node    0    1
 0: 10 21
 1: 21 10
```

These are half of the available places for this chip

# Side trip: the lscpu command

[interesting “affinity” information highlighted in bold red]

```
ME759node ~> lscpu
```

```
Architecture:          x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:           Little Endian
CPU(s):              32
On-line CPU(s) list: 0-31
Thread(s) per core:  2
Core(s) per socket: 8
Socket(s):          2
NUMA node(s):        2
Vendor ID:            GenuineIntel
CPU family:           6
Model:                63
Model name:           Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
Stepping:              2
```

(output of lscpu continued  
in the block to the right) →

```
CPU MHz:                  3116.890
CPU max MHz:               3400.0000
CPU min MHz:               1200.0000
BogoMIPS:                  5187.98
Virtualization:            VT-x
L1d cache:                 32K
L1i cache:                 32K
L2 cache:                  256K
L3 cache:                  20480K
NUMA node0 CPU(s):        0-7,16-23
NUMA node1 CPU(s):        8-15,24-31
```

# OMP\_PLACES: stating the places explicitly

- Via environment variable: export `OMP_PLACES=<place_list>`
- Here's what `place_list` above can be:
  - A place: `{0}`
  - Enumeration: `{0},{1},{2},{3}`
  - Interval notation: `<place>:<len>:<stride>` → e.g. `{0}:4:1`
  - NOTE: `{0},{1},{2},{3}` is the same as `{0}:4:1`
- What these numbers in red above mean:
  - with HyperThreading: list of ids of the HW-threads (virtual cores)
  - without HyperThreading: list of cores

List of [Linux] CPU ids

# Example [Assume 2 sockets & 4 cores per socket & no hyper-threading enabled]



```
export OMP_NUM_THREADS = 4  
export OMP_PROC_BIND = close
```

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```

```
export OMP_NUM_THREADS = 4  
export OMP_PROC_BIND = spread
```

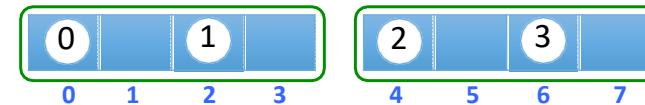
←bash shell

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```

←C code



COMPACT PACKING



SCATTER PACKING

←thread id (tid)  
←cpu id

# Example [Assume 2 sockets & 4 cores per socket & no hyper-threading enabled]



```
export OMP_NUM_THREADS = 4  
export OMP_PLACES = '{0},{1},{2},{3}'
```

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```



COMPACT PACKING

```
export OMP_NUM_THREADS = 4  
export OMP_PLACES = '{0},{2},{4},{6}'
```

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```



SCATTER PACKING

←thread id (tid)  
←cpu id

# Example [Assume 2 sockets & 4 cores per socket & hyper-threading not enabled]



```
export OMP_NUM_THREADS = 4  
export OMP_PLACES = '{0}:4'
```

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```



COMPACT PACKING

```
export OMP_NUM_THREADS = 4  
export OMP_PLACES = '{0}:4:2'
```

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```



SCATTER PACKING

# Example [Assume 2 sockets & 4 cores per socket & hyper-threading enabled]



```
export OMP_NUM_THREADS = 8  
export OMP_PLACES = threads
```

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```



Binding to Single HW-thread

```
export OMP_NUM_THREADS = 8  
export OMP_PLACES = cores
```

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```



Binding to core

←bash shell

←C code

←thread id (tid)

←cpu id



# Example [Assume 2 sockets & 4 cores per socket & hyper-threading enabled]



```
export OMP_NUM_THREADS = 8  
export OMP_PLACES = '{0}:8' (#1)  
export OMP_PLACES = '{8}:8' (#2)
```

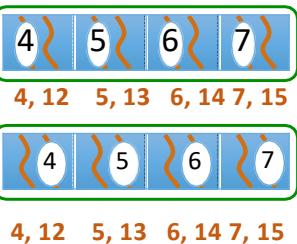
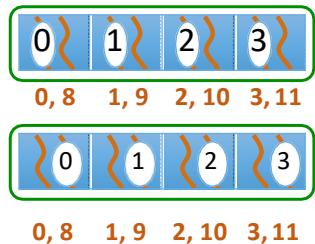
```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```

```
export OMP_NUM_THREADS = 8  
export OMP_PLACES = cores
```

←bash shell

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```

←C code



(#1)  
(#2)

Binding to Single HW-thread



←thread id (tid)  
←cpu id  
} HW-thread

Binding to core

# Example [Assume 2 sockets & 4 cores per socket & hyper-threading enabled]



```
export OMP_NUM_THREADS = 4  
export OMP_PLACES = threads  
export OMP_PROC_BIND = close
```

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```

```
export OMP_NUM_THREADS = 4  
export OMP_PLACES = cores  
export OMP_PROC_BIND = spread
```

←bash shell

```
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // code here...  
}
```

←C code



Binding to Single HW-thread



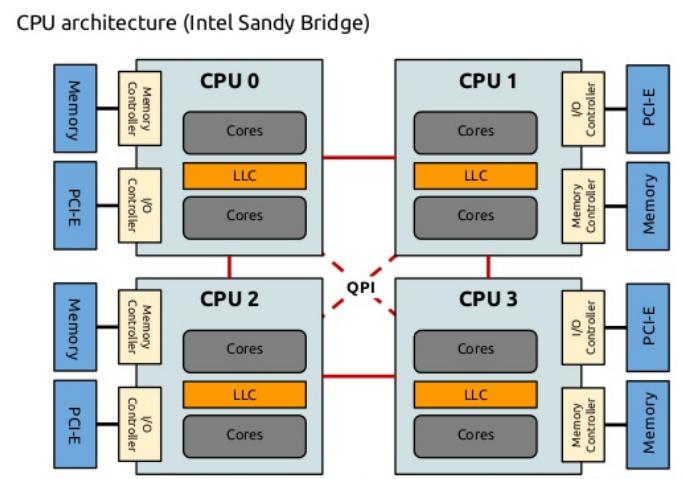
Binding to core

←thread id (tid)  
←cpu id



# Affinity & cc-NUMA

- There are two reasons why affinity is good
  - First: we avoid remote data NUMA-node access; i.e., costly local network hops
  - Second: overhead of cache-coherence mechanism starts kicking in
    - Lack of affinity in multi-threaded execution (OpenMP):
      - A small amount of data out of a big chunk of data that is physically next to CPU0 can get stored in cache of CPU3 as well
      - Changes made by CPU3 are going to invalidate cache of CPU0; not good
- cc-NUMA: **cache-coherent** NUMA
  - The run time must maintain coherence between caches in different NUMA nodes
  - Topic of next section

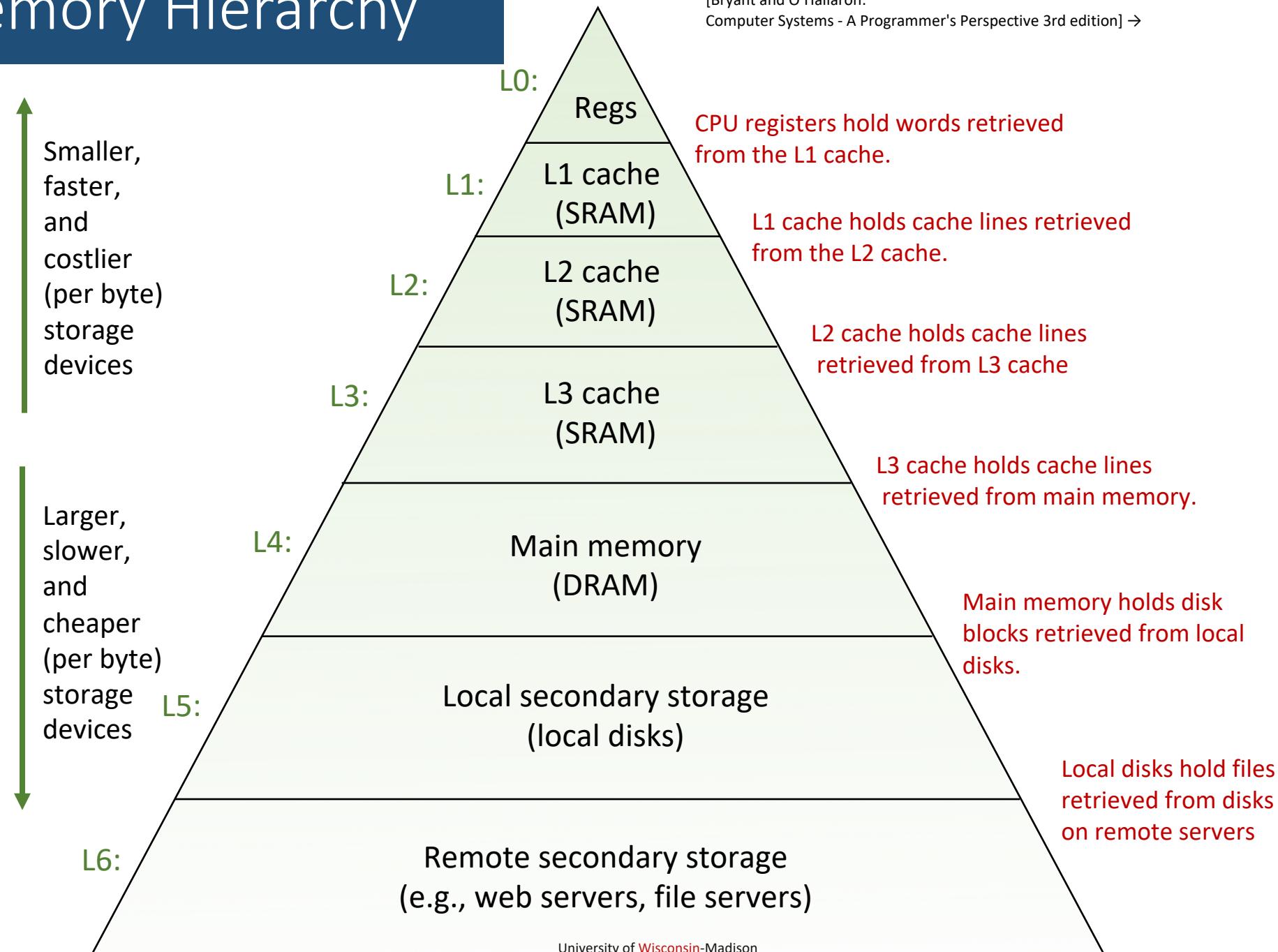


# Loose ends: topics discussed

- Non-uniform memory access (NUMA)
- Cache issues

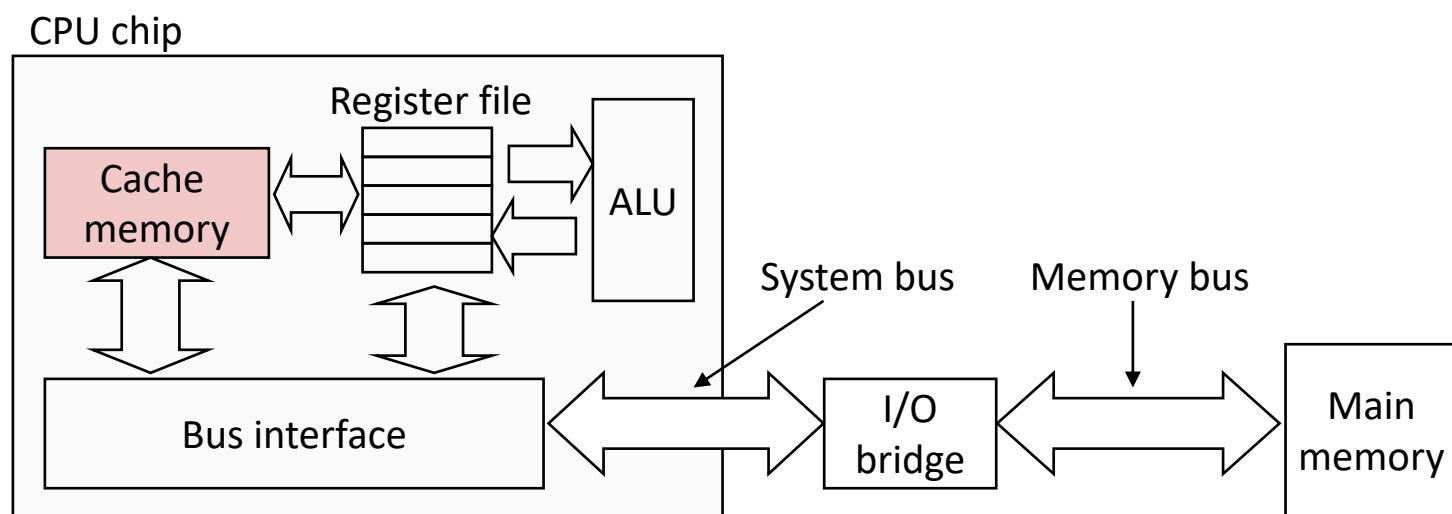
# The Memory Hierarchy

[Bryant and O'Hallaron:  
Computer Systems - A Programmer's Perspective 3rd edition] →



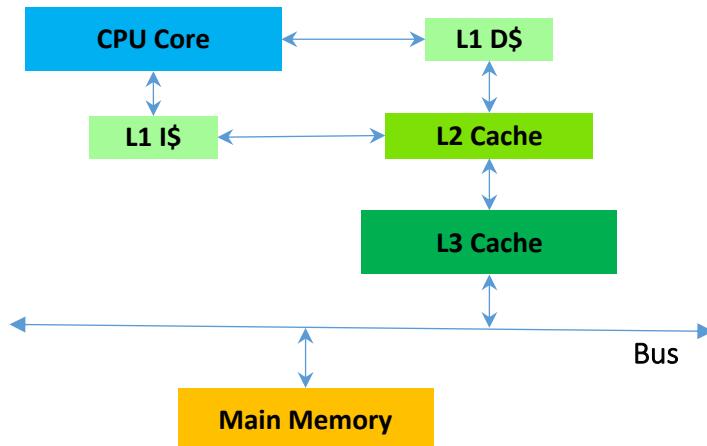
# Cache Memories [old slide]

- Cache memories: small, fast SRAM-based memories managed transparently
  - Purpose: Hold frequently accessed blocks of main memory
  - When very useful: when execution displays **locality** (spatial/temporal)
- Typical system structure:



# How Cache Comes into Play [old slide, with a facelift]

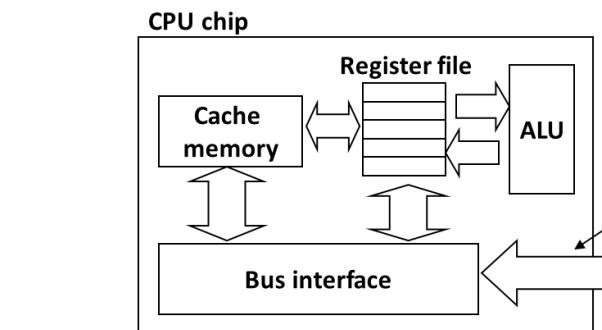
- Typically\*, the read/write operations go through cache
- Caches used to hold both data and instructions
  - L1: tens of KB (per core)
  - L2: hundreds of KB (per core)
  - L3: tens of MB (per CPU)



(Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz - from ten slides ago or so...)

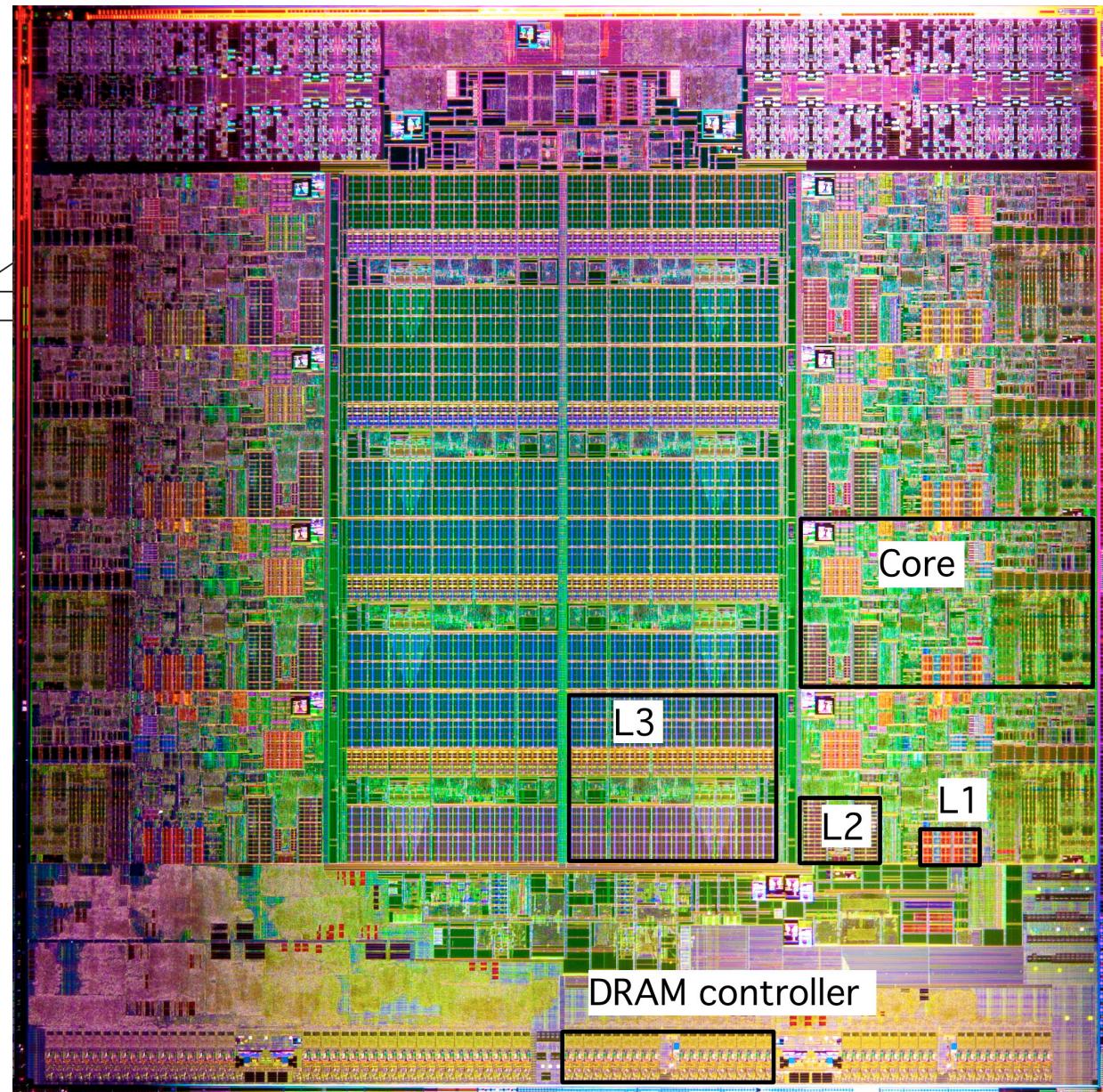
CPU MHz:	3116.890
CPU max MHz:	3400.0000
CPU min MHz:	1200.0000
BogoMIPS:	5187.98
Virtualization:	VT-x
<b>L1d cache:</b>	<b>32K</b>
<b>L1i cache:</b>	<b>32K</b>
<b>L2 cache:</b>	<b>256K</b>
<b>L3 cache:</b>	<b>20480K</b>
NUMA node0 CPU(s):	0-7,16-23
NUMA node1 CPU(s):	8-15,24-31

# What it Really Looks Like



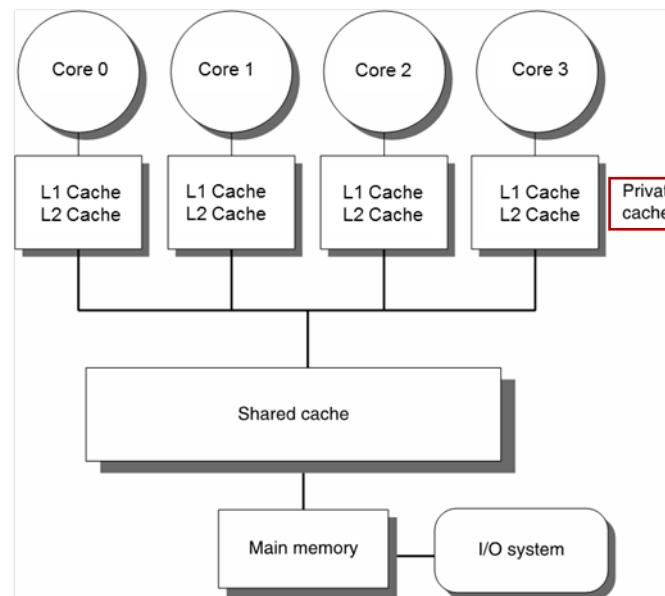
Intel Sandy Bridge  
Processor Die

L1: 32KB Instruction + 32KB Data  
L2: 256KB  
L3: 3–50MB



# What We Are Interested in: Caches in a Multi-Core Setup

- We almost exclusively discussed caches in conjunction with one core
- What if there is another core, and this other core changes the memory in a block that you just cached?



# Multi-Core Architecture: Two Memory Aspects Coming Into Play

- [Memory consistency model](#)
- [Memory coherence](#)
- Computer Science at UW-Madison: active in this area
  - Mark Hill, (Emeritus) David Wood, Karu Sankaralingam, Matt Sinclair
  - [Book](#) on this very topic
- PhDs granted on consistency models and/or coherence mechanisms
  - Topic is deep

# The concept of “memory consistency” [OLD SLIDE]

- Backdrop
  - Thread\_1 executes writeXY()
  - Thread\_2 executes readXY()
- Question: what values can A and B assume?
- For sequential consistency (“strongly-ordered mem. model”), the only possible scenarios are
  - A=2 and B=1
  - A=2 and B=10
  - A=20 and B=10
- For weak consistency (“weakly-ordered memory model”), it is possible to have this
  - A=20 and B=1

```
__device__ volatile int X = 1, Y = 2;  
  
__device__ void writeXY()  
{  
    X = 10;  
    Y = 20;  
}  
  
__device__ void readXY()  
{  
    int A = Y;  
    int B = X;  
}
```

# Consistency vs. Coherence

- Consistency establishes a set of rules that governs the collective actions of the threads relative to the **entire system memory**
- Coherence regards expected behavior that **\*one\* memory location** must display relative to transactions carried out by multiple threads running on multiple cores

# Consistency vs. Coherence

- Earning the attribute of “consistent memory” (for instance “sequential consistency”) imposes a set of **stronger/stricter requirements** that need to be met compared to what it takes to earn the “coherent memory” attribute
- Coherence: with some hand waving, it captures the expectation that if a thread running on core A changes a value in a memory location, other thread running on core B will read the updated value even though the read might occur **relatively soon** after the change of the value takes place per A’s action
- “relatively soon”: a bit of a mystery to me

# Consistency vs. Coherence

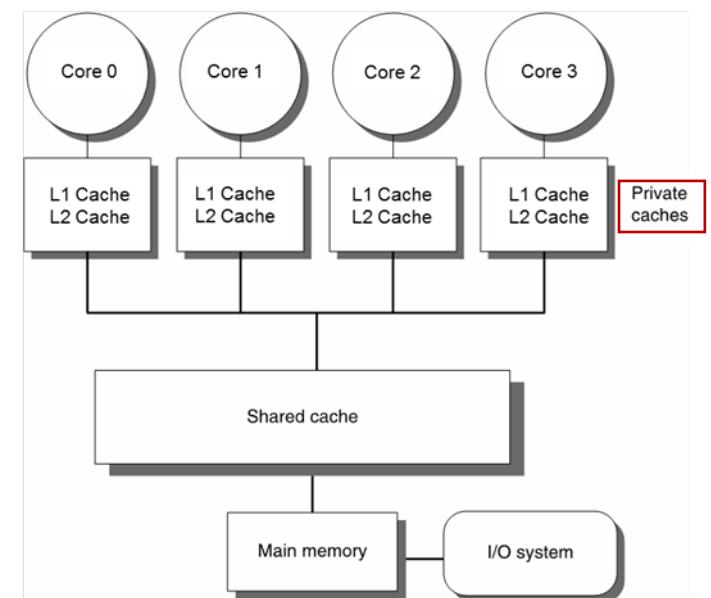
- Cache coherence is a mechanism, a hardware protocol to ensure that memory updates propagate to other cores. Cores will then be able to agree on the values of information stored in memory, as if there were no cache at all
- Cache consistency defines a programming model: when do memory writes become visible to other cores?
  - Defines the ordering of memory updates
  - A contract between the hardware and the programmer: if we follow the rules, the results of memory operations are guaranteed to be predictable

# Caches and Coherency, bottom line

- Speeding up **sequential** computing accomplished through use of large caches
  - Note that each OpenMP thread actually does engage in sequential computing
- Caching consequence: **multiple copies** of a physical memory location may exist at different hardware locations
- For program correctness, caches must be kept **coherent**

# Caches Coherence Mechanisms

- Significant **overhead** associated with implementing cache coherence
  - One of the main reasons why OpenMP doesn't scale
  - Another reason: memory speed is low, bandwidth saturates way before cores saturate
- Two established approaches for enforcing cache coherence
  - Directory-based
  - Snooping-based



# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 24

03/25/2020

# The quote the day

“I have never let my schooling interfere with my education.”

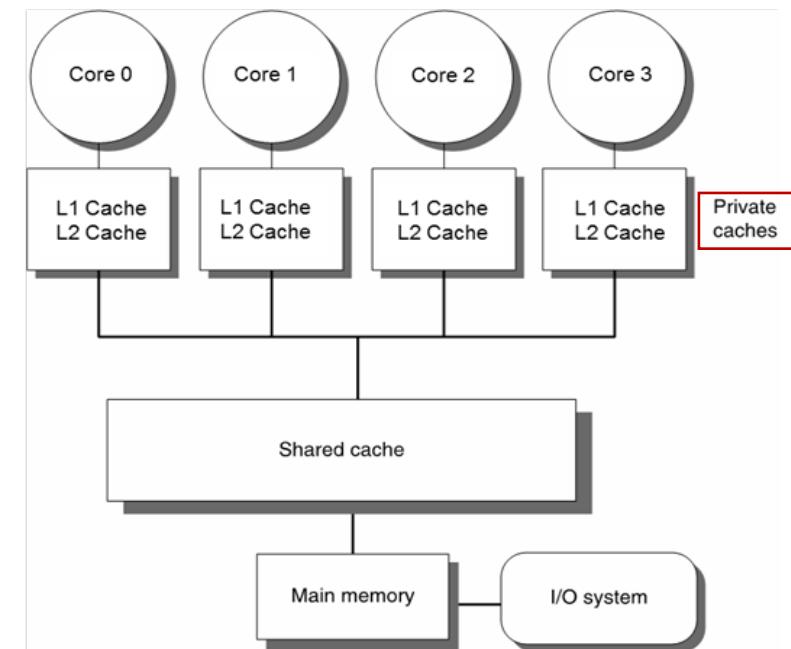
-- Mark Twain (more likely though, Grant Allen - Canadian science writer and novelist [1848 – 1899])

# Things to do, for Canvas live-session

- Dan: start the Canvas recording
- Everybody else:
  - Please mute your microphone
  - Please ask questions (limit: two per person per lecture; after that, post on Piazza)
  - When you ask questions, first unmute your mic

# Before we get going...

- Last time:
  - NUMA aspects, wrap up
  - Cache issues
- Today:
  - Cache issues
  - Optimization aspects
- Other tidbits:
  - Lectures held in Canvas Blackboard Ultra
    - Previous lecture [recording](#)
  - ME759 Exam: April 15, at 7:15 PM, in Canvas
    - Review: Tu, April 14, at 7:00 PM, in Canvas
  - Office hours moved to Canvas (same time/date)
  - Assignment due on Th, 9 PM
  - Final Project Proposal due on Friday, 9 PM



# Cache Coherence, in ME759 – disclaimer

- We discuss cache coherence just to better understand how OpenMP gets impacted
  - No deep dive, by any stretch of imagination
- Cache Coherence: ongoing research topic, particularly for large core counts
- Courses both in Electrical & Computer Engineering and in Computer Sciences

# Caches and Coherency, bottom line

- Speeding up **sequential** computing accomplished through use of large caches
  - Note that each OpenMP thread actually does engage in sequential computing
- Caching consequence: **multiple copies** of a physical memory location may exist at different hardware locations
- For program correctness, caches must be kept **coherent**

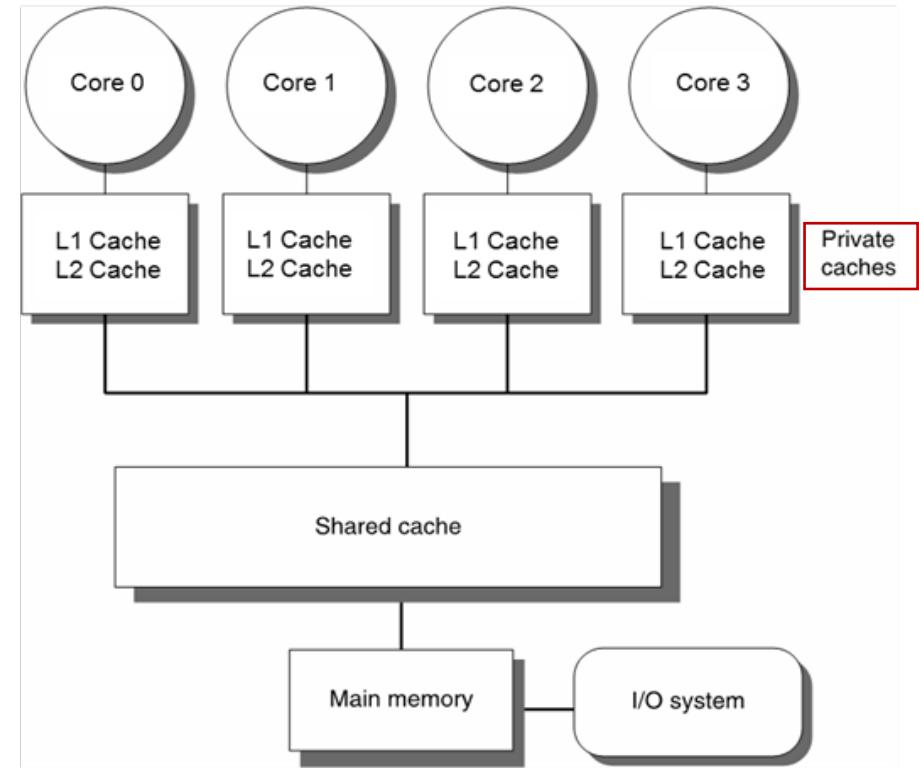
# Cache Coherence: Why is this relevant for OpenMP?

- When it comes to the code below, OpenMP is butchering the cache
  - Below, assume N=1,000,000 and 32 threads

```
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
#pragma omp atomic
    a++;
}
```

# Caches Coherence Mechanisms

- Cache coherence (CC) doesn't come for free
- Significant **overhead** associated with implementing CC
  - One of the main reasons why OpenMP doesn't scale
- Two established approaches for enforcing cache coherence
  - Directory-based
  - Snooping-based



# Cache Coherence Mechanisms: Directory-Based [1/2]

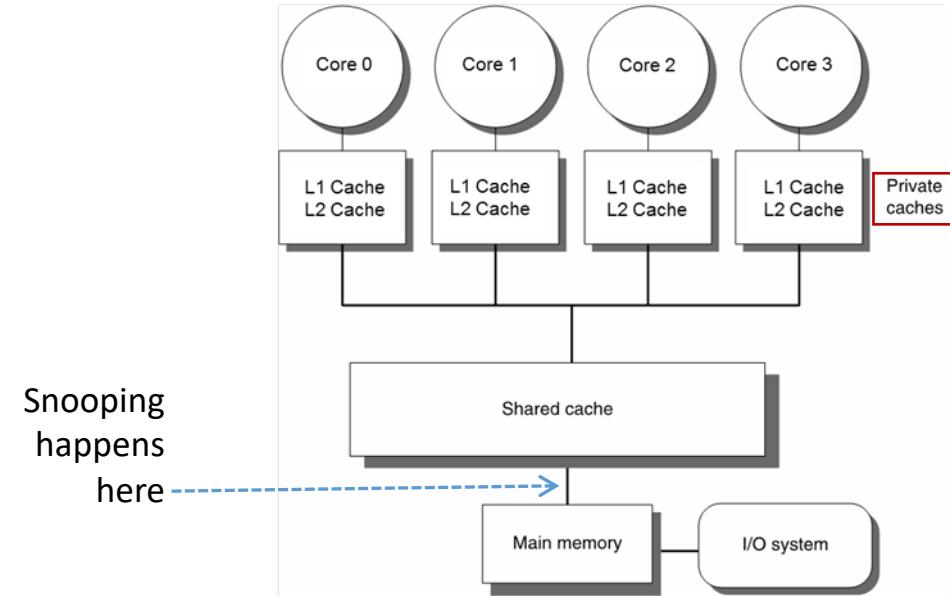
- Encountered in systems with large core counts
  - Example: Intel Xeon Phi 7290, had 72 core count (chip now discontinued)
- Builds on the assumption that there is a **fast point-to-point connection** between the cores
  - Employs a directory that indicates which data is stored by which core's cache
    - You can think of this directory as something similar to the Page Table that we introduced when discussing the concept of Virtual Memory
    - Keep in mind that this directory is not overwhelmingly large (caches are not that large)
- Pros: scales better than the snooping-based alternative
- Cons: complex, relatively low time overhead (“relatively”: compared to the alternatives; still very costly)

# Cache Coherence Mechanisms: Directory-Based [2/2]

- Directory typically split into pieces; each piece stored by one core
- Each mem. transaction needs to consider the possibility of invalidating a cache line in some other core's cache
- CC becomes particularly taxing if you have NUMA architectures
- NOTE: Ultimately, it's not OpenMP's fault
  - Taxing overhead for CC: interplay between the underlying hardware and the shared memory model embraced on the multi-core CPU

# Cache Coherence Mechanisms: Snooping/Sniffing-Based

- Encountered in systems with small core counts
- Builds on the assumption of a front-side bus; i.e., all memory transactions initiated by any core are processed through one connection/junction to the system memory
- Pros: simple
- Cons: doesn't scale

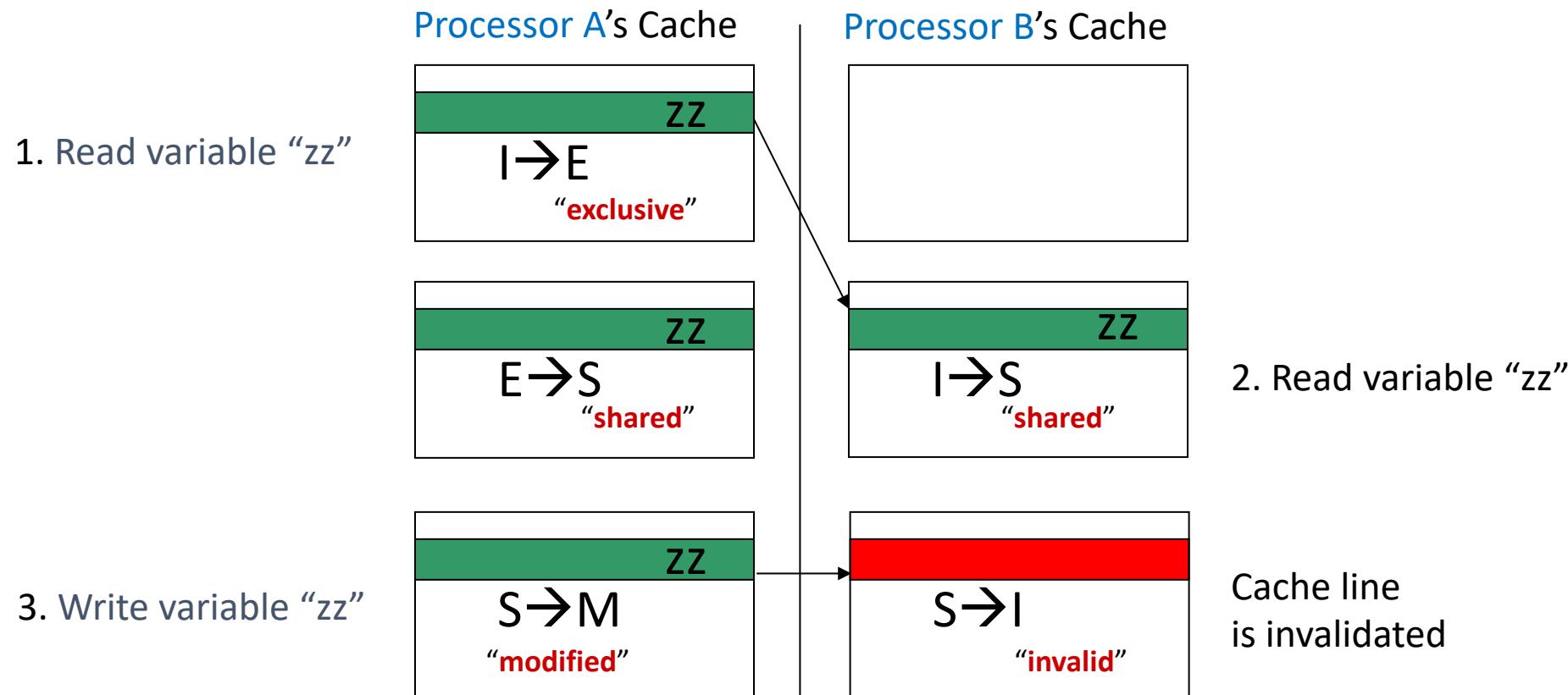


# MESI Protocol: Example of Snooping-Based Cache Coherence Protocol



# MESI: Invalidation-Based Coherence Protocol

- Cache lines have **state bits** used to characterize state of one cache line
- MESI** Protocol has four states: **M: Modified**, **E: Exclusive**, **S: Shared**, **I: Invalid**

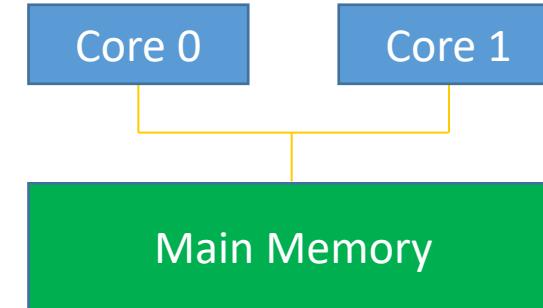


# The four states in which a cache line (CL) can find itself

- Modified (M)
  - CL present only in the current cache, and is dirty; i.e., there's discrepancy w.r.t. what's in main memory
  - Cache required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state
  - NOTE: The write-back changes the CL to (S)
- Exclusive (E)
  - CL present only in the cache and it's clean; i.e., it matches what's in main memory
  - NOTE: Might change to (S) in response to a read request. Might change to (M) when writing to it
- Shared (S)
  - CL may be stored in other caches of the machine and is clean; i.e., it matches what's in main memory
  - NOTE: The CL may be discarded; i.e., changed to (I), at any time.
- Invalid (I)
  - CL is invalid; i.e., it should not be used

# Example: Possible state combinations (✓), for a pair of caches

- System can't be M & M (that'd be too sweet)
- One cache can be I while the other one is M
- One cache can be S while the other is S
- System can't be E & E
- Etc.



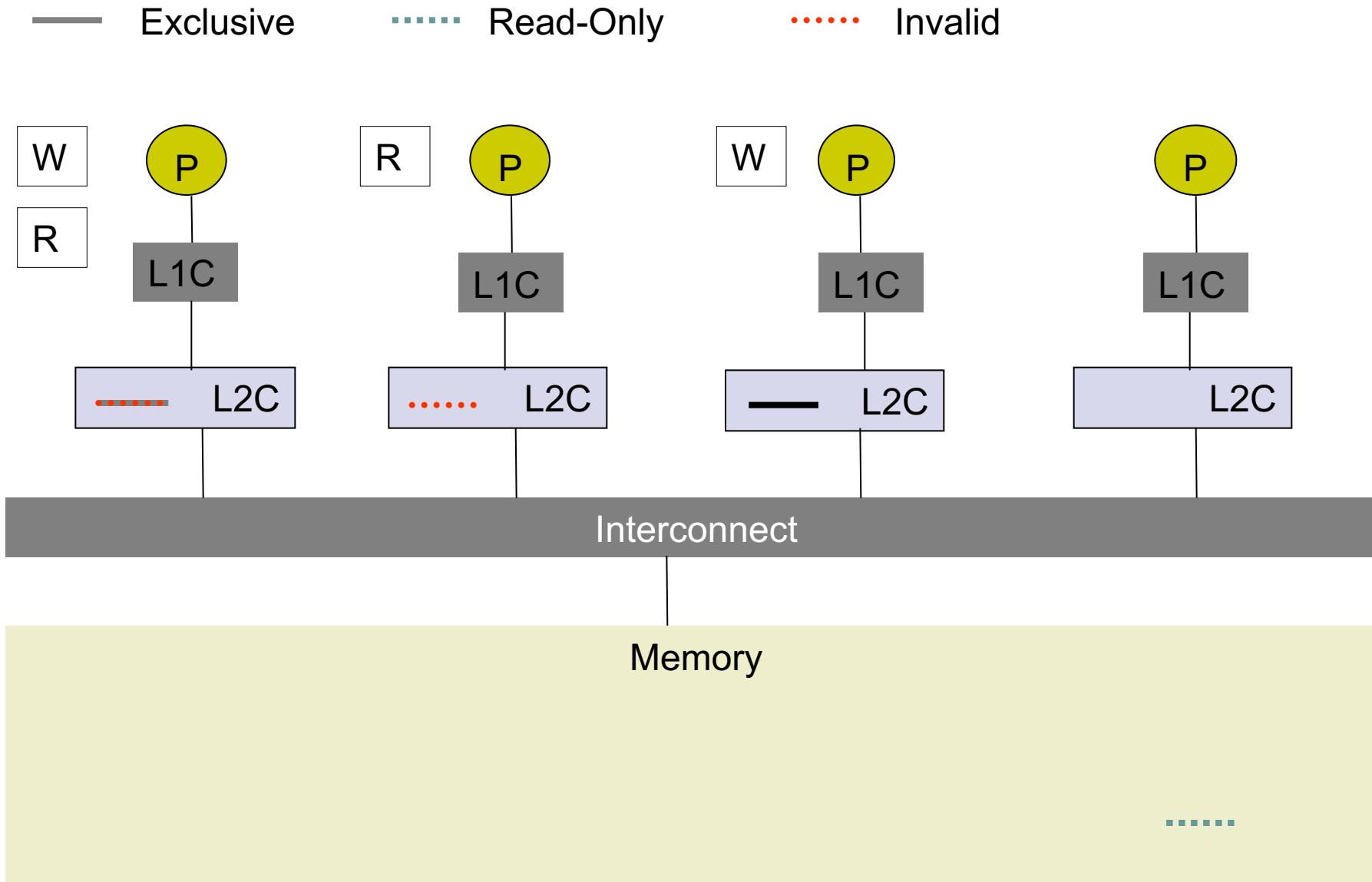
	M	E	S	I
M	x	x	x	✓
E	x	x	x	✓
S	x	x	✓	✓
I	✓	✓	✓	✓

# Coherency – Simplified Further

[cooked up example]

- Further simplify MESI for sake of simple discussion on next slide
  - Assume now that each cache line can exist in one of 3 states:
    - **Exclusive**: the only valid copy in any cache
    - **Read-only**: a valid copy but other caches may contain it
    - **Invalid**: out of date and cannot be used
- In this simplified coherency model
  - A READ on an *invalid* or *absent* cache line will be cached as *read-only* or *exclusive*
  - A WRITE on a line not in an *exclusive* state will cause all other copies to be marked *invalid* and the written line to be marked *exclusive*

# Cache coherency cartoon



[new sub-topic] Preamble: the “false sharing” issue in OpenMP

(below, assume a, and b, and c are arrays of float-type variables)

```
#pragma omp parallel for schedule(dynamic,2)
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

# The “False Sharing” Pitfall

- Recall cache lines typically are 64 byte long
  - Can store, for instance, 8 double precision values or 16 integers
- IMPORTANT NOTE: as soon as one entry in a cache line is changed, all the other values in cache line get dirty
- **False sharing:** happens when two threads are both writing into **different** locations within the **same** cache line
  - One write by Processor A will invalidate the cache line on Processor B
  - Memory transactions galore, tied to the cache coherence requirement

# False Sharing: symptoms

- Poor performance
- High, non-deterministic numbers of cache misses (particularly if NUMA comes into play)
- Mild, non-deterministic, unexpected load imbalance

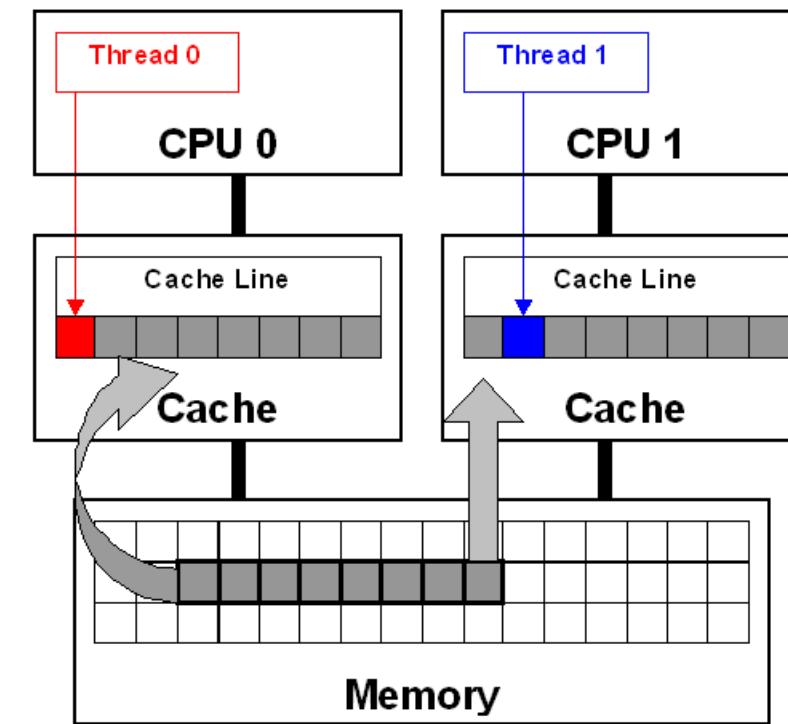
# False Sharing Example [1/2]

```
double sum_local[8];
double sum = 0.0;

#pragma omp parallel num_threads(8)
{
    int me = omp_get_thread_num();
    sum_local[me] = 0.0;

#pragma omp for
    for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];

#pragma omp atomic
    sum += sum_local[me];
}
```



How can this instance of false sharing be fixed?

# False Sharing Example [2/2]

- Easy fix: use `reduction(+,sum)`
- Look into other alternatives
  - Basic idea, use private/local variables
  - Fall back on “false sharing” only once, at the end

# Sometimes This Fixes It

- Reduce the frequency of false sharing by using thread-local copies of data
  - The thread-local copy will be read and modified frequently
  - When complete, copy the result back to the data structure

```
struct ThreadParams {  
    // For the following 4 variables: 4*4 = 16 bytes -> fits in one cache line...  
    unsigned long thread_id;  
    unsigned long v; // Frequently accessed variable (for reads/writes)  
    unsigned long start, end;  
};  
  
void threadFunc(void *parameter) {  
    ThreadParams *p = (ThreadParams *)parameter;  
    // local copy for read/write access variable  
    unsigned long local_v = p->v;  
  
    for (unsigned long local_dummy = p->start; local_dummy < p->end; local_dummy++) {  
        // Functional computation, would read/write the "v" member.  
        // Keep reading/writing local_v instead  
    }  
  
    p->v = local_v; // Update shared data structure only once  
}
```

# Another Way to Fix This [Ugly + Architecture Dependent]

- When using an array of data structures, pad the structure to the end of a cache line to ensure that the array elements begin on a cache-line boundary
  - If you cannot ensure that the array is aligned on a cache line boundary, pad the data structure to twice the size of a cache line

```
struct ThreadParams {  
    // For the following 4 variables: 4*4 = 16 bytes  
    unsigned long thread_id;  
    unsigned long v; // Frequent read/write access variable  
    unsigned long start, end;  
  
    // expand to 64 bytes to avoid false-sharing  
    // (4 unsigned long variables + 12 padding)*4 = 64  
    int padding[12];  
};  
  
__declspec(align(64)) struct ThreadParams Array[10];
```

# OpenMP, departing thoughts

- The cornerstone of OpenMP was the concept of parallel region: `omp parallel`
- However, rarely does a parallel region witness all threads executing the same code
- Why?
  - The parallel region construct was qualified via other OpenMP directive that changed the interplay between the code and the threads:
    - `for`
    - `sections`
    - `single`
    - `task`
    - `critical`

# OpenMP: What Speedup Should You Expect?

- For N cores, to get a speed-up of a factor of N is rare
  - Problem should be embarrassingly parallel, and you should not max out mem bandwidth (you need high arithmetic intensity, that is)
- In general scaling doesn't materialize beyond a certain core count
  - CC overhead
  - Limited amount of main memory bandwidth
  - Synchronization demands on the part of the algorithm you implement
  - NUMA collateral damage
- Question: **Is fewer fat chips better than more slim chips?**
  - slim/fat: higher/lower core count per chip

# Parallel computing: GPU CUDA vs. OpenMP

- A CUDA kernel is like an OpenMP parallel region without embellishments such as `for`, `single`, `task`, etc.
  - All CUDA/OpenMP threads execute the code in the kernel/parallel region, respectively
  - Caveat, CUDA: parallel execution proceeds in lock-step fashion<sup>\*</sup> (for the threads in a warp, that is)
  - Reflect on the advantages and disadvantages of the memory ecosystem/memory operation when you compare GPU and OpenMP parallel computing

Critical Thinking in Code Design/Development,  
with an Eye Towards Improving Performance

# Back of the envelope, ME759 opportunities for efficiency gains

- Use of right compiler flags, ILP-friendly code, etc. [discussed in this segment]
  - Speed up factor: 2X
- Inspired use of cache [already discussed this]
  - Speed up factor: 2-3X
- Use of vectorization (AVX2/AVX512) – use mips in OpenMP parallel for [discussed before, touched upon in this segment]
  - Speed up factor: 1.2X
- Use of multiple cores [already discussed this, OpenMP]
  - Speed up factor: 5 – 6X
- If the planets align, ME759 knowledge should help you to gain a 30X speed up factor

# Motivation, ME759 segment on “critical thinking/code optimization”

- Understanding how things work under the hood can translate into improved performance  
(to that end, we'll hit on various ME759 aspects)

# Acknowledgement

- Some slides in this segment come from
  - Material from class 15-213 at CMU
  - Nathan Bell (while he was at NVIDIA)
  - Georg Hager, Jan Eitzinger, Gerhard Wellein – material presented at Supercomputing 2019
  - Matt Godbolt, maker of [Compiler Explorer](#)

# Mindsets, approaching software development

- Thinking first about **correctness**
- Then one might add **productivity & convenience** in software development
- Other factors that come into play
  - Programming for **speed** of execution
  - Accounting for **legacy/growth** (want the code to be around for 50+ years)
  - Want to sell to everybody → **Portability**
- There are trade-offs that come into play; tough to get all bases covered
  - I cannot be amazing in wrestling and weightlifting and track and field and ski jumping and sumo. And math.

# Angle of attack, this module

- Our end goal: write faster programs, to handle large datasets fast
- Focus of the discussion
  - Particularly relevant when your code runs on the CPU; thus, for OpenMP code
  - However, some lessons carry over to the GPU as well

# Angle of attack, this module

- Module brings together concepts we've covered thus far; i.e., knowledge about:
  - a) Compilers, and how they work [covered mostly in ME459, we'll further elaborate on this]
  - b) Memory aspects: pointers, hierarchy, latencies, bandwidths
  - c) Instruction level parallelism (pipelining, jump instructions, branch prediction, etc.)
- We'll use a couple of examples to show how a) through c) above come into play
  - The hope is that the points made in these example will help you down the road

# Assessing performance, common sense things to keep in mind

- Second quote of the day: “You can't manage what you can't measure.”

Peter Drucker -- Austrian-born American management consultant, educator

- Timing and profiling needed to measure the performance of your code
  - Issue 1: make sure you understand the level of resolution of your timers
    - Example: if your code runs in 1 millisecond, your timer's level of resolution should be at least 1 microsecond
  - Issue 2: steep learning curve to get to point where you get return on investment for profiling

# Assessing performance, common sense things to keep in mind

- Find out where the bottleneck of the execution is. Time/profile heavily that part
  - Almost always this is tied to some [loop](#); hopefully, you have control over what happens in that loop
- Try to understand how execution time is split between “user time” and “system time”
  - There is nothing that you can do about the speed of `printf`
  - There is nothing that you can do if the “user time” for your program is 10%

# Assessing performance, common sense things to keep in mind

- Measure repeatedly to get statistical data about your program's execution time
- Timers: [command-line timers](#) or [code-embedded timers](#)
- “command-line timers”: you measure the time it takes the OS to load your program from disk into memory, launch it, and then execute it
  - You care about the execution time only
  - Bottom line: when given a choice, prefer “code-embedded timers” to “command-line timers”
- **NOTE:** if you measure repeatedly, your code is not going to be brought over from disk
  - Thus, you're not going to have a “[cold start](#)” if you measure repeatedly

# Assessing performance, common sense things to keep in mind

- “cold start” also applies to loading data into memory
- Example:
  - Suppose your program uses 40 KB of data
    - This would fit in L1 cache, on a **quiet machine** with **a decent chip**
  - 40KB of data:
    - It can be buried on the disk since you haven’t used it since last year: cold start
    - It can be already in L1 since you run your code 1000 times to get statistical info about run times: hot start
- Thus, if you measure repeatedly, your data is not going to be brought over from disk
  - In other words, you’re not going to have a “cold start” if you measure repeatedly

# Assessing performance, common sense things to keep in mind

- “quiet machine”
  - Recall difference between “wall-clock time” and “execution time”
  - Good idea to time on a quiet machine
    - Alternative 1: You are one of 40 programs running, and you do so at some sad priority level
    - Alternative 2: Other than the kernel, you are perhaps the only show in town, on that machine

# Assessing performance, common sense things to keep in mind

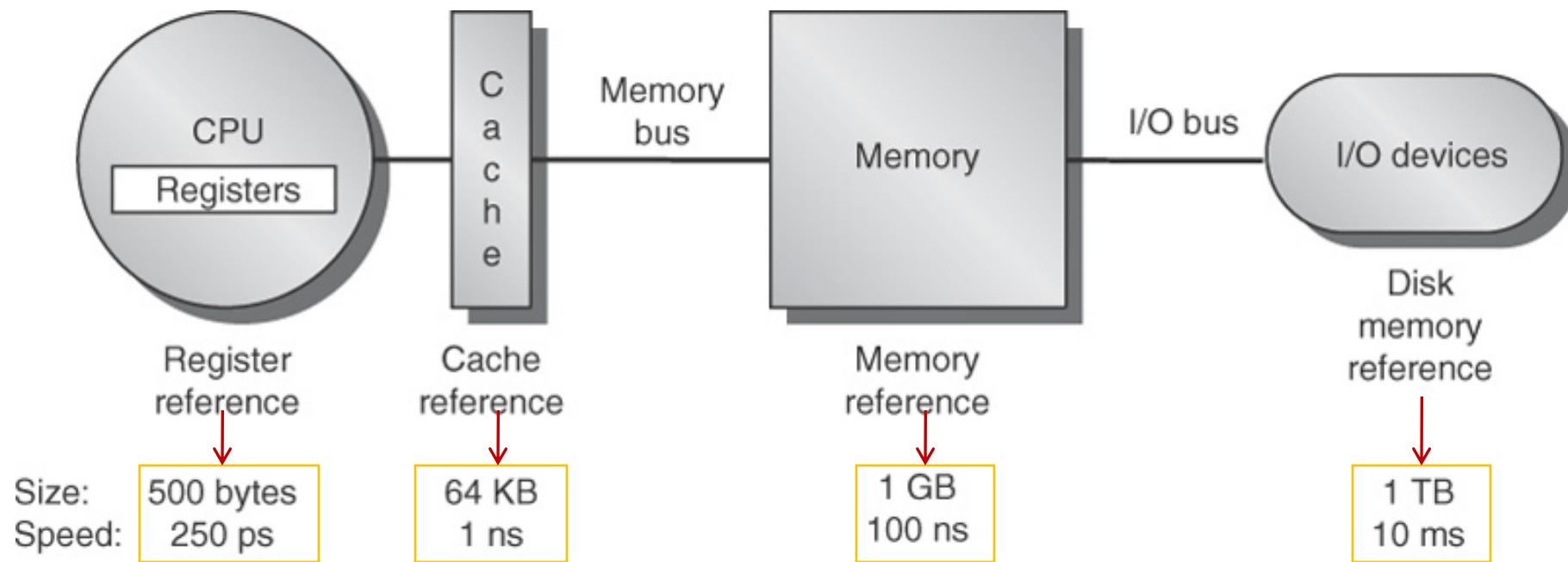
- In general, if you are assessing performance for improving speed, carry out timing/profiling exercise on the platform/chip on which you'll be running the code
  - Assessing & improving performance for on a Windows machine w/ Microsoft compiler and a fat Intel chip with large caches will produce different results than running gcc-compiled code on Raspberry Pi w/ Linux

# Rules of the thumb, squeezing performance out of solution

- Think critically about the code you're writing
  - Get to know your hardware: core count, how much memory, what latencies, bandwidth, etc.
  - Write compiler-friendly code [more to come, on this topic]
    - Watch out for optimization blockers; i.e., procedure calls & memory references
    - Look carefully at innermost loops (where most work is done)
  - Choose the right algorithm for your problem

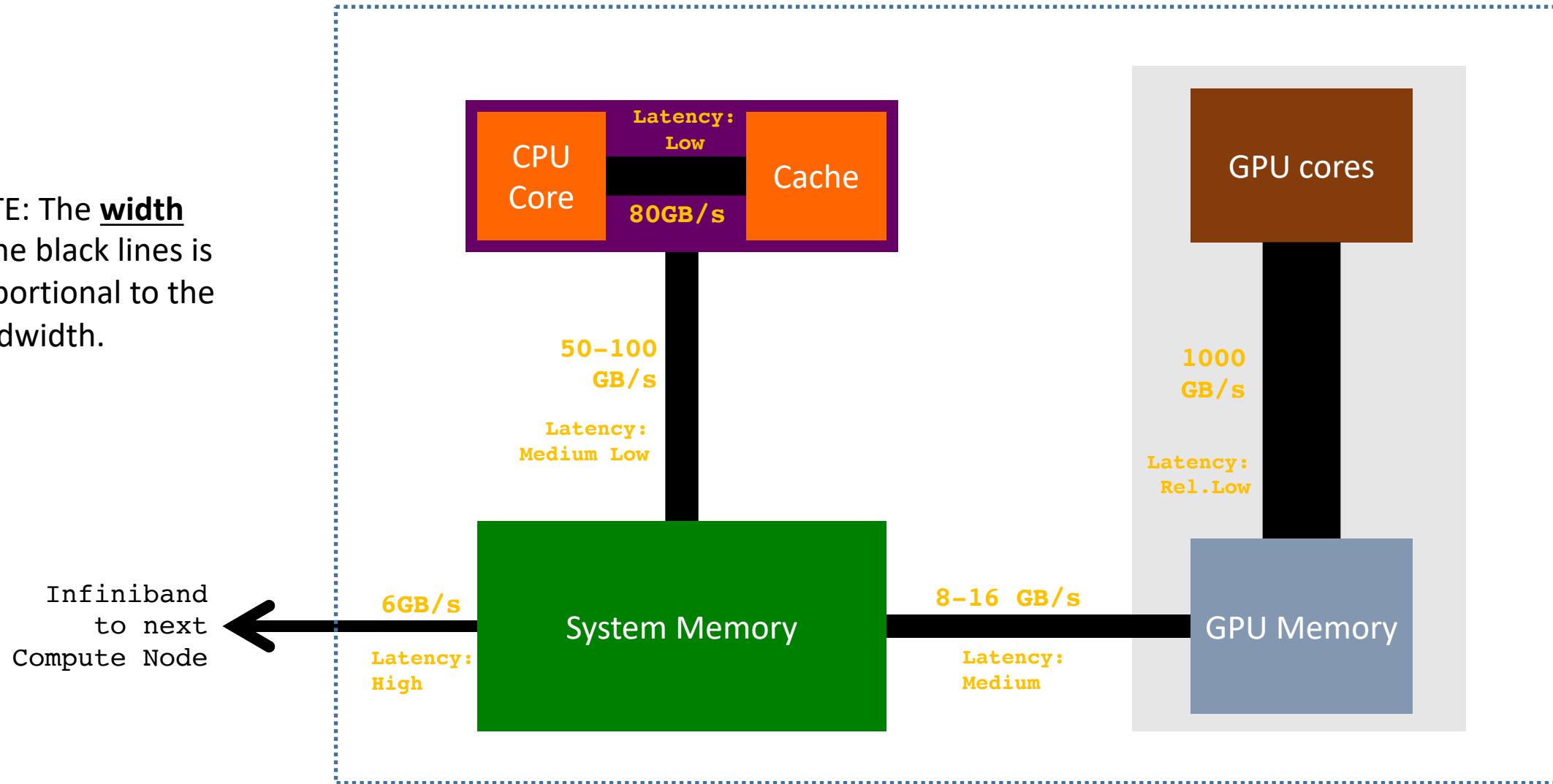
# Know your memory latencies

[typical embedded, desktop, and server computers]



# Know your Bandwidths in your system

NOTE: The width of the black lines is proportional to the bandwidth.



# Thinking critically about the expected speedup outcome

- Dealing w/ Big Data: You are trying to optimize your code for speed
- What makes for a good speed-up? What would make you happy?
  - 1.2X speedup?
  - 1.5X speedup?
  - 2X speedup?
  - 10X speedup?
- Answer depends on how polished the code is when you pick it up
- Rule of thumb in engineering: if your job finishes overnight, it's most often ok
- “Job”: might be running a long program once, or running a short one a million times
  - “Job”: something that when completed produces data that yields information that leads to knowledge that produces insights that assist your decision making

# Starting off on the right foot: choose the right algorithm

- Choose the algorithm that is a good fit for your architecture and type of data you're handling
  - A mediocre algorithm choice ices all your other efforts
  - Reflect on level of computation (pen-and-paper) needed to solve the problem (e.g., for matrix-vector multiplication probably  $O(N^2)$ )
  - Consider what data the algorithm needs, how much of it, where you read from/write to, how often. That is, data access/movement
  - Select data structures in a way that leads to advantageous memory accesses
- Concept good to know: “asymptotic complexity” of an algorithm
  - What you study in an “Algorithms” class, tells you about level of effort to solve a certain class of problems
    - Example:
      - Solution of a linear system - LU factorization:  $O(2n^3/3)$  vs Cholesky factorization:  $O(n^3/3)$
      - Prefix scan – Hillis & Steele  $O(n \log(n))$  vs Harris  $O(n)$
- Note: Algorithm selection – done before writing any line of code

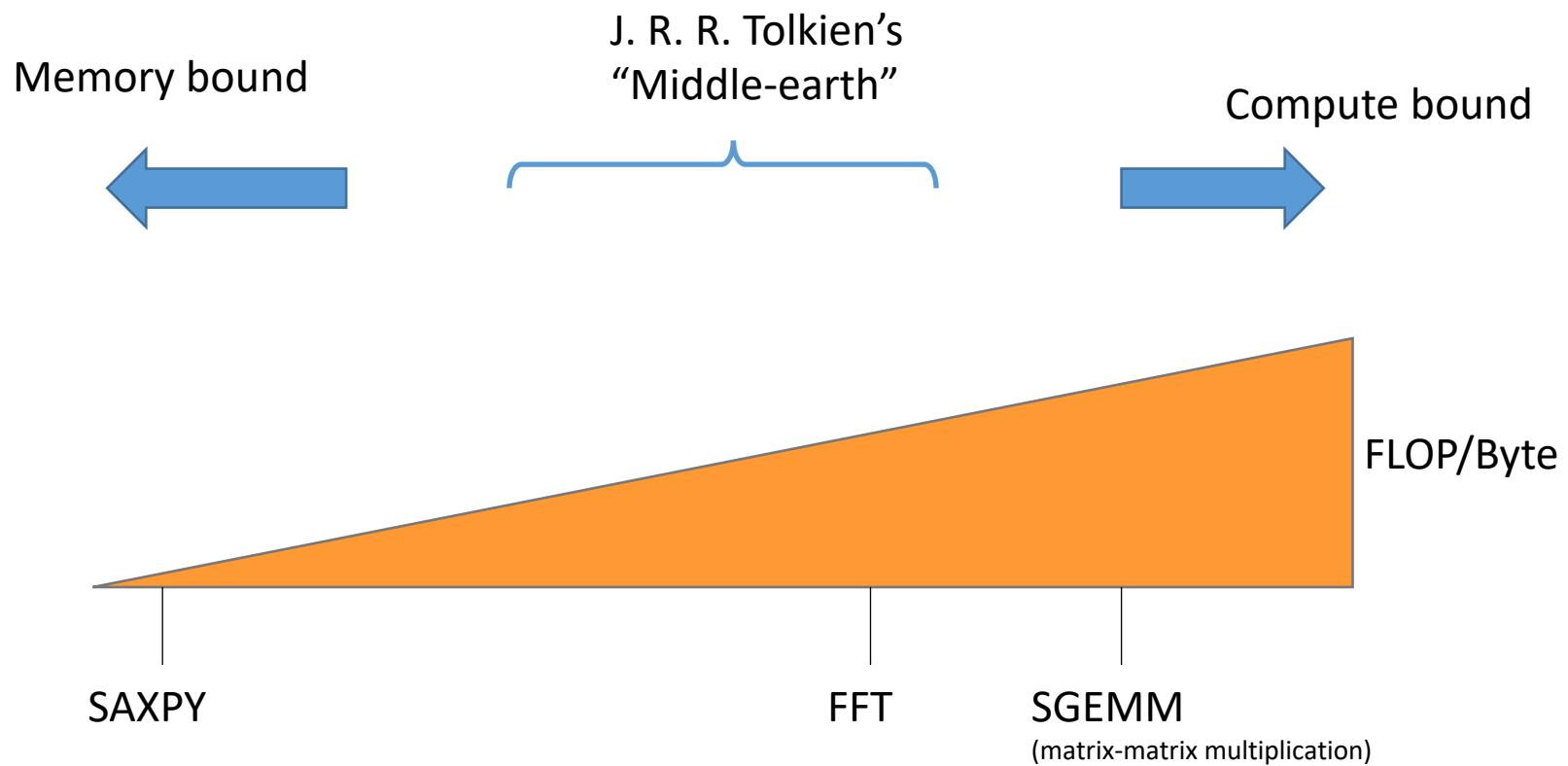
# Starting off on the right foot: choose the right algorithm

- When working on problem, there's more to performance than asymptotic complexity
  - Why?
    - Because asymptotic complexity is most often defined by number of operations
    - Memory transactions are rarely considered – they are specific to the hardware you'll eventually use to run
- If you understand how the computer works, it can guide you in designing a good/fast solution
  - Indeed, there is hope:
    - A problem can be solved by many algorithms (or approaches)
    - Also, one algorithm can have many implementations
- Spend time reflecting on your solution before writing any line of code!

# Asses the “Arithmetic Intensity” associated w/ your problem

- **Arithmetic intensity:** how much math you do per byte of data you bring from memory
- You can be in one of three cases:
  - Case 1: compute bound
    - That is, high arithmetic intensity – the ALU works a lot, not a whole amount of data movement to/from memory
  - Case 2: memory bound
    - You are moving data back and forth, likely only small parts of the data moved gets changed and/or used
    - Think cache misses
  - Case 3: “middle-earth” (J. R. R. Tolkien)
    - Somewhere in between compute-bound and memory-bound.
    - Your mission in this case: Try to move to Case 1 or Case 2 above

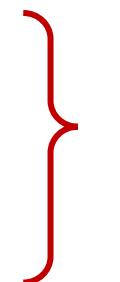
# Arithmetic Intensity: Putting things in perspective



# Simple Optimization Technique: Fusing Transformations

- One way to move the needle from “memory bound” towards “compute bound”
- Basic idea
  - Do not bring data into cache twice
  - If you use data X, Y, and Z – bring this data from memory once and then forget about it
- Why is it better?
  - You avoid memory traffic

```
for (int i = 0; i < N; i++)
    U[i] = F(X[i], Y[i], Z[i]);
for (int i = 0; i < N; i++)
    V[i] = G(X[i], Y[i], Z[i]);
```

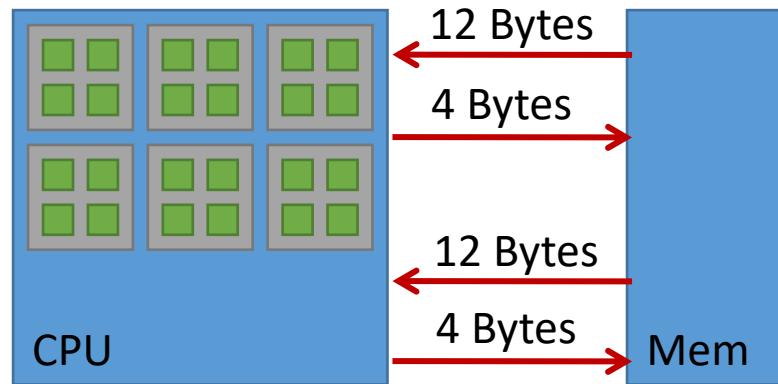


```
for (int i = 0; i < N; i++)
{
    U[i] = F(X[i], Y[i], Z[i]);
    V[i] = G(X[i], Y[i], Z[i]);
}
```

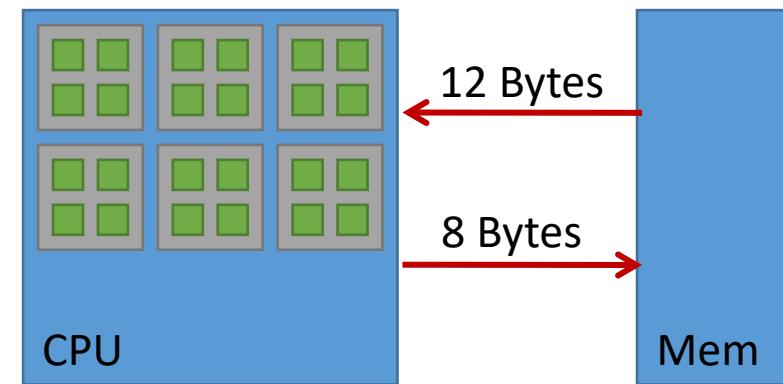
Loop Fusion

# Fusing transformations on previous slide

Original implementation



More thoughtful implementation



- Memory traffic cut by a factor of 1.6 (=32/20)

# Fusing Transformations

```
for (int i = 0; i < N; i++)
    Y[i] = F(X[i]);
```

```
for (int i = 0; i < N; i++)
    sum += Y[i];
```

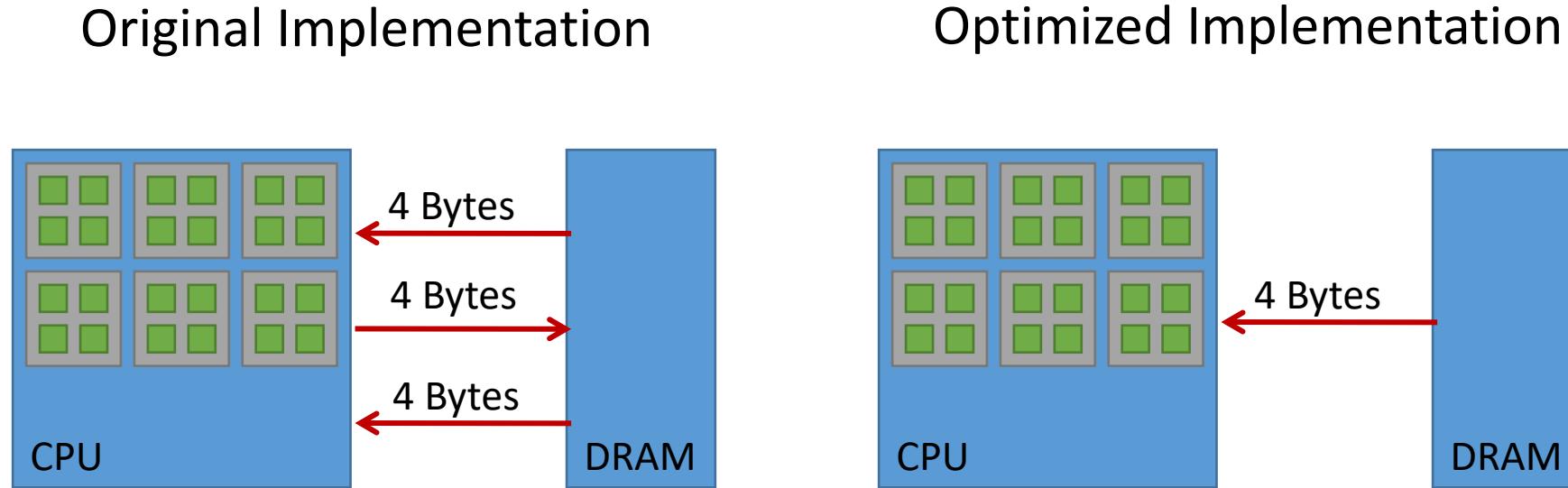


```
for (int i = 0; i < N; i++)
    sum += F(X[i]);
```

## Loop Fusion

- Mem traffic cut by a factor of 3
- Less memory used (no need to have Y)

# Fusing Transformations in Previous Example



- TAKE AWAY MESSAGE: Think if you can reduce in your code data movement since it is expensive:
  - Power costs
  - Time waiting for data to move around

# High Performance Computing: Questions to Ask

- Am I compute bound or memory bound?
- Do I understand the performance behavior of my code?
  - Does the performance match a model I have made? For instance, matrix-matrix multiplication should be  $O(N^3)$
- What is the optimal performance for my code on a given machine?
  - High Performance Computing == Computing at the bottleneck
- Can I change my code so that the “optimal performance” gets higher?
  - Circumventing/ameliorating the impact of the bottleneck

# Outside the scope of course: The roofline model

- I encourage you to read about the “roofline model” to assess code performance
  - “Are you compute bound?” ; “Am I memory bound?” ; “What chip/architecture might help me?”
- Access [here](#) “roofline model” material from Supercomputing 2019

# Let's talk about the compiler...

- When you time/profile your code, you'll have to play with the compile flags
  - Execution speed impacted heavily by compiling flags (3-4X speedup)
- NOTE: Aggressive optimization done by compiler might change behavior of your code
  - Might change results
  - Might uncover skeletons in the closet
- Side note: fair run-time comparisons is tricky
  - Different compilers have different flags
    - Tons of compilers out there: gcc, clang, g++, icl, cl, xl, etc.
  - Some compilers very adept at leveraging the underlying processor

# How compilers come into play

- A compiler has a tough job
  - Register allocation
  - Instruction selection and ordering
  - Dead code elimination
  - Fix minor inefficiencies
- The expectation is that the compiler will provide an efficient mapping of program to machine
  - Compilers may or may not support features that have been included in a language recently (C++11, C++14, C++17, C++20, C++23, etc.)
- Something to think about:
  - At least in theory, Intel compiler ought to be best positioned when it comes to compiling code for Intel chips
    - It ought to know all the dark corners of the microarchitecture
    - Well positioned to first implement any new/exotic ISA features

# How compilers come into play

- Compilers don't improve asymptotic efficiency of your algorithm
  - Up to programmer to select the right algorithm for the problem & the architecture
- Compilers have difficulty overcoming “optimization blockers”
  - Potential memory aliasing
  - “function call” side-effects

# The compiler has a tough job

- Often, compiler ends up doing analysis only at the function level
- Newer versions of gcc do inter-procedural analysis within **individual** files
- Whole-program analysis for some reason not done too often
  - More on this later

```
int functionFromOtherFile(double*);  
double someOtherRemoteFunc(double*);  
  
double myFunc() {  
    const size_t N = 1024;  
    double foo[N];  
    double accumulator = 0.;  
    for (int i = 0; i < functionFromOtherFile(foo); i++) {  
        accumulator += someOtherRemoteFunc(foo);  
    }  
    return accumulator;  
}
```



Compiler in a tough spot: function bodies are often hidden from call sites, compiler can only see their declaration

# The compiler has a tough job

- The compiler operates under tight constraints
  - It must obey all the rules in the C/C++ standard
    - Often prevents compiler from making optimizations that would only affect behavior under pathological conditions
  - Rule above possibly ignored though when your code is making use of nonstandard language features
    - This is what gets the compiler confused and it needs to play it super safe so that you are not mad at it
- Behavior that may be obvious to the programmer can be obfuscated by coding styles and poor understanding of the language
- Use of pointers is a big stumbling block

# The compiler has a tough job

- Most analysis based only on static information
  - Compiler can't anticipate run-time inputs (in general)
- One way in which we don't help compilers is by writing code that is ambiguous, open to interpretation
  - The compilers can't read our minds
- When in doubt, the compiler must be conservative

# Tricks of the trade, done by compiler

- Function inlining – this is a big one
  - Wherever you call a function, the compiler simply replaces the call to function foo with the entire body of function foo (the entire function body dropped in)
  - Not always possible, you must explicitly call for it (by function decoration and/or compile flags)
    - Nowadays you can ask the compiler to report back what functions it successfully inlined
- Consequences, if you can inline:
  - You have one less jump
  - More important: the compiler sees way more code and therefore engages in more fruitful optimizations
    - It's one thing to optimize when you have 5 lines of code, and a different one for 500 lines of code

# Other tricks of the trade, done by compiler

- Constant propagation `const double myApplePie = 3.1415;`
  - Compiler substitutes all references to `myApplePie` with the constant value
- Common subexpression elimination
- Dead code removal
  - There may be areas of the code that have no effect on the output, and these can be removed
    - This includes loads and stores whose values are unused, as well as entire functions and expressions
- Tail call removal
  - Set of recursive calls to same function get replaced by a loop

# How can you help the compiler?

## A) Allow it to see as much code as possible

- You set it free, so to speak
- Recall that typically compiler sees only one file at a time
  - Can't peek at some function you use that is defined in a different file

## B) Provide flags to convey to it information

- E.g.: telling compiler your target architecture can help a lot

# Compilers, and link time optimization (LTO)

- Compilers are becoming faster and smarter
- It's affordable and profitable to do program-level, not function level optimization
- LTO (also known as LTCG, for “link time code generation”) can be used to allow the compiler to see across translation unit boundaries
- Likely to provide good speed up for complex codes with many source files

# Compilers, and link time optimization (LTO)

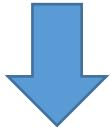
- Individual translation units are compiled to an intermediate form instead of machine code
- During the link process—when the entire program is visible—machine code is generated
- The compiler can take advantage of this to inline across translation units, or at least use information about the side effects of called functions to optimize
- NOTE: Invoking LTO can be done today; controlled through compile flags

# Other tricks pulled off by the compiler [called “strength reduction”]

You wrote this:

```
for (int i = 0; i < 100; ++i)
{
    func(i * 1234);
}
```

This is what the compiler will  
generate on your behalf:



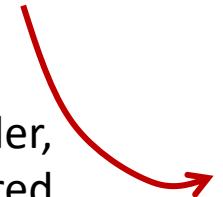
```
for (int iTimes1234 = 0; iTimes1234 < 100 * 1234; i += 1234)
{
    func(iTimes1234);
}
```

# One common theme for speeding up code: **code motion**

- Code Motion
  - Reduce frequency with which some computation is performed
    - Particularly rewarding when a piece of code can get moved out of a loop
    - Typically done by the compiler, under higher optimization levels – if this is possible (more later)
  - Caveat: Make sure the new code produces same result (if you, rather than the compiler goes for it)
- Code motion, basic idea, captured in this code snippet:

```
void set_row(double *a, double *b, int i, int n) {  
    for (int j = 0; j < n; j++)  
        a[n*i + j] = b[j];  
}
```

Typically done by compiler,  
if no red flags encountered



```
void set_row(double *a, double *b, int i, int n) {  
    int dummy = n * i;  
    for (int j = 0; j < n; j++)  
        a[dummy + j] = b[j];  
}
```

# Assembly code, no optimization

```
1 void set_row(double *a, double *b, int i, int n) {  
2     for (int j = 0; j < n; j++)  
3         a[n*i + j] = b[j];  
4 }
```

See line 3 (left) and line 19 (right)

x86-64 gcc 9.2 Compiler options...  
A Output... Filter... Libraries Add new...  
1 set\_row(double\*, double\*, int, int):  
2 push rbp  
3 mov rbp, rsp  
4 mov QWORD PTR [rbp-24], rdi  
5 mov QWORD PTR [rbp-32], rsi  
6 mov DWORD PTR [rbp-36], edx  
7 mov DWORD PTR [rbp-40], ecx  
8 mov DWORD PTR [rbp-4], 0  
9 .L3:  
10 mov eax, DWORD PTR [rbp-4]  
11 cmp eax, DWORD PTR [rbp-40]  
12 jge .L4  
13 mov eax, DWORD PTR [rbp-4]  
14 cdqe  
15 lea rdx, [0+rax\*8]  
16 mov rax, QWORD PTR [rbp-32]  
17 add rdx, rax  
18 mov eax, DWORD PTR [rbp-40]  
19 imul eax, DWORD PTR [rbp-36]  
20 mov ecx, eax  
21 mov eax, DWORD PTR [rbp-4]  
22 add eax, ecx  
23 cdqe  
24 lea rcx, [0+rax\*8]  
25 mov rax, QWORD PTR [rbp-24]  
26 add rax, rcx  
27 movsd xmm0, QWORD PTR [rdx]  
28 movsd QWORD PTR [rax], xmm0  
29 add DWORD PTR [rbp-4], 1  
30 jmp .L3  
31 .L4:  
32 nop  
33 pop rbp  
34 ret

# Assembly code, optimization -O1

Multiplication moved out of loop by compiler!

```
A ▾ Save/Load + Add new... ▾ Vim CppInsights Quick-bench
1 void set_row(double *a, double *b, int i, int n) {
2     for (int j = 0; j < n; j++)
3         a[n*i + j] = b[j];
4 }
```

Take home message: the compiler tries hard to help you

x86-64 gcc 9.2 (Editor #1, Compiler #1) C++ x  
x86-64 gcc 9.2 ✓ -O1

A ▾ Output... ▾ Filter... ▾ Libraries ▾ + Add new... ▾

```
1 set_row(double*, double*, int, int):
2     test    ecx, ecx
3     jle     .L1
4     lea     r8d, [rcx-1]
5     imul   ecx, edx
6     movsx  rcx, ecx
7     lea     rcx, [rdi+rcx*8]
8     mov     eax, 0
9 .L3:
10    movsd  xmm0, QWORD PTR [rsi+rax*8]
11    movsd  QWORD PTR [rcx+rax*8], xmm0
12    mov     rdx, rax
13    add     rax, 1
14    cmp     rdx, r8
15    jne     .L3
16 .L1:
17     ret
```

[Next thing to consider:] Function calls are optimization blockers

- Procedure to Convert String to Lower Case

```
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

# Convert Loop To Goto Form [first order approx. for assembly]

- `strlen` executed at every iteration (for a total of  $N$  times, if number of chars in  $s$  is  $N$ )

```
void lower(char *s) {
    size_t i = 0;
    if (i >= strlen(s))
        goto done;
loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
done:
}
```



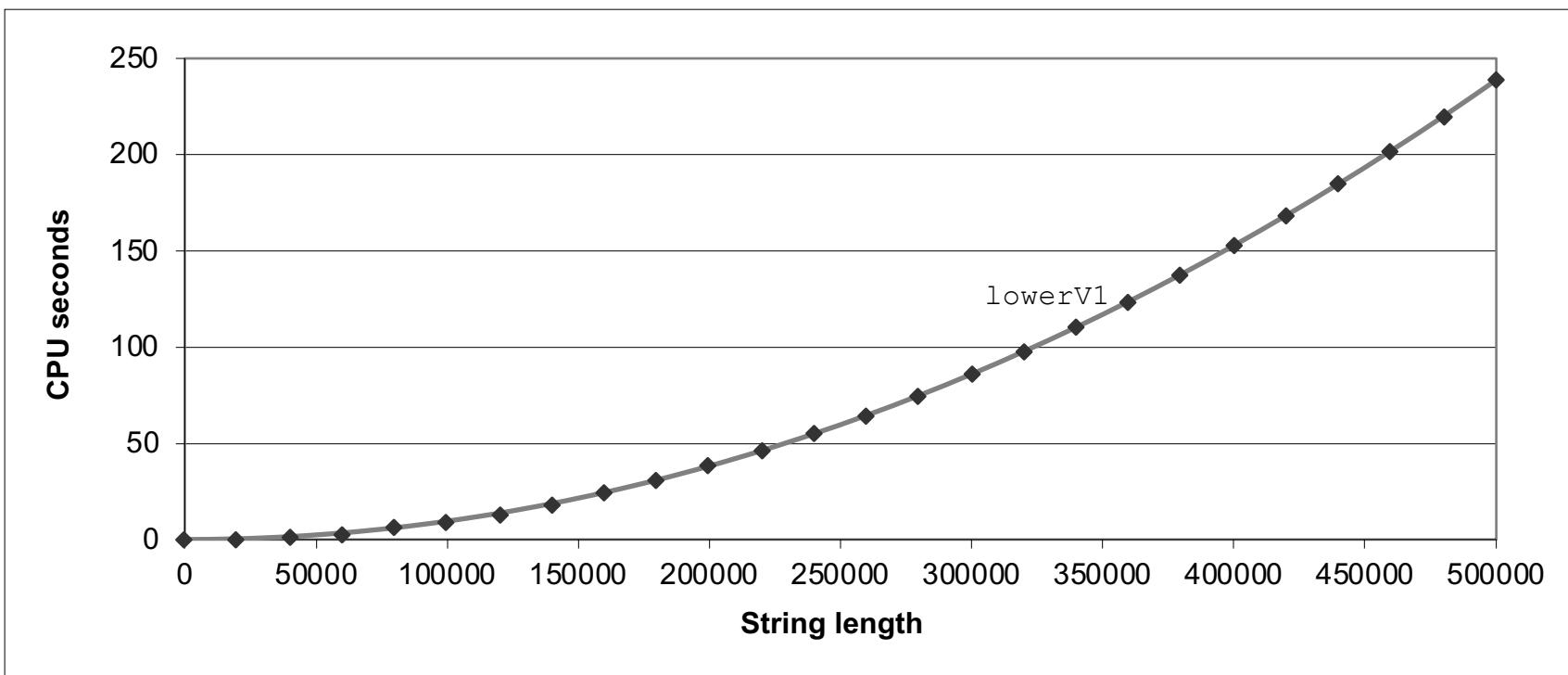
# Calling strlen

- **strlen performance**
  - Only way to determine length of string is to scan its entire length, looking for null '\0' character
- What's the impact of this?
  - N calls to strlen
  - Each call runs loop over N elements to find '\0'
  - Overall  $O(N^2)$  performance

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

# Lower Case Conversion Performance: version V1

- Time quadruples when double string length
- Quadratic performance



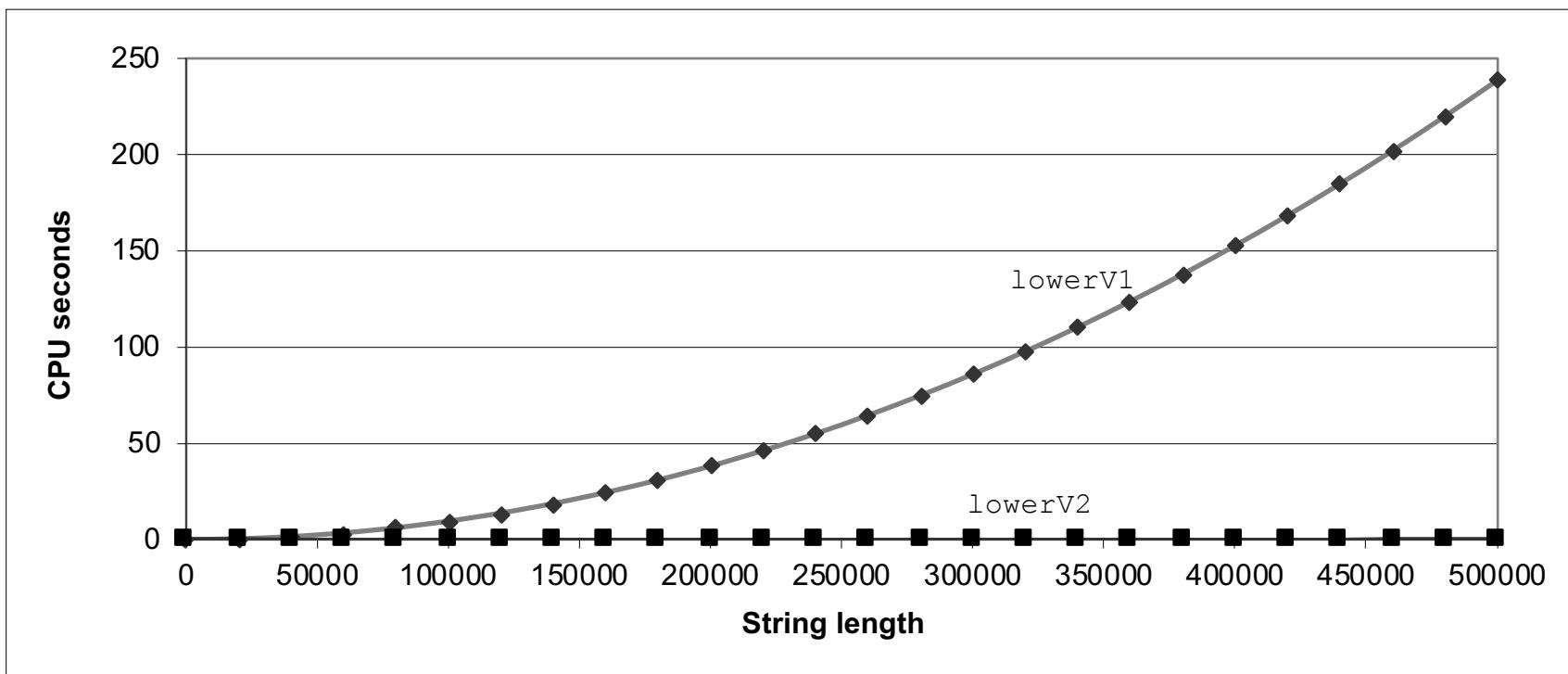
# Improving Performance

- Move call to **strlen** outside of loop
- Since result does not change from one iteration to another
- Perform “code motion”

```
void lower(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

# Lower Case Conversion Performance: version V2

- Time doubles when double string length
- Linear performance of lower2



# Optimization Blocker: Procedure Calls

- Why couldn't compiler move `strlen` out of inner loop?
  - A function may have side effects
  - e.g., alters global state each time called
  - Function may not return same value for given arguments
  - e.g., depends on global state (for instance, in multi-core parallelism)
- The takeaway message
  - Compiler treats procedure call as a black box
  - Weak optimizations near/around procedure call (stumbling block)
- Remedies:
  - Do your own code motion
  - C++: Use of inline functions
    - GCC does this with `-O1`
    - Within single file



```
size_t lencnt = 0;  
  
size_t strlen(const char *s)  
{  
    size_t length = 0;  
    while (*s != '\0') {  
        s++; length++;  
    }  
    lencnt += length;  
    return length;  
}
```

In this example, each time this `strlen` is called a variable is incremented (cooked up, but makes a point)

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 25

03/27/2020

# Things to do, for Canvas live-session

- Dan: start the Canvas recording
- Everybody else:
  - Please mute your microphone
  - Please ask questions (limit: two per person per lecture; after that, post on Piazza)
  - When you ask questions, first unmute your mic

# The quote the day

**“The best fighter is never angry.”**

-- Lao Tzu [ballpark 6th to 4th century BC]

# Before we get going...

- Last time:
  - Caches
    - Caches coherence & mechanisms to enforce it
    - False sharing
  - Optimization aspects
- Today:
  - Optimization aspects, wrap up
- Other tidbits:
  - ME759 Exam: April 15, at 7:15 PM, in Canvas
    - Review: Tu, April 14, at 7:00 PM, in Canvas
  - Office hours moved to Canvas (same time/date)
  - Assignment 9 went out; due on Th, 9 PM
    - We'll have 10 instead of 12 assignments
  - Final Project Proposal due tonight, 9 PM



# Starting off on the right foot: choose the right algorithm

- Choose the algorithm that is a good fit for your architecture and type of data you're handling
  - A mediocre algorithm choice ices all your other efforts
  - Consider what data the algorithm needs, how much of it, where you read from/write to, how often. That is, data access/movement
  - Select data structures in a way that leads to advantageous memory accesses
- Concept good to know: “asymptotic complexity” of an algorithm
  - What you study in an “Algorithms” class, tells you about level of effort to solve a certain class of problems
    - Example:
      - Solution of a linear system - LU factorization:  $O(2n^3/3)$  vs Cholesky factorization:  $O(n^3/3)$
      - Prefix scan – Hillis & Steele  $O(n \log(n))$  vs Harris  $O(n)$

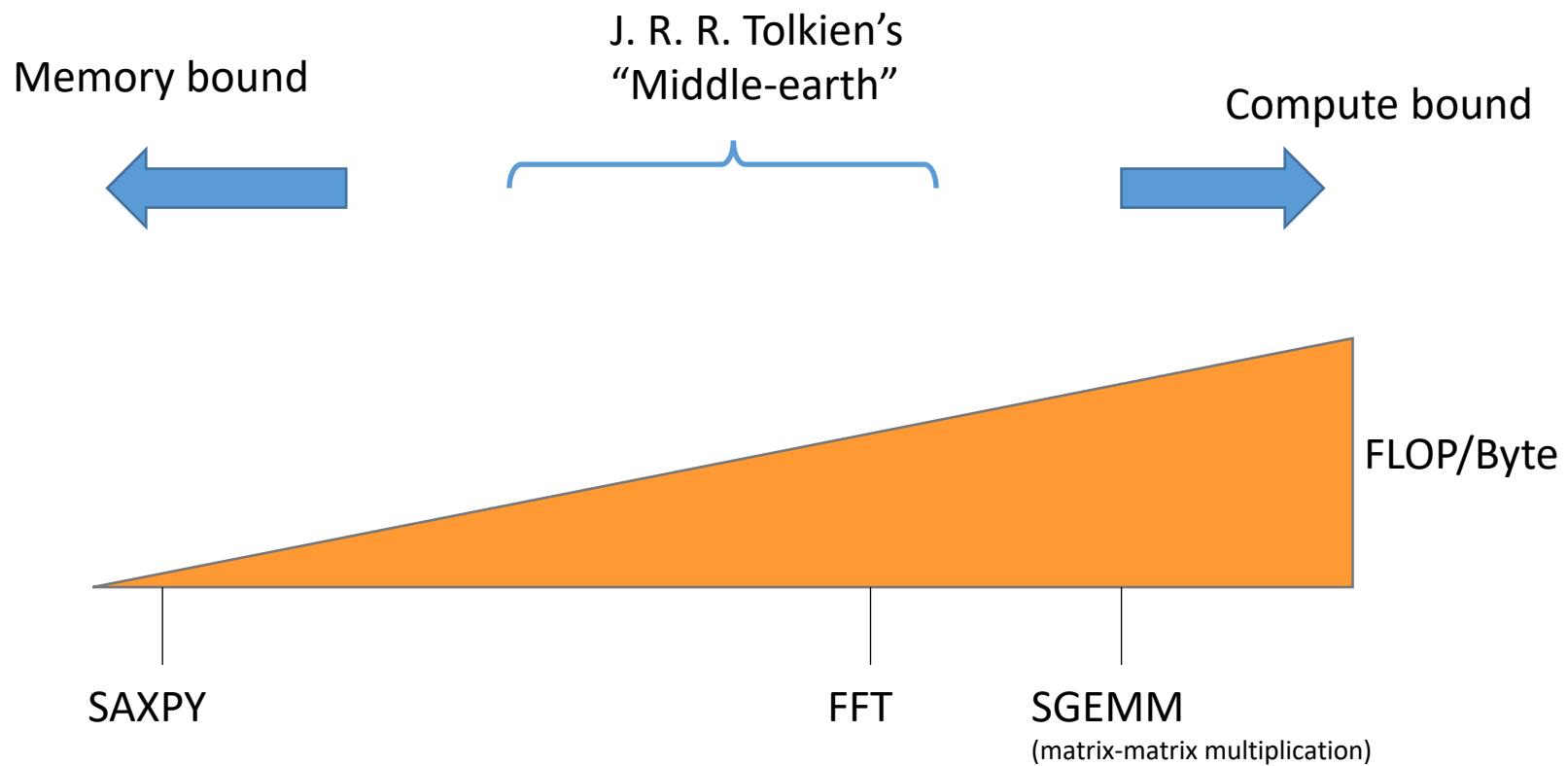
# Starting off on the right foot: choose the right algorithm

- When working on problem, there's more to performance than asymptotic complexity
  - Why?
    - Because asymptotic complexity is most often defined by number of operations
    - Memory transactions are rarely considered – they are specific to the hardware you'll eventually use to run
- If you understand how the computer works, it can guide you in designing a good/fast solution
  - Indeed, there is hope:
    - A problem can be solved by many algorithms (or approaches)
    - Also, one algorithm can have many implementations
- NOTE: You should spend time reflecting on your solution before writing any line of code!

# Asses the “Arithmetic Intensity” associated w/ your problem

- **Arithmetic intensity:** how much math you do per byte of data you bring from memory
- You can be in one of three cases:
  - Case 1: compute bound
    - That is, high arithmetic intensity – the ALU works a lot, not a whole amount of data movement to/from memory
  - Case 2: memory bound
    - You are moving data back and forth, likely only small parts of the data moved gets changed and/or used
    - Think cache misses
  - Case 3: “middle-earth” (J. R. R. Tolkien)
    - Somewhere in between compute-bound and memory-bound.
    - Your mission in this case: Try to move to Case 1 or Case 2 above

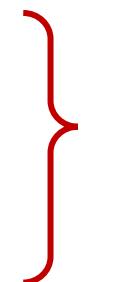
# Arithmetic Intensity: Putting things in perspective



# Simple Optimization Technique: Fusing Transformations

- One way to move the needle from “memory bound” towards “compute bound”
- Basic idea
  - Do not bring data into cache twice
  - If you use data X, Y, and Z – bring this data from memory once and then forget about it
- Why is it better?
  - You avoid memory traffic

```
for (int i = 0; i < N; i++)
    U[i] = F(X[i], Y[i], Z[i]);
for (int i = 0; i < N; i++)
    V[i] = G(X[i], Y[i], Z[i]);
```

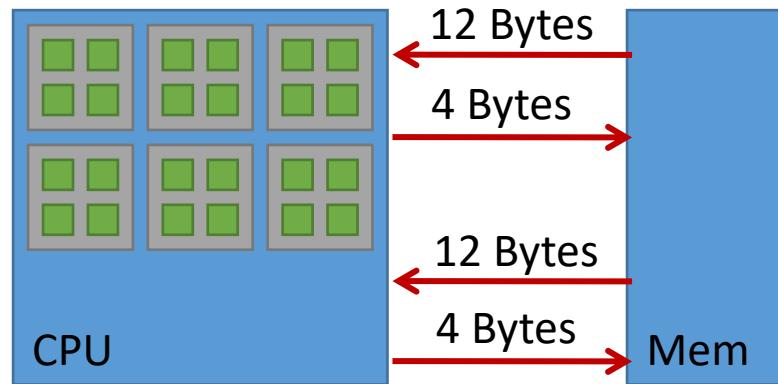


```
for (int i = 0; i < N; i++)
{
    U[i] = F(X[i], Y[i], Z[i]);
    V[i] = G(X[i], Y[i], Z[i]);
}
```

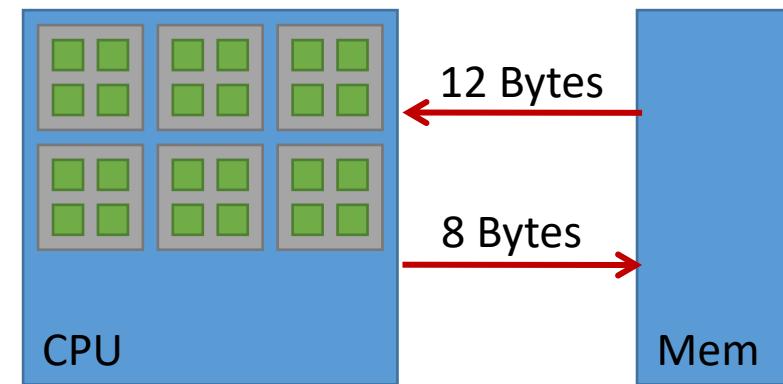
Loop Fusion

# Fusing transformations on previous slide

Original implementation



More thoughtful implementation



- Memory traffic cut by a factor of 1.6 (=32/20)

# Fusing Transformations

```
for (int i = 0; i < N; i++)
    Y[i] = F(X[i]);
```

```
for (int i = 0; i < N; i++)
    sum += Y[i];
```

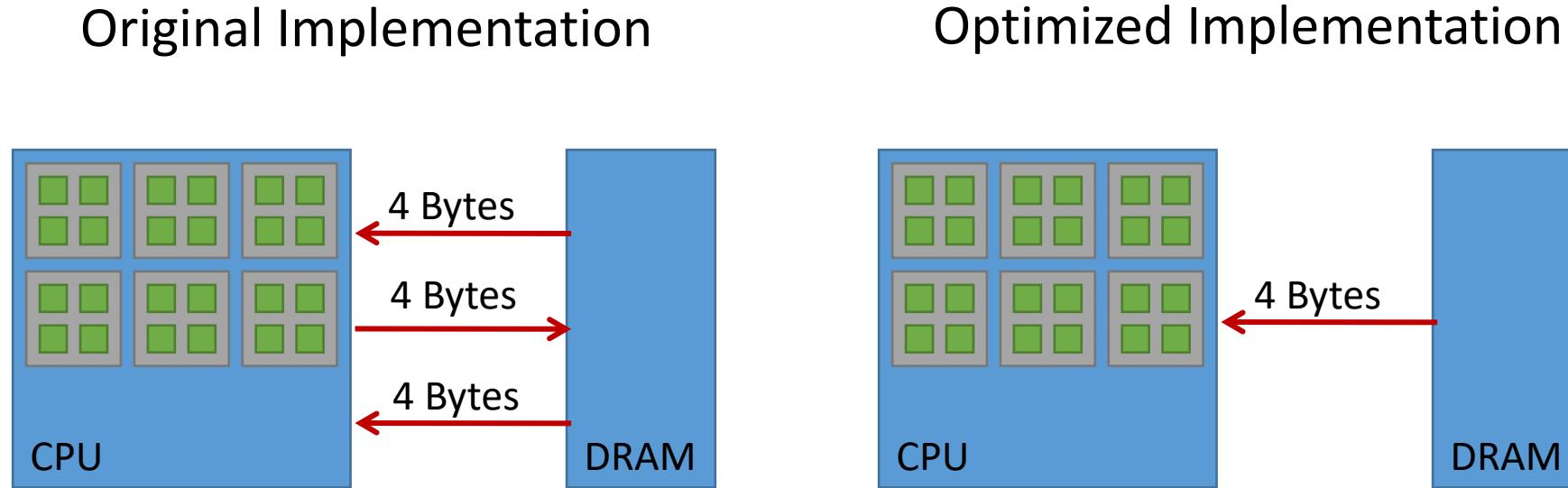


```
for (int i = 0; i < N; i++)
    sum += F(X[i]);
```

## Loop Fusion

- Mem traffic cut by a factor of 3
- Less memory used (no need to have Y)

# Fusing Transformations in Previous Example



- TAKE AWAY MESSAGE: Think if you can reduce in your code data movement since it is expensive:
  - Power costs
  - Time waiting for data to move around

# High Performance Computing: Questions to Ask

- Am I compute bound or memory bound?
- Do I understand the performance behavior of my code?
  - Does the performance match a model I have made? For instance, matrix-matrix multiplication should be  $O(N^3)$
- What is the optimal performance for my code on a given machine?
  - High Performance Computing == Computing at the bottleneck
- Can I change my code so that the “optimal performance” gets higher?
  - Circumventing/ameliorating the impact of the bottleneck

# Outside the scope of course: The roofline model

- I encourage you to read about the “roofline model” to assess code performance
  - “Are you compute bound?” ; “Am I memory bound?” ; “What chip/architecture might help me?”
- Access [here](#) “roofline model” material from Supercomputing 2019

# Let's talk next about the **compiler**...

- When you time/profile your code, you'll have to play with the compile flags
  - Execution speed impacted heavily by compiling flags (3-4X speedup)
- NOTE: Aggressive optimization done by compiler might change behavior of your code
  - Might change results
  - Might uncover skeletons in the closet
- Side note: fair run-time comparisons is tricky
  - Different compilers have different flags
    - Tons of compilers out there: gcc, clang, g++, icl, cl, xl, etc.
  - Some compilers very adept at leveraging the underlying processor

# How compilers come into play

- A compiler has a tough job
  - Register allocation
  - Instruction selection and ordering
  - Dead code elimination
  - Fix minor inefficiencies
- The expectation is that the compiler will provide an efficient mapping of program to machine
  - Compilers may or may not support features that have been included in a language recently (C++11, C++14, C++17, C++20, C++23, etc.)
- Something to think about:
  - At least in theory, Intel compiler ought to be best positioned when it comes to compiling code for Intel chips
    - It ought to know all the dark corners of the microarchitecture
    - Well positioned to first implement any new/exotic ISA features

# How compilers come into play

- Compilers don't improve asymptotic efficiency of your algorithm
  - Up to programmer to select the right algorithm for the problem & the architecture
- Compilers have difficulty overcoming “optimization blockers”
  - Potential memory aliasing
  - “function call” side-effects

# The compiler has a tough job

- Often, compiler ends up doing analysis only at the function level
- Newer versions of gcc do inter-procedural analysis within **individual** files
- Whole-program analysis for some reason not done too often
  - More on this later

```
int functionFromOtherFile(double*);  
double someOtherRemoteFunc(double*);  
  
double myFunc() {  
    const size_t N = 1024;  
    double foo[N];  
    double accumulator = 0.;  
    for (int i = 0; i < functionFromOtherFile(foo); i++) {  
        accumulator += someOtherRemoteFunc(foo);  
    }  
    return accumulator;  
}
```



Compiler in a tough spot: function bodies are often hidden from call sites, compiler can only see their declaration

# The compiler has a tough job

- The compiler operates under tight constraints
  - It must obey all the rules in the C/C++ standard
    - Often prevents compiler from making optimizations that would only affect behavior under pathological conditions
  - Rule above possibly ignored though when your code is making use of nonstandard language features
    - This is what gets the compiler confused and it needs to play it super safe so that you are not mad at it
- Behavior that may be obvious to the programmer can be obfuscated by coding styles and poor understanding of the language
- Use of pointers is a big stumbling block (see more in 10 slides)

# The compiler has a tough job

- Most analysis based only on static information
  - Compiler can't anticipate run-time inputs (in general)
- One way in which we don't help compilers is by writing code that is ambiguous, open to interpretation
  - The compilers can't read our minds
- When in doubt, the compiler must be conservative

# Tricks of the trade, done by compiler

- Function inlining – this is a big one
  - Wherever you call a function, the compiler simply replaces the call to function foo with the entire body of function foo (the entire function body dropped in)
  - Not always possible, you must explicitly call for it (by function decoration and/or compile flags)
    - Nowadays you can ask the compiler to report back what functions it successfully inlined
- Consequences, if you can inline:
  - You have one less jump
  - More important: the compiler sees way more code and therefore engages in more fruitful optimizations
    - It's one thing to optimize when you have 5 lines of code, and a different one for 500 lines of code

# Other tricks of the trade, done by compiler

- Constant propagation `const double myApplePie = 3.1415;`
  - Compiler substitutes all references to `myApplePie` with the constant value
- Common subexpression elimination
- Dead code removal
  - There may be areas of the code that have no effect on the output, and these can be removed
    - This includes loads and stores whose values are unused, as well as entire functions and expressions
- Tail call removal
  - Set of recursive calls to same function get replaced by a loop

# How can you help the compiler?

## A) Allow it to see as much code as possible

- You set it free, so to speak
- Recall that typically compiler sees only one file at a time
  - Can't peek at some function you use that is defined in a different file

## B) Provide flags to convey to it information

- E.g.: telling compiler your target architecture can help quite a lot

# Compilers, and link time optimization (LTO)

- Compilers are becoming faster and smarter
- It's affordable and profitable to do program-level, not function level optimization
- LTO (also known as LTCG, for “link time code generation”) can be used to allow the compiler to see across translation unit boundaries
- Likely to provide good speed up for complex codes with many source files

# LTO: Done how?

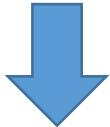
- Individual translation units are compiled to an intermediate form instead of machine code
- During the link process—when the entire program is visible—machine code is generated
- The compiler can take advantage of this to inline across translation units, or at least use information about the side effects of called functions to optimize
- NOTE: Invoking LTO can be done today; controlled through compile flags

# Other tricks pulled off by the compiler [called “strength reduction”]

You wrote this:

```
for (int i = 0; i < 100; ++i)
{
    func(i * 1234);
}
```

This is what the compiler will  
generate on your behalf:



```
for (int iTimes1234 = 0; iTimes1234 < 100 * 1234; i += 1234)
{
    func(iTimes1234);
}
```

# Other lesser trick: Share Common Subexpressions

- Reuse portions of expressions
- GCC will take care of this for you with `-O1` flag

```
/* Sum neighbors of i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n      + j-1];
right = val[i*n      + j+1];
sum = up + down + left + right;
```

Up  
Down  
Left  
Right

Stencil Operations

```
long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications:  $i \cdot n$ ,  $(i-1) \cdot n$ ,  $(i+1) \cdot n$

1 multiplication:  $i \cdot n$

```
leaq  1(%rsi), %rax # i+1
leaq  -1(%rsi), %r8  # i-1
imulq %rcx, %rsi    # i*n
imulq %rcx, %rax    # (i+1)*n
imulq %rcx, %r8    # (i-1)*n
addq  %rdx, %rsi    # i*n+j
addq  %rdx, %rax    # (i+1)*n+j
addq  %rdx, %r8    # (i-1)*n+j
```

```
imulq  %rcx, %rsi  # i*n
addq  %rdx, %rsi  # i*n+j
movq  %rsi, %rax  # i*n+j
subq  %rcx, %rax  # i*n+j-n
leaq  (%rsi,%rcx), %rcx # i*n+j+n
```

# Other lesser tricks: Replace costly operations with simpler ones

- Examples:

- Compiler will use “shift” whenever you multiply by a power of 2
  - $16 * x \rightarrow x << 4$
  - Do not do this yourself, let the compiler do it, since it's machine dependent
- Compiler will go to great lengths to avoid divisions [of integers](#)
  - Around 20 times slower than an addition, and 5 times more expensive than multiplication
  - Division by power of 2 is simple (shift the other way, let the compiler do it)
  - Here's a divide by 3, there is no division, actually:

```
unsigned divideByThree(unsigned x){  
    return x / 3;  
}
```

```
divideByThree(unsigned int):  
    mov    eax, edi          ; eax = edi  
    mov    edi, 2863311531   ; edi = 0xaaaaaaaaab  
    → imul   rax, rdi       ; rax = rax * 0xaaaaaaaaab  
    shr    rax, 33           ; rax >= 33  
    ret
```

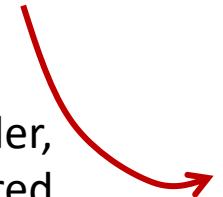
Computes 0.33333333337213844 of the input

# One common theme for speeding up code: **code motion**

- Code Motion
  - Reduce frequency with which some computation is performed
    - Particularly rewarding when a piece of code can get moved out of a loop
    - Typically done by the compiler, under higher optimization levels – if this is possible (more later)
  - Caveat: Make sure the new code produces same result (if you, rather than the compiler goes for it)
- Code motion, basic idea, captured in this code snippet:

```
void set_row(double *a, double *b, int i, int n) {  
    for (int j = 0; j < n; j++)  
        a[n*i + j] = b[j];  
}
```

Typically done by compiler,  
if no red flags encountered



```
void set_row(double *a, double *b, int i, int n) {  
    int dummy = n * i;  
    for (int j = 0; j < n; j++)  
        a[dummy + j] = b[j];  
}
```

# Assembly code, no optimization

```
1 void set_row(double *a, double *b, int i, int n) {  
2     for (int j = 0; j < n; j++)  
3         a[n*i + j] = b[j];  
4 }
```

See line 3 (left) and line 19 (right)

x86-64 gcc 9.2 Compiler options...  
A Output... Filter... Libraries Add new...  
1 set\_row(double\*, double\*, int, int):  
2 push rbp  
3 mov rbp, rsp  
4 mov QWORD PTR [rbp-24], rdi  
5 mov QWORD PTR [rbp-32], rsi  
6 mov DWORD PTR [rbp-36], edx  
7 mov DWORD PTR [rbp-40], ecx  
8 mov DWORD PTR [rbp-4], 0  
9 .L3:  
10 mov eax, DWORD PTR [rbp-4]  
11 cmp eax, DWORD PTR [rbp-40]  
12 jge .L4  
13 mov eax, DWORD PTR [rbp-4]  
14 cdqe  
15 lea rdx, [0+rax\*8]  
16 mov rax, QWORD PTR [rbp-32]  
17 add rdx, rax  
18 mov eax, DWORD PTR [rbp-40]  
19 imul eax, DWORD PTR [rbp-36]  
20 mov ecx, eax  
21 mov eax, DWORD PTR [rbp-4]  
22 add eax, ecx  
23 cdqe  
24 lea rcx, [0+rax\*8]  
25 mov rax, QWORD PTR [rbp-24]  
26 add rax, rcx  
27 movsd xmm0, QWORD PTR [rdx]  
28 movsd QWORD PTR [rax], xmm0  
29 add DWORD PTR [rbp-4], 1  
30 jmp .L3  
31 .L4:  
32 nop  
33 pop rbp  
34 ret

# Assembly code, optimization -O1

Multiplication moved out of loop by compiler!

```
A Save/Load + Add new... Vim CppInsights Quick-bench
1 void set_row(double *a, double *b, int i, int n) {
2     for (int j = 0; j < n; j++) {
3         a[n*i + j] = b[j];
4     }
```

Take home message:  
the compiler is your friend

The screenshot shows the Godbolt Compiler Explorer interface. On the left, the C++ code for the `set_row` function is displayed. On the right, the generated assembly code for x86-64 architecture using gcc 9.2 is shown. A red arrow points from the C++ code to the assembly code. Another red arrow points from the "-O1" button in the toolbar to the assembly code, indicating that the optimizer has moved the multiplication out of the loop.

Line	Assembly Instruction	Description
1	set_row(double*, double*, int, int):	
2	test    ecx, ecx	
3	jle     .L1	
4	lea     r8d, [rcx-1]	
5	imul   ecx, edx	
6	movsx  rcx, ecx	
7	lea     rcx, [rdi+rcx*8]	
8	mov    eax, 0	
9	.L3:	
10	movsd  xmm0, QWORD PTR [rsi+rax*8]	Bring b[j] into xmm0
11	movsd  QWORD PTR [rcx+rax*8], xmm0	Move xmm0 into a
12	mov    rdx, rax	Move value of j into rdx
13	add    rax, 1	Increment rax; i.e., j, by 1
14	cmp    rdx, r8	Compare j with n (in r8)
15	jne     .L3	Jump to .L3 if not equal
16	.L1:	
17	ret	

[Next thing to consider:] Function calls are optimization blockers

- Procedure to Convert String to Lower Case

```
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

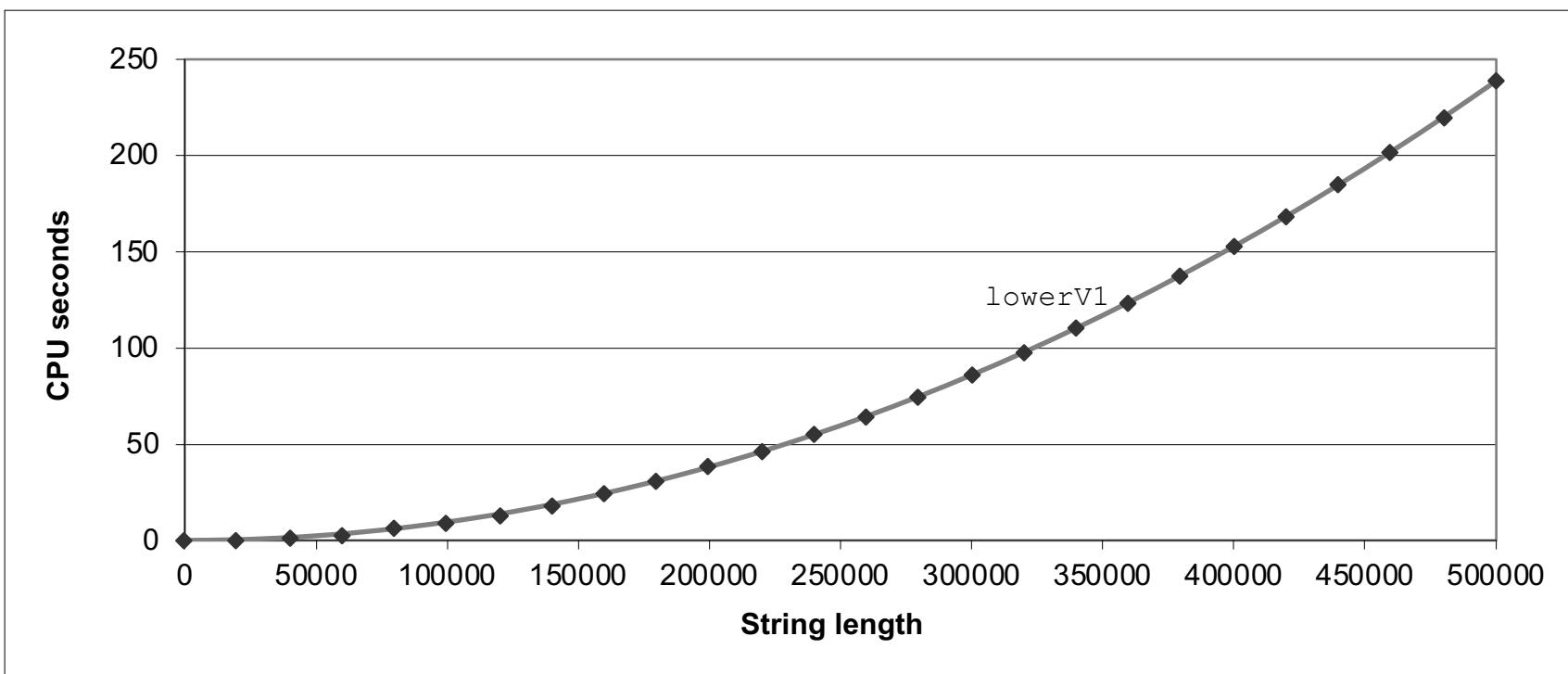
# Calling `strlen`

- `strlen` performance
  - Only way to determine length of string is to scan its entire length, looking for null '\0' character
- We have N calls to `strlen`. What's the impact?
  - Each call runs loop over N elements to find '\0'
  - Impact: Overall,  $O(N^2)$  performance

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

# Lower Case Conversion Performance: version V1

- Time quadruples when double string length
- Quadratic performance



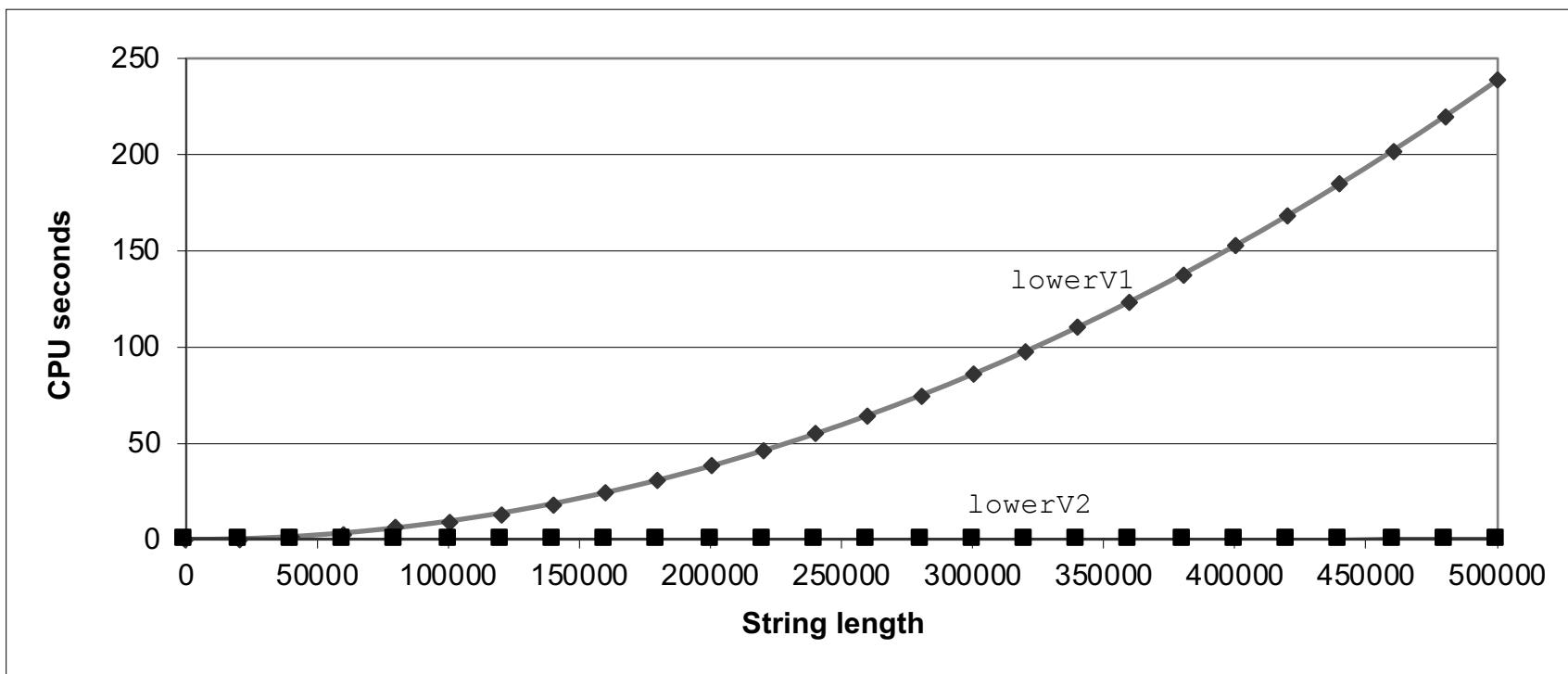
# Improving Performance

- Move call to **strlen** outside of loop
  - Note that you know that the length `len` does not change from one iteration to another
  - You took it upon yourself to perform “code motion”

```
void lower(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

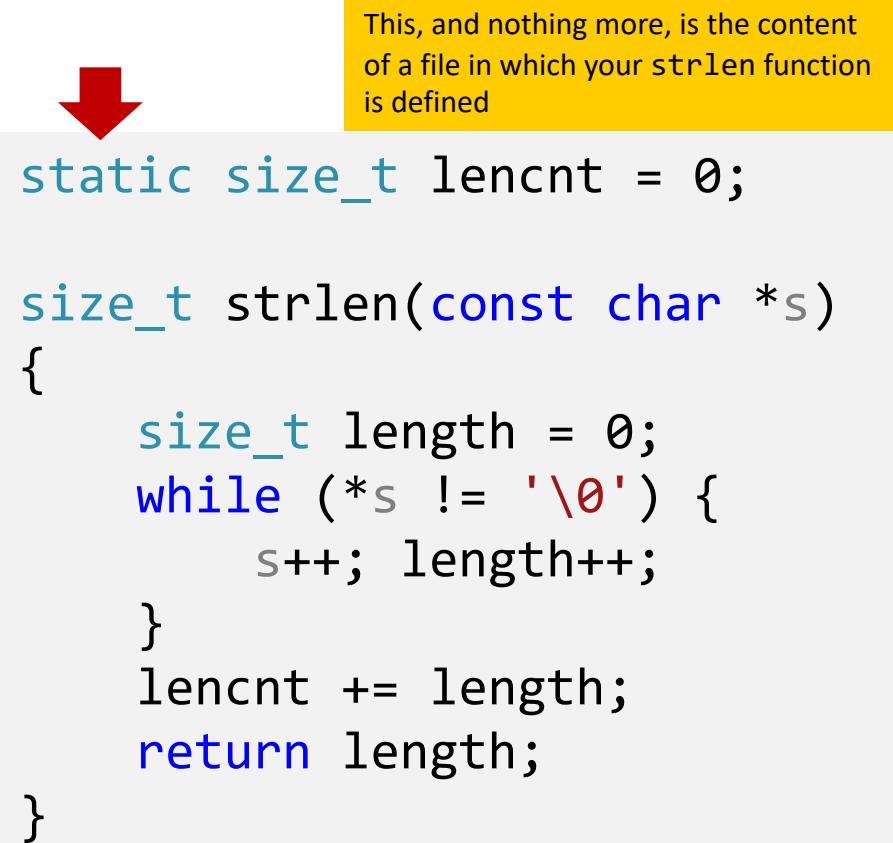
# Lower Case Conversion Performance: version V2

- Time doubles when double string length
- Linear performance of lower2



# Optimization Blocker: Procedure Calls

- Why couldn't compiler move `strlen` out of inner loop?
  - A function might have side effects
    - e.g., alters global state each time called
  - Function might not return same value for given arguments
    - e.g., depends on global state (for instance, in multi-core parallelism)
- The takeaway message
  - Compiler must treat procedure call as a black box
  - Weak optimizations near/around procedure call (stumbling block)
  - Compiler cannot read your mind; it does exactly what you told it to do
- Possible remedies:
  - Try to do your own code motion (you should know if it's safe or not)
  - C++: Use of inline functions
    - GCC does this with `-O1`
      - Within single file
  - Use LTO (more on this later)



This, and nothing more, is the content of a file in which your `strlen` function is defined

```
static size_t lencnt = 0;  
  
size_t strlen(const char *s)  
{  
    size_t length = 0;  
    while (*s != '\0') {  
        s++; length++;  
    }  
    lencnt += length;  
    return length;  
}
```

In this example, each time this `strlen` is called a variable is incremented (cooked up, but makes a point)

# [Next thing to consider:] Memory matters, and the trickiness of pointers

```
// sum entries in each row of A and store in vector b
void sum_rows1(double *A, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += A[i*n + j];
    }
}
```



Assembly code, sum\_rows1, inner loop (after j)  
(gcc, optimized w/ -O1, from <https://godbolt.org/>)

.L3:

movsd	xmm0, QWORD PTR [rdx]	Bring b[i] from memory into register xmm0
addsd	xmm0, QWORD PTR [rax]	Add to register xmm0 the right A entry from memory
movsd	QWORD PTR [rdx], xmm0	Move updated b[i] value back into memory
add	rax, 8	Make rax have the address to next entry in A (double takes 8 bytes)
cmp	rax, rcx	Compare i with n (latter stored in rcx)
jne	.L3	Jump to .L3 if not equal

- Code updates b[i] on every iteration
- Why couldn't compiler optimize this away?

# Memory Aliasing

```
#include <stdlib.h>
/* Sum rows of n X n matrix A and store in vector b */
void sum_rows1(double *A, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += A[i * n + j];
    }
}
```

```
int main() {
    double A[9] = {0, 1, 2, 4, 8, 16, 32, 64, 128};

    double *b;
    b = A + 3;
    sum_rows1(A, b, 3);
    return 0;
}
```

```
double A[9] =
{ 0, 1, 2,
  3, 22, 224,
  32, 64, 128};
```

Value of b:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

A,9	0x000000ff3a8ffc60	double[9]
[0]		0double
[1]		1double
[2]		2double
[3]		3double
[4]		22double
[5]		224double
[6]		32double
[7]		64double
[8]		128double
b,3	0x000000ff3a8ffc78	double[3]
[0]		3double
[1]		22double
[2]		224double

- The code updates **b[i]** on every iteration. It does so because:
  - Must consider possibility that these updates will affect program behavior
  - Value of “b” might change from when **i=3** to when **i=4** (parallel thread might interfere)

# Removing Aliasing

- Assume that there is no aliasing
  - That is, no monkey business in main driver
- You can do code motion
  - Result: Accessing array b only once!!!
- Number of instructions drops down
  - Performance goes up

.L3:

```
addsd    xmm0, QWORD PTR [rax]
add      rax, 8
cmp      rax, rcx
jne     .L3
```

```
/* Sum rows of n X n matrix A and store in vector b */
void sum_rows2(double *A, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += A[i*n + j];
        b[i] = val;
    }
}
```



Assembly code, sum\_rows2, inner loop (after j)  
(gcc, optimized w/ -O2, from <https://godbolt.org/>)

Bring b[i] from memory and add to what's in xmm0

Make rax point to next entry in row of A

Have you exhausted all entries in row "i" of A?

Jump back and do another trip if not more elems in row "i"

# [Next thing to consider:] Memory Matters

.L3:

```
movsd  xmm0, QWORD PTR [rdx]
addsd  xmm0, QWORD PTR [rax]
movsd  QWORD PTR [rdx], xmm0
add    rax, 8
cmp    rax, rcx
jne    .L3
```

.L3:

```
addsd  xmm0, QWORD PTR [rax]
add    rax, 8
cmp    rax, rcx
jne    .L3
```

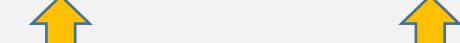
- Bottom line: → **QWORD PTR** shows up: three times (left) vs. one time (right)
  - That is, three vs. one memory accesses. Leads to **significant speedup**

# Optimization Blocker: Memory Aliasing

- Aliasng

- Two different memory references specify single location
- Easy to have happen in C - you have low level access to memory through pointers
- Way around it:
  - Used local variables, like in the previous example – handy to accumulate within loops
    - You'll help the compiler – it can issue better instructions
  - Use the **restrict** keyword, see code snippet
    - Feature added to the standard in 1999 (C99)

```
#include <stdlib.h>
/* Sum rows of n X n matrix A and store in vector b */
void sum_rows1(double * restrict A, double * restrict b, long n)
{
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += A[i * n + j];
    }
}
```



# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 26

03/30/2020

# Things to do, for Canvas live-session

- Dan: start the Canvas recording
- Everybody else:
  - Please mute your microphone
  - Please ask questions (limit: two per person per lecture; after that, post on Piazza)
  - When you ask questions, first unmute your mic

# The quote the day

“Kids, you tried your best and you failed miserably. The lesson is, never try.”

-- Homer Simpson, Nuclear Safety Inspector [1989 - ]

# Before we get going...

- Last time:
  - Optimization aspects
    - Help the compiler help you
    - Memory aliasing
- Today:
  - Optimization aspects, wrap up
  - Start parallel computing w/ MPI
- Other tidbits:
  - ME759 Exam: April 15, at 7:15 PM, in Canvas
    - Review: Tu, April 14, at 7:00 PM, in Canvas
  - Final Project Proposal: Dan to provide feedback this week
  - Last ME759 lecture is on Friday

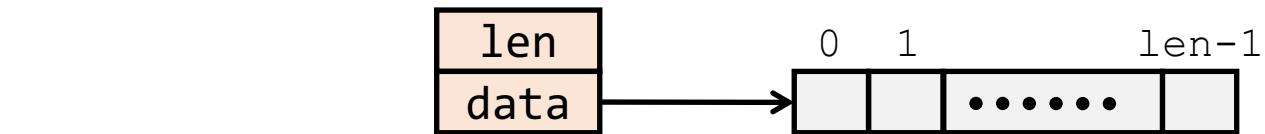
```
// sum entries in each row of A and store in vector b
void sum_rows1(double *A, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += A[i*n + j];
    }
}
```

# New Example, w/ 3 actors – a struct + two functions

- Our struct will use various Data Types

- Use different declarations for `data_t`:

- `int`
- `long`
- `float`
- `double`



```
/* data structure for vectors */
typedef struct {
    size_t len;
    data_t *data;
} vec;
```

```
/* retrieve vector element and store in val */
void get_vec_element(vec* v, size_t idx, data_t *val) {
    if (idx >= v->len || idx < 0)
        *val = 0;
    else
        *val = v->data[idx];
}
```

The first function  
that comes into play

# Wrapping up the setup, for the new example

```
void combine1(vec_ptr v, data_t *dest) {
    *dest = IDENT; // the identity value for OP
    for (int i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

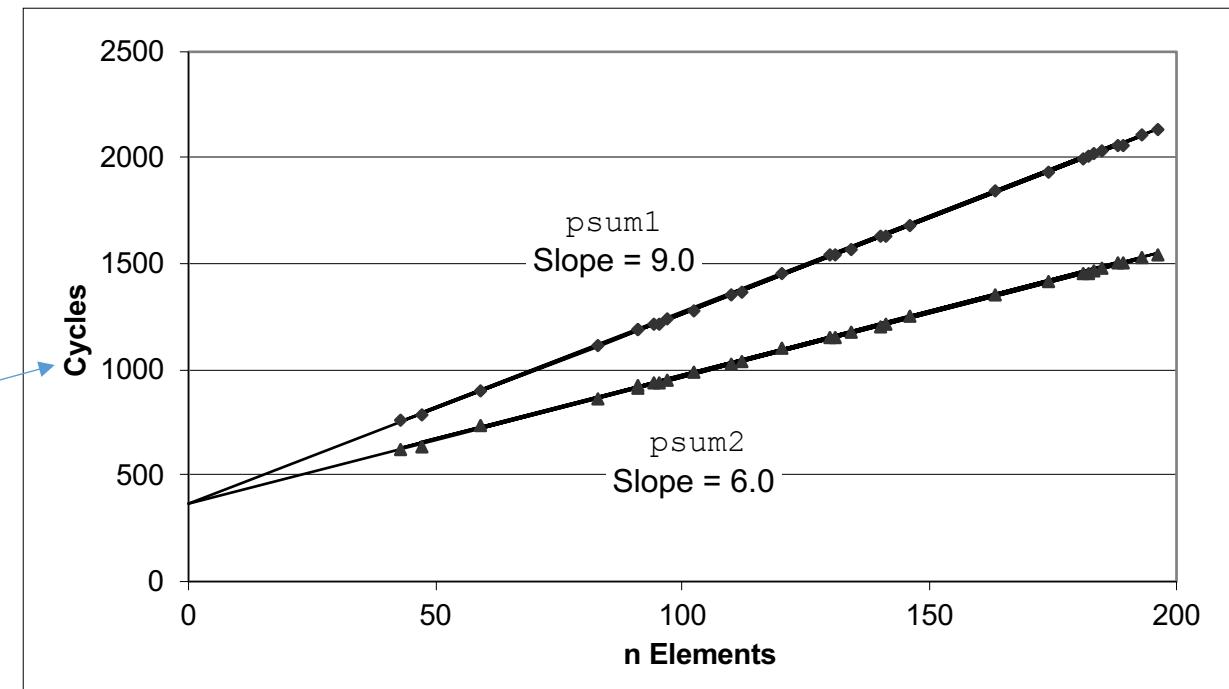
Compute sum or product of vector elements (based on **OP**)

The second function that comes into play

- Data Types
  - Use different declarations for **data\_t**
    - **int**
    - **long**
    - **float**
    - **double**
- Operations (**OP**)
  - Use different definitions of **OP** and **IDENT**
    - “+” : 0
    - “\*” : 1

# Cycles Per Element (CPE)

- Assume a program operates on vectors or lists
  - Vector Length =  $n$
- Assume we're interested in sweep over the vector
  - Number of OPs is  $n$  ( $n-1$ , to be correct)
- Thus, **cycles per OP  $\approx$  cycles per element (CPE)**
- Sweep, total # of cycles:  $C = \text{CPE} * n + \text{Overhead}$ 
  - CPE is slope of line
  - See figure



# Benchmark Performance

```
void combine1(vec_ptr v, data_t *dest) {
    *dest = IDENT; // the identity value for OP
    for (int i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Sweep to compute sum or product of vector elements

Results in CPE (cycles per element) – Smaller is BETTER!

Method	data_t: Integer		data_t: Double FP	
Operation (OP)	add	mult	add	mult
combine1 unoptimized	22.68	20.02	19.98	20.18
combine1 with -O1	10.12	10.12	10.17	11.14
combine1 with -O3	4.5	4.5	6	7.8

# Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    data_t *d = get_vec_start(v); ← No function call here anymore
    data_t temp = IDENT;
    for (int i = 0; i < length; i++)
        temp = temp OP d[i];
    *dest = temp;
}
```

Shortcut: you get the raw pointer, and index into it...  
No need to call `get_vec_element`

- Modification #1: `vec_length` moved out of loop
  - One function call avoided at each iteration
- Modification #2: bypassed `get_vec_element` call
- Modification #3: Accumulate in temporary “temp”

# Effect of Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t temp = IDENT;
    for (int i = 0; i < length; i++)
        temp = temp OP d[i];
    *dest = temp;
}
```

Method	Integer		Double FP	
Operation	add	mult	add	mult
combine1 -01	10.12	10.12	10.17	11.14
combine4	1.27	3.01	3.01	5.01

- Eliminates overhead in loop & can do prefetching

# Opportunities for Efficiency Gains

Cluster (distributed mem.)	Group of nodes communicate through fast interconnect	MPI, Charm++, Chapel
Multi-Socket Node	Group of CPUs on the same node, talk through main mem.	OpenMP, MPI
Acceleration (GPU/Phi)	Compute devices accelerating parallel computation on one node	CUDA, OpenCL, OpenMP
Multi-Core Processor	Communication through shared caches and main mem.	OpenMP, TBB, pthreads
Vectorization	Higher operation throughput via special/fat registers	AVX, SSE
Pipelining	Sequence of instruction sharing functional units	Assembly
Superscalar	Non-sequence instructions sharing functional units	Assembly

We have full control → 

We have  little to no control → 

Little, but might pack a punch!

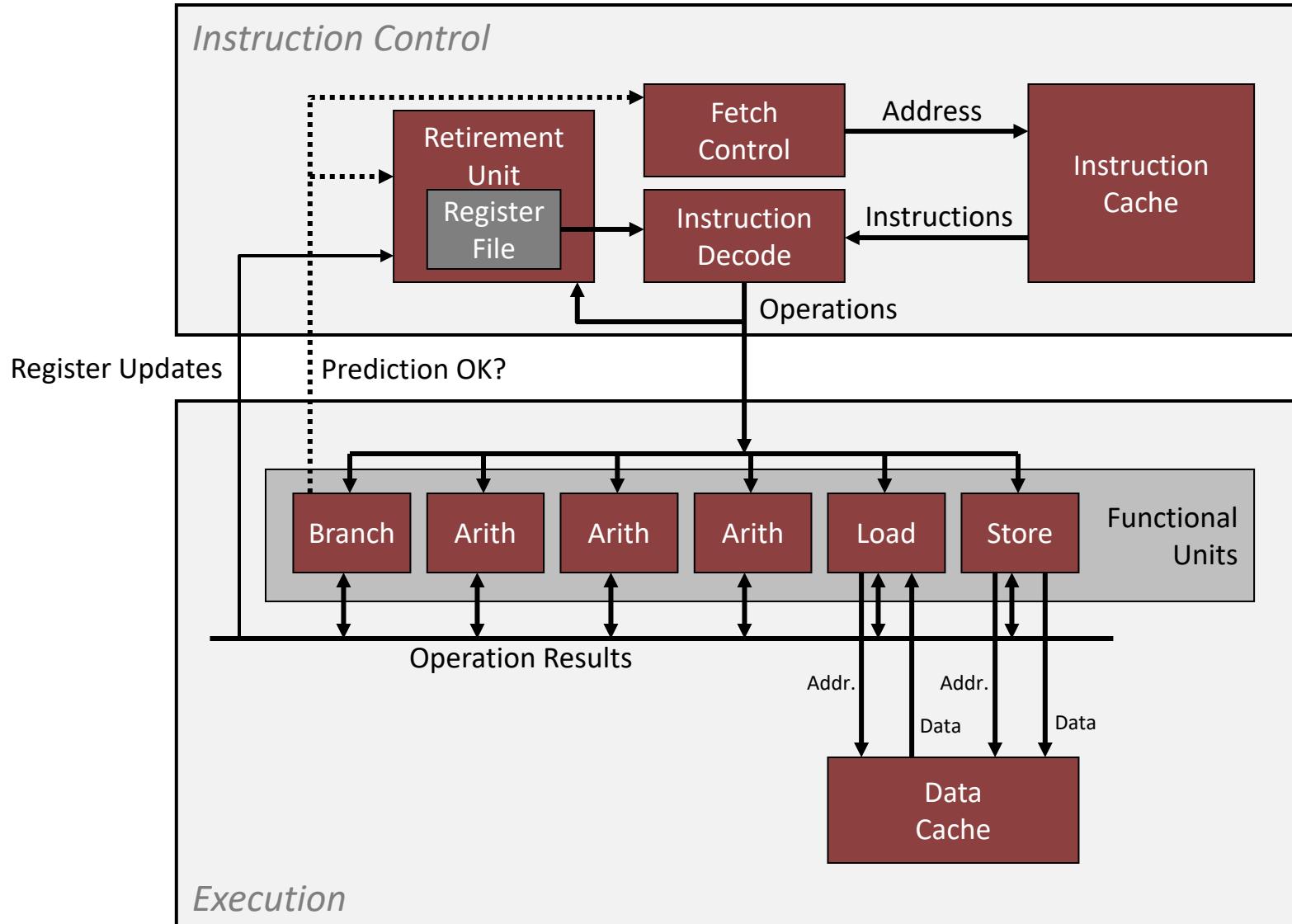
[Next thing to consider:] Exploiting Instruction-Level Parallelism

- Simple ILP-targeted transformations can yield good performance improvements
  - Compilers often cannot make these transformations since they don't know if it's safe to do so or not
- ILP
  - Hardware can execute multiple instructions in parallel
  - Performance limited by ILP hazards

# Curbing our enthusiasm, ILP hazards

- Structural hazards
  - The scheduler: “Unfortunately, there aren’t enough hammers in this plant”
- Data dependency hazards
  - The scheduler: “This next instruction depends on the outcome of the instruction right ahead of it”
- Control hazards
  - The scheduler: “I’m not sure what is supposed to happen next”

# Modern CPU Design [old slide]

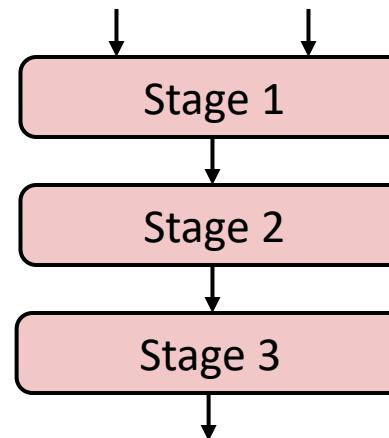


# Recall ILP discussion, 1.5 months ago: superscalar attribute

- **Superscalar processor**: one that can issue more than one *instruction in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically
- **Superscalar solution**: one manifestation of the *instruction level parallelism* idea
- Note: Most modern CPUs are superscalar
  - Intel: since Pentium (1993)

# Old ILP discussion: the pipelining attribute

```
int mult_example(int a, int b, int c) {  
    int p1 = a * b;  
    int p2 = a * c;  
    int p3 = p1 * p2;  
    return p3;  
}
```



	Time							
	1	2	3	4	5	6	7	
Stage 1	a*b	a*c			p1*p2			
Stage 2		a*b	a*c			p1*p2		
Stage 3			a*b	a*c			p1*p2	

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage s can start on new computation once values passed to s+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles
- Note when  $p1*p2$  is issued – classical data dependency situation

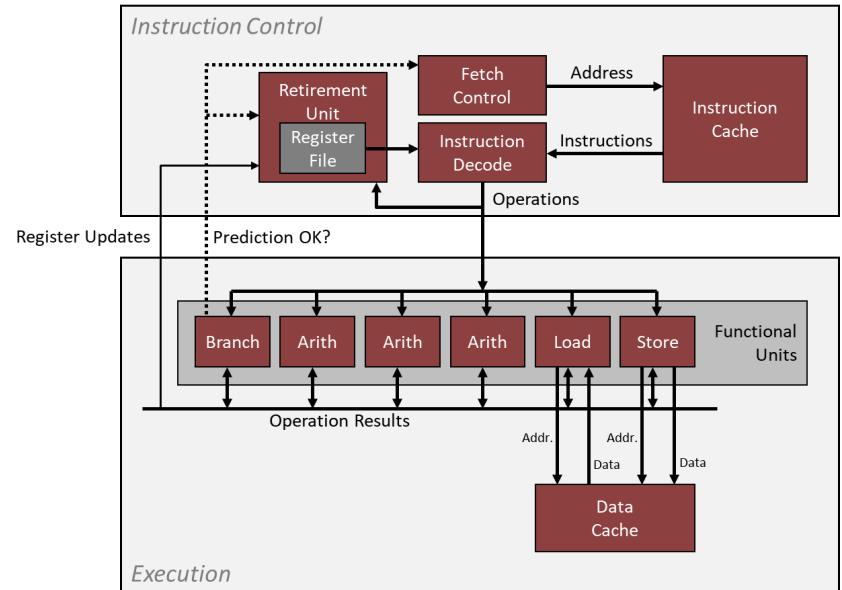
# Example: The Intel Haswell CPU – aspects that are ILP relevant

- Haswell: 8 Total Functional Units
- Multiple instructions can execute in parallel

2 load, with address computation  
1 store, with address computation  
4 integer ops (doing address arithmetic too – explains why you have more)  
2 FP multiply  
1 FP add  
1 FP divide

- Some instructions take > 1 cycle, but can be pipelined

<i>Instruction</i>	<i>Latency, cycles</i>	<i>Cycles/Issue</i>
Load / Store	4	1
Integer Multiply	3	1
Integer Add	1	1
<b>Integer/Long Divide</b>	<b>3-30</b>	<b>3-30</b>
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
<b>Single/Double FP Divide</b>	<b>3-15</b>	<b>3-15</b>



For this type of info: see Agner Fog's optimization [manual](#) (part of "ME759 assigned reading" list)

# x86-64 Compilation of **combine4** function

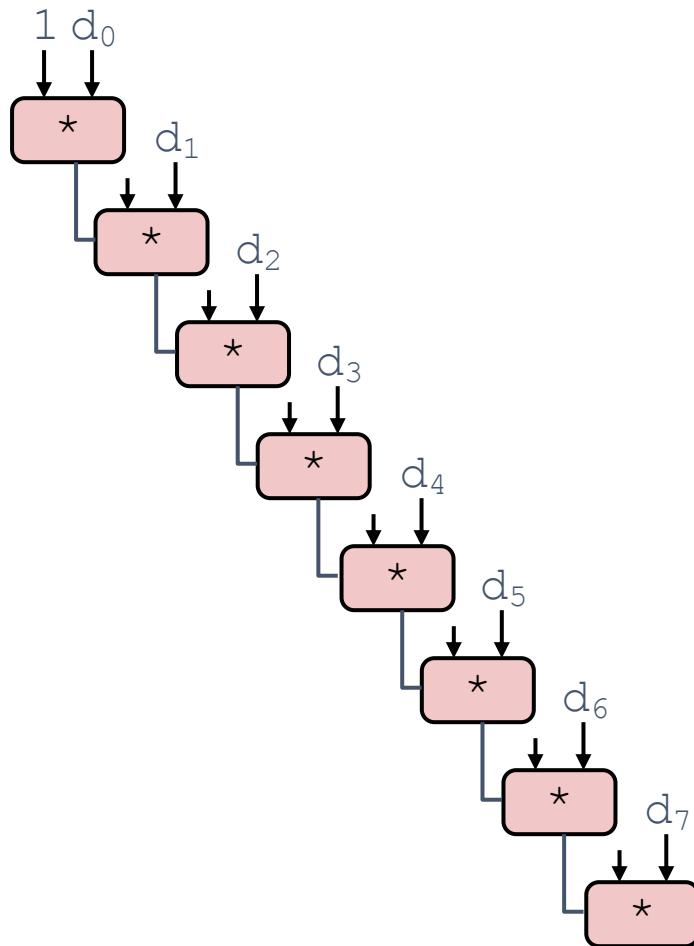
- Inner Loop (Case: Integer Multiply)

```
.L519:                                # Loop:  
    imull  (%rax,%rdx,4), %ecx  # t = t * d[i]  
    addq   $1, %rdx            # i++  
    cmpq   %rdx, %rbp          # Compare length:i  
    jg     .L519               # If >, goto Loop
```

Method	Integer		Double FP	
Operation	add	mult	add	mult
combine4	1.27	3.01	3.01	5.01
<i>Latency Bound</i>	1.00	3.00	3.00	5.00

# combine4 = Serial Computation

Assume **OP**  $\equiv$  multiplication (\*)



- Computation, assume length=8:  
$$(((((((1 * d[0]) * d[1]) * d[2]) * d[3]) * d[4]) * d[5]) * d[6]) * d[7])$$
- You have **sequential dependence**
  - Performance: determined by latency of **OP**

# First trick: Loop Unrolling (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length - 1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    // Combine 2 elements at a time
    for (i = 0; i < limit; i += 2) {
        x = (x OP d[i]) OP d[i + 1];
    }
    // Finish any remaining elements
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

# Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	add	mult	add	mult
combine4	1.27	3.01	3.01	5.01
unroll2a	1.01	3.01	3.01	5.01
<i>Latency Bound</i>	1.00	3.00	3.00	5.00

- Helps integer add
  - Achieves latency bound
- Others don't improve. *Why?*
  - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

# Loop Unrolling with Reassociation (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length - 1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2) {
        x = x OP(d[i] OP d[i + 1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

- Can this change the result of the computation?
- Yes, for FP. *Why?*

# Effect of Reassociation

Method	Integer		Double FP	
Operation	add +	mult. ×	add +	mult. ×
combine4	1.27	3.01	3.01	5.01
unroll 2x1	1.01	3.01	3.01	5.01
unroll 2x1a	1.01	1.51	1.51	2.51
<i>Latency Bound</i>	1.00	3.00	3.00	5.00
<i>Throughput Bound</i>	0.50	1.00	1.00	0.50

4 func. units for “INT +”  
2 func. units for load

2 func. units for “FP ×”  
2 func. units for load

- Nearly 2x speedup for “INT ×”, “FP +”, “FP ×”

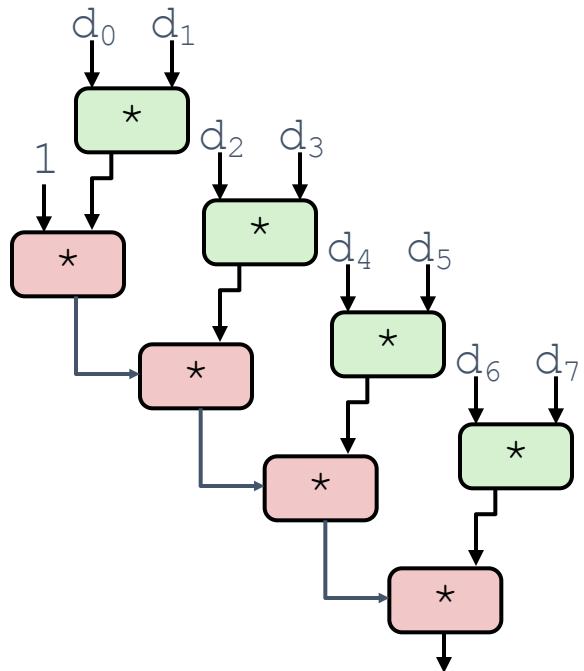
- Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

- Why is that? (next slide)

# Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```



- What changed:
  - Ops in the next iteration can be started early (no dependency)
  - Prefetching kicks in
- Overall Performance
  - N elements, D cycles latency/op
  - $(N/2+1)*D$  cycles: **CPE = D/2**

# Loop Unrolling with Separate Accumulators (2x2)

- Different form of reassociation

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length - 1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    // Combine 2 elements at a time
    for (i = 0; i < limit; i += 2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i + 1];
    }
    // Finish any remaining elements
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

# Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
<i>Latency Bound</i>	1.00	3.00	3.00	5.00
<i>Throughput Bound</i>	0.50	1.00	1.00	0.50

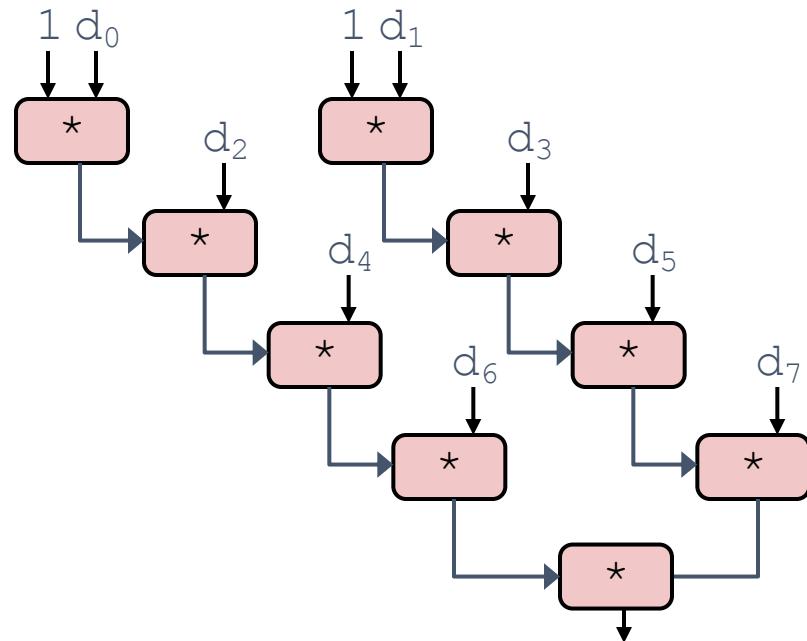
- “INT +” makes use of two load units

```
x0 = x0 OP d[i];  
x1 = x1 OP d[i+1];
```

- 2x speedup (over unroll2) for “INT \*”, “FP +”, “FP \*”

# Separate Accumulators

```
x0 = x0 OP d[i];  
x1 = x1 OP d[i+1];
```



## ■ What changed:

- Two independent “streams” of operations

## ■ Overall Performance

- N elements, D cycles latency/op
- Should be  $(N/2+1)*D$  cycles: **CPE = D/2**
- CPE matches prediction!

*What Now?*

# Unrolling & Accumulating

- Idea
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K
- Limitations
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially

# Unrolling & Accumulating: Floating Point Multiplication

- Case

- Intel Haswell
- Double FP Multiplication
- Latency bound: 5.00. Throughput bound: 0.50

FP *	Unrolling Factor L								
	K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51			2.51		
3			1.67						
4				1.25		1.26			
6					0.84			0.88	
8						0.63			
10							0.51		
12								0.52	

# Unrolling & Accumulating: Integer Addition

- Case
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 0.50

FP *	Unrolling Factor L								
	K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69			0.54		
3			0.74						
4				0.69		1.24			
6					0.56			0.56	
8						0.54			
10							0.54		
12								0.56	

# Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
<i>Latency Bound</i>	1.00	3.00	3.00	5.00
<i>Throughput Bound</i>	0.50	1.00	1.00	0.50

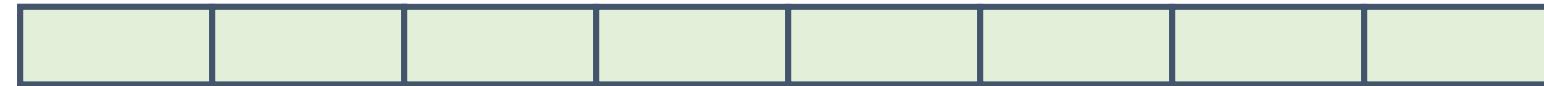
- Limited only by throughput of functional units
- Up to 42X improvement over original, un-optimized code

# Vector instructions

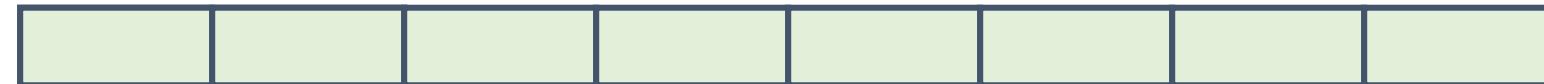
- Basic idea: Use fat registers to perform the same operation on all variables stored in register
- X86 has, for instance, 512 bit-wide registers
  - Technology called AVX512
- Since chip in this example is Haswell, it only has 256-bit registers
  - Technology called AVX2

# Programming with AVX2, YMM Registers related

- 16 total, each 32 bytes
- What can you do with them?
  - Store 8 32-bit integers:



- Store 8 single-precision floats:



- Store 4 double-precision floats:



- Store 1 single-precision float but then waste bits:



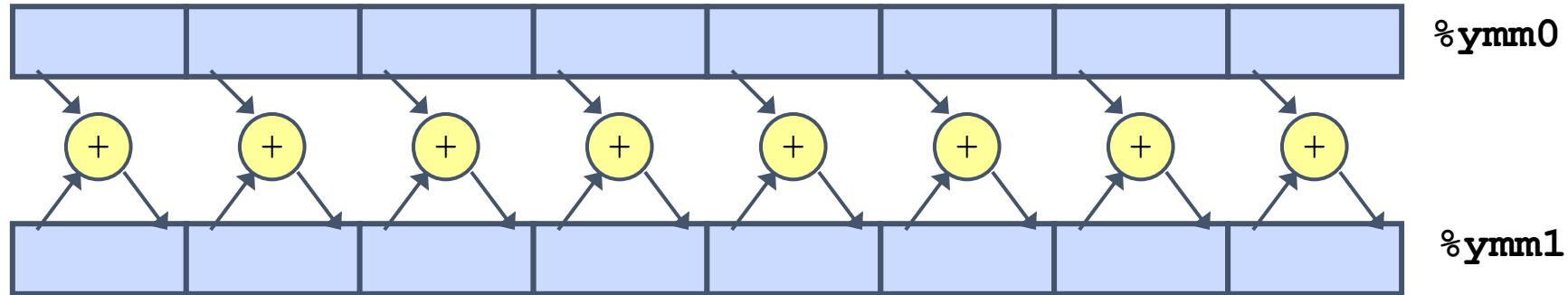
- Store 1 double-precision float but then waste bits:



# SIMD Operations, with AVX2

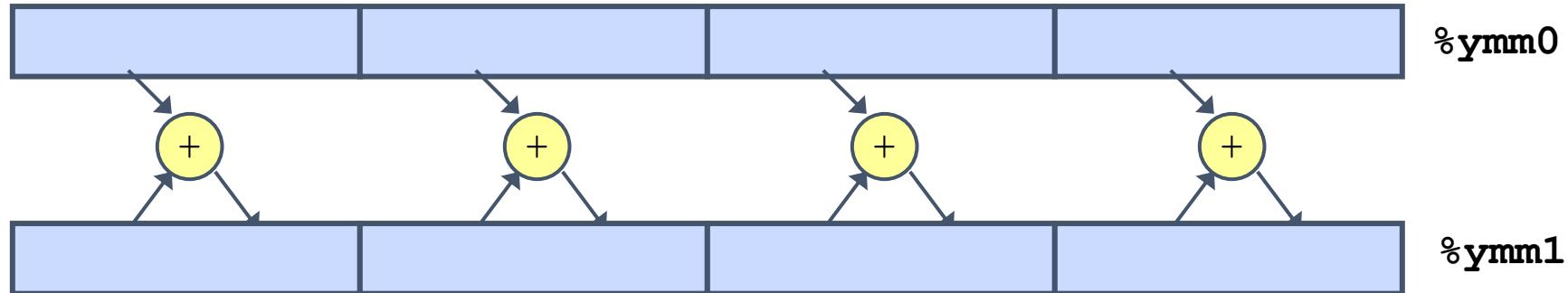
## ■ SIMD Operations: Single Precision

**vaddsd %ymm0, %ymm1, %ymm1**



## ■ SIMD Operations: Double Precision

**vaddpd %ymm0, %ymm1, %ymm1**



# Using Vector Instructions

Method	Integer		Double FP	
Operation	add	mult	add	mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
<i>Latency Bound</i>	<b>0.50</b>	<b>3.00</b>	<b>3.00</b>	<b>5.00</b>
<i>Throughput Bound</i>	<b>0.50</b>	<b>1.00</b>	<b>1.00</b>	<b>0.50</b>
<b>Vec Throughput Bound</b>	<b>0.06</b>	<b>0.12</b>	<b>0.25</b>	<b>0.12</b>



This is where  
we are now

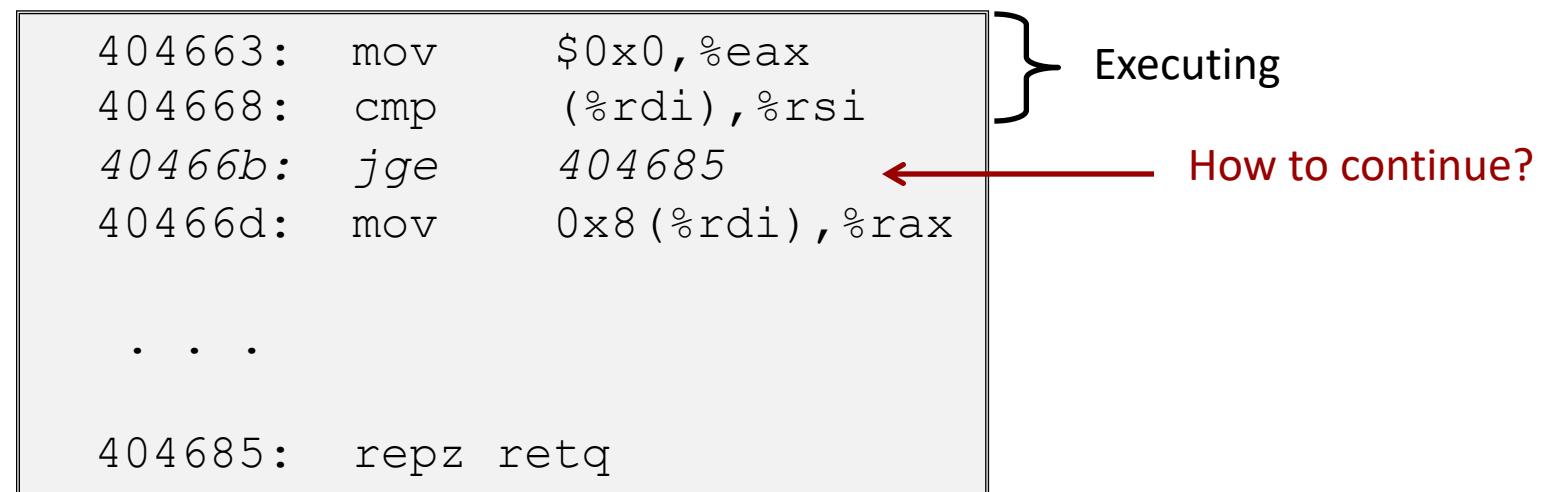
- Additional speedup owing to use of AVX Instructions
  - Parallel operations on multiple data elements
  - The “mips” in OpenMP

This is where  
we started

Method	data_t: Integer		data_t: Double FP	
Operation (OP)	add	mult	add	mult
combine1 unoptimized	22.68	20.02	19.98	20.18
combine1 with -O1	10.12	10.12	10.17	11.14
combine1 with -O3	4.5	4.5	6	7.8

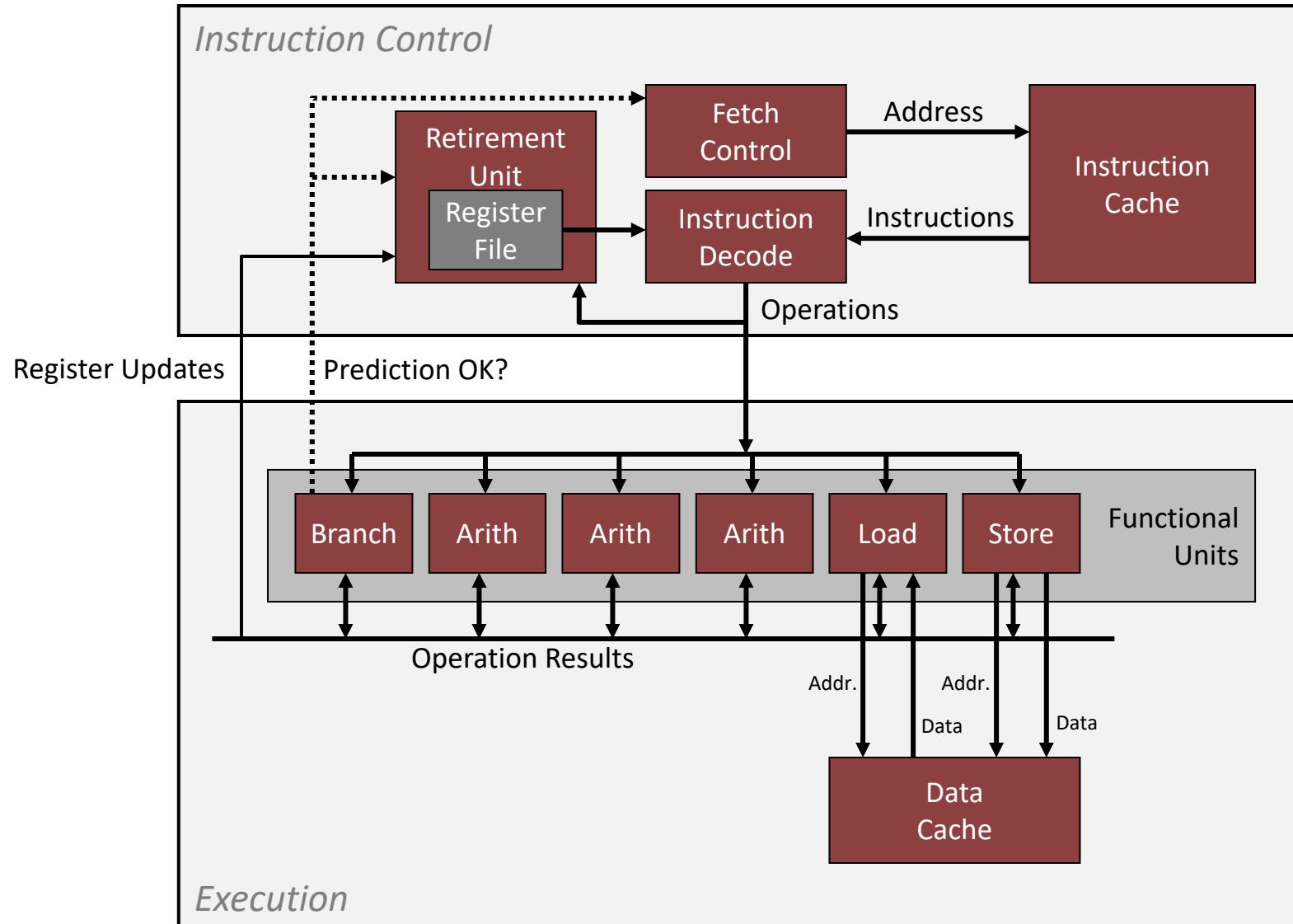
# Supplemental Material: Branch Prediction (another ILP trick)

- Challenge
  - Instruction Control Unit must work ahead of Execution Unit to generate enough operations to keep EU busy



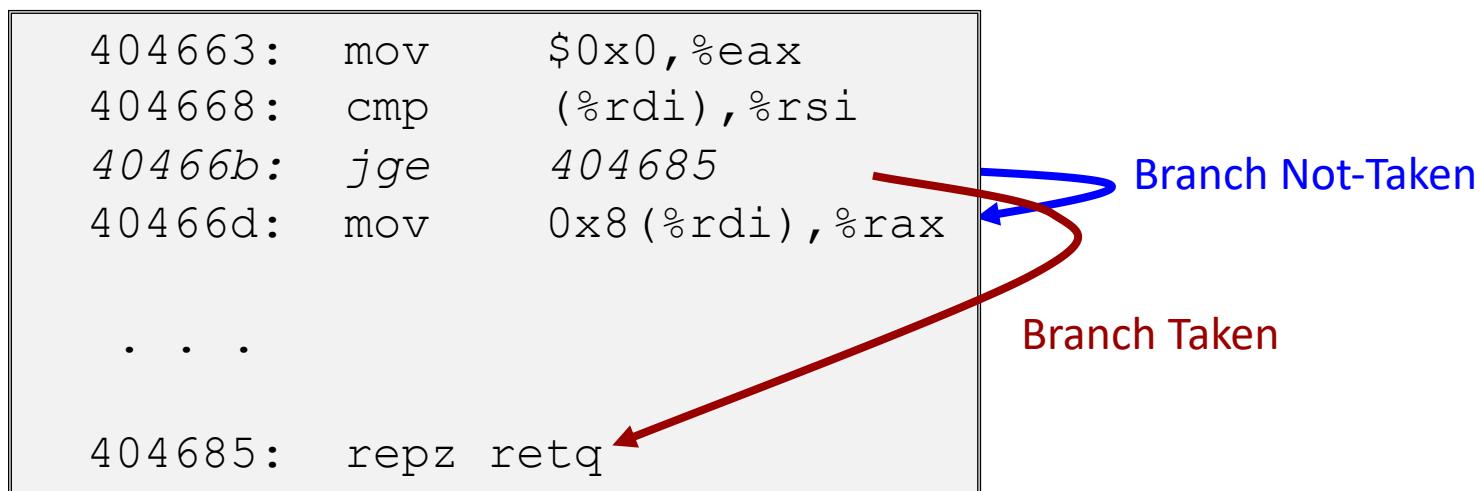
- When encounters conditional branch, cannot reliably determine where to continue fetching

# Modern CPU Design



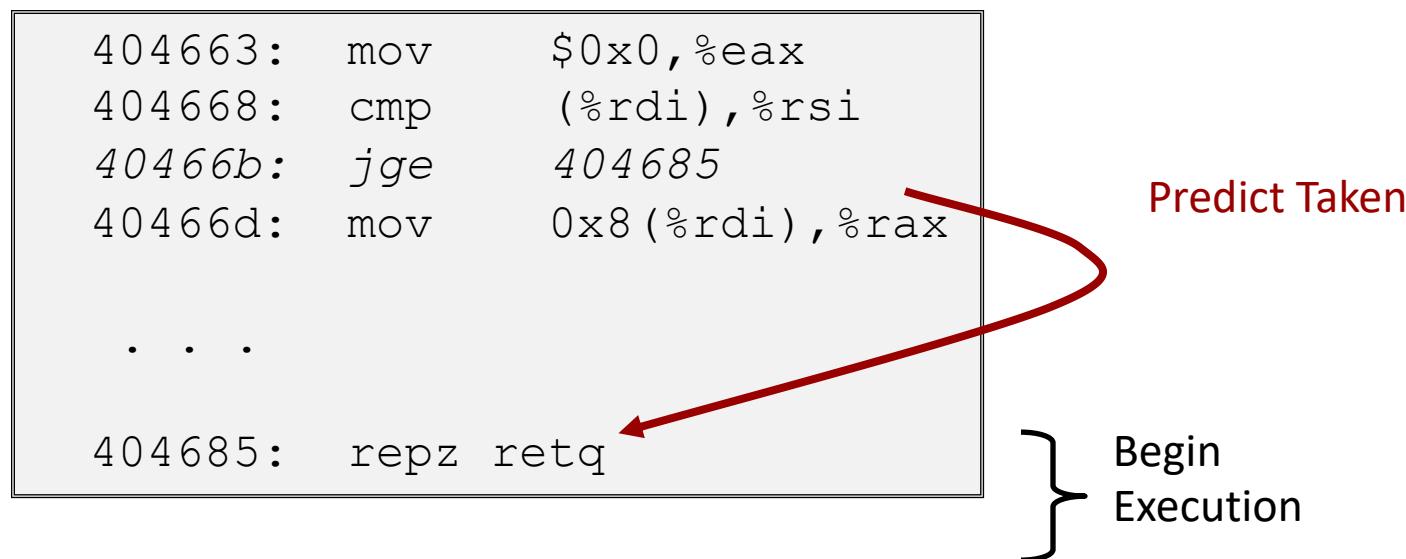
# Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit



# Branch Prediction

- Idea
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don't actually modify register or memory data (draws on RAT & ROB, see later)



# Branch Prediction Through Loop

```
401029: vmulsd (%rdx),%xmm0,%xmm0  
40102d: add    $0x8,%rdx  
401031: cmp    %rax,%rdx  
401034: jne    401029
```

*i = 98*

Assume  
vector length = 100

```
401029: vmulsd (%rdx),%xmm0,%xmm0  
40102d: add    $0x8,%rdx  
401031: cmp    %rax,%rdx  
401034: jne    401029
```

*i = 99*

Predict Taken (OK)

```
401029: vmulsd (%rdx),%xmm0,%xmm0  
40102d: add    $0x8,%rdx  
401031: cmp    %rax,%rdx  
401034: jne    401029
```

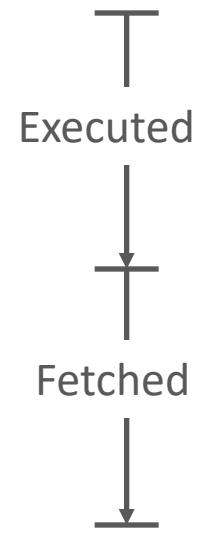
*i = 100*

Predict Taken  
(Oops)

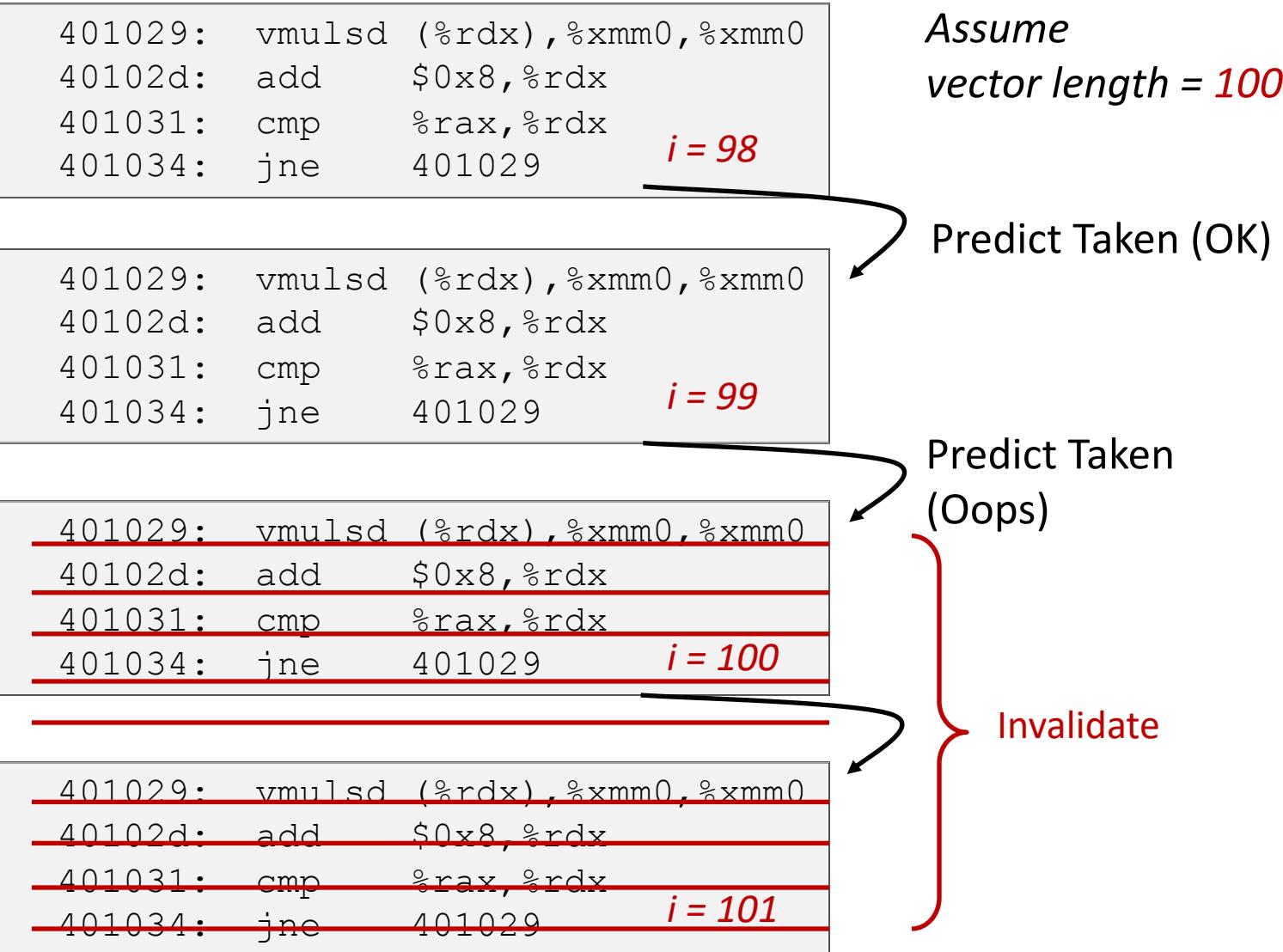
Read  
invalid  
location

```
401029: vmulsd (%rdx),%xmm0,%xmm0  
40102d: add    $0x8,%rdx  
401031: cmp    %rax,%rdx  
401034: jne    401029
```

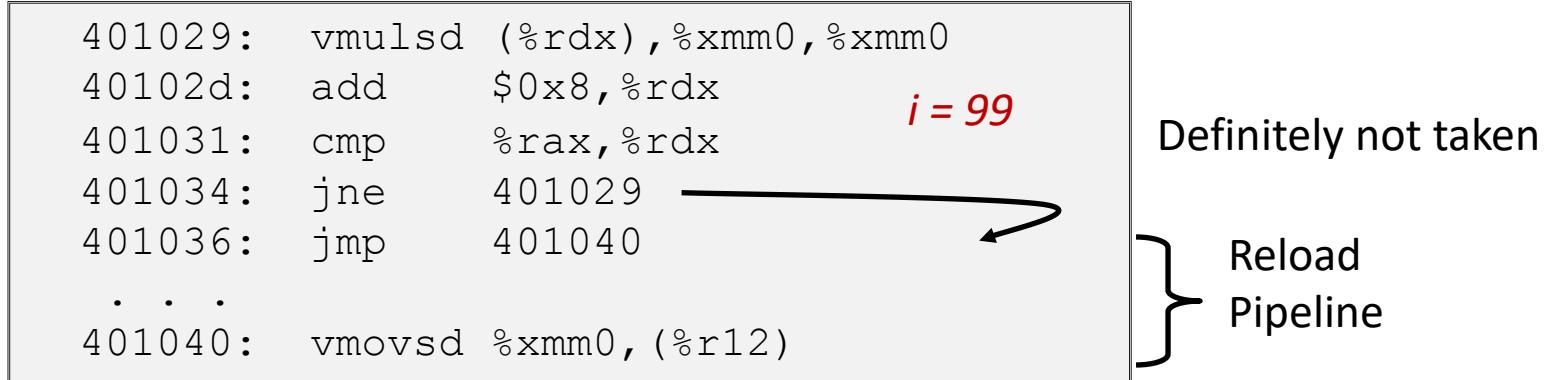
*i = 101*



# Branch Misprediction Invalidation



# Branch Misprediction Recovery



- Performance Cost
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

# Branch Prediction Numbers

- Default behavior:
  - Backwards branches are often loops so predict taken
  - Forwards branches are often if so predict not taken
- Predictors average better than 95% accuracy
  - Most branches are already predictable.
- Bonus material:
  - <http://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-an-unsorted-array>

# Short trip: actors, the “speculative execution” trick [it’s tricky]

- “instruction issue phase” vs. “instruction retirement phase”: two different animals
- ROB: reorder buffer
  - At issue time, an instruction is assigned an entry at the tail of the ROB. It’s there while in flight, before being committed
- RAT: register alias table
  - RAT needed because of “register renaming”, which is a technique that abstracts logical registers from physical registers
    - Every logical register has a set of physical registers associated with it
    - The physical registers are opaque and cannot be referenced directly but only via the canonical names
- The interplay between ROB and RAT allows an unwinding of the instructions without compromising the state of the program (since things were not committed yet)
- Resources: see [here](#) and [here](#), it takes 7 mins total.

# Looking back at examples, what we did to improve performance

- Used good compiler and the right flags
- Helped the compiler get optimization going (use “restrict” keyword on pointer arguments)
- Exploited instruction-level parallelism

# Parallel Computing with the Message Passing Interface (MPI)

# Acknowledgments

- Parts of the MPI material covered draws on slides made available by Irish Centre for High-End Computing (ICHEC)
  - The ICHEC material was based on the MPI course developed by Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart in collaboration with the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh
- Slides have been changed, which probably explains any inaccuracies in the material

# ME759: where we are right now

- Behind us: OpenMP & CUDA
  - Commodity parallel computing (eight cores, or one GPU, etc.)
- Next segment: the non-commodity parallel computing
  - Reliance on supercomputers
- We will focus on the Message Passing Interface (MPI) standard
  - MPI: The software environment that makes supercomputers tick
  - MPI is for supercomputers what CUDA is for the GPU and OpenMP for the multi-core chip

# Opportunities for Efficiency Gains

Cluster (distributed mem.)	Group of nodes communicate through fast interconnect	MPI, Charm++, Chapel
Multi-Socket Node	Group of CPUs on the same node, talk through main mem.	OpenMP, MPI
Acceleration (GPU/Phi)	Compute devices accelerating parallel computation on one node	CUDA, OpenCL, OpenMP
Multi-Core Processor	Communication through shared caches and main mem.	OpenMP, TBB, pthreads
Vectorization	Higher operation throughput via special/fat registers	AVX, SSE
Pipelining	Sequence of instruction sharing functional units	Assembly
Superscalar	Non-sequence instructions sharing functional units	Assembly

- Any ILP issues that we can address?
- Hitting the cache?
- Any opportunities for vectorization?
- Do the threads work together harmoniously?
- Any NUMA issues to be aware of?
- Communication/synchronization reduced as much as possible?

We have full control → 

We have little to no control → 

What is left at this point → 

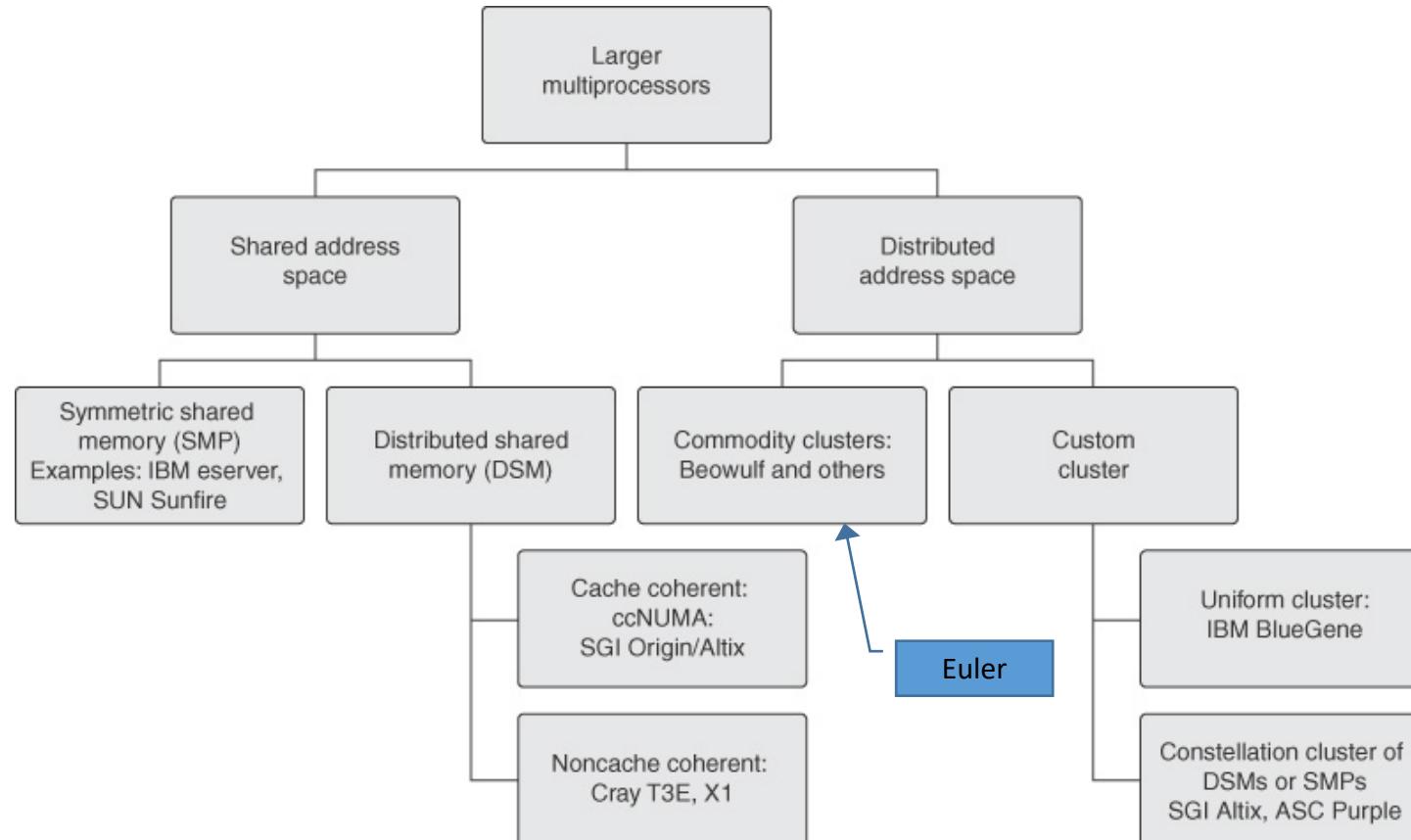
# Supercomputers, example



# Supercomputers, examples (\$20 million and such)



# Overview of Large Multiprocessor Hardware Configurations (“supercomputers”)



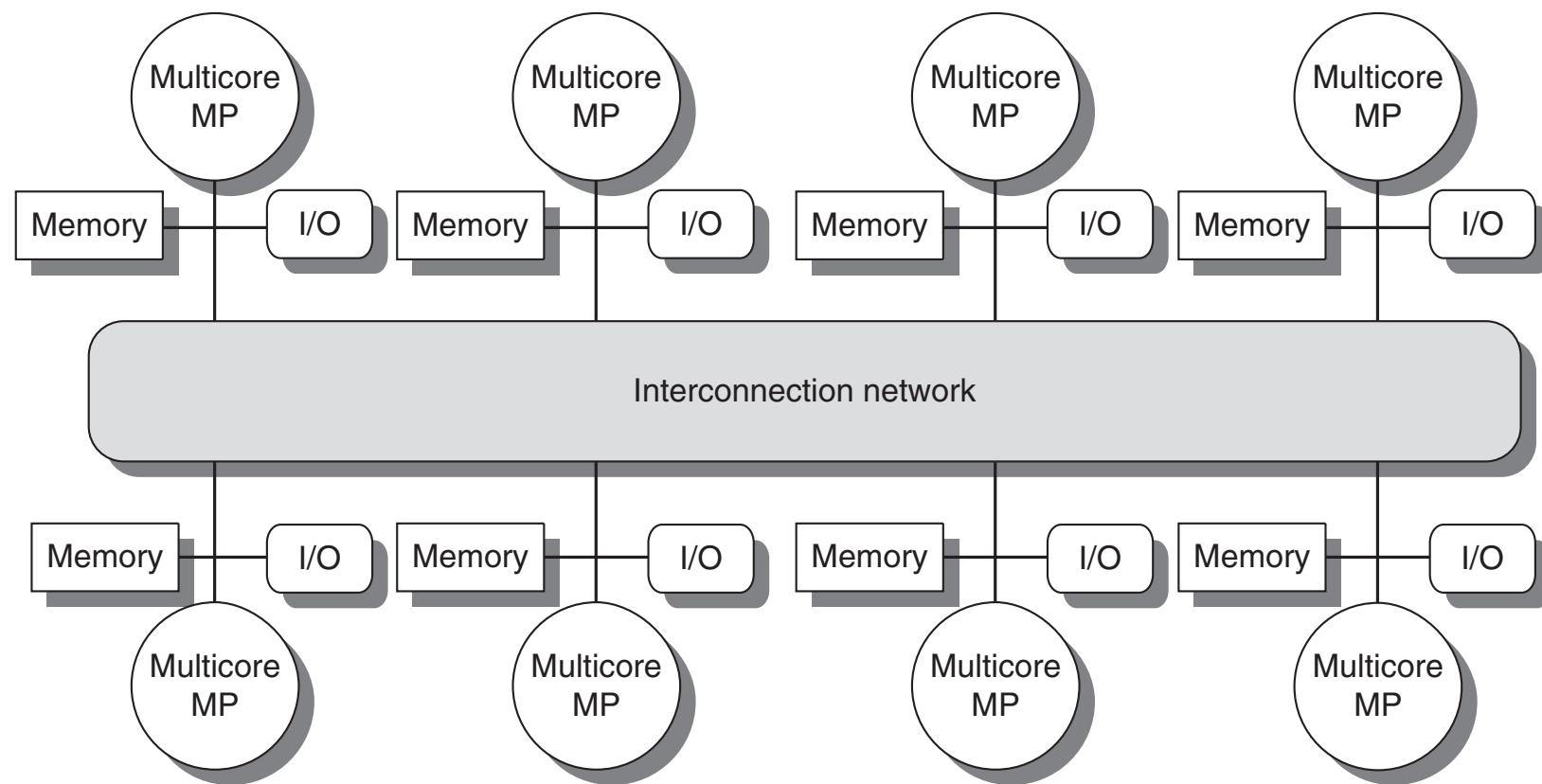
© 2007 Elsevier, Inc. All rights reserved.

# Nomenclature Issues

- Shared address space: when you invoke address “0x0043fc6f” on one machine and then invoke “0x0043fc6f” on a different machine they actually point to the same global memory space
  - Issues: memory coherence
    - Fix: software-based or hardware-based
- Distributed address space: the opposite of the above
- Symmetric Multiprocessor (SMP): you have one machine that shares amongst all its processing units a certain amount of memory (same address space)
  - Mechanisms should be in place to prevent data hazards (**RAW**, **WAR**, **WAW**). Raises the issue of memory coherence
- Distributed shared memory (DSM) – aka distributed global address space (DGAS)
  - Although physically memory is distributed, it shows as one uniform memory
  - Memory latency is highly unpredictable

# Example

- Distributed-memory multiprocessor (MP) architecture
  - Euler, for instance



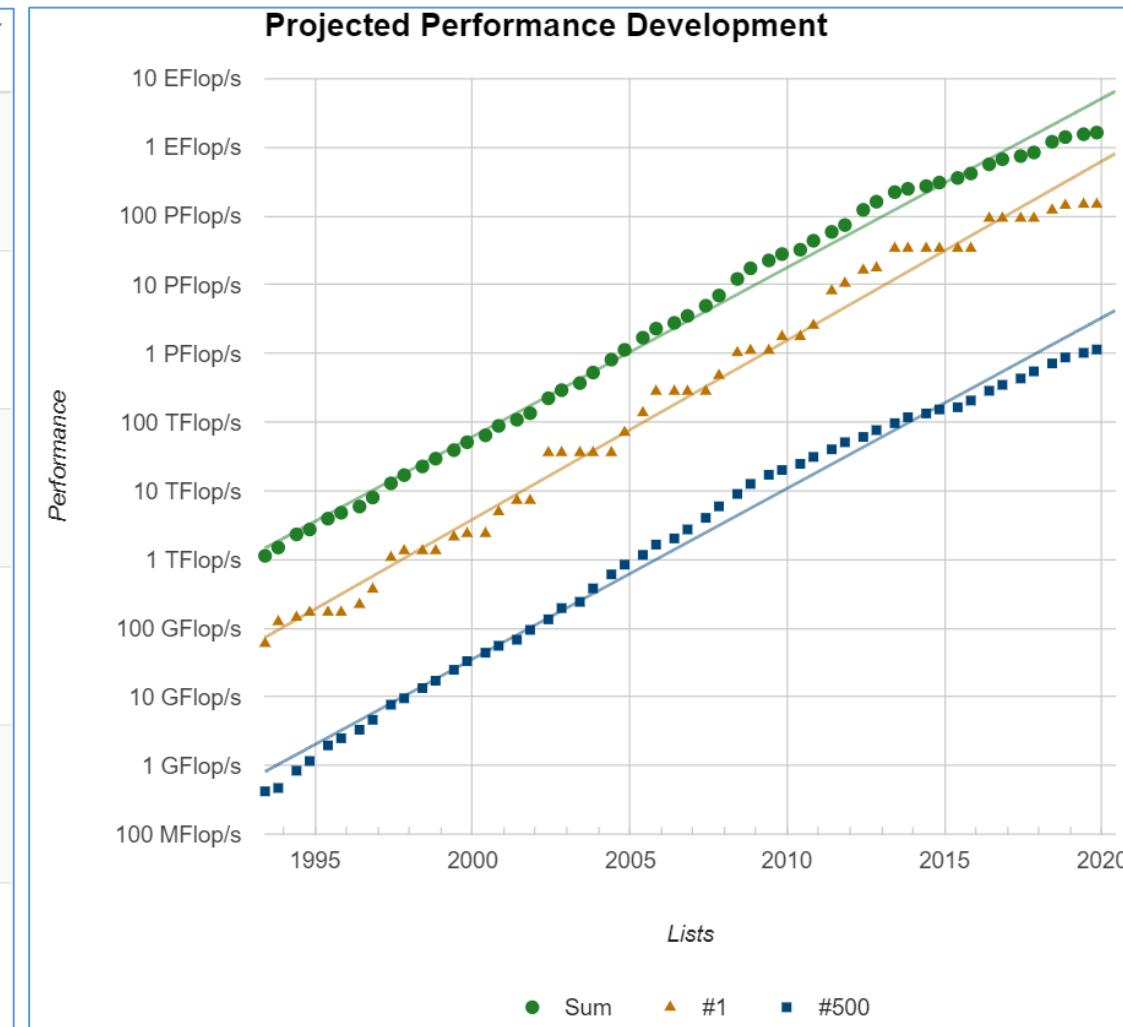
# Comments, distributed-memory multiprocessor architecture

- Basic architecture consists of nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all nodes
- Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is less scalable than the global interconnection network
- Popular interconnection network: Mellanox and Qlogic InfiniBand
  - Bandwidth range: 1 through 200 Gb/sec (about 25 GB/s)
  - Latency: in the microsecond range ( $\approx 1E-6$  to  $1E-7$  second); i.e., relatively high
  - Requires special network cards: HCA – “Host Channel Adaptor”

# Supercomputers: Where Are We Today?

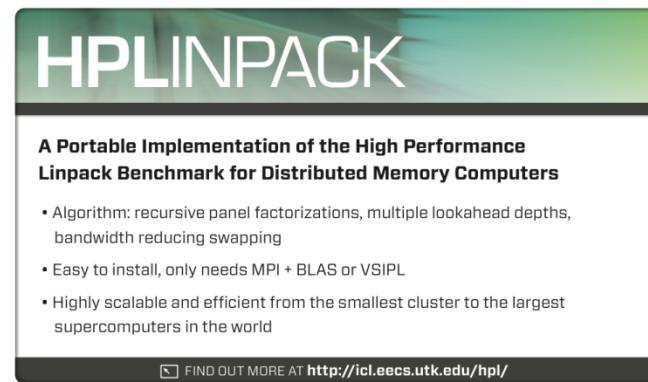
[Info lifted from Top500 website: <http://www.top500.org/>]

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	<b>Summit</b> - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
2	<b>Sierra</b> - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	<b>Sunway TaihuLight</b> - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
4	<b>Tianhe-2A</b> - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	<b>Frontera</b> - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR , Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
6	<b>Piz Daint</b> - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384



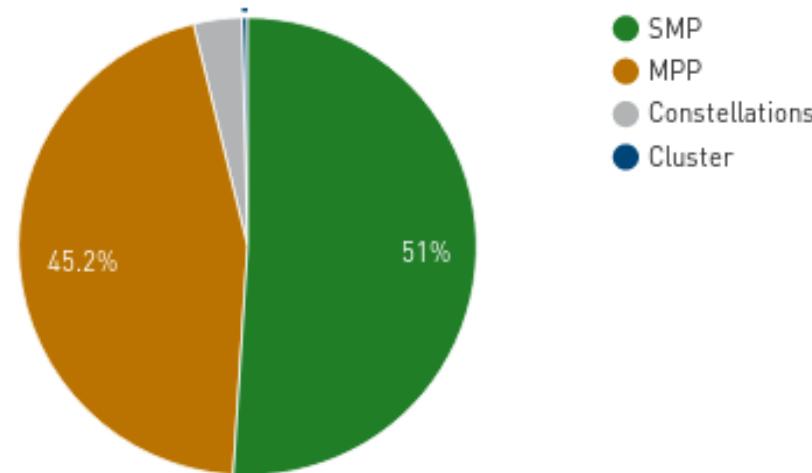
# How is a supercomputer's speed measured?

- How is the speed measured to put together the Top500?
  - Basically reports how fast you can solve a dense linear system



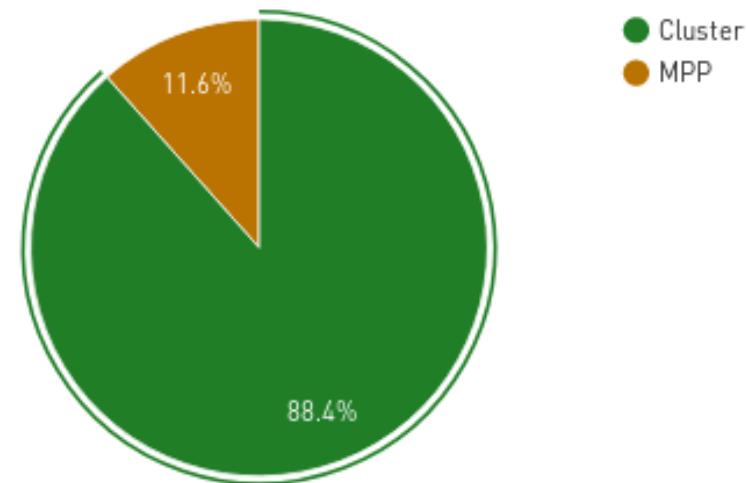
# Supercomputers: Where Are We Today? [Cntd.]

Architecture System Share



1998

Architecture System Share

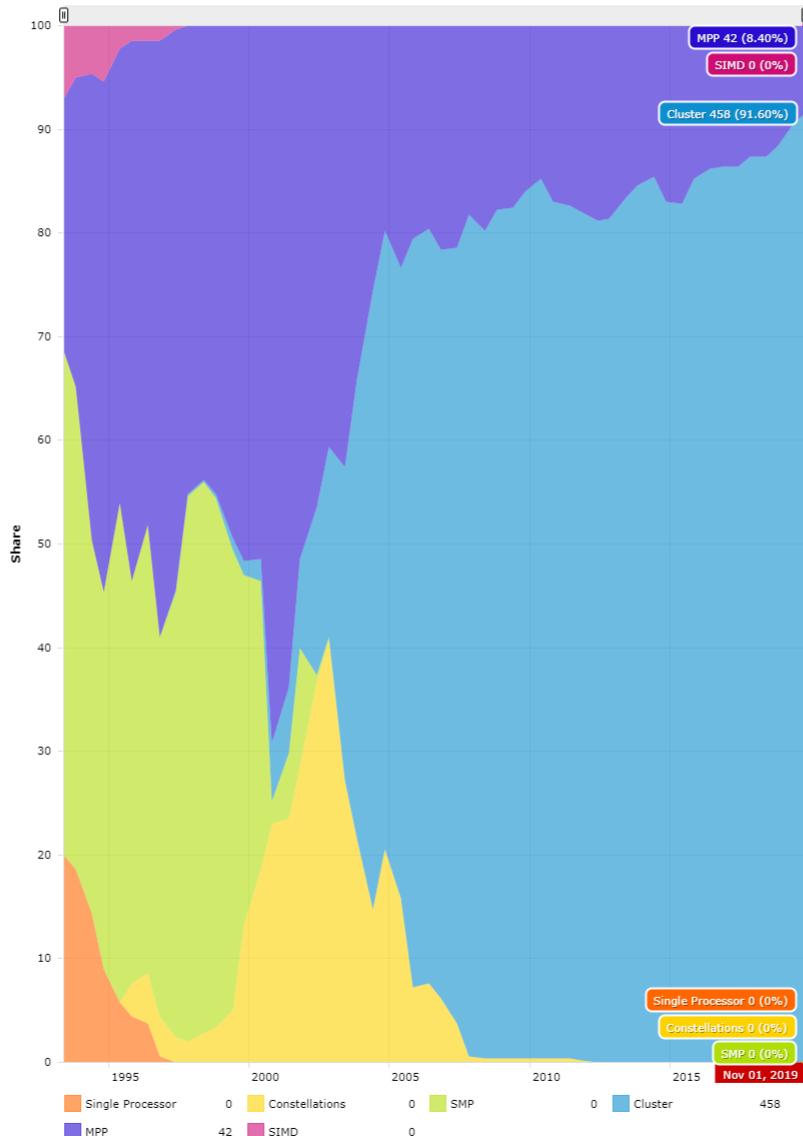


2018

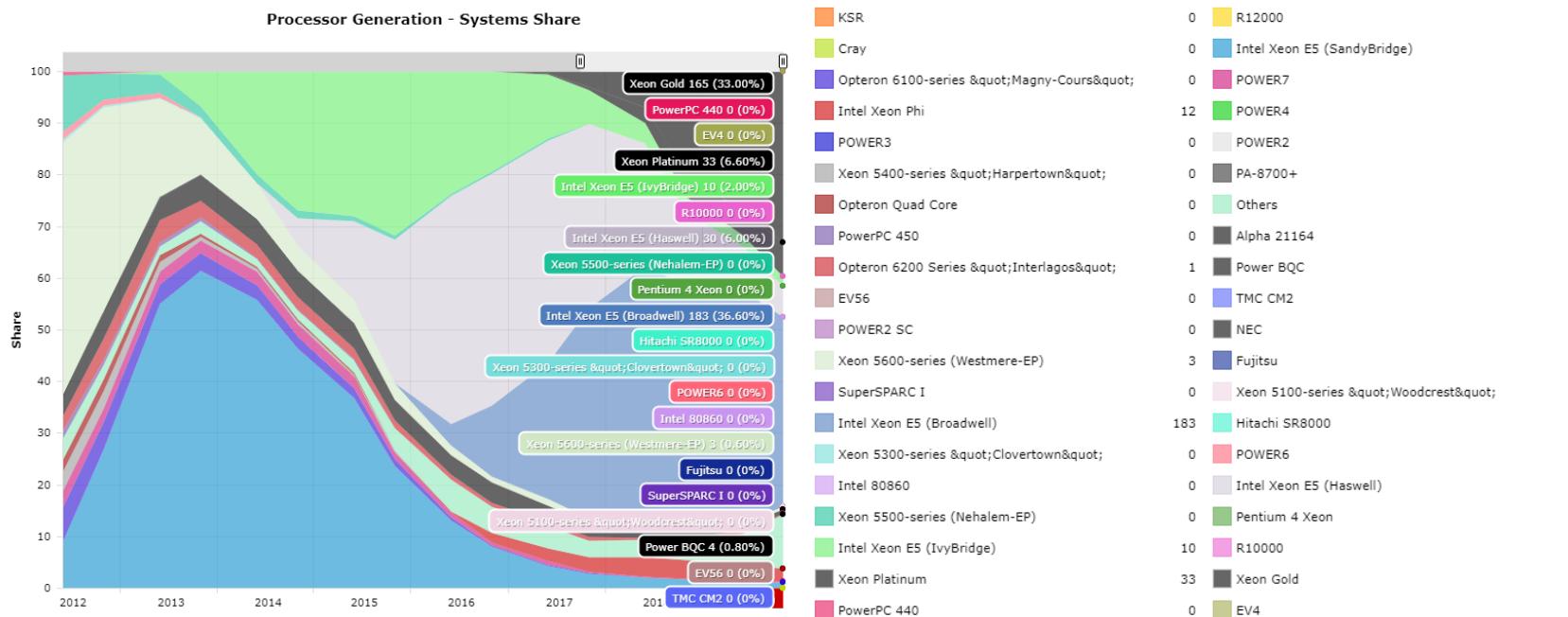
- Abbreviations/Nomenclature
  - MPP – Massively Parallel Processing
  - Constellation – subclass of cluster architecture envisioned to capitalize on data locality

# Supercomputers: Where Are We Today? [Cntd.]

Architecture - Systems Share



Processor Generation - Systems Share



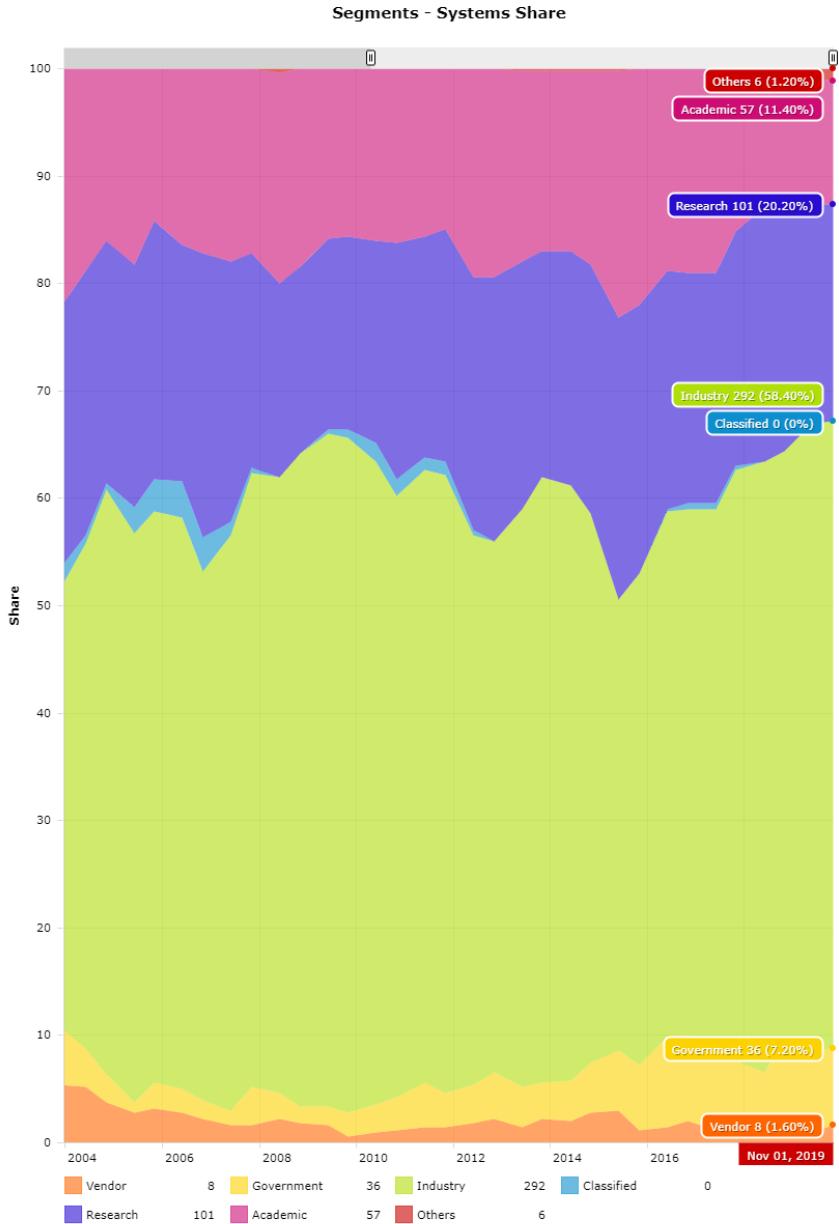
## • Abbreviations/Nomenclature

- MPP – Massively Parallel Processing
- Constellation – subclass of cluster architecture envisioned to capitalize on data locality
- MIPS – “Microprocessor without Interlocked Pipeline Stages”, a chip design of the MIPS Computer Systems of Sunnyvale, California
- SPARC – “Scalable Processor Architecture” is a RISC instruction set architecture developed by Sun Microsystems (now Oracle) and introduced in mid-1987
- Alpha - a 64-bit reduced instruction set computer (RISC) instruction set architecture developed by DEC (Digital Equipment Corporation was sold to Compaq, which was sold to HP) – adopted by Chinese chip manufacturer (see primer)

# Nomenclature: What is an MPP? How's it different from a cluster?

- MPP: A very large-scale computing system with commodity processing nodes interconnected with a **custom-made** high-bandwidth low-latency interconnect
  - A cluster is an MPP for which the manufacturer forgot to put a high-bandwidth low-latency interconnect
- Memories are physically distributed
- Nodes often run a microkernel
  - Perhaps this is a second difference, a cluster might run some rather pedestrian Linux distro
- Rather blurred line between MPPs and clusters
- Example:
  - Euler (our machine) is a cluster
  - An IBM BG/Q machine is an MPP (because of the interconnect)

# Supercomputers: Where Are We Today? [Cntd.]



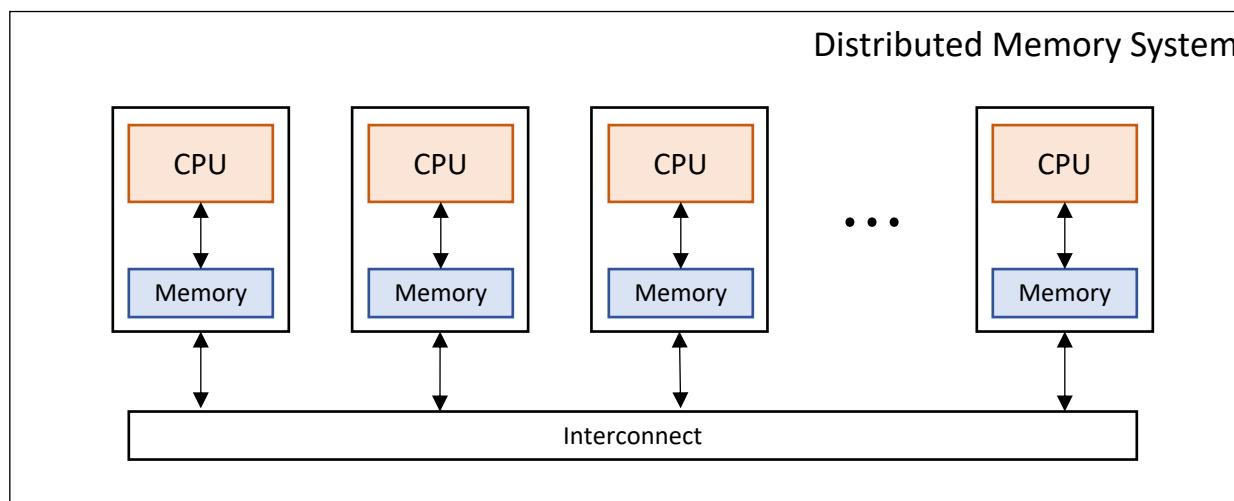
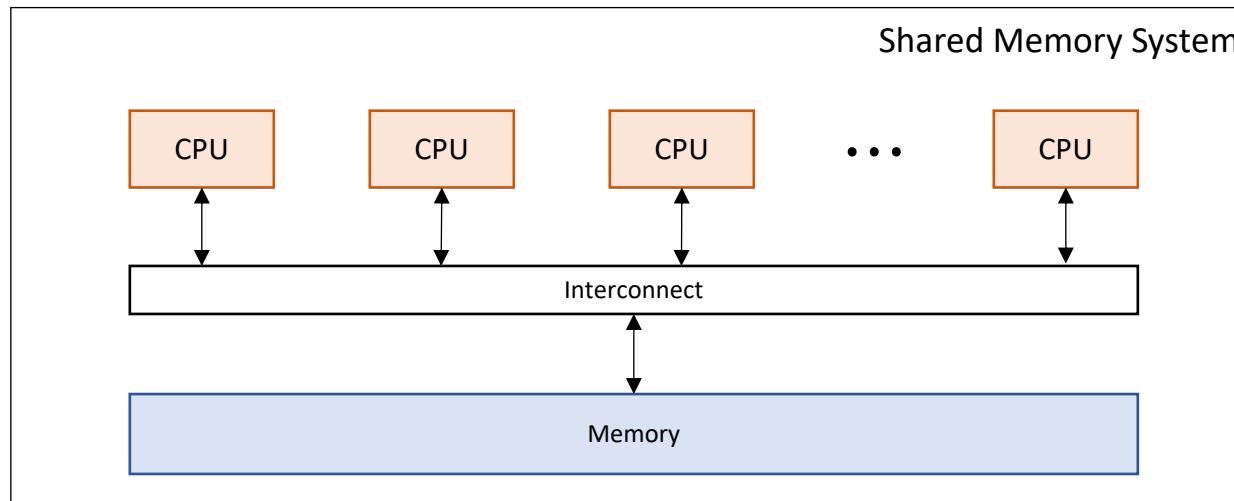
## Top 500: What accelerators do they use?

NVIDIA Tesla P100	29	PEZY-SC2 500Mhz	0
Intel Xeon Phi SE10P	0	Intel Xeon Phi 5120D	2
PEZY-SC2 700Mhz	1	Intel Xeon Phi 31S1P	1
NVIDIA Tesla V100	79	Intel Xeon Phi SE10X	0
Matrix-2000	1	Intel Xeon Phi	0
NVIDIA 2050	2	NVIDIA 2090	0
Intel MIC	0	IBM PowerXCell 8i	0
ATI GPU	0	NVIDIA Tesla GP100	0
Intel Xeon Phi 7120P	1	NVIDIA Tesla P100 NVLink	1
NVIDIA Tesla K40m	1	Intel Xeon Phi 7120X	0
Intel Xeon Phi 7110P	0	NVIDIA Tesla K20	0
None	355	NVIDIA K20/K20x, Xeon Phi 5110P	0
Deep Computing Processor	1	NVIDIA Tesla P4	0
NVIDIA Tesla K40	3	AMD FirePro S9150	0
NVIDIA Volta GV100	4	AMD FirePro S10000	0
NVIDIA Tesla K20m	1	Nvidia Titan Black	0
NVIDIA 2075	0	NVIDIA Tesla V100 SXM2	11

# Shared Memory Hardware Systems

- Memory is shared; i.e., visible to all processors and threads
  - Typical scenario, on a budget: one node with four CPUs, each with 16 cores
  - Typically one node systems; i.e., a workstation or a laptop
- Example:
  - Symmetric Multi-Processors (SMP) – like one node of Euler
    - Each processor has equal access to RAM
- Traditionally, this represents the hardware setup that supports OpenMP-enabled parallel computing
- Two issues, with SMP-leveraging OpenMP:
  - Challenging applications might **need more memory** than available on the typical multi-core node
  - Solution **scales poorly** with system size due to cache coherence aspects

# SMS vs. Distributed Memory Systems (DMS)

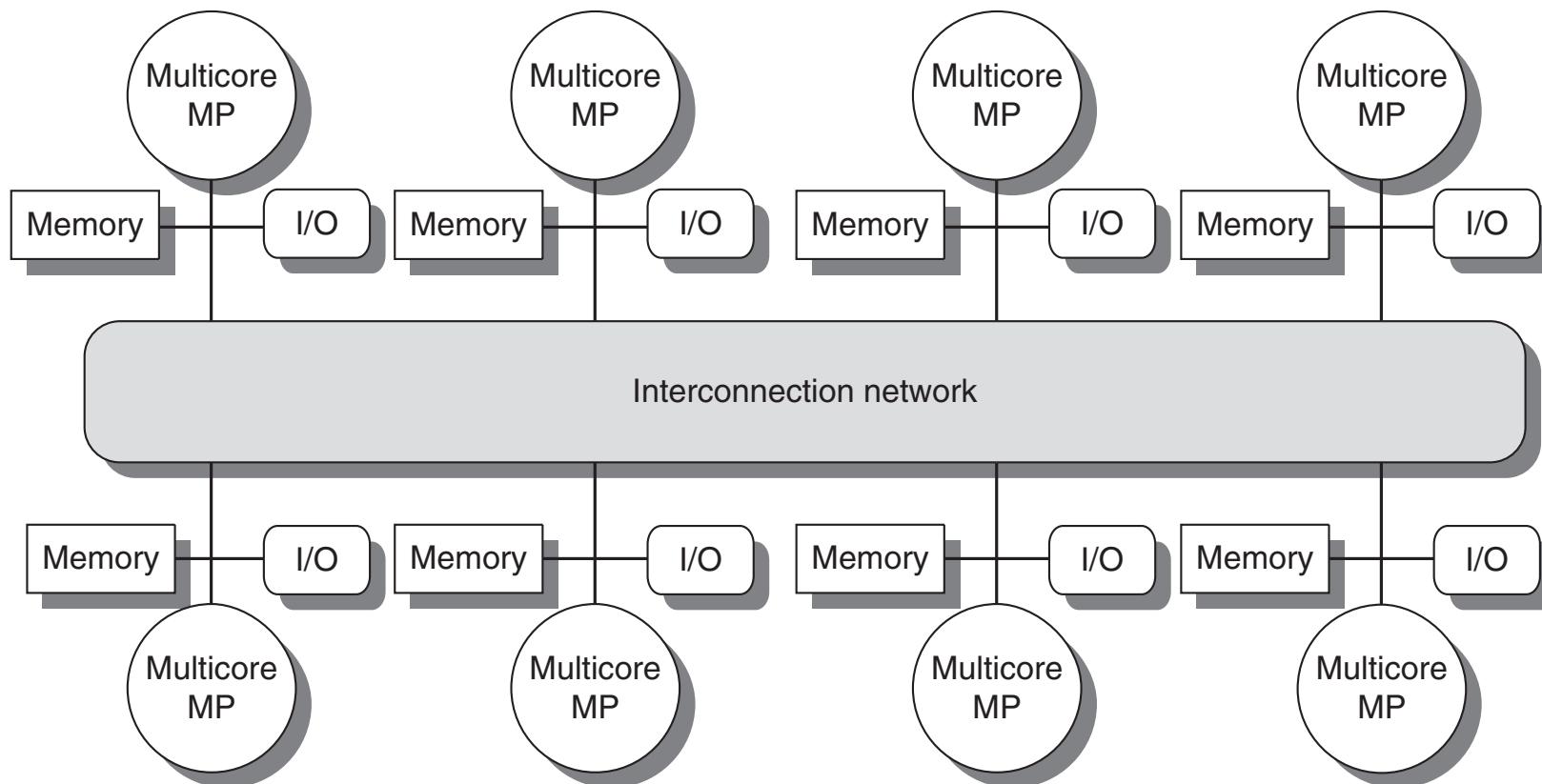


# Distributed Memory Hardware Systems: The Virtual Memory Angle

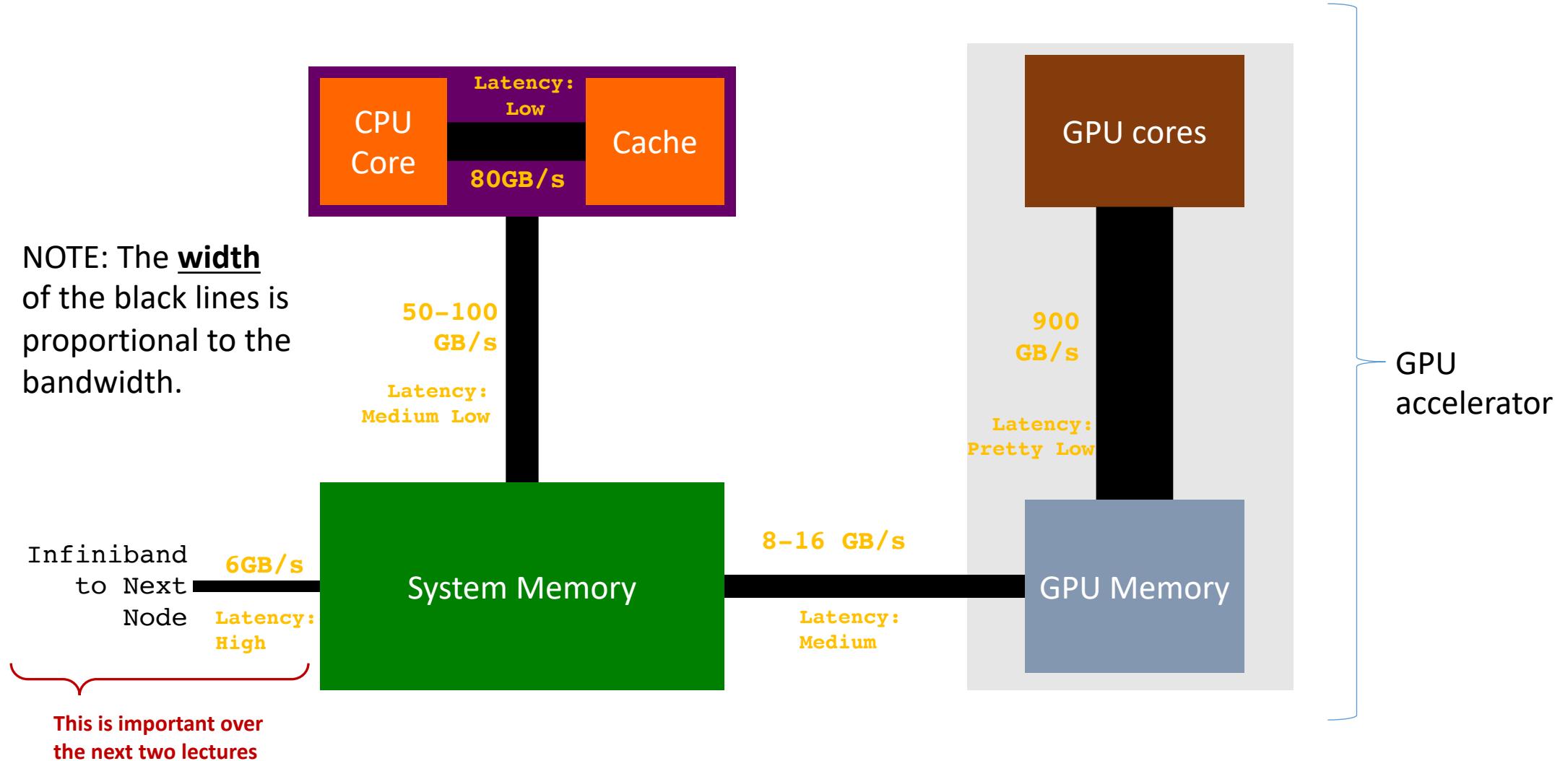
- Typically, parallelism on supercomputers accomplished by interplay between different *\*processes\** and not threads
- Consequence: each process sees its own virtual memory, there's no overlap
- More difficult to write programs for distributed memory systems since the programmer must keep track of memory usage

# Schematic, Typical Example

- Distributed-memory multiprocessor architecture (Euler, for instance)



# Bandwidths in a multi-node, CPU-GPU System



# A “High Performance Computing” Digression

- High Performance Computing, to an expert, has a very precise meaning: supercomputing
  - HPC entails the idea of a bunch of nodes linked through fast interconnect trying to solve **together** one big problem
- This class “759: High Performance Computing for Applications in Engineering”
  - High performance like in advanced, very efficient
- The terms in **blue** and **red**: same syntax, different semantics
  - A **way of computing** vs. an **attribute** of material covered in this class

# High Performance Computing (HPC) vs. High Throughput Computing (HTC)

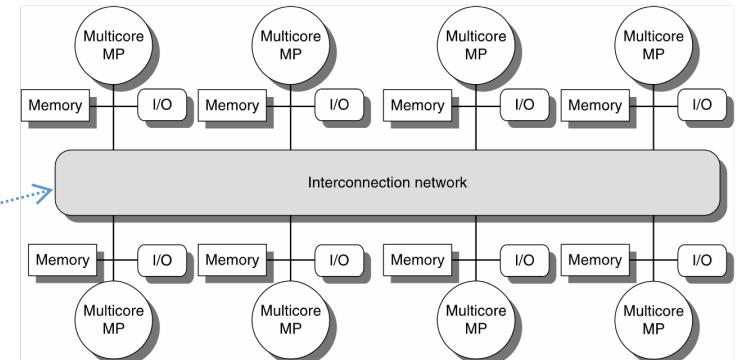
- High Performance Computing
  - The idea: run **one** executable as fast as you can
    - Might spend one month running one DFT job or a week on a CFD job...
- High Throughput Computing
  - The idea: run **many** executables at the same time on different machines
  - Example: bone analysis in ABAQUS
    - You have uncertainty in the length of the bone (20 possible lengths) in the material of the bone (10 values for Young's modulus) in the loading of the bone (50 force values with different magnitude/direction) . Grand total: 10,000 ABAQUS runs
    - We have 1400 workstations hooked up together on-campus -> use Condor to schedule the 10,000 independent ABAQUS jobs and have them run on scattered machines overnight

# High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- High Performance Computing

- Usually one cluster (e.g. Euler) or one massively parallel machine (e.g. IBM Blue Gene or Cray) that is dedicated to running one large application that requires **a lot of memory, a lot of compute power, and a lot of communication**
  - Example: each particle in a MD simulation requires (due to long range electrostatic interaction) to keep track of a large number of particles that it interacts with. Needs to query and figure out where these other particles are at any time step of the numerical integration
- What is crucial is the interconnect between the processing units
  - Typically some fast dedicated interconnect (e.g. InfiniBand), which operates at 40 GB/s
    - Euclid@UW-Madison: 1 GB/s Ethernet, Bluewaters@UIUC: 100 GB/s, Tianhe-I claims double the speed of Infiniband
- Typically **uniform** hardware components: e.g. 100,000 Intel Xeon 5520, etc.
- Expensive

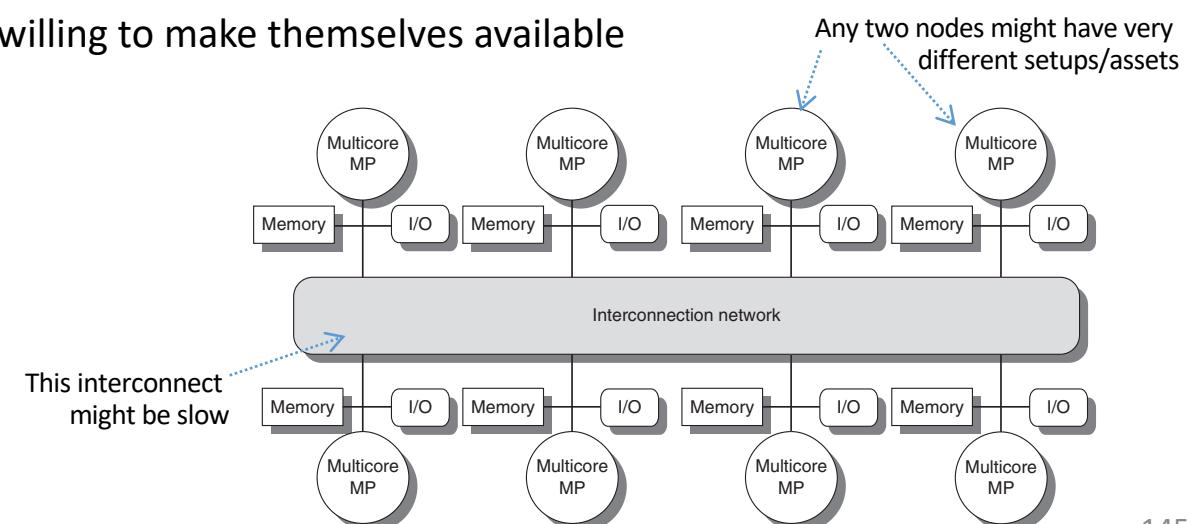
HPC: this interconnect  
expected to be fast...



# High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- **High Throughput Computing**

- Usually a collection of heterogeneous compute resources linked through a slow connection, most likely Ethernet
  - Example: 120 Windows workstations in the CAE labs (all sorts of machines, some new, some old)
- When CAE machine 58 runs an ABAQUS bone simulation there is no communication needed with CAE machine 83 that runs a different ABAQUS scenario
- Don't need to spend any money, you can piggyback on resources willing to make themselves available
- Very effective to run Design of Experiments type analyses



# High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- You can do HPC on a configuration that has slow interconnect
  - It will run very, very slow...
- Conversely, you can do HTC on an IBM Blue Gene
  - You will use the processors but will waste the fast interconnect that made the machine expensive in the first place
  - You need to have the right licensing system in place to “check out” 10,000 ABAQUS licenses
- University of Wisconsin-Madison well known due to pioneering work in the area of HTC
  - Professor Miron Livny in CS
  - UW-Madison solution for HTC: Condor, used by a broad spectrum of organizations from academia and industry

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- HPC w/ MPI: Closing Remarks

# MPI: High-level overview

- N instances of the same program are launched for execution independently as N distinct processes
- [Flynn Taxonomy](#) classification: MIMD
  - MPI & Distributed Memory: enables a MIMD computing mechanism
    - Multiple Instruction and Multiple Data
- Another way to look at it: [Single Program Multiple Data \(SPMD\)](#)

# Parallel Computing w/ MPI: One way to look at it

- CUDA: a small snippet of code (the “kernel”) run by all threads spawned via your exec. config.
- OpenMP: all threads execute an `omp parallel` region; name of the game is work sharing
- MPI: the **\*entire\*** code is executed in parallel by all processes

# A First MPI Program

```
#include "mpi.h"
#include <iostream>

int main(int argc, char **argv) {
    int my_rank, n;
    char hostname[128];

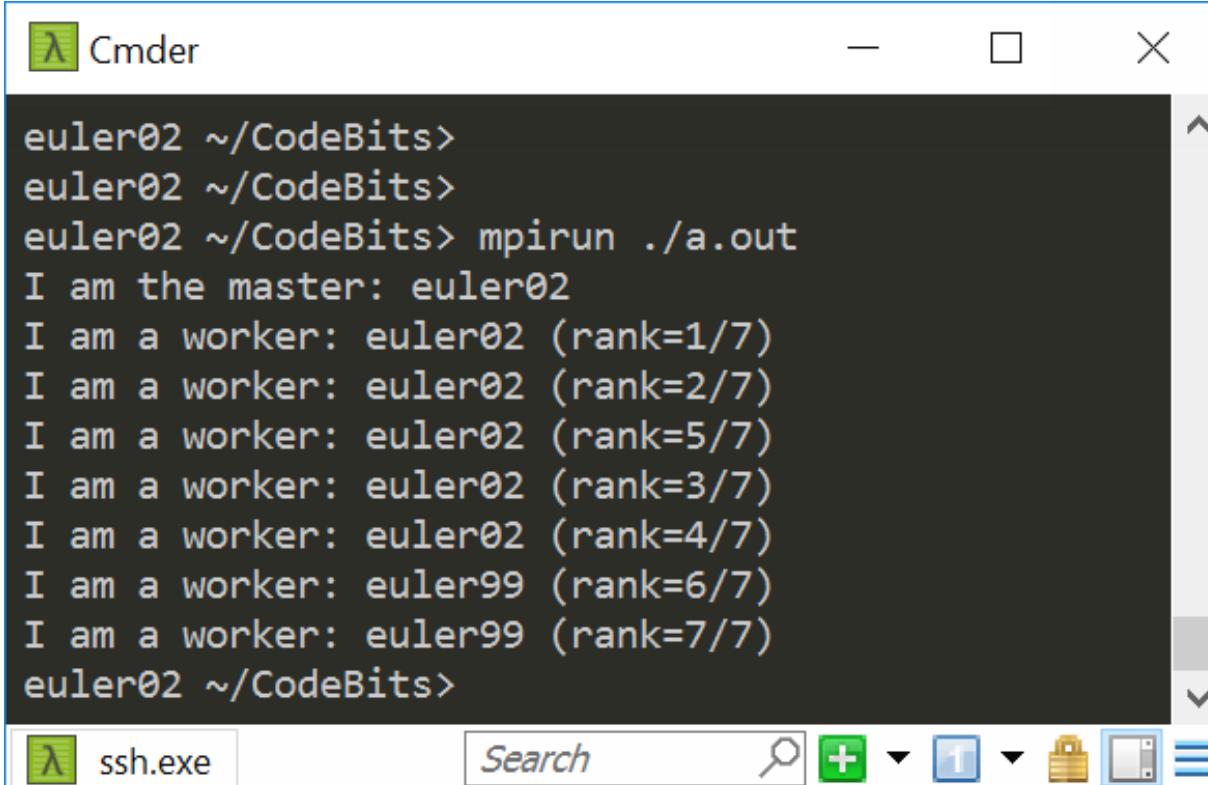
    MPI_Init(&argc,&argv); Has to be called first, and once only
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n);

    gethostname(hostname, 128);
    if (my_rank == 0) { /* master */
        printf("I am the master: %s\n", hostname);
    }
    else { /* worker */
        printf("I am a worker: %s (rank=%d/%d)\n", hostname, my_rank, n-1);
    }

    MPI_Finalize(); Has to be called last, and once only
    return 0;
}
```

# Program Output

- I asked for 8 ranks
- Got my 8 ranks spread out over two nodes: euler02 and euler99
- Please use Slurm to run your jobs; instructions in the assignment. Colin can assist if you have questions



The screenshot shows a Cmder terminal window with the title bar "Cmder". The terminal content displays the following text:

```
euler02 ~/CodeBits>
euler02 ~/CodeBits>
euler02 ~/CodeBits> mpirun ./a.out
I am the master: euler02
I am a worker: euler02 (rank=1/7)
I am a worker: euler02 (rank=2/7)
I am a worker: euler02 (rank=5/7)
I am a worker: euler02 (rank=3/7)
I am a worker: euler02 (rank=4/7)
I am a worker: euler99 (rank=6/7)
I am a worker: euler99 (rank=7/7)
euler02 ~/CodeBits>
```

The terminal interface includes a search bar labeled "Search" and various icons for file operations at the bottom.

# MPI: High-level Overview

- Each MPI process, which runs most often on a core, executes the **same** executable at roughly the same time
  - Synchronization calls can be invoked; useful to coordinate the execution of the MPI job
  - Unlike in CUDA, here synchronization can be **global**

# MPI: High-level Overview

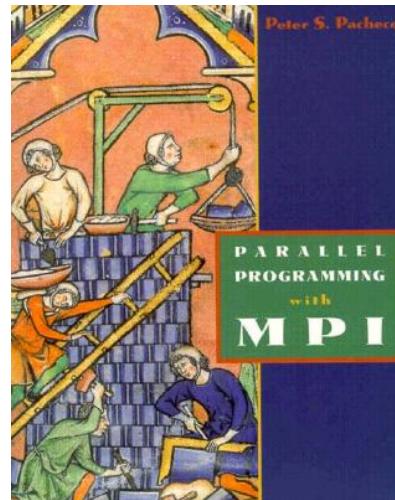
- What differentiates processes is their **rank**: processes with different ranks do different things (“branching based on the process rank”)
  - Very similar to GPU computing, where one thread did work based on its thread index
  - Very similar to OpenMP function `omp_get_thread_num()`

# The Message-Passing Model

- Each process has its own program counter and address space
- Message passing enables communication among processes that have separate address spaces
- Inter-process communication typically consists of
  - Synchronization, followed by...
  - ... movement of data from one process's address space to another process's address space

# MPI: A Second Example Application

Example from Peter Pacheco's book:  
“Parallel Programming with MPI”



```
/* greetings.c -- greetings program
 *
 * Send a message from all processes with rank != 0 to process 0.
 *   Process 0 prints the messages received.
 *
 * Input: none.
 * Output: contents of messages received by process 0.
 *
 * See Chapter 3, pp. 41 & ff in PPMPI.
 */
```

# MPI: A Second Example Application

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int      my_rank;      /* rank of process */
    int      p;            /* number of processes */
    int      source;       /* rank of sender */
    int      dest;         /* rank of receiver */
    int      tag = 0;       /* tag for messages */
    char    message[100];  /* storage for message */
    MPI_Status status;     /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
```

Type of each sent item

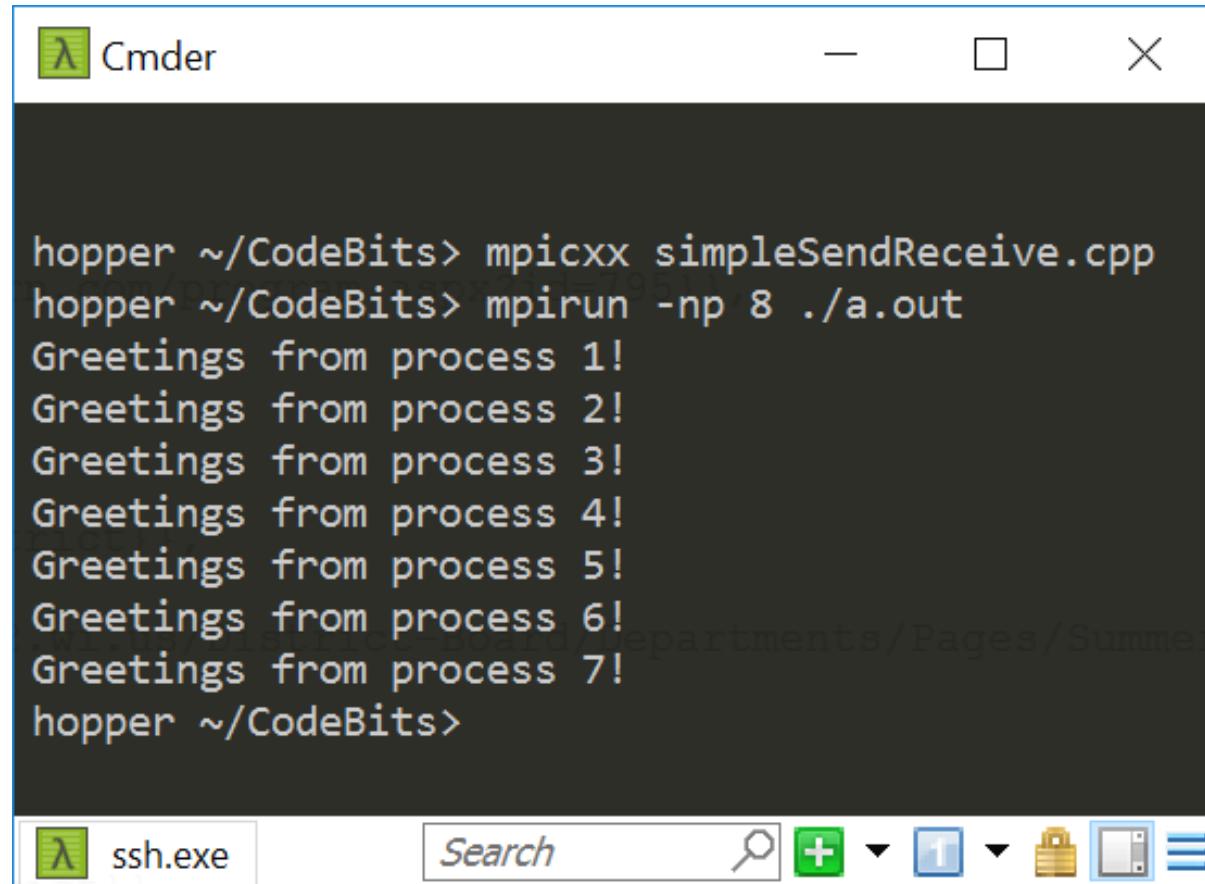
Destination process rank

Identifier for this message

Buffer to be sent

Number of items to send

# Program Output



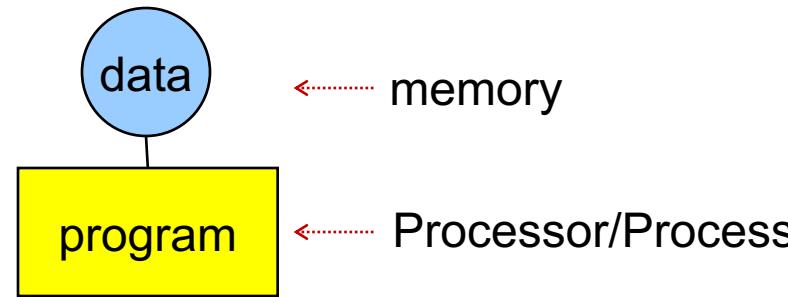
A screenshot of a Cmder terminal window titled "Cmder". The window contains the following text output:

```
hopper ~/CodeBits> mpicxx simpleSendReceive.cpp
hopper ~/CodeBits> mpirun -np 8 ./a.out
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
hopper ~/CodeBits>
```

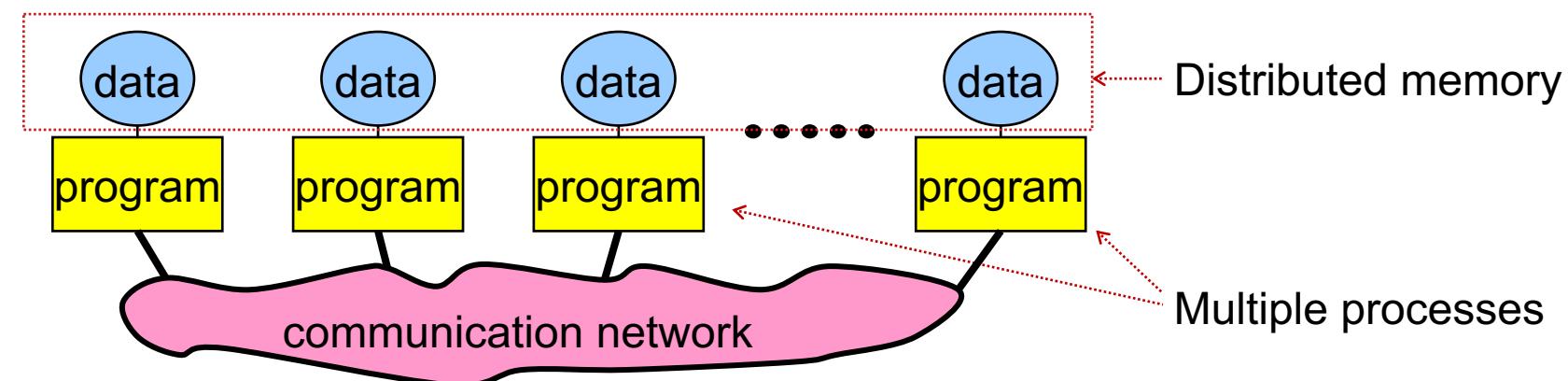
The terminal has a dark background and light-colored text. The command prompt "hopper ~/CodeBits>" appears at the bottom and top of the output. The "mpicxx" and "mpirun" commands are used to compile and run the program respectively. The program itself prints "Greetings from process 1!" through "Greetings from process 7!" to the console.

# The Message-Passing Programming Paradigm

- The Sequential Programming Paradigm

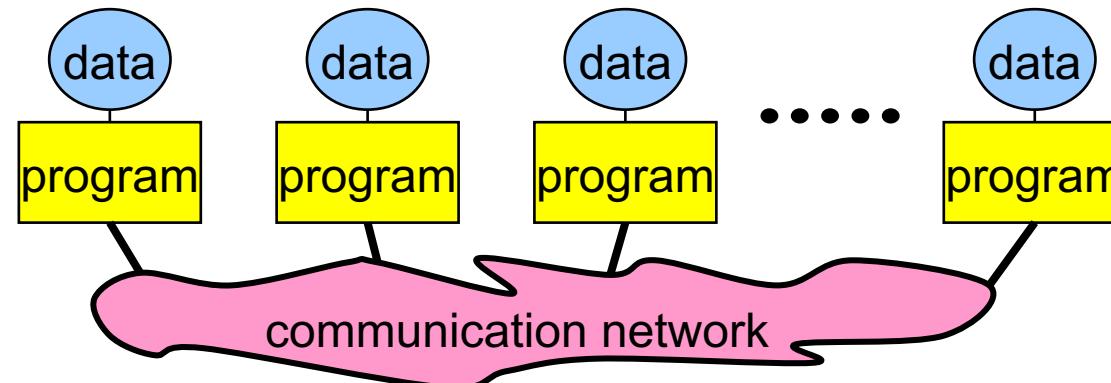


- Message-Passing Programming Paradigm



# Fundamental Concepts: Process/Program/Processor

- A **process** is a **software program** executing a task on a **processor**
- In other words, each process in an MPI job runs an instance of a **program**:
  - Program written in a conventional sequential language, e.g., C, C++, Fortran
  - The variables of each program have the **same name** but **different locations** (distributed memory) and **different values**
  - Communicate via special send & receive routines (**message passing**)



# What is MPI?

- A **standard** that specifies **rules** & **functionality** to facilitate parallel programming through message passing
  - Specifies a set of operations, but says nothing about their implementation
- MPI: a widely accepted standard, many vendors implemented it ⇒ MPI code **is portable**
- Note: MPI is not a library. It's a specification of services as well as guarantees regarding data safety

# What is MPI?

- Early implementation of MPI: [MPICH](#) (this is a library that implements [some of] the MPI specifications)
  - The CH comes from Chameleon, the software layer used in the original MPICH to provide portability to the existing message-passing systems
- [OpenMPI](#): another library that implements [some of] the MPI specifications
  - Joint effort of several groups from Los Alamos, Tennessee, Indiana University, Europe
- Just as we discussed for OpenMP, MPI is a standard and different vendor provide libraries that implement a certain version of the standard and a certain set of features defined in that version of the standard

# Where Can We Use MPI?

- Message passing can be used wherever it is possible for processes to exchange messages:
  - Distributed memory systems
  - Networks of workstations
  - Even on a shared memory system; i.e., on one workstation that has many cores
- **NOTE [VERY IMPORTANT]:**
  - Two OpenMP threads share the same virtual memory
  - Two MPI ranks do not share the same virtual memory (but they share the cache if they run on the same processor w/ multiple cores)
    - Two MPI ranks correspond to two *\*distinct\** processes, which therefore have two different virtual memories they operate in

# Why Care about MPI?

- Today, MPI is what enables supercomputers to run at PFlops rates
  - At the node level:
    - A process might use GPU acceleration at the node level
    - A process might use OpenMP the multiple cores available on a node
- Widely used in Scientific Computing, MPI has **FORTRAN** and **C** bindings

# MPI: Pluses and Minuses

- Pluses:
  - Access to lots of memory (across many nodes...)
  - Access to a lot of processors (across many nodes...)
  - No cache coherence overhead (if they don't run on the same node)
- Minuses:
  - High latency in communication
  - Low bandwidth compared to what we've seen so far (GPU & OpenMP on a workstation)
  - You have to manage the process of sharing data amongst processes
- NOTE: up-to-date typical latencies & bandwidth of various systems available [here](#)
  - Ranges: 5-10 usec & 1-80 GB/s (BW depends heavily on size of the big-iron machine; more like 1-5 GB/s)

# MPI vs. CUDA

- When would you use GPU computing and when would you use MPI-based parallel programming?
  - Use GPU
    - If your data fits the memory constraints associated with GPU computing
    - You have parallelism at a fine grain so that the SIMD paradigm applies
    - Example: Image processing
  - Use MPI-enabled parallel programming
    - If you have a very large problem, with a lot of data that needs to be spread out across several machines
    - Example: Large-scale CFD, FEA
- NOTE: there are many machines on which HPC is done using both MPI and CUDA

# MPI on Euler [Selecting MPI Distribution]

- What's available: OpenMPI, MVAPICH, MVAPICH2
- OpenMPI is default on Euler
- To load OpenMPI environment variables:
  - Typically not needed, should be done automatically for you

```
>> module load openmpi
```

# Compiling MPI Code by Hand

- Most MPI distributions provide wrapper scripts named `mpicc` or `mpicxx`
  - Adds in `-L`, `-l`, `-I`, etc. flags for MPI
  - Passes any options to your native compiler (`gcc`)
  - Very similar to what `nvcc` did for CUDA – it's a compile driver...

```
>> mpicxx -o integrate_mpi integrate_mpi.cpp
```

# Example, Running on Euler

[Context: running the executable integrate\_mpi]

```
### BEGINNING OF submit_mpi.sh SCRIPT ###

#!/bin/bash
#SBATCH -t 0-5:0:0
#SBATCH -o output.txt

cd $SLURM_SUBMIT_DIR
mpirun ./integrate_mpi

### END OF SCRIPT ###
```

```
euler $ sbatch -p wacc --gres=infiniband:1 -N 2 --ntasks-per-node=4 submit_mpi.sh
euler $ cat output.txt
8 32.121040666358297 in 2.171963s

euler $ sbatch -p wacc --gres=infiniband:1 -N 2 --ntasks-per-node=2 submit_mpi.sh
euler $ cat output.txt
4 32.121040666358297 in 4.600204s

euler $ sbatch -p wacc --gres=infiniband:1 -N 1 --ntasks-per-node=1
euler $ cat output.txt
1 32.121040666358297 in 15.163330s
```

# Running MPI, In General

```
mpirun [-np #] [-machinefile file] <program> [<args>]
```

Number of processors.  
Inside SLURM, this is  
handled automatically.

List of hostnames to use.  
Inside SLURM, this is  
handled automatically.

Your program and its  
arguments

- **-np** will be set automatically by SLURM. Do not use it.
- **-machinefile** will be set automatically by SLURM. Do not use it.
- See the **mpirun** manpage for more options

# MPI with CMake

- Using MPI in CMake a lot like using OpenMP
- To enable MPI, the FindMPI CMake module is called using `find_package`
- The MPI module sets a few flags which should be **added to your target**

```
find_package(MPI REQUIRED)
add_executable(problemX problemx.c)
target_include_directories(problemX ${MPI_C_INCLUDE_PATH})
target_compile_options(problemX ${MPI_C_COMPILE_FLAGS})
target_link_libraries(problemX ${MPI_C_LIBRARIES})
```

NOTE: Other language bindings such as MPI\_CXX\_\* or MPI\_FORTRAN\_\* may also be used.

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 27

04/01/2020

# Things to do, for Canvas live-session

- Dan: start the Canvas recording
- Everybody else:
  - Please mute your microphone
  - Please ask questions (limit: two per person per lecture; after that, post on Piazza)
  - When you ask questions, first unmute your mic

# The deep thinking of the day

Just like the folks who for the entire semester have been watching me @ home at 1.5X speed while enjoying the favorite brew, now I can do the same: reaching for a cold one while lecturing ME759 over Canvas...  
Does that cold one have any collateral effect? Perhaps my accent improves???

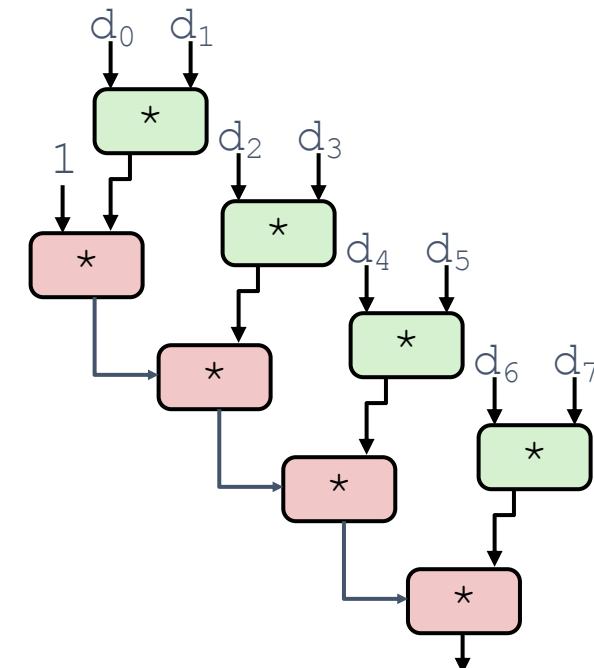
# Quote of the day

“There is nothing new in the world except the history you do not know.”  
-- Harry S. Truman, 33<sup>rd</sup> president of the United States [1884 –1972]

# Before we get going...

- Last time:
  - Optimization aspects, wrapped up
    - ILP & vectorization aspects
  - Distributed memory parallel computing via MPI
    - Discussion of underlying hardware
    - TOP500
    - Message Passing Interface (MPI): warmup
- Today:
  - MPI: point-to-point communication
- Other tidbits:
  - ME759 Exam: April 15, at 7:15 PM, in Canvas
    - Review: Tu, April 14, at 7:00 PM, in Canvas
  - Final Project Proposal: Dan to provide feedback this week
  - Last ME759 lecture is on Friday
  - Assignment is due on Friday

```
x = x OP (d[i] OP d[i+1]);
```



# HPC with the Message Passing Interface (MPI): Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- HPC w/ MPI: Closing Remarks

# MPI: High-level overview

- N instances of the same program are launched for execution independently as N distinct processes
- Flynn Taxonomy classification: MIMD
  - MPI & Distributed Memory: enables a MIMD computing mechanism
    - Multiple Instruction and Multiple Data
- Another way to look at it: Single Program Multiple Data (SPMD)

# Parallel Computing w/ MPI: One way to look at it

- CUDA: a small snippet of code (the “kernel”) run by all threads spawned via your exec. config.
- OpenMP: all threads execute an `omp parallel` region; name of the game is work sharing
- MPI: the **\*entire\*** code is executed in parallel by all processes

# A First MPI Program

```
#include "mpi.h"
#include <iostream>

int main(int argc, char **argv) {
    int my_rank, n;
    char hostname[128];

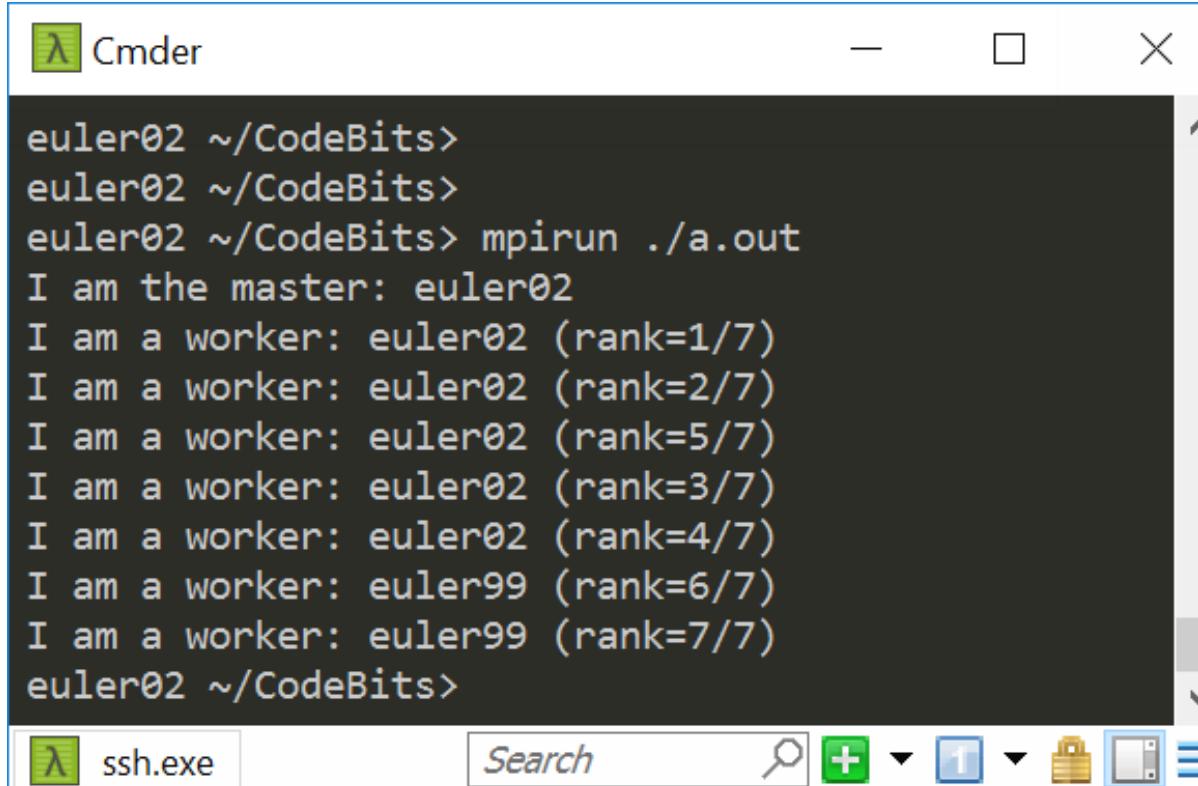
    MPI_Init(&argc,&argv); Has to be called first, and once only
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n);

    gethostname(hostname, 128);
    if (my_rank == 0) { /* master */
        printf("I am the master: %s\n", hostname);
    }
    else { /* worker */
        printf("I am a worker: %s (rank=%d/%d)\n", hostname, my_rank, n-1);
    }

    MPI_Finalize(); Has to be called last, and once only
    return 0;
}
```

# Program Output

- I asked for 8 ranks
- Got my 8 ranks spread out over two nodes: euler02 and euler99
- Please use Slurm to run your jobs; instructions in the assignment. Colin can assist if you have questions



The screenshot shows a Cmder terminal window with the title bar "Cmder". The terminal content displays the following text:

```
euler02 ~/CodeBits>
euler02 ~/CodeBits>
euler02 ~/CodeBits> mpirun ./a.out
I am the master: euler02
I am a worker: euler02 (rank=1/7)
I am a worker: euler02 (rank=2/7)
I am a worker: euler02 (rank=5/7)
I am a worker: euler02 (rank=3/7)
I am a worker: euler02 (rank=4/7)
I am a worker: euler99 (rank=6/7)
I am a worker: euler99 (rank=7/7)
euler02 ~/CodeBits>
```

The terminal interface includes a search bar labeled "Search" and various icons for file operations at the bottom.

# MPI: High-level Overview

- Each MPI process, which runs most often on one core, executes the **same** executable at roughly the same time
  - Synchronization calls can be invoked (more later); useful to coordinate the execution of the MPI job
  - Unlike in CUDA, synchronization w/ MPI can be **global**

# MPI: High-level Overview

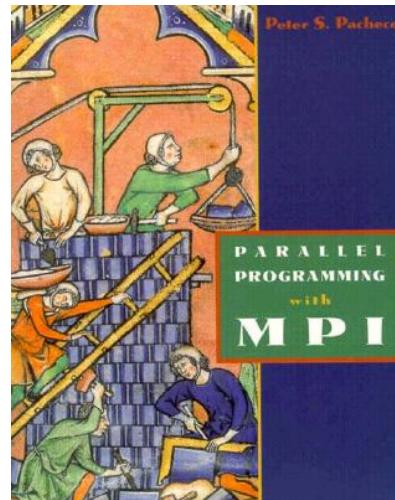
- What differentiates processes is their **rank**: processes with different ranks do different things (“branching based on the process rank”)
  - Very similar to GPU computing, where one thread did work based on its thread index
  - Very similar to OpenMP function `omp_get_thread_num()`

# The Message-Passing Model

- Each process has its own program counter **\*and\*** address space
- Message passing enables communication among processes that have separate address spaces
- Inter-process communication typically consists of
  - Synchronization, followed by...
  - ... movement of data from one process's address space to another process's address space

# MPI: A Second Example Application

Example from Peter Pacheco's book:  
“Parallel Programming with MPI”



```
/* greetings.c -- greetings program
 *
 * Send a message from all processes with rank != 0 to process 0.
 *   Process 0 prints the messages received.
 *
 * Input: none.
 * Output: contents of messages received by process 0.
 *
 * See Chapter 3, pp. 41 & ff in PPMPI.
 */
```

# MPI: A Second Example Application

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int      my_rank;      /* rank of process */
    int      p;            /* number of processes */
    int      source;       /* rank of sender */
    int      dest;         /* rank of receiver */
    int      tag = 0;       /* tag for messages */
    char    message[100];  /* storage for message */
    MPI_Status status;     /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
```

Type of each sent item

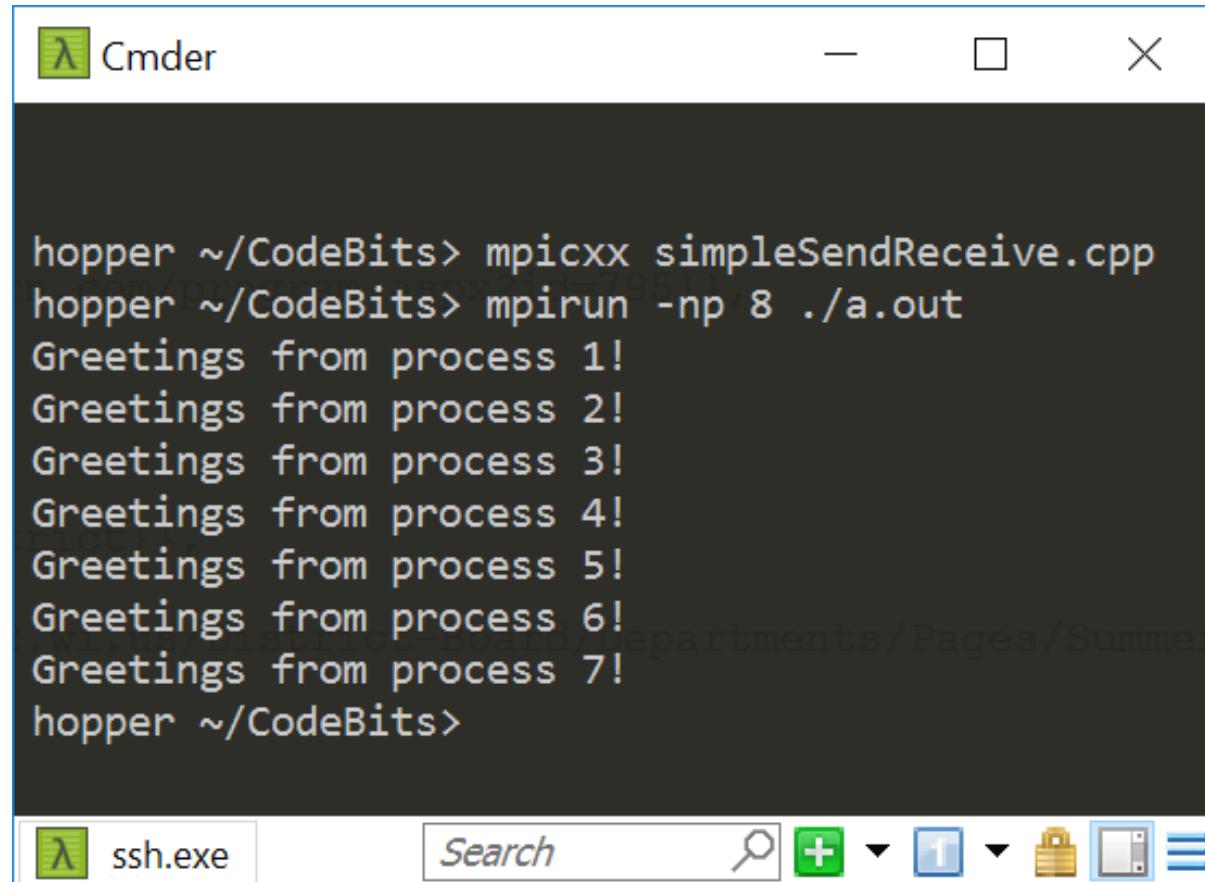
Destination process rank

Identifier for this message

Buffer to be sent

Number of items to send

# Program Output



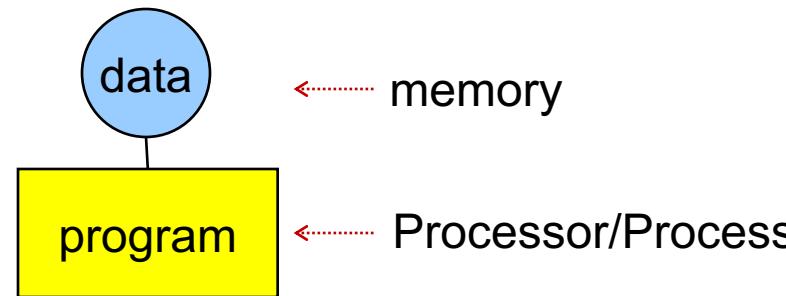
A screenshot of a Cmder terminal window titled "Cmder". The window contains the following text:

```
hopper ~/CodeBits> mpicxx simpleSendReceive.cpp
hopper ~/CodeBits> mpirun -np 8 ./a.out
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
hopper ~/CodeBits>
```

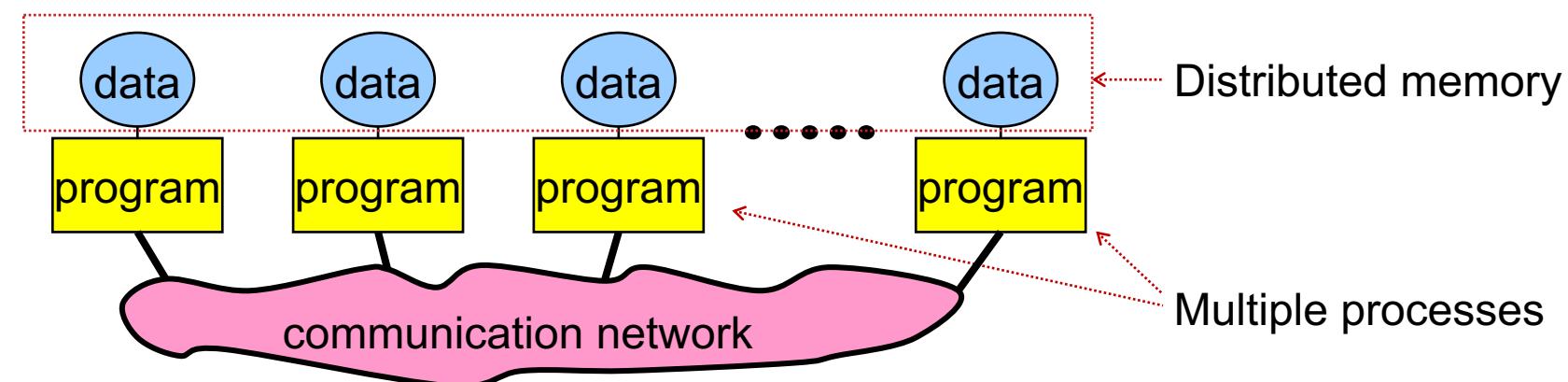
The terminal has a dark background and light-colored text. The command prompt "hopper ~/CodeBits>" appears at the bottom and top of the terminal window. The window has standard operating system window controls (minimize, maximize, close) at the top right. At the bottom, there is a taskbar with icons for "ssh.exe" and a search bar labeled "Search". To the right of the search bar are several small icons: a green plus sign, a blue downward arrow, a blue square with a white letter "T", a blue downward arrow, a gold padlock, a white square with a blue border, and a blue square with three horizontal lines.

# The Message-Passing Programming Paradigm

- The Sequential Programming Paradigm

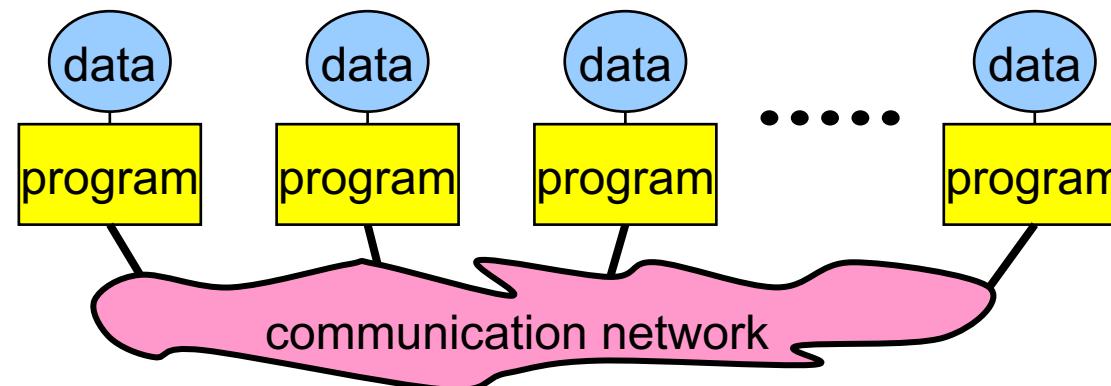


- Message-Passing Programming Paradigm



# Fundamental Concepts: Process/Program/Processor

- A **process** is a **software program** executing a task on a **processor**
- In other words, each process in an MPI job runs an instance of a **program**:
  - Program is written in C, C++, or Fortran. Reason: these are the languages for scientific computing, which MPI serves
  - The variables of each program have the **same name** but **different locations** (distributed memory) and **different values**
  - Communicate via special send & receive routines (**message passing**)



# What is MPI?

- A **standard** that specifies **rules** & **functionality** to facilitate parallel programming through message passing
  - Specifies a set of operations, but says nothing about their implementation
- MPI: a widely accepted standard, many vendors implemented it ⇒ MPI code **is portable**
- Note: MPI is not a library. It's a specification of services as well as guarantees regarding data safety

# Two MPI implementations

- Early implementation of MPI: [MPICH](#) (this is a library that implements [some of] the MPI specifications)
  - The CH comes from Chameleon, the software layer used in the original MPICH to provide portability to the existing message-passing systems
- [OpenMPI](#): another library that implements [some of] the MPI specifications
  - Joint effort of several groups from Los Alamos, Tennessee, Indiana University, Europe
- Just as we discussed for OpenMP, MPI is a standard and different vendor provide libraries that implement a certain version of the standard and a certain set of features defined in that version of the standard

# Where can we use MPI?

- Message passing can be used wherever it is possible for processes to exchange messages:
  - Distributed memory systems
  - Networks of workstations
  - Even on one workstation that has many cores
    - An MPI job run on one workstation does not pass data over a network; it does so through the main memory

# 16 threads/ranks on one workstation (assume one socket, 16-core processor)

- If 16 OpenMP threads, they will operate against the same virtual memory
- If 16 MPI ranks, they do not operate against the same virtual memory
  - There will be 16\*distinct\* processes, each with its virtual memory space
  - Consequences:
    - The 16 cores running the 16 MPI ranks are going to compete for the LLC (perhaps L3?)
    - Compared to OpenMP, there will be less cache coherence overhead since ranks don't operate on the same memory
    - Compared to OpenMP, there will be a similar level of contention for the memory bandwidth

# Why care about MPI?

- Today, MPI is what enables supercomputers to run at PFlops rates
  - At the node level:
    - A process might use GPU acceleration at the node level
    - A process might use OpenMP the multiple cores available on a node
- Widely used in Scientific Computing, MPI has **FORTRAN** and **C** bindings

# MPI: Pluses and Minuses

- Pluses:
  - Access to lots of memory (across many nodes...)
  - Access to a lot of processors (across many nodes...)
  - Better scalability - No cache coherence overhead at all when two ranks run on different nodes
    - Large MPI jobs involve tens of thousands of processes, each running heavy weight code
- Minuses:
  - High latency in node-to-node communication
  - Low bandwidth compared to what we've seen so far (GPU & OpenMP on a workstation)
  - You have to manage the process of sharing data amongst ranks
- NOTE: up-to-date typical latencies & bandwidth of various systems available [here](#)
  - Ranges: 5-10 usec & 1-80 GB/s (BW depends heavily on size of the big-iron machine; more like 1-5 GB/s)

# MPI vs. CUDA

- When would you use GPU computing and when would you use MPI-based parallel programming?
  - Use GPU
    - If your data fits the memory constraints associated with GPU computing
    - You have parallelism at a fine grain so that the SIMD paradigm applies
    - Example: Image processing
  - Use MPI-enabled parallel programming
    - If you have a very large problem, with a lot of data that needs to be spread out across several machines
    - Example: Large-scale CFD, FEA – sophisticated, multi-stage algorithms
- NOTE: there are many machines on which HPC is done using both MPI and CUDA

# MPI on Euler [Selecting MPI Distribution]

- What's available: OpenMPI, MVAPICH, MVAPICH2
- OpenMPI is default on Euler
- To load OpenMPI environment variables:
  - Typically not needed, should be done automatically for you

```
>> module load openmpi
```

# Compiling MPI Code by Hand

- Most MPI distributions provide wrapper scripts named `mpicc` or `mpicxx`
  - Adds in `-L`, `-l`, `-I`, etc. flags for MPI
  - Passes any options to your native compiler (`gcc`)
  - Very similar to what `nvcc` did for CUDA – it's a compile driver...

```
>> mpicxx -o integrate_mpi integrate_mpi.cpp
```

# Example, Running on Euler

[Context: running the executable integrate\_mpi]

```
### BEGINNING OF submit_mpi.sh SCRIPT ###

#!/bin/bash
#SBATCH -t 0-5:0:0
#SBATCH -o output.txt

cd $SLURM_SUBMIT_DIR
mpirun ./integrate_mpi

### END OF SCRIPT ###
```

```
euler $ sbatch -p wacc --gres=infiniband:1 -N 2 --ntasks-per-node=4 submit_mpi.sh
euler $ cat output.txt
8 32.121040666358297 in 2.171963s

euler $ sbatch -p wacc --gres=infiniband:1 -N 2 --ntasks-per-node=2 submit_mpi.sh
euler $ cat output.txt
4 32.121040666358297 in 4.600204s

euler $ sbatch -p wacc --gres=infiniband:1 -N 1 --ntasks-per-node=1
euler $ cat output.txt
1 32.121040666358297 in 15.163330s
```

# Running MPI, In General

```
mpirun [-np #] [-machinefile file] <program> [<args>]
```

Number of processors.  
Inside SLURM, this is  
handled automatically.

List of hostnames to use.  
Inside SLURM, this is  
handled automatically.

Your program and its  
arguments

- **-np** will be set automatically by SLURM. Do not use it.
- **-machinefile** will be set automatically by SLURM. Do not use it.
- See the **mpirun** manpage for more options

# MPI with CMake

- Using MPI in CMake a lot like using OpenMP
- To enable MPI, the FindMPI CMake module is called using `find_package`
- The MPI module sets a few flags which should be **added to your target**

```
find_package(MPI REQUIRED)
add_executable(problemX problemx.c)
target_include_directories(problemX ${MPI_C_INCLUDE_PATH})
target_compile_options(problemX ${MPI_C_COMPILE_FLAGS})
target_link_libraries(problemX ${MPI_C_LIBRARIES})
```

NOTE: Other language bindings such as MPI\_CXX\_\* or MPI\_FORTRAN\_\* may also be used.

# Quick note...

- ME759 is not about remembering how to launch (or lunch?) a job on a cluster
  - This is why we rely on sysadmin folks. Somebody like Colin will be there to take care of you
- Instead, the focus is on understanding how things work, how you can get the code to run fast

# Outline, Parallel Computing w/ MPI

- Introduction to message passing and MPI
- Point-to-Point (P2P) Communication
- Collective Communication
- MPI Closing Remarks

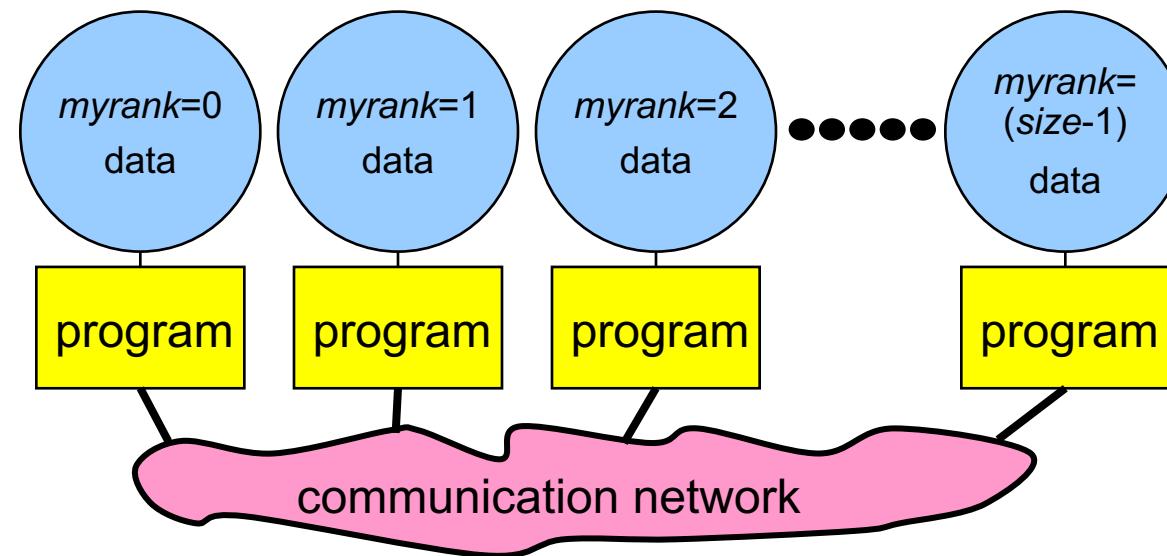
# Parallel computing issue: Who am I? What's my problem/task?

- Parallel computing on the GPU, on multi-core chips, and multi-node (MPI) share one thing:
  - For each thread/rank a mechanism is in place to provide a **unique ID**
    - ID then used to figure out what that thread/rank ought to be doing
  - Specifically,
    - In CUDA you have `threadIdx` and `blockIdx`
    - In OpenMP you have the `thread ID`
    - In MPI you have a process `rank`

# The Rank & The Communicator

[As Facilitators for Data and Work Distribution]

- To communicate with each other MPI processes need identifiers: **rank = identifying number**
- Work distribution decisions are based on the **rank**
  - Helps establish which process works on which data
  - Just like we had thread and block indices in CUDA

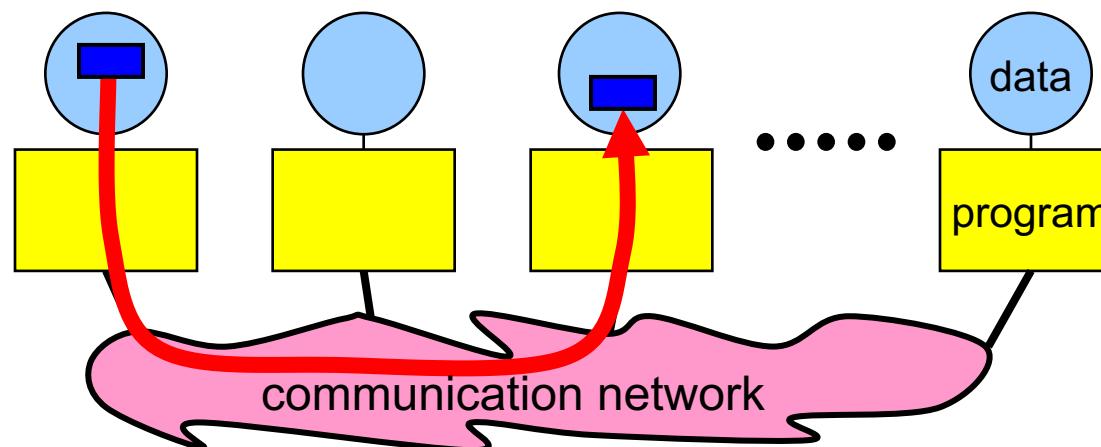


# Message Passing: The Actors Involved

- Messages are packets of data moving between different processes
- Necessary information for the message passing system:

• sending process        +        receiving process        } i.e., the two “ranks”

• source **mem.location**        +        destination **mem.location**  
• source **data type**        +        destination **data type**  
• source **data size**        +        destination **buffer size**



# Point-to-Point (P2P) Communication

- P2P: Simplest form of message passing communication
- One process sends a message to another process
  - `MPI_Send`
  - `MPI_Recv`

# MPI: Revisiting Previous Example

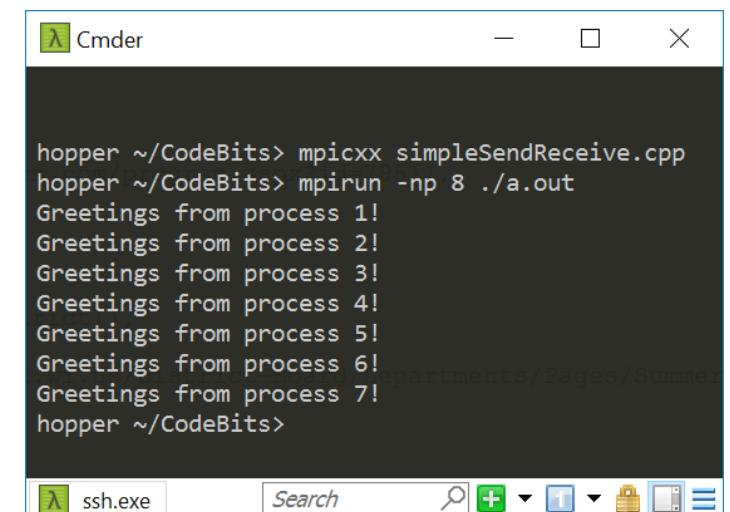
```
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int my_rank; /* rank of process */
    int p; /* number of processes */
    int source; /* rank of sender */
    int dest; /* rank of receiver */
    int tag = 0; /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }
    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
```

## Example: MPI\_Send / MPI\_Recv



The screenshot shows a terminal window titled 'Cmder' running on a Linux system. The command 'mpicxx simpleSendReceive.cpp' is run to compile the code, followed by 'mpirun -np 8 ./a.out' to execute it. The output shows greetings from all 8 processes (rank 1 to 8). The terminal also shows the path '/Users/.../Desktop/CodeBits/Summer...'.

```
hopper ~/CodeBits> mpicxx simpleSendReceive.cpp
hopper ~/CodeBits> mpirun -np 8 ./a.out
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
hopper ~/CodeBits>
```

# P2P Communication, Syntax Issues: Sending a Message

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- **buf** is the starting point of the message with **count** elements, each described with **datatype**
- **dest** is the rank of the destination process within the communicator **comm**
- **tag** is an additional nonnegative integer piggyback information, additionally transferred with the message
  - The **tag** can be used to distinguish between different messages
  - Not very common, needed in some situations

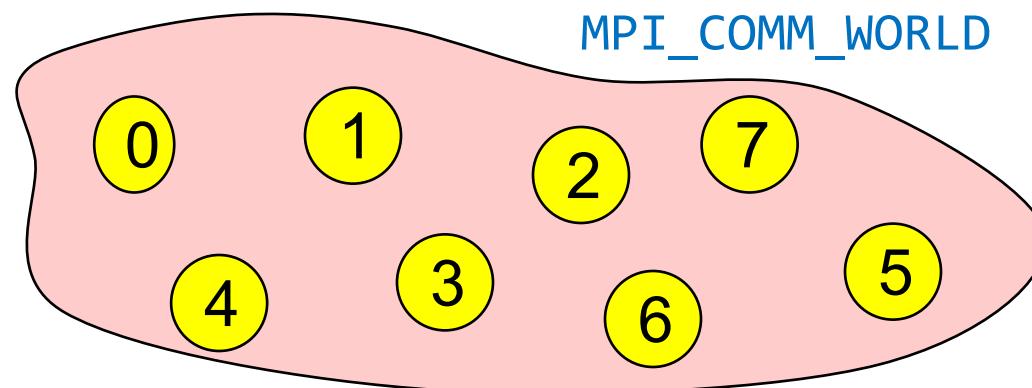
# P2P Communication, Syntax Issues: Receiving a Message

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- **buf/count/datatype** describe the receive buffer
  - Note: It better be that the memory chunk that starts at address buf can store count variables of type datatype
- Receiving the message sent by process with rank **source** in **comm**
- Only messages with matching **tag** are received
- Envelope information is returned in an **MPI\_Status** object whose address **status** is passed as argument

# The MPI\_COMM\_WORLD Communicator

- All processes of an MPI program are members of the default communicator `MPI_COMM_WORLD`
- `MPI_COMM_WORLD` is a predefined **handle** in `mpi.h`
- Each process has its own **rank** in a given communicator:
  - starting with 0
  - ending with (size-1)



- You can define a new communicator in case you find it useful
  - To create a new communicator `MY_COMM_WORLD`

```
MPI_Comm_create(MPI_COMM_WORLD, new_group, &MY_COMM_WORLD);
```

# MPI\_Comm\_create

- Synopsis

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);
```

- Input Parameters

- `comm` - communicator (handle)
- `group` - subset of the family of processes making up the `comm` (handle)

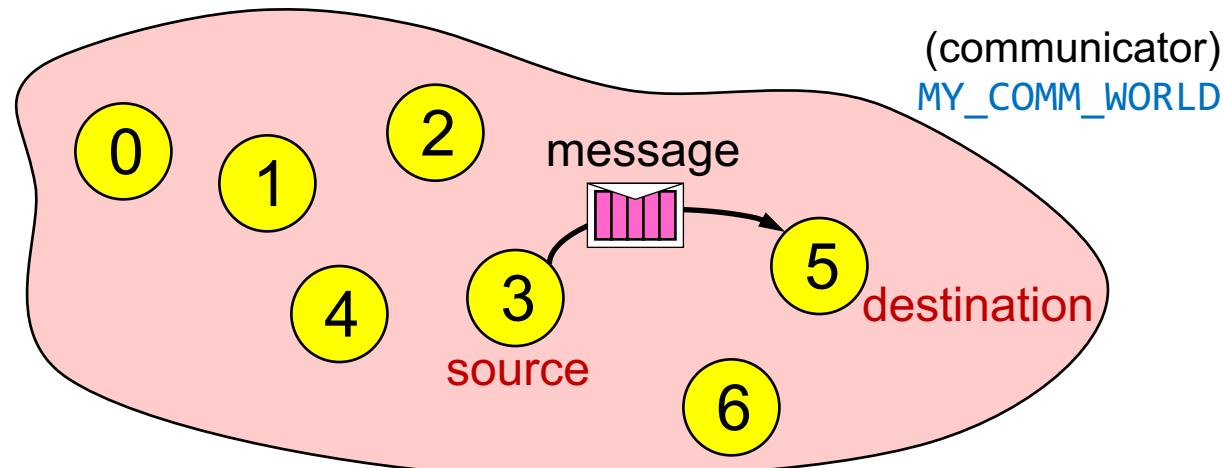
- Output Parameter

- `newcomm` - new communicator (handle)

- NOTE: Same process has different ranks in different communicators

# Point-to-Point Communication

- Communication between two processes
- In pic. below: **source** process sends message to **destination** process
- Communication takes place within a communicator, e.g., `MY_COMM_WORLD`
- Processes are identified by their ranks **within** the communicator
  - Recall that the same process has different ranks in different communicators



# The Data Type

- A message contains a number of elements of some particular data type
- MPI data types:
  - Basic data type
  - User-defined data types – not emphasized here
- Data type **handles** describe the type of the data moved around

Example: message with 5 integers

2345	654	96574	-12	7676
------	-----	-------	-----	------

# MPI-defined data types, and C counterparts

<b>MPI Datatype</b>	<b>C datatype</b>
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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

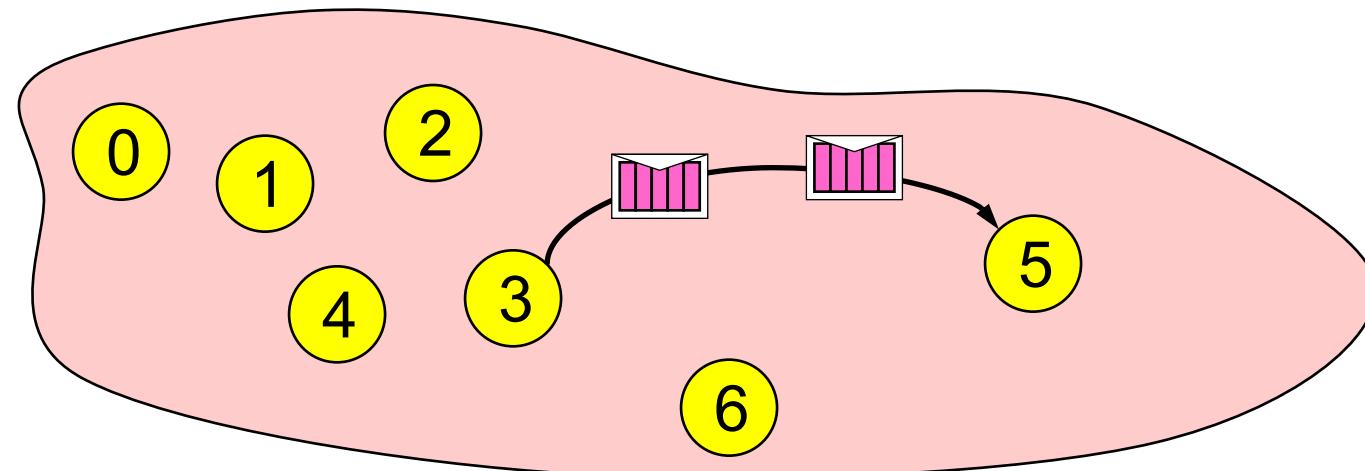
**Example:**

2345	654	96574	-12	7676
------	-----	-------	-----	------

count=5  
datatype=MPI\_INT  
**int arr[5]**

# Message Order Preservation

- Rule for messages on the same connection; i.e., same communicator, source, and destination rank:
  - **Messages do not overtake each other**



- If both receives match both messages, then the order is preserved

# The Mechanics of P2P Communication: Wildcarding

- Receiver can wildcard
  - To receive from any source – `source = MPI_ANY_SOURCE`
  - To receive from any tag – `tag = MPI_ANY_TAG`
  - Actual source and tag returned in receiver's `status` argument
    - One more instance where the `status` argument comes into play

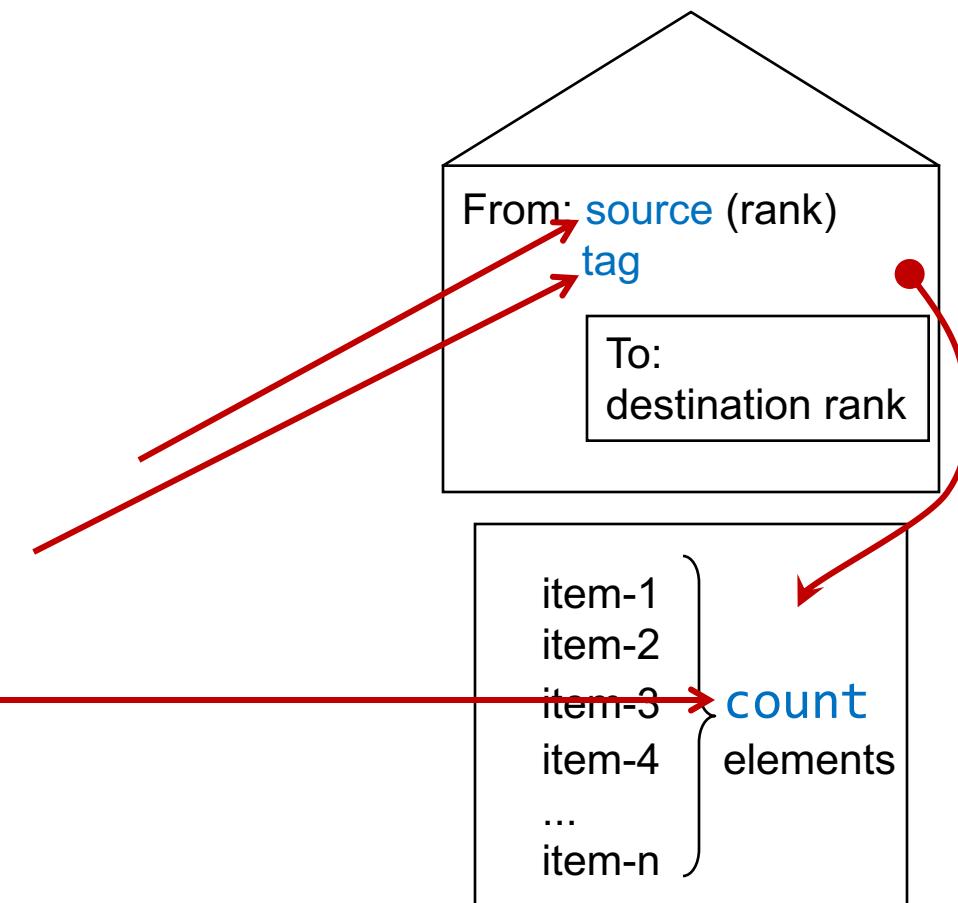
# MPI\_Recv: The Need for an MPI\_Status Argument

- The **MPI\_Status** object updated by the **MPI\_Recv** call settles a series of questions:
  - The receive call does not specify the size of an incoming message, but only an upper bound on the receive buffer
    - The actual size only known to the sender (think about it)
  - The source or tag of a received message may not be known if wildcard values were used in a receive operation

# The Mechanics of P2P Communication: Communication Envelope

- Envelope information is returned from MPI\_RECV via the status pointer

status->MPI\_SOURCE  
status->MPI\_TAG  
count via MPI\_Get\_count()



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

INPUT: status and datatype. OUTPUT: count

# The Mechanics of P2P Communication: Some Rules of Engagement

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
- Message data types must match
- Receiver's buffer must be large enough

# MPI\_Send & MPI\_Recv: The Eager and Rendezvous Flavors

- If you send **small** messages, the content of the buffer is sent to the receiving partner immediately
  - Operation happens in “**eager mode**”
- If you send a large amount of data, the sender function waits for the receiver to post a receive before sending the actual data of the message
  - Operation happens in “**rendezvous mode**”
- Why this eager-rendezvous dichotomy?
  - Because of the size of the data and the desire to have a safe implementation
  - If you send a small amount of data, the MPI implementation can afford to buffer the content and subsequently carry out the transaction later on, when the receiving process asks for that data
    - Doesn't fly if you attempt to move around a big chunk of data

# MPI\_Send & MPI\_Recv: Eager OR Rendezvous?

- Each implementation of MPI has a default value (which might change at run time) beyond which a larger `MPI_Send` stops acting in “eager mode”
  - The MPI standard doesn’t provide specifics
  - You don’t know how large is too large...
  - Consult your implementation’s documentation or measure it yourself
- Does it matter if it’s Eager or Rendezvous?
  - **Yes**, sometimes the code can hang – example to come
- **Remark:** the Eager vs. Rendezvous modes → they exclusively concern the `MPI_Send` and `MPI_Recv` send/receive flavors.
  - There are other ways to send data, beyond plain vanilla `MPI_Send`. More to come

# Blocking Type: Communication Modes

- Send communication modes:
  - Synchronous send → `MPI_SSEND`
  - Buffered [asynchronous] send → `MPI_BSEND`
  - Standard send → `MPI_SEND`
  - Ready send → `MPI_RSEND`
- Receiving all modes → `MPI_RECV`

# Point-to-Point Communication: The MPI\_Bsend flavor

- Another blocking P2P communication alternative that reduces the overhead associated with data transmission
  - Realized through `MPI_Bsend` (**b**uffered)
  - Your data is quickly moved to a “staging buffer” and then the call returns and renders the control back to you
  - NOTE: \*you\* need to **provide** this additional buffer that stages the transfer. \*You\* should also decide **how big** it is
    - Relevant question: how large should \*that\* staging buffer be?
      - If you keep posting buffered sends that are not matched by corresponding “`MPI_Recv`” operations, you are going to overflow this staging buffer sooner or later

# More on the Buffered Send [1/2]

- Relies on the existence of a buffer, which is set up through a call

```
int MPI_Buffer_attach(void* buffer, int size);
```

- **MPI\_Bsend**: a local operation. Does not depend on the occurrence of a matching receive in order to complete
  - If a bsend operation is started and no matching receive is posted, the outgoing message is buffered to allow the send call to complete
- Return from an **MPI\_Bsend** does not guarantee the message was sent
  - Message remains in the buffer until a matching receive is posted

## More on the Buffered Send [2/2]

- Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is not enough buffer space
- The amount of buffer space needed to be safe depends on the expected peak of pending messages. The sum of the sizes of all of the pending messages at that point plus (`MPI_BSEND_OVERHEAD*number_of_messages`) should be sufficient
- `MPI_Bsend` lowers bandwidth since it requires an extra memory-to-memory copy of the outgoing data
- The `MPI_Buffer_attach` subroutine provides MPI a buffer in the user's memory. This buffer is used only by messages sent in buffered mode, and only one buffer is attached to a process at any time

# Further comments on the four flavors for sending data in MPI

## 1. Synchronous with MPI\_Ssend

- In synchronous mode, a send will not complete until a matching receive is posted.
  - The sender has to wait for a receive to be posted
  - No buffering of data
  - Used for ensuring the code is healthy and doesn't rely on buffering

## 2. Buffered with MPI\_Bsend

- Send completes once message has been buffered internally by MPI
  - Buffering incurs an extra memory copy
  - Does not require a matching receive to be posted
  - May cause buffer overflow if many Bsend posted and no matching receives have been posted yet

# Four flavors for sending data in MPI

## 3. Standard with MPI\_Send

- Up to the MPI implementation to decide whether to do rendezvous or eager, for performance reasons
  - NOTE: If it does rendezvous, in fact the behavior is that of MPI\_Ssend
- Very commonly used

## 4. Ready with MPI\_Rsend

- Will work correctly *only* if the matching receive has been posted
- Can be used to avoid handshake overhead when it is known for sure that the receive has been posted
- Rarely used, can cause major problems
  - Undefined behavior if a matching receive has not been posted

# Cheat Sheet, Blocking Options

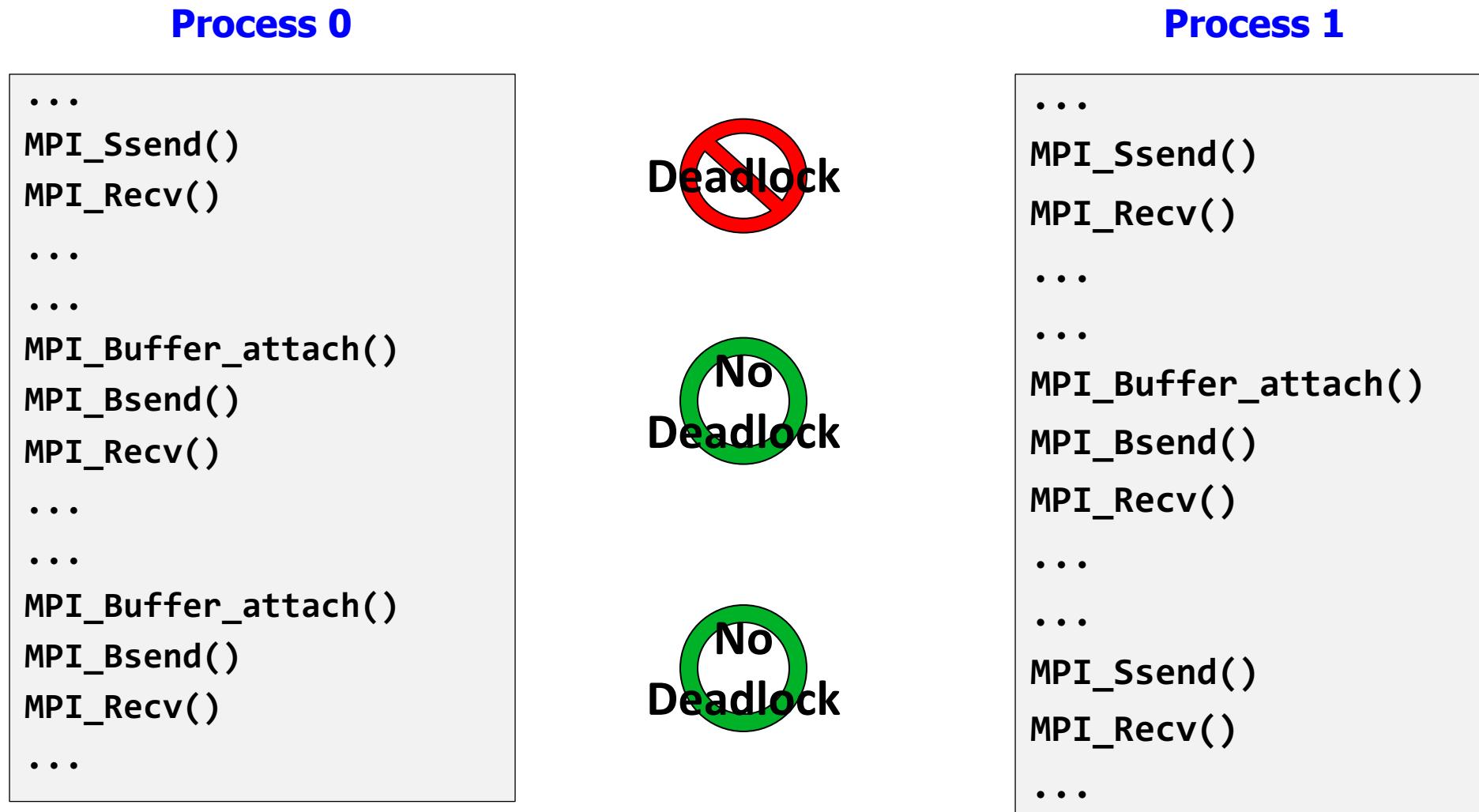
Sender modes	Definition	Notes
Synchronous send <b>MPI_SSEND</b>	Only completes when the receive has started	
Buffered send <b>MPI_BSEND</b>	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with <b>MPI_BUFFER_ATTACH</b>
Classic <b>MPI_SEND</b>	Standard send	Rendezvous or eager mode. Decided at run time
Ready send <b>MPI_RSEND</b>	Started right away. Will do the job <b>only</b> if the matching receive has already been posted!	Blindly do a send. Avoid, might cause unforeseen problems...
Receive <b>MPI_RECV</b>	Completes when a the message (data) has arrived	

# Side-trip: TCP/IP vs. UDP Analogy

- There is a parallel between how MPI\_Send and MPI\_Rsend different, on the one hand, and how TCP/IP and UDP are different, on the other hand
- TCP/IP: you send something out
  - You do not declare victory until the receiver pings you that all packets have been received and it's all good
  - Very safe. You pay a premium for this
- UDP: you send something out
  - You declare victory right away
  - Not safe, you don't know if the receiver actually got your packet[s]. It's fast
- UDP and TCP/IP otherwise similar – just two APIs for sending information out over a network
- Note: UDP used in online gaming, for instance

# P2P Communication: Deadlocking

- Deadlock situations: due to a certain sequence of commands the MPI execution hangs



# Deadlocking, Another Example

- `MPI_Send` can respond in eager or rendezvous mode
- Example, on a certain machine running MPICH v1.2.1:



# Avoiding Deadlocking

- Easy way to eliminate deadlock is to pair `MPI_Ssend` and `MPI_Recv` operations the right way:



- Conclusion: understand how the implementation works and what its pitfalls/limitations are

# Example

- Always succeeds, even if no buffering is done

```
if(rank==0)
{
    MPI_Send(...); // send to 1
    MPI_Recv(...); // recv from 1
}
else if(rank==1)
{
    MPI_Recv(...); // recv from 0
    MPI_Send(...); // send to 0
}
```

# Example

- Will always deadlock, no matter the buffering mode

```
if(rank==0)
{
    MPI_Recv(...); // recv from 1
    MPI_Send(...); // send to 1
}
else if(rank==1)
{
    MPI_Recv(...); // recv from 0
    MPI_Send(...); // send to 0
}
```

# Timing an MPI Job

```
int main()
{
    double starttime, endtime;
    starttime = MPI_Wtime();
    // .... stuff to be timed ...
    endtime   = MPI_Wtime();
    printf("That took %f seconds\n", endtime - starttime);
    return 0;
}
```

- Returns time in seconds since an arbitrary time in the past.
- Resolution is typically 1E-3 seconds (see [MPI\\_Wtick](#))
- Time of different processes might actually be synchronized, controlled by the variable [MPI\\_WTIME\\_IS\\_GLOBAL](#)

# Determine the eager/rendezvous threshold [useful for HW as well]

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define LIMIT 500000
#define NLOOPS 1000

int main(int argc, char **argv) {
    int rank, npes, bsize, i;
    int source = 0, dest = 1, tag = 100;
    double start, end;
    char buf[LIMIT];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    if (npes != 2)
        MPI_Abort(MPI_COMM_WORLD, 1);

    if (rank == 0)
        bsize = (argc == 2) ? atoi(argv[1]) : 0;

    MPI_Bcast(&bsize, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (bsize <= 0 || bsize > LIMIT) {
        MPI_Finalize(); // drop out, if size is not OK
        return 0;
    }
```

```
if (rank == source) {
    for (i = 0; i < bsize; i++)
        buf[i] = 'x'; // fill up the buffer with data to be sent over

    start = MPI_Wtime();
    for (i = 0; i < NLOOPS; i++)
        MPI_Send(buf, bsize, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
    end = MPI_Wtime();
    printf("Send: %d\t%f\n", bsize, (end - start) / NLOOPS);

    start = MPI_Wtime();
    for (i = 0; i < NLOOPS; i++)
        MPI_Ssend(buf, bsize, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
    end = MPI_Wtime();
    printf("Ssend: %d\t%f\n\n", bsize, (end - start) / NLOOPS);
}

if (rank == dest) {
    for (i = 0; i < NLOOPS; i++)
        MPI_Recv(buf, bsize, MPI_BYTE, source, tag, MPI_COMM_WORLD, &status);
    for (i = 0; i < NLOOPS; i++)
        MPI_Recv(buf, bsize, MPI_BYTE, source, tag, MPI_COMM_WORLD, &status);
}

MPI_Finalize();
return 0;
```

# Results

```
[serban@euler24:~/CODES/MPI]$ cat my_hostfile
euler24 slots=1
euler23 slots=1
[serban@euler24:~/CODES/MPI]$ mpiexec -np 2 -machinefile my_hostfile ./a.out 64000
Send: 64000 0.000543
Ssend: 64000 0.000753

[serban@euler24:~/CODES/MPI]$ mpiexec -np 2 -machinefile my_hostfile ./a.out 65000
Send: 65000 0.000551
Ssend: 65000 0.000767

[serban@euler24:~/CODES/MPI]$ mpiexec -np 2 -machinefile my_hostfile ./a.out 66000
Send: 66000 0.000846
Ssend: 66000 0.000848

[serban@euler24:~/CODES/MPI]$ mpiexec -np 2 -machinefile my_hostfile ./a.out 67000
Send: 67000 0.000846
Ssend: 67000 0.000856

[serban@euler24:~/CODES/MPI]$
```

Two processes on two different nodes

Threshold between 64 KB and 65 KB

```
[serban@euler24:~/CODES/MPI]$ cat my_hostfile
euler24 slots=2
euler23 slots=1
[serban@euler24:~/CODES/MPI]$ mpiexec -np 2 -machinefile my_hostfile ./a.out 64000
Send: 640000 0.000151
Ssend: 640000 0.000151

[serban@euler24:~/CODES/MPI]$ mpiexec -np 2 -machinefile my_hostfile ./a.out 650000
Send: 650000 0.000150
Ssend: 650000 0.000149

[serban@euler24:~/CODES/MPI]$ mpiexec -np 2 -machinefile my_hostfile ./a.out 660000
Send: 660000 0.000158
Ssend: 660000 0.000157

[serban@euler24:~/CODES/MPI]$ mpiexec -np 2 -machinefile my_hostfile ./a.out 670000
Send: 670000 0.000160
Ssend: 670000 0.000160

[serban@euler24:~/CODES/MPI]$
```

Two processes on the same node

No difference between Send and Ssend (regardless of message size)

# ME759

High Performance Computing for Applications in Engineering

[Spring 2020]

Lecture 28

04/03/2020

# Things to do, for Canvas live-session

- Dan: start the Canvas recording
- Everybody else:
  - Please mute your microphone
  - Please ask questions (limit: two per person per lecture; after that, post on Piazza)
  - When you ask questions, first unmute your mic

# Quote of the day

“Human beings only use ten percent of their brains. Ten percent! Can you imagine how much we could accomplish if we used the other sixty percent?”

Ellen DeGeneres, American comedian, television host, actress, writer, and producer [1958 - ]

# And how “Quote of the Day” applies to ME759

In most cases, we use 10% of the compute power at our disposal. Ten percent!  
Can you imagine what we could accomplish if we used the other 60 %?  
**That's what ME759 is all about – using the other 60%.**

Dan Negrut, Romanian instructor, television host, actor, writer, and producer [1968 - ]

# Before we get going...

- Last time:
  - MPI: point-to-point communication
    - MPI\_Send, MPI\_Ssend, MPI\_Bsend, MPI\_Rsend
    - MPI\_Receive
- Today:
  - Wrap up MPI P2P communication
    - Nonblocking flavors
  - MPI collective communication
  - ME759 wrap up
- Other tidbits:
  - ME759 Exam: April 15, at 7:15 PM, in Canvas
    - Review: Tu, April 14, at 7:00 PM, in Canvas
  - Final Project Proposal feedback went out on Wd
  - This is last ME759 lecture
  - Last assignment goes out today; due next Friday, 9PM

# ME759: Final Exam related

- Exam is comprehensive
  - Exam: “open everything”
  - Exam requirement: you should not communicate w/ anybody for the duration of the exam
    - You’ll have to electronically sign and date this statement: “I have neither given nor received, nor have I tolerated others’ use of unauthorized aid on this exam.”
- Exam pertains material that was **either** covered in the PPT slides **or** part of the homework
  - Assigned reading is for you to know more, but no probing there
- When: April 15, 04/15/2020, 7:15PM – 9:15PM
  - Exam is **online**

# ME759: Final Exam related

- For taking the exam, you'll need:
  - Fully charged or plugged laptop that can connect to the internet
  - A good/reliable internet connection
  - Access to Canvas
- Final Exam format:
  - A set of 20-30 multiple-choice questions
  - Some four or so other problems where you need to think and provide an answer
  - You might have to write pseudo-code, or look at a piece of code and answer questions
  - You will not have to write code, compile it, link, etc.
- Link to [sample exam](#) (do not read too much into this, might be changed without notice)

# ME759: Review session for the Final Exam

- Review for exam
  - Tu, April 14, at 7:00 PM, in Canvas
  - I will record the session, for those of you who can't make it
  - Questions to be compiled in a forum post. We'll go through questions until we cover them all
    - Link to forum thread where you can post questions: <https://piazza.com/class/k5crmnymtx95c0?cid=363>
  - Please post your questions in advance

# Non-blocking, P2P Communication

# P2P Communication: Drawbacks of the MPI\_Send (and friends)

- Two problems with plain vanilla blocking send/receive covered thus far (that is, Send, Ssend, Bsend, Rsend):
  1. When MPI “send” function is called, calling rank is blocked (there is no overlap of compute & data movement) (calling rank needs to wait until the send completed, which can be a while)
  2. If not done properly, the processes executing the MPI code can hang
- There is a second class of send/receive operations, which eliminates the two issues above (but brings in others)
  - Called “non-blocking” sends/receives

# Send/Receives: Blocking vs. Non-blocking [important slide]

- MPI send/receive operations: classified as **blocking** or **non-blocking**
  - **Blocking send:** upon return from a send operation, you can modify the content of the buffer in which you stored data to be sent since the data has been sent
    - `MPI_Send`, `MPI_Ssend`, `MPI_Bsend`, `MPI_Rsend`
  - **Non-blocking send:** the send call returns immediately; there is no guarantee that the data has actually been transmitted upon return from send call
    - `MPI_Isend`, `MPI_Issend`, `MPI_Ibsend`, `MPI_Irsend`
- The blocking/non-blocking **receive** op has the same semantics
  - Upon return from a non-blocking receive call, it's not clear whether the receive buffer has fresh data in it

# Non-Blocking P2P Communication: How things play out

- There's three acts to the “non-blocking communication” play:
  1. Initiate non-blocking communication
    - Returns **I**mmmediately
    - Routine name starting with **MPI\_I**...
  2. The calling rank gets to do some other useful work upon return from non-blocking call
    - This allows for the “latency hiding”
  3. Later, wait for non-blocking communication to complete (accomplished via a synchronization call)
- Bottom line: Like with CUDA kernel launch, you can overlap communication with execution

# P2P Communication: The looming danger w/ non-blocking MPI\_Isend

- The “I” flavor switches you to an asynchronous operation mode. Improves efficiency, but **no free lunch**
- If non-blocking, then data “lives” in your buffer – think twice before changing its content since not clear when transaction has completed. This means it’s not clear when it’s safe to mess with the content of the buffer
  - There are mechanisms to check for `MPI_Isend` completion (to give you a green light to change its content)
    - However, they add overhead :-)
  - Same looming issue for “receive” as well...

# Non-blocking Send/Receive

There was no such thing in the good old MPI\_Send (and the other blocking flavors)

- Syntax

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

- buf - [in] initial address of send buffer (choice)
- count - [in] number of elements in send buffer (integer)
- datatype - [in] datatype of each send buffer element (handle)
- dest - [in] rank of destination (integer)
- tag - [in] message tag (integer)
- comm - [in] communicator (handle)
- req - [out] communication request (handle) 

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *req);
```

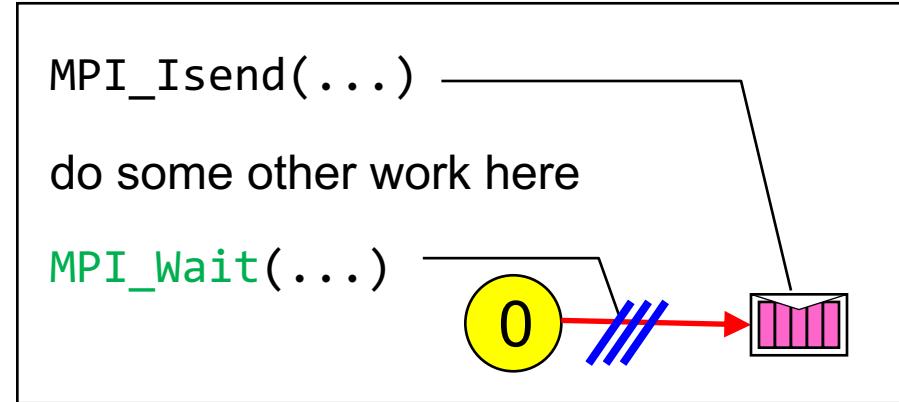
For the good old MPI\_Recv, this used to be MPI\_Status 

# Non-Blocking Send/Receive: Tools of the Trade

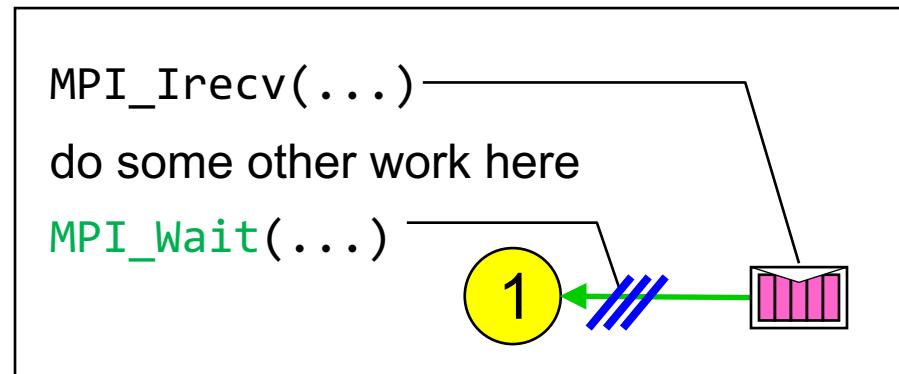
- Key observation: the nonblocking call returns immediately. Therefore, user must worry whether ...
  - Data to be sent is out of the send buffer before trampling over the buffer
  - Data to be received has arrived before using the content of the buffer
- Tools that will come in handy for immediate sends & receives:
  - `MPI_Wait` – blocking: once called, makes you wait till a certain request completed
  - `MPI_Test` – non-blocking: returns quickly with status information
- These helper tools: all use the `MPI_Request` object – it is like the “fingerprint” of a certain MPI operation
  - You use this fingerprint as input when you want to find information about the status of the MPI operation

# The Screenplay: Non-Blocking P2P Communication

- Non-blocking send



- Non-blocking receive



= waiting until operation locally completed

# Waiting for Irecv/Irecv to complete

- Waiting on a single send

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);  
          ↑  
          [IN]           ↑  
                      [OUT]
```

- Waiting on multiple sends (get status of all)

- Till all complete, as a barrier

```
int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses);
```

- Till at least one completes

```
int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status);
```

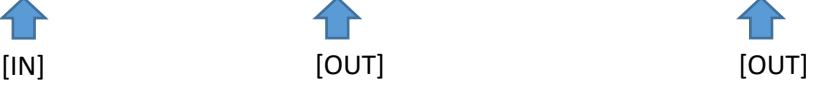
- Helps manage progressive completions

```
int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indices, MPI_Status *statuses);
```

# MPI\_Test

- When completed, flag is true

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```



The diagram shows the `MPI_Test` function signature with three parameters. Above the first parameter, `*request`, is a blue arrow pointing up with the label "[IN]" below it. Above the second parameter, `*flag`, is a blue arrow pointing up with the label "[OUT]" below it. Above the third parameter, `*status`, is a blue arrow pointing up with the label "[OUT]" below it.

- Similar in meaning to `MPI_Waitsome`, except nonblocking

```
int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indices, MPI_Status *statuses);
```

```
int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses);
```

```
int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status);
```

## Another helper: MPI\_Probe / MPI\_Iprobe

- The `MPI_PROBE` and `MPI_IPROBE` operations allow incoming messages to be queried prior to receiving them
- The user can then decide how to receive them, based on the information returned by the probe
  - The information of interest returned in the status handler
- In particular, user may allocate memory for the receive buffer, according to the length of the probed message

# Probe prior to receive

- Function comes in two flavors: blocking and non-blocking

- Blocking probing, wait till match

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

- Non-blocking probing, flag set to true if ready

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);
```

# MPI P2P Communication: Take Away Points

- Two types of communication:
  - Blocking:
    - Safe to change content of buffer holding on to data in the MPI send call
  - Non-blocking:
    - Be careful with the data in the buffer, since you might step on/use it too soon
- MPI provides four modes for these two types
  - standard, synchronous, buffered, ready
- Beware of possible deadlock

# Quiz

- How is `MPI_Issend` supposed to work?
- How is `MPI_Ibsend` supposed to work?
  - Starts a nonblocking buffered send

# Outline, Parallel Computing w/ MPI

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Actions
- MPI Closing Remarks

# Collective Actions

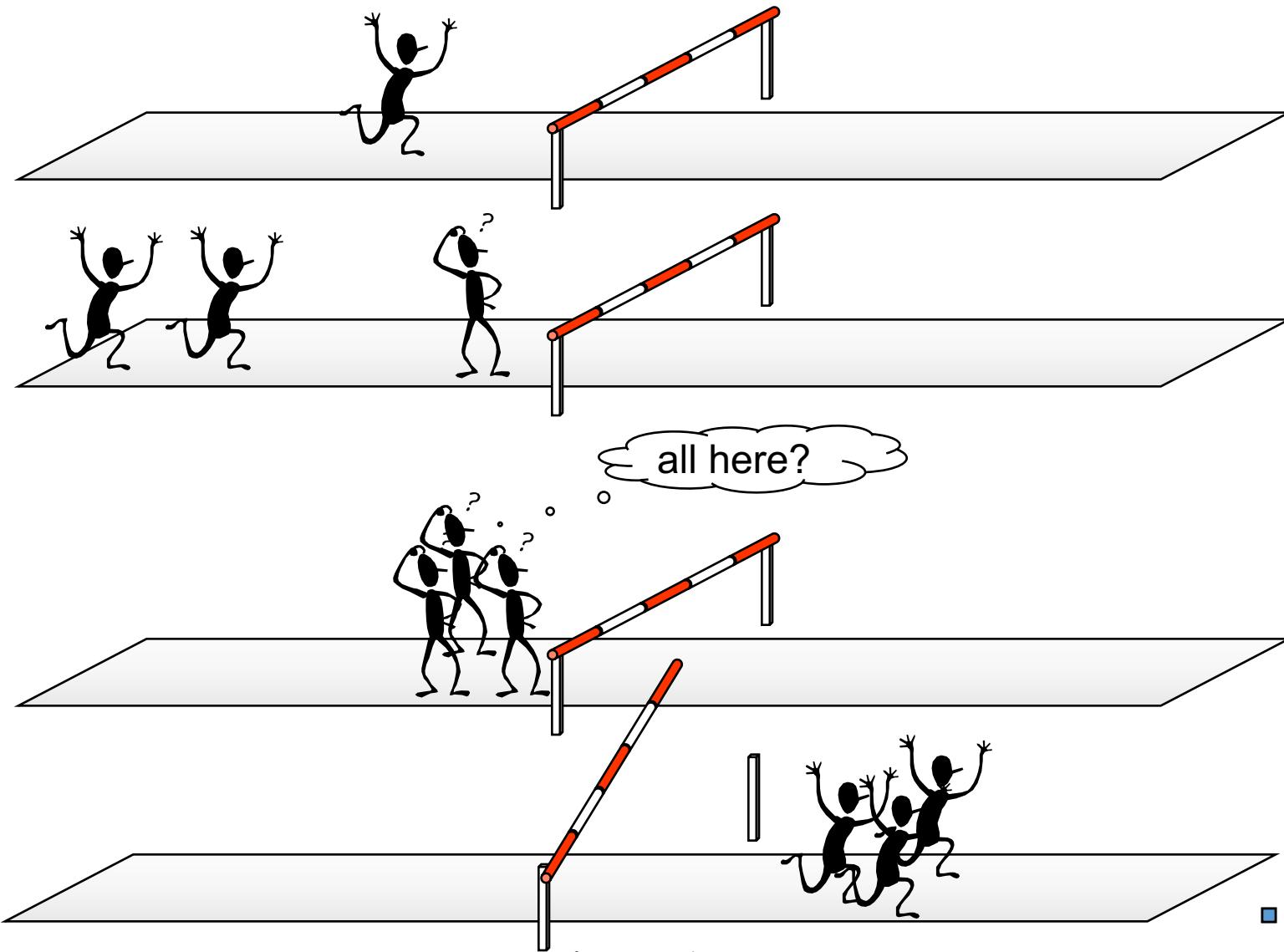
- Collective Actions: MPI actions involving a **group** of processes
  - Thus far, we've talked about P2P communication only
- Must be called by **all** processes in a communicator
- Many collective actions come in two flavors (just like P2P communication)
  - **Blocking** (covered in these slides)
  - **Non-blocking** - just like the "I" version of MPI\_Send; i.e., MPI\_Isend. Not covered here

# Collective Actions

- Three types of MPI Collective Actions:
  - Collective **Synchronization** (barrier synchronization)
  - Collective **Communication** (broadcast, scatter, gather, etc.)
  - Collective **Operation** (reduce, prefix scan, user-defined, etc.)

# Collective Actions: Synchronization Barrier

- Used implicitly or explicitly to synchronize processes



# Collective Actions: Synchronization Barrier

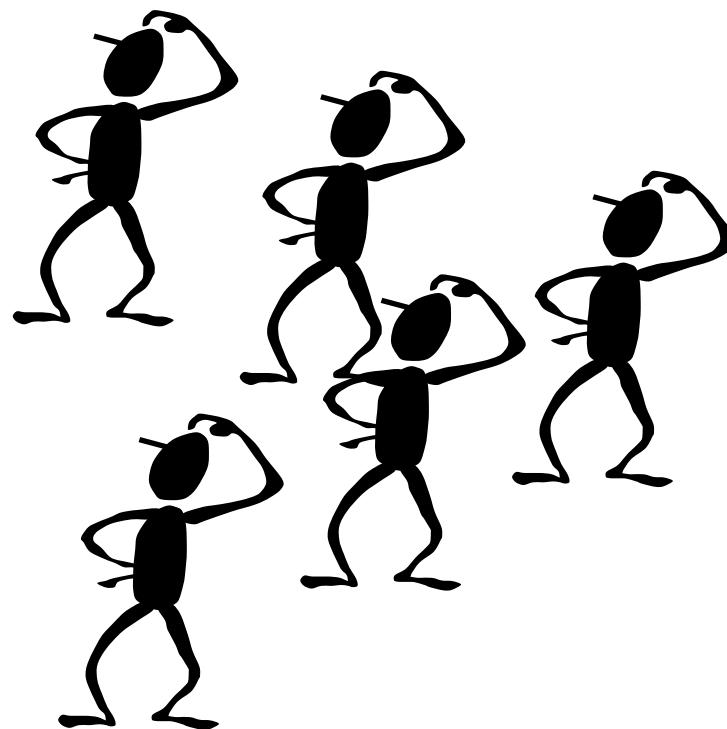
- Function prototype:

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

- **MPI\_Barrier** not needed that often:
  - All synchronization is done automatically by data communication (when using blocking actions)
    - When using blocking actions, a process cannot continue before it has the data that it needs
  - If barriers used for debugging, remember to remove for production release

# Collective Actions: Communication, Broadcast-type

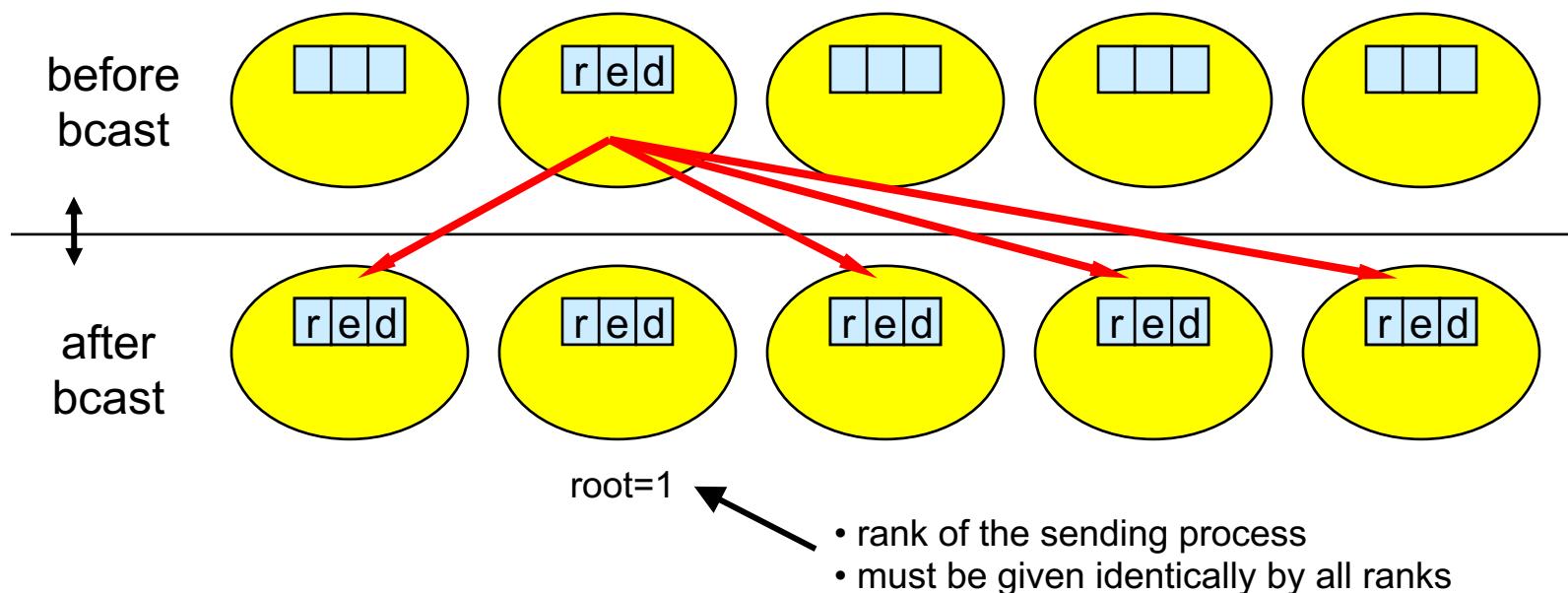
- Broadcast: A one-to-many communication



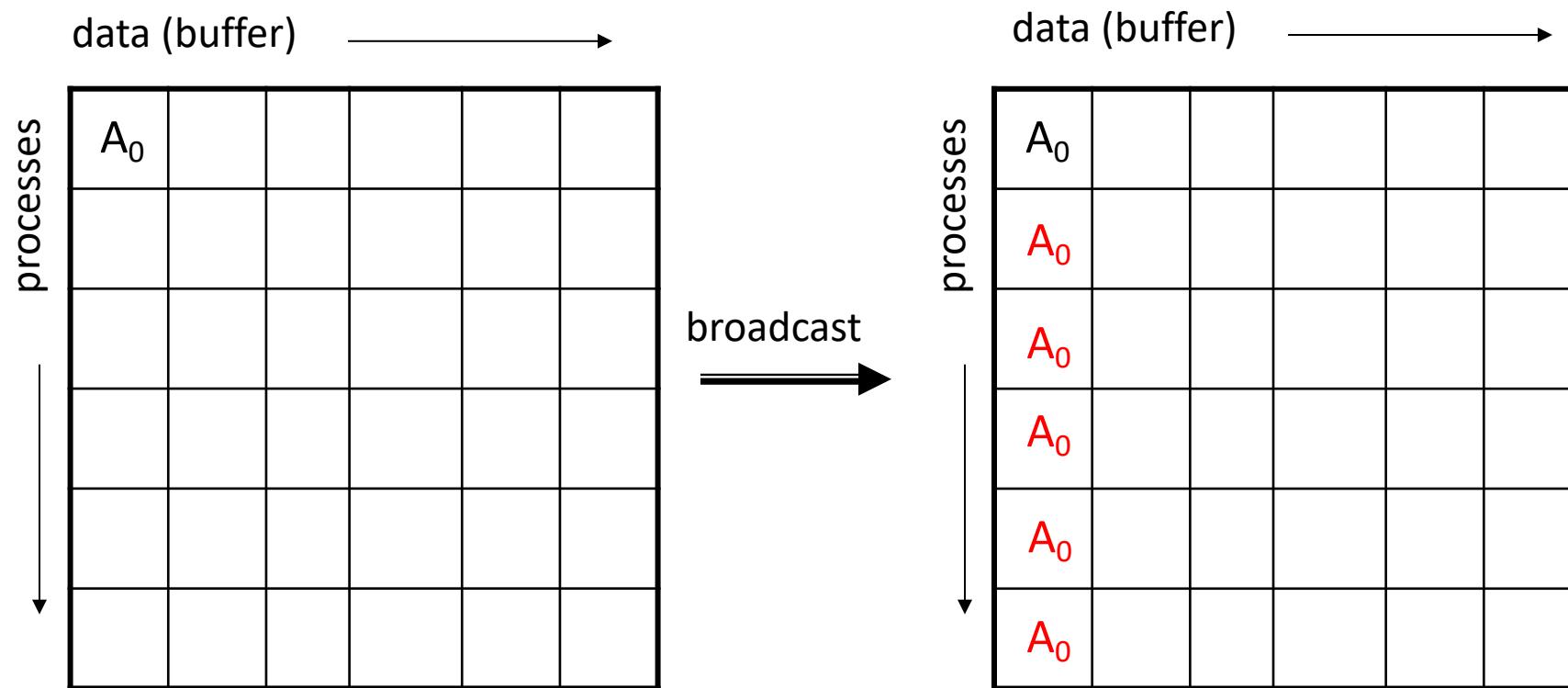
# Communication Action: Broadcast

- Function prototype:

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



# MPI\_Bcast



$A_0$  : any chunk of contiguous data described with MPI\_Datatype and count

# MPI\_Bcast

- Function prototype:

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

INOUT : **buffer** (starting address, as usual)

IN : **count** (number of entries in buffer)

IN : **type** (can be user-defined)

IN : **root** (rank of broadcast root)

IN : **com** (communicator)

- Broadcasts message from **root** to all processes (including **root**)
- **comm** and **root** must assume identical values in all processes
- On return, contents of **buffer** is copied to all processes in **comm**

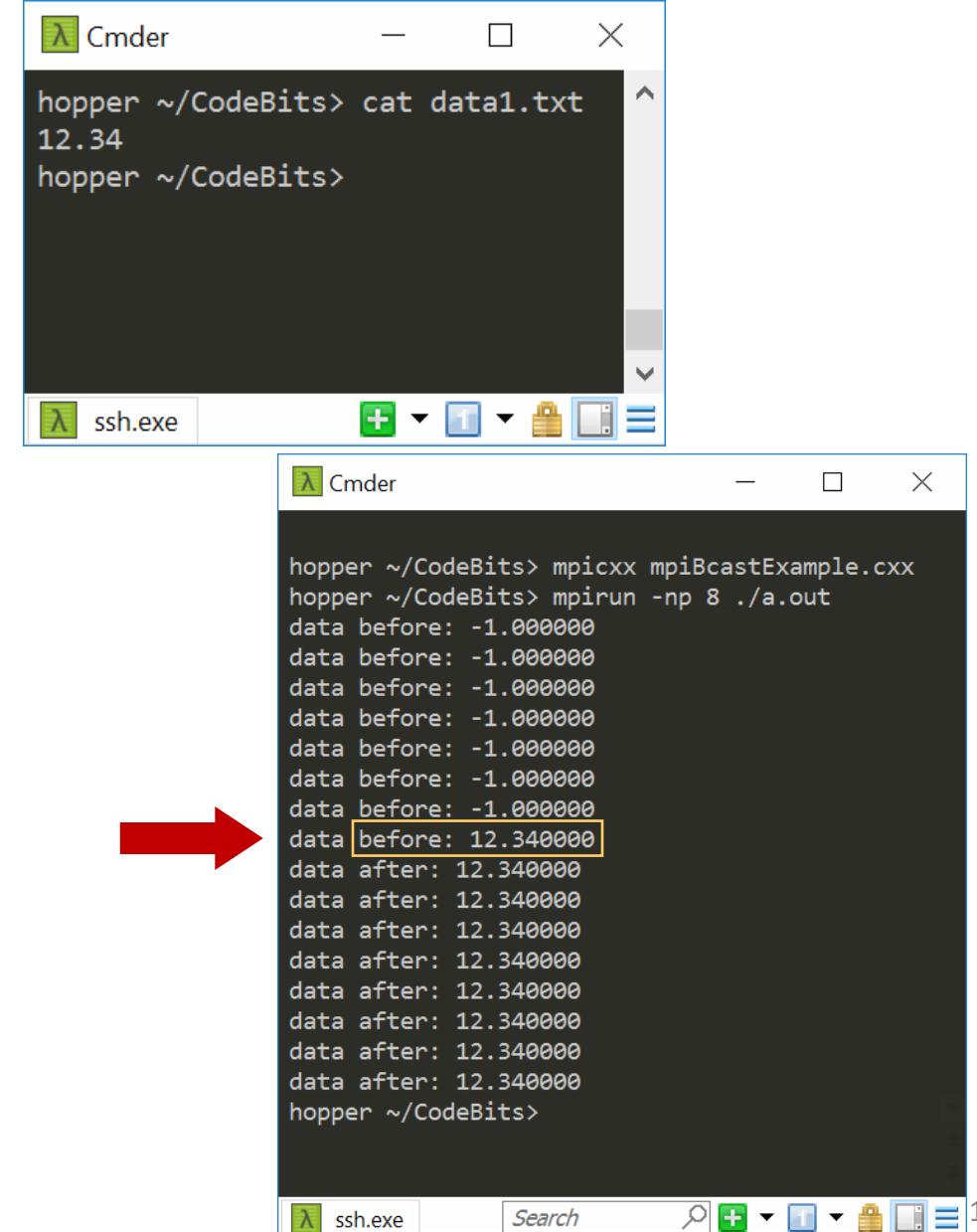
# Example, MPI\_Bcast

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
// Read parameter from file and send to all processes
void main(int argc, char** argv){
    int myRank, nprocs;
    float data = -1.0;
    FILE* file;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    if( myRank==0 ) {
        char input[100];
        file = fopen("data1.txt", "r");
        fscanf(file, "%s\n", input);
        fclose(file);
        data = atof(input);
    }
    printf("data before: %f\n", data);
    MPI_Bcast(&data, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    printf("data after: %f\n", data);

    MPI_Finalize();
}
```



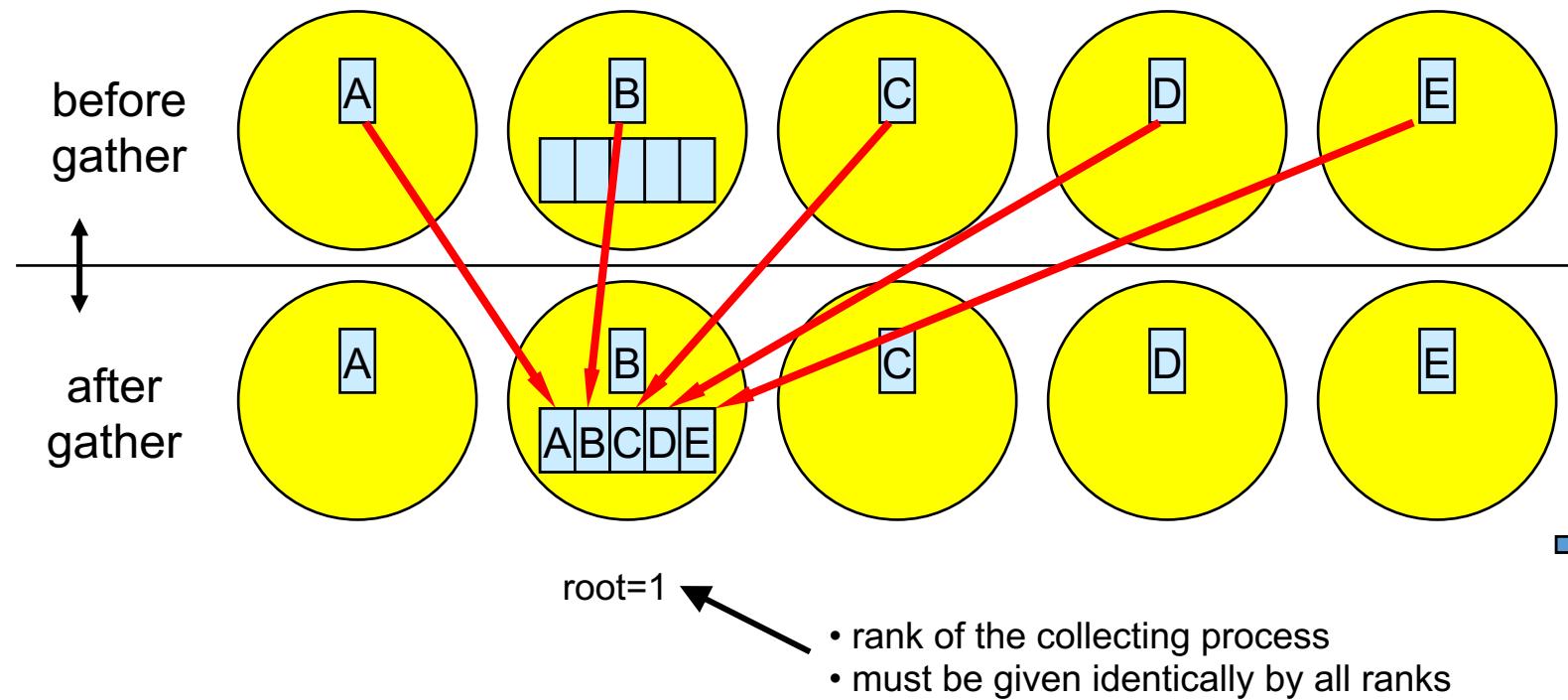
The screenshot shows two terminal windows side-by-side. The left window is titled 'ssh.exe' and displays the C code for MPI\_Bcast. The right window is titled 'Cmder' and shows the execution of the code. A red arrow points from the left window to the right window, indicating the flow of data. The right window's output shows the initial value of 'data' (12.34) followed by multiple iterations of the broadcast process, where 'data' is updated to 12.34 at each rank.

```
hopper ~/CodeBits> cat data1.txt
12.34
hopper ~/CodeBits>
hopper ~/CodeBits> mpicxx mpiBcastExample.cxx
hopper ~/CodeBits> mpirun -np 8 ./a.out
data before: -1.000000
data before: 12.340000
data after: 12.340000
hopper ~/CodeBits>
```

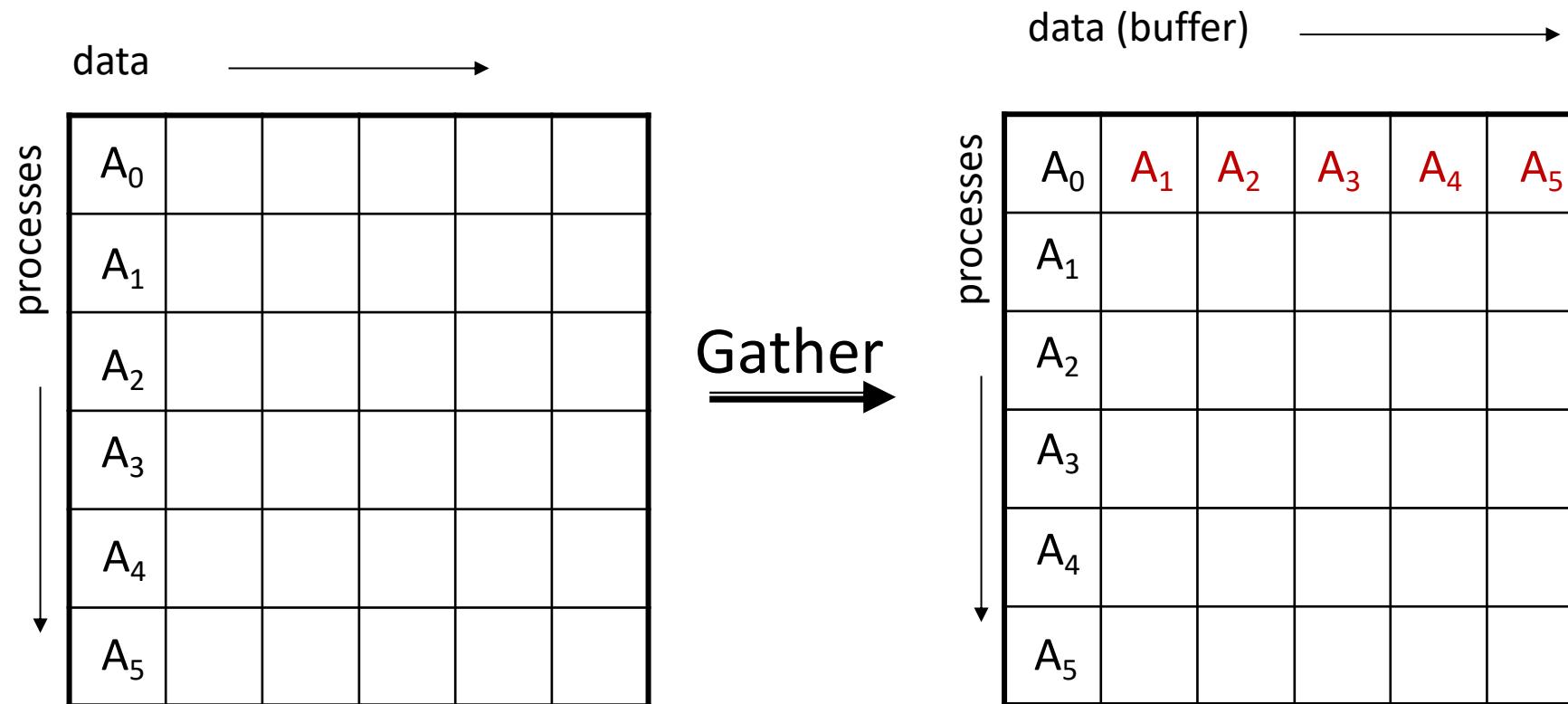
# Communication Action: Gather

- Function Prototype

```
int MPI_Gather(void *sndbuf, int sndcount, MPI_Datatype sndtype, void *rcvbuf, int rcvcount, MPI_Datatype rcvtype, int root, MPI_Comm comm);
```



# MPI\_Gather



# MPI\_Gather

- Function prototype:

```
int MPI_Gather (void *sndbuf, int sndcount, MPI_Datatype sndtype, void *rcvbuf, int rcvcount, MPI_Datatype rcvtype, int root, MPI_Comm comm);
```

- IN **sndbuf** (starting address of send buffer)
- IN **sndcount** (number of elements in send buffer)
- IN **sndtype** (type)
- OUT **rcvbuf** (address of receive buffer)
- IN **rcvcount** (number of elements **gathered from each party**)
- IN **rcvtype** (data type of recv buffer elements)
- IN **root** (rank of receiving process)
- IN **comm** (communicator)

# MPI\_Gather

- Each process sends content of `sndbuf` to the root process
- NOTE: Root receives and stores in **rank order**
- Remarks:
  - Receive buffer argument ignored for all non-root processes (also `rcvtype`, etc.)
  - `rcvcount` on root indicates number of items received **from each process**, not total. Common error

# Example, MPI\_Gather

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
void main(int argc, char** argv){
    int myRank, nprocs, n_loc=2, i;
    float *data, *data_loc;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

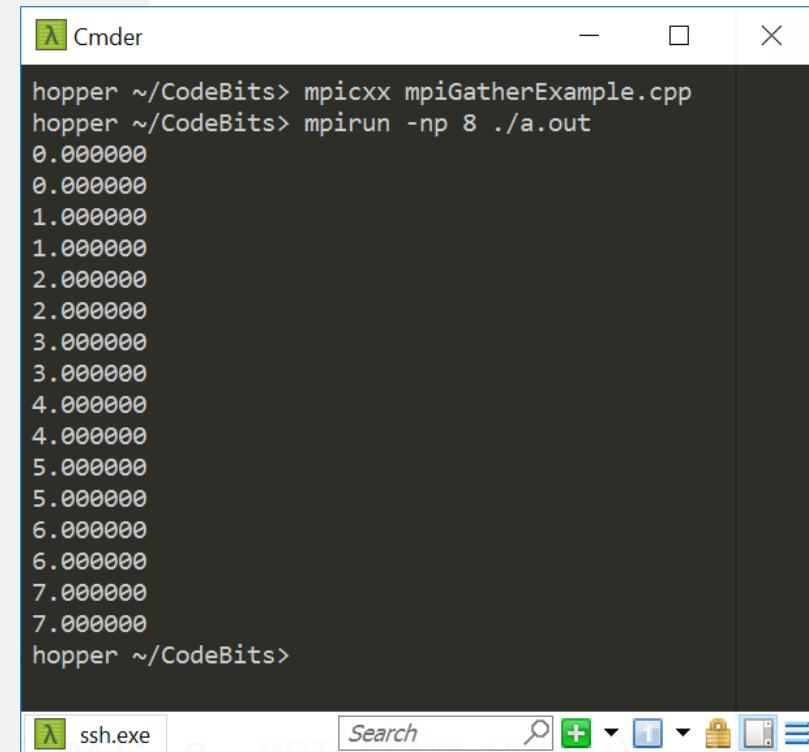
    /* local array size on each proc = n_loc */
    data_loc = (float*) malloc(n_loc*sizeof(float));
    for (i = 0; i < n_loc; i++) data_loc[i] = myRank;

    if (myRank == 0)
        data = (float*) malloc(nprocs * n_loc*sizeof(float));

    MPI_Gather(data_loc, n_loc, MPI_FLOAT, data, n_loc, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if (myRank == 0)
        for (i = 0; i < n_loc*nprocs; i++)
            printf("%f\n", data[i]);

    MPI_Finalize();
}
```



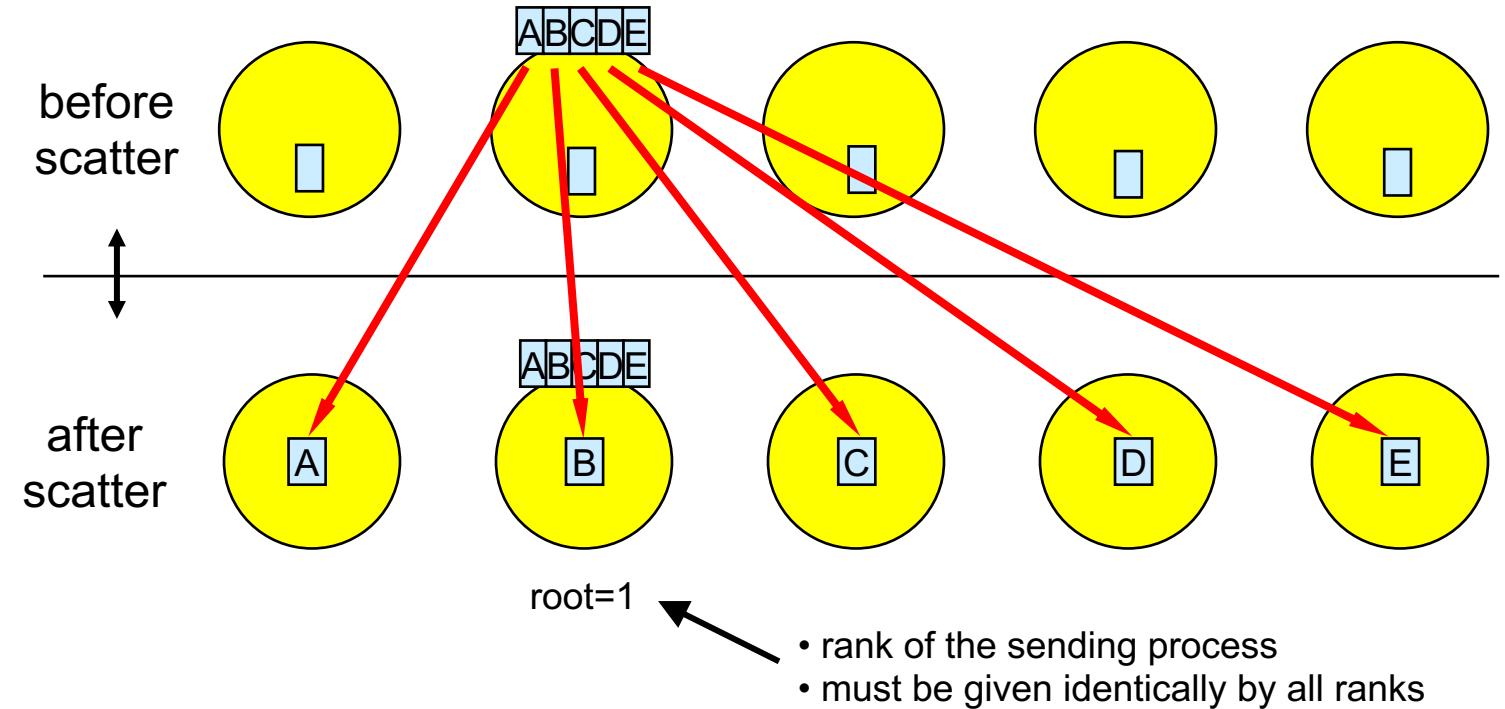
The terminal window shows the following session:

```
hopper ~/CodeBits> mpicxx mpiGatherExample.cpp
hopper ~/CodeBits> mpirun -np 8 ./a.out
0.00000
0.00000
1.00000
1.00000
2.00000
2.00000
3.00000
3.00000
4.00000
4.00000
5.00000
5.00000
6.00000
6.00000
7.00000
7.00000
hopper ~/CodeBits>
```

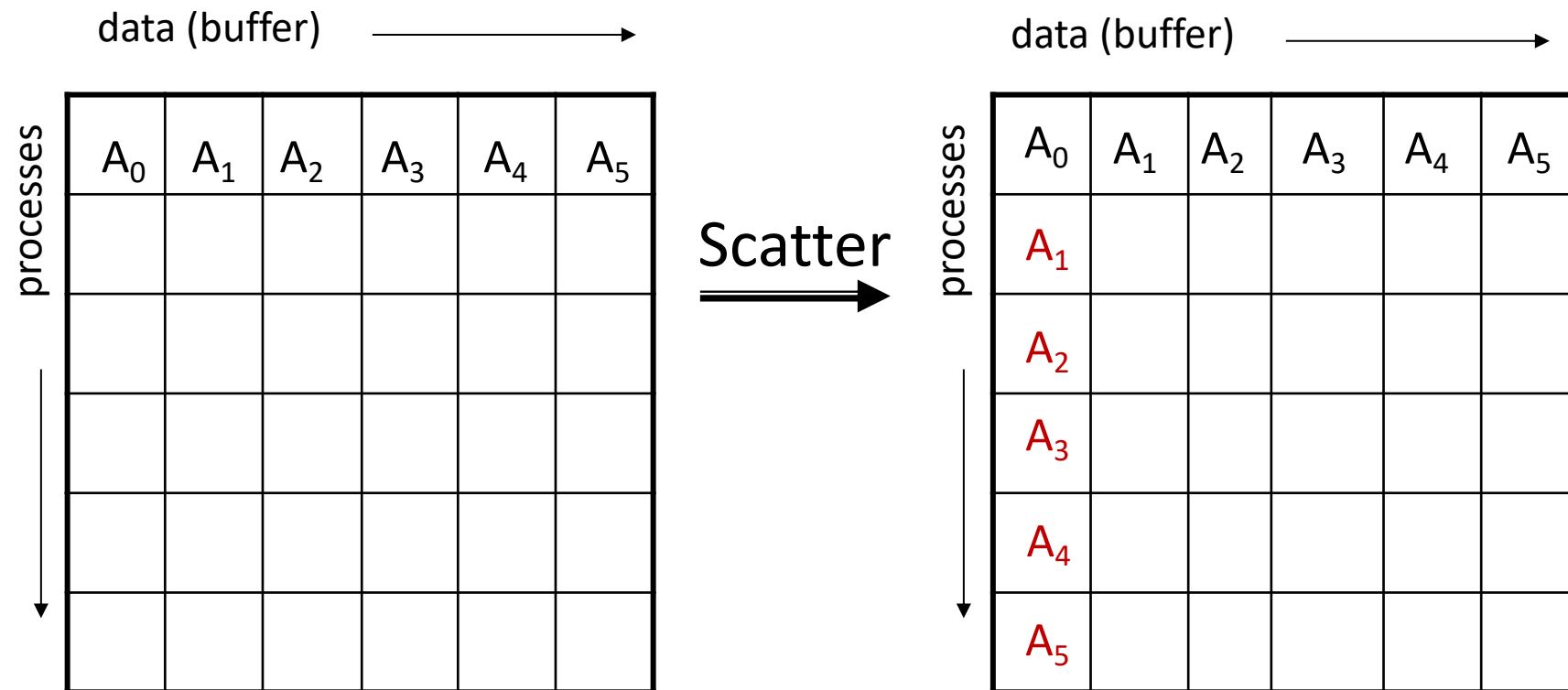
# Communication Action: Scatter

- Function prototype

```
int MPI_Scatter(void *sndbuf, int sndcount, MPI_Datatype sndtype, void *rcvbuf, int rcvcount, MPI_Datatype rcvtype, int root, MPI_Comm comm);
```



# MPI\_Scatter



# MPI\_Scatter

- Function prototype

```
int MPI_Scatter(void *sndbuf, int sndcount, MPI_Datatype sndtype, void *rcvbuf, int rcvcount, MPI_Datatype rcvtype, int root, MPI_Comm comm);
```

- IN `sndbuf` (starting address of send buffer)
- IN `sndcount` (number of elements **scattered to each process**)
- IN `sndtype` (type)
- OUT `rcvbuf` (address of receive bufer)
- IN `rcvcount` (number of elements **in receive buffer**)
- IN `rcvtype` (data type of receive elements)
- IN `root` (rank of sending process)
- IN `comm` (communicator)

# MPI\_Scatter

- Inverse of **MPI\_Gather**
- Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter
- Remarks:
  - All arguments are significant on **root**, while on other processes only **recvbuf**, **recvcount**, **recvtype**, **root**, and **comm** are significant

# Example, MPI\_Scatter

```
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
void main(int argc, char **argv){
    int myRank, nprocs, n_lcl=2;
    float *data, *data_l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    /* local array size on each proc = n_lcl */
    data_l = (float *) malloc(n_lcl*sizeof(float));

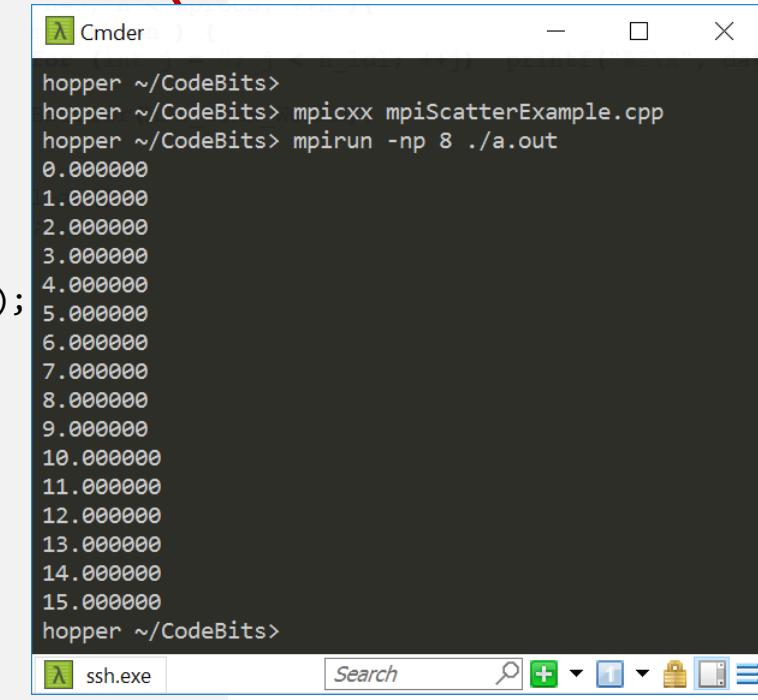
    if( myRank==0 ) {
        data = (float *) malloc(nprocs*sizeof(float)*n_lcl);
        for( int i = 0; i < nprocs*n_lcl; ++i) data[i] = i;
    }

    MPI_Scatter(data, n_lcl, MPI_FLOAT, data_l, n_lcl, MPI_FLOAT, 0, MPI_COMM_WORLD);

    for( int n=0; n < nprocs; ++n ){
        if( myRank==n ) {
            for (int j = 0; j < n_lcl; ++j) printf("%f\n", data_l[j]);
        }
        MPI_Barrier(MPI_COMM_WORLD); // this is interesting...
    }

    MPI_Finalize();
}
```

This is interesting...  
What happens here?



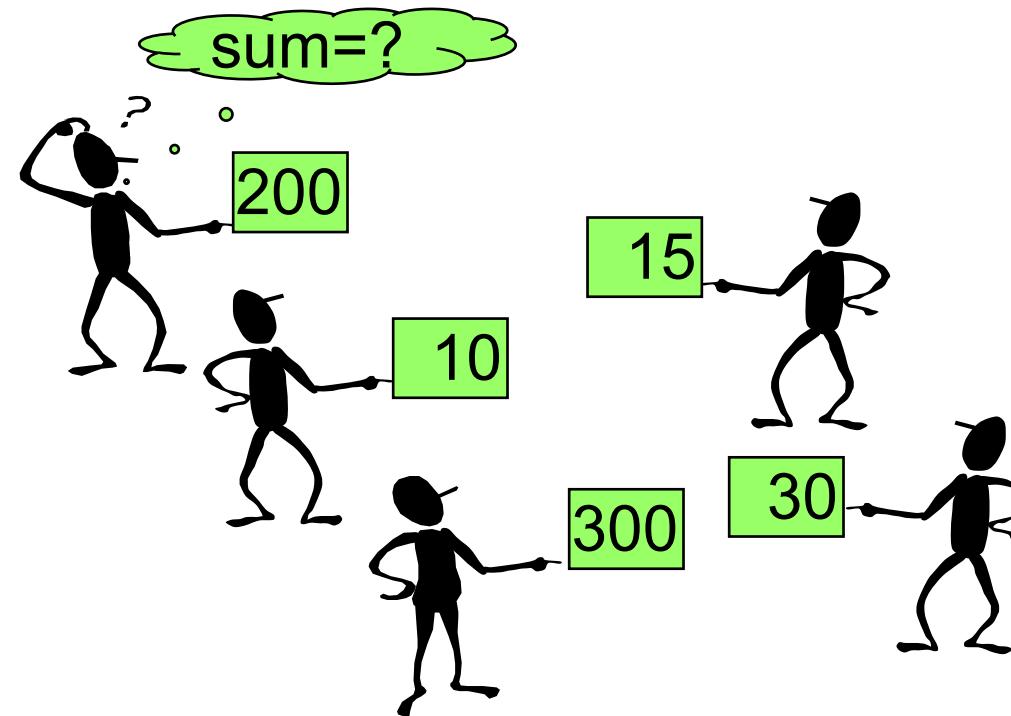
```
hopper ~/CodeBits>
hopper ~/CodeBits> mpicxx mpiScatterExample.cpp
hopper ~/CodeBits> mpirun -np 8 ./a.out
0.00000
1.00000
2.00000
3.00000
4.00000
5.00000
6.00000
7.00000
8.00000
9.00000
10.00000
11.00000
12.00000
13.00000
14.00000
15.00000
hopper ~/CodeBits>
```

# MPI Communication Actions: Putting Things in Perspective...

- Scatter: create a distributed array from a serial one
  - Also a more general form: `MPI_Scatterv`
    - Gaps are allowed between messages in source data
      - But the individual messages must still be contiguous
    - Irregular message sizes are allowed
    - Data can be distributed to processes in any order
- Gather: create a serial array from a distributed one
  - Also a more general form: `MPI_Gatherv`
    - Counterpart to `MPI_Scatterv`
  - NOTE: there is a `MPI_Allgather` flavor, more on this later (Allreduce)

[New subtopic] **Collective Actions: Operations**

- Combine data from several processes to produce a single result



# Global Reduction Operations

- Performs a global reduce operation across all members of a group
  - Abstracted as:  $d_0 \textcolor{red}{o} d_1 \textcolor{red}{o} d_2 \textcolor{red}{o} d_3 \textcolor{red}{o} \dots \textcolor{red}{o} d_{s-2} \textcolor{red}{o} d_{s-1}$ 
    - $d_i$  = data in process rank i
      - single variable, or
      - vector
    - $\textcolor{red}{o}$  = associative operation
  - Example:
    - global sum or product
    - global maximum or minimum
    - global user-defined operation
- NOTE: Floating point rounding may depend on usage of associative law
  - These two expressions might produce different results:
    - $[(d_0 \textcolor{red}{o} d_1) \textcolor{red}{o} (d_2 \textcolor{red}{o} d_3)] \textcolor{red}{o} [\dots \textcolor{red}{o} (d_{s-2} \textcolor{red}{o} d_{s-1})]$
    - $(((((d_0 \textcolor{red}{o} d_1) \textcolor{red}{o} d_2) \textcolor{red}{o} d_3) \textcolor{red}{o} \dots) \textcolor{red}{o} d_{s-2}) \textcolor{red}{o} d_{s-1}$

# Example of Global Reduction

- Global integer sum
- Sum of all `inbuf` values should be returned in `resultbuf`

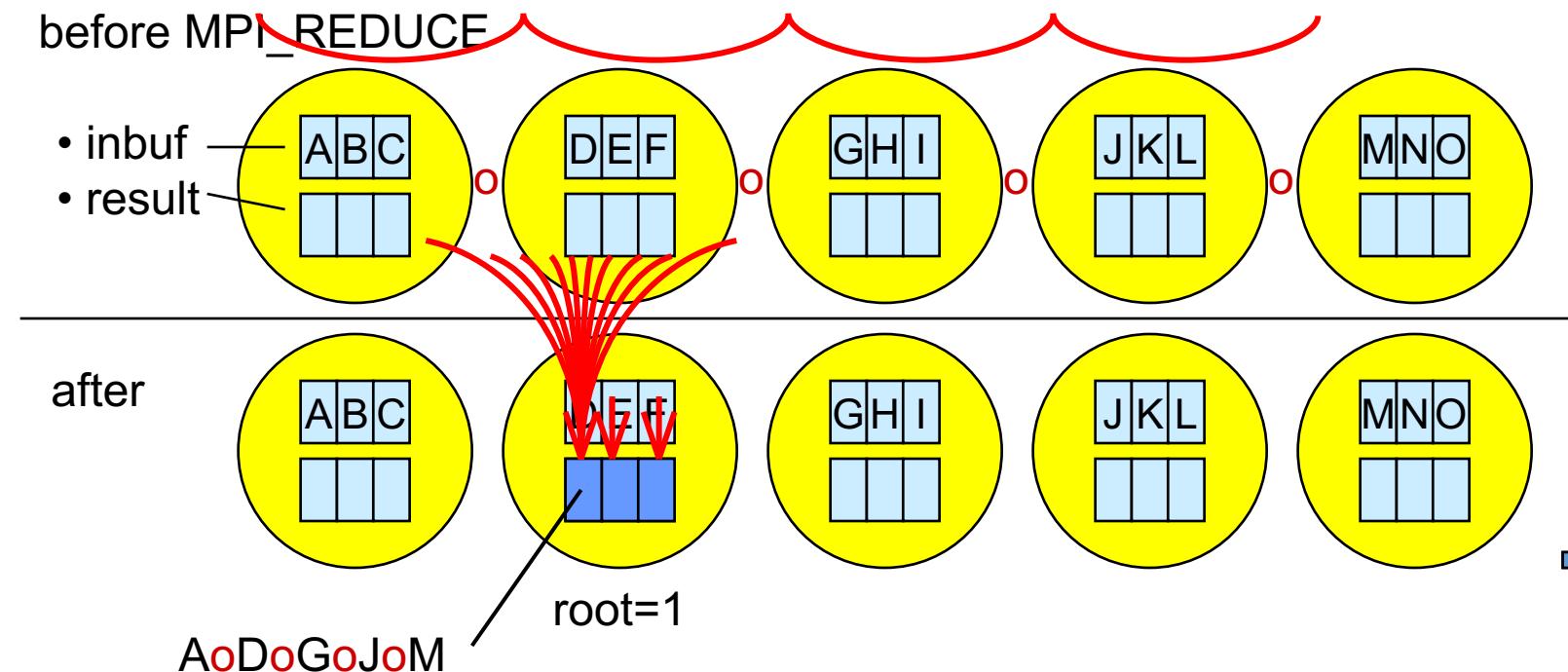
```
MPI_Reduce(&inbuf, &resultbuf, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
```

- The result is only placed in `resultbuf` at the `root` process.

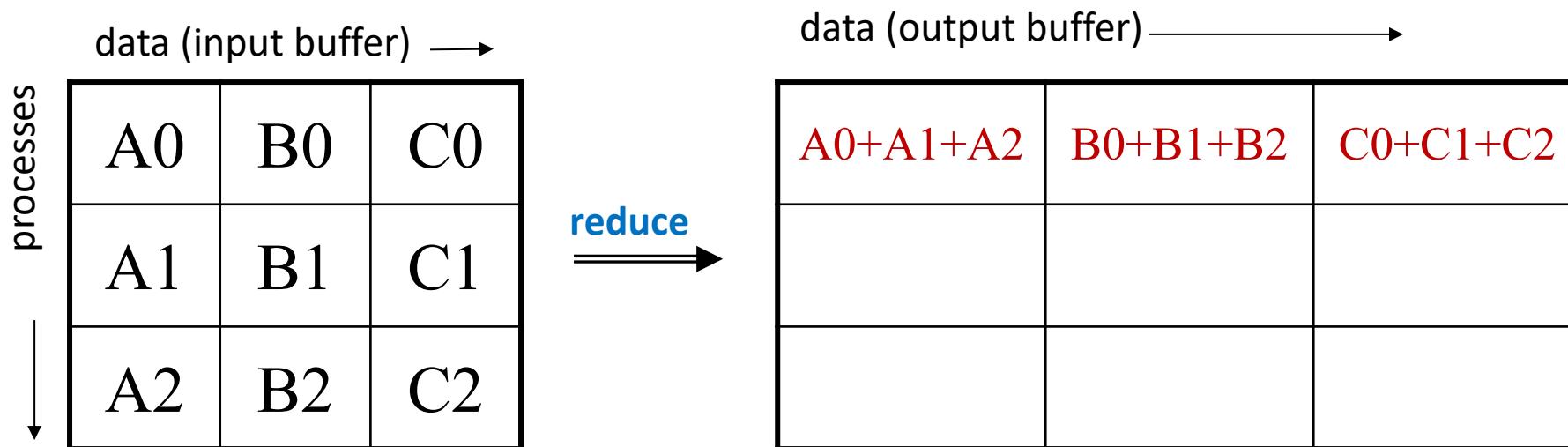
# Predefined Reduction Operations

Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

# MPI\_Reduce

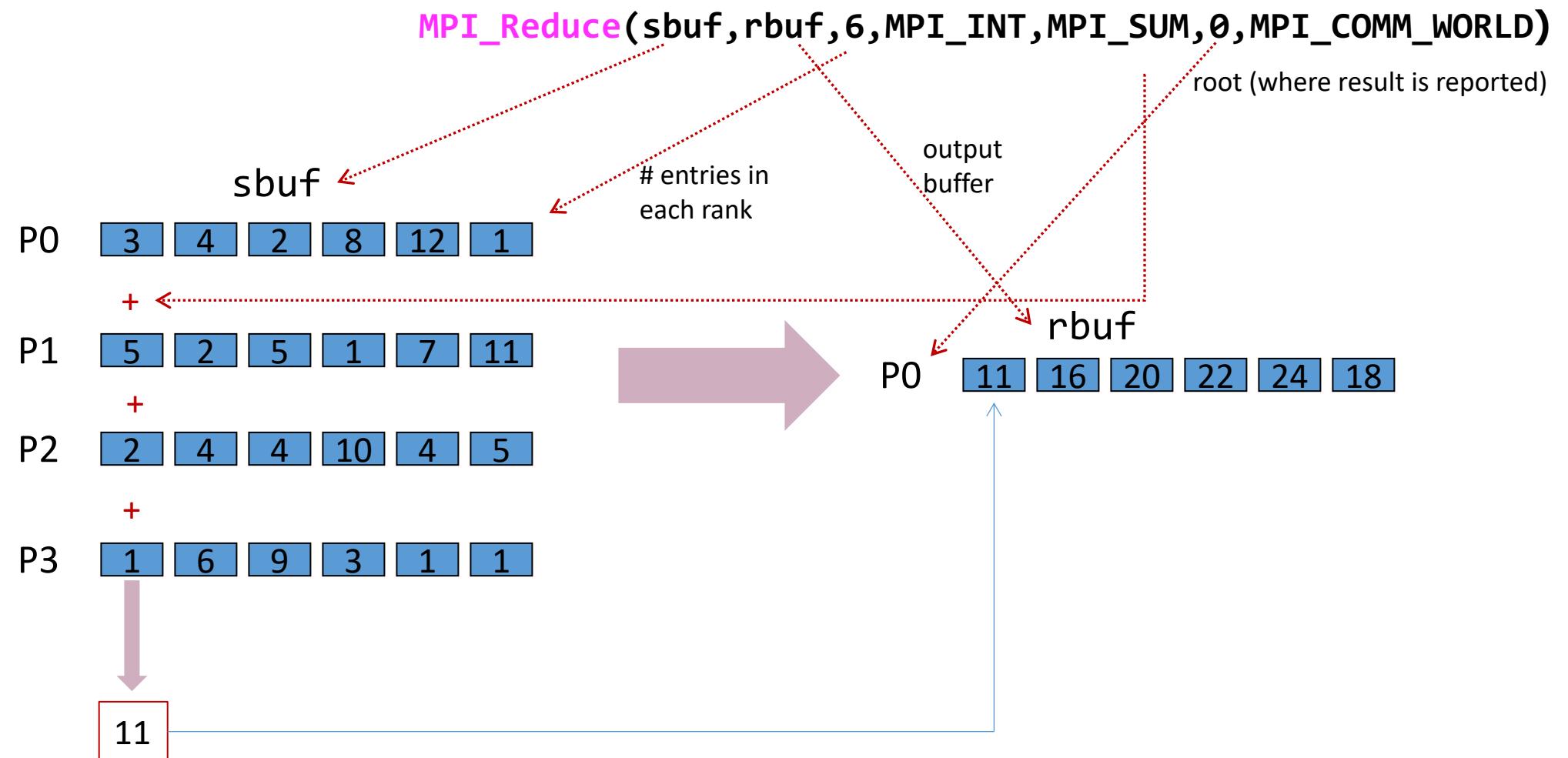


# Reduce Operation



Assumption: Rank 0 is the root

# Example, MPI\_Reduce



# MPI\_Reduce

- Function prototype

```
int MPI_Reduce(void *sndbuf, void *rcvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

- IN  **sndbuf** (address of send buffer)
- OUT  **rcvbuf** (address of receive buffer)
- IN  **count** (number of elements in send buffer)
- IN  **datatype** (data type of elements in send buffer)
- IN  **op** (reduce operation)
- IN  **root** (rank of root process)
- IN  **comm** (communicator)

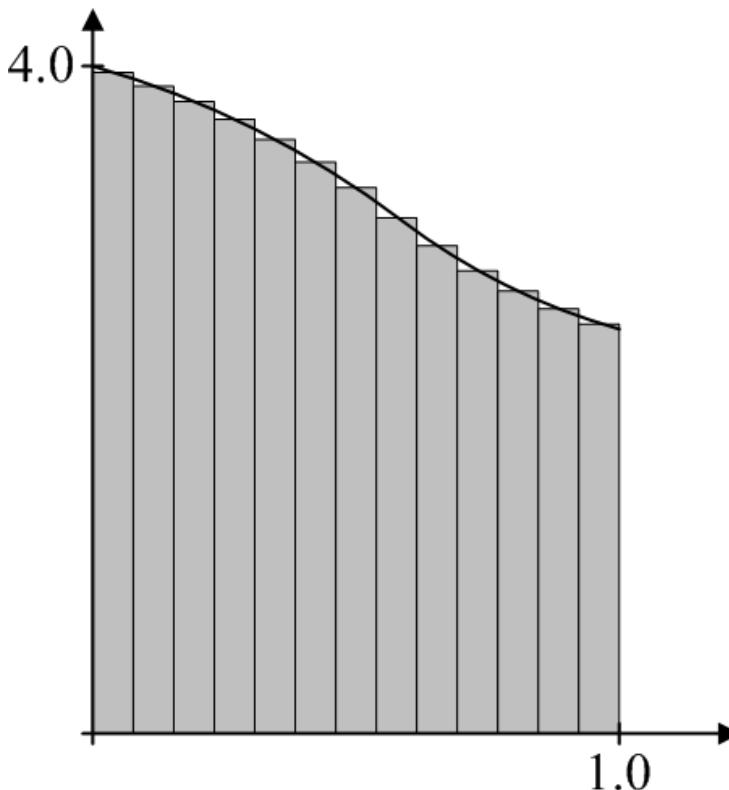
# MPI Example: computing an integral (approximating $\pi$ )

$$\int_0^1 \frac{4}{1+x^2} dx = 4 \tan^{-1}(1) = \pi$$

Numerical Integration: Midpoint rule

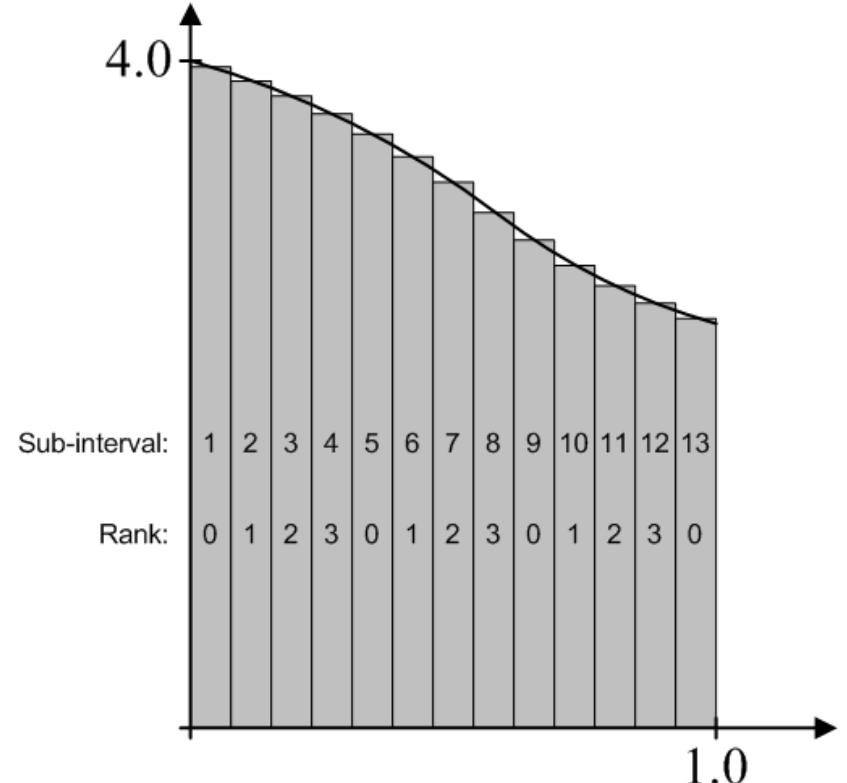
$$\int_0^1 f(x) dx \approx \sum_{i=1}^n h f((i - 0.5)h)$$

$$h = \frac{1}{n}$$



# MPI Example: computing an integral (approximating $\pi$ )

- For sake of this discussion, assume 4 MPI processes (rank 0 through 3)
- In the picture, the interval  $[0,1]$  split into  $n=13$  sub-intervals
- Sub-intervals are assigned to ranks in a round-robin manner
  - Rank 0: handles sub-intervals 1,5,9,13
  - Rank 1: handles sub-intervals 2,6,10
  - Rank 2: handles sub-intervals 3,7,11
  - Rank 3: handles sub-intervals 4,8,12
- Each rank computes the area in its associated sub-intervals
- **`MPI_Reduce`** is used to sum the areas computed by each rank yielding final approximation to  $\pi$



# MPI Example: computing an integral (approximating $\pi$ )

```
#include "mpi.h"
#include <math.h>
#include <iostream>

using namespace std;

void main(int argc, char *argv[])
{
    int n, rank, size, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int namelen;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(processor_name, &namelen);

    cout << "Hello from process " << rank << " of " << size << " on " << processor_name << endl;
```

# MPI Example: computing an integral (approximating $\pi$ )

```
if (rank == 0) {  
    if (argc<2 || argc>2)  
        n=0;  
    else  
        n=atoi(argv[1]);  
}  
  
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);  
if (n>0) {  
    h = 1.0 / (double) n;  
    sum = 0.0;  
    for (i = rank + 1; i <= n; i += size) {  
        x = h * ((double)i - 0.5);  
        sum += (4.0 / (1.0 + x*x));  
    }  
    mypi = h * sum;  
  
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);  
    if (rank == 0)  
        cout << "pi is approximately " << pi << ", Error is " << fabs(pi - PI25DT) << endl;  
}  
MPI_Finalize();
```

How many instances of this data type are moved around

Data type we are moving around

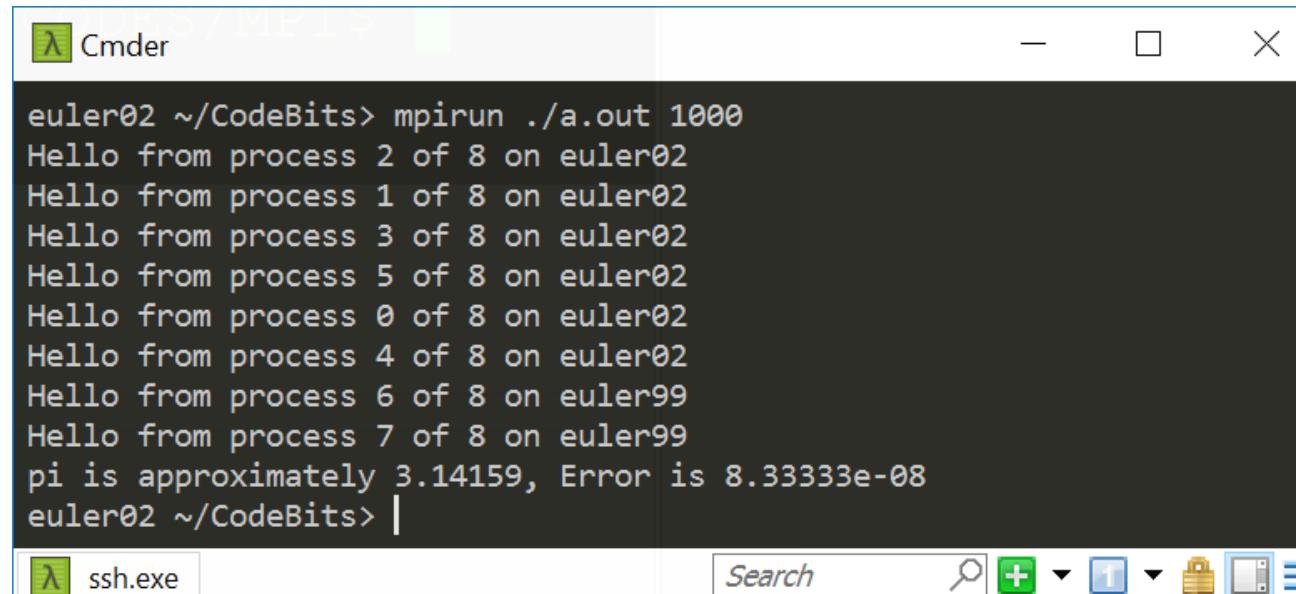
Reduce through a “sum” operation

Root process, it ends up storing the result

Partial contribution of “this” process

Where the reduce operation stores the result

# Program Output [uses 8 ranks to compute integral]

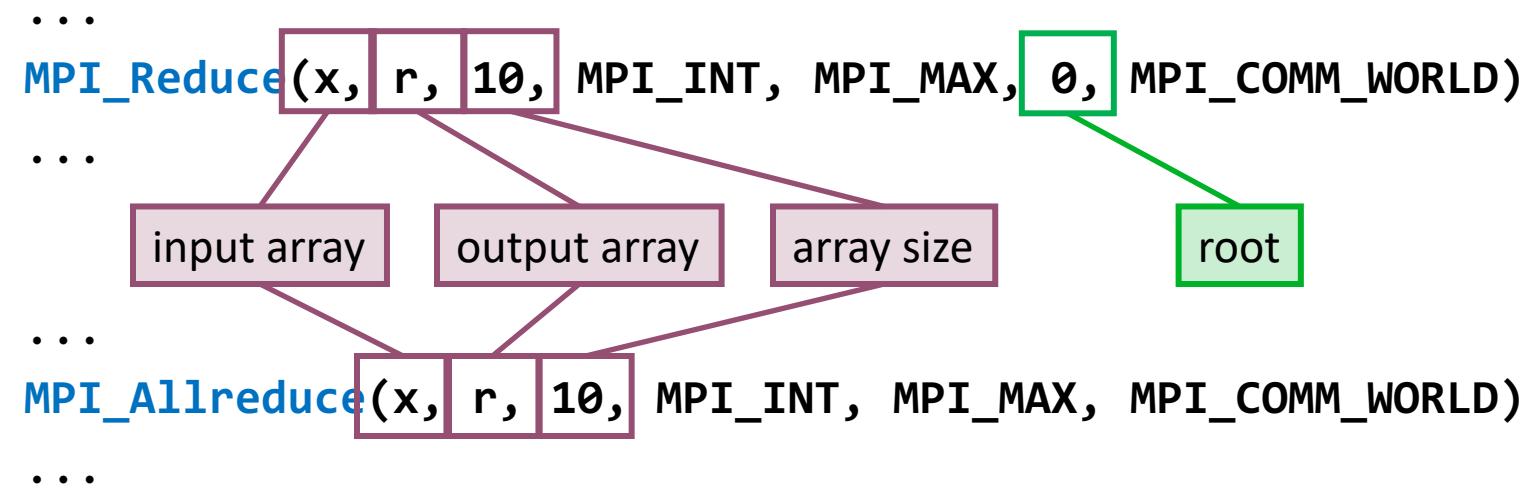


The screenshot shows a Cmder terminal window with the title bar "λ Cmder". The command entered is "euler02 ~/CodeBits> mpirun ./a.out 1000". The output consists of eight "Hello from process" messages, one for each rank (0 to 7), followed by the calculated value of pi and its error. The terminal has a dark theme with light-colored text. The bottom bar includes tabs for "ssh.exe", a search bar, and various icons.

```
euler02 ~/CodeBits> mpirun ./a.out 1000
Hello from process 2 of 8 on euler02
Hello from process 1 of 8 on euler02
Hello from process 3 of 8 on euler02
Hello from process 5 of 8 on euler02
Hello from process 0 of 8 on euler02
Hello from process 4 of 8 on euler02
Hello from process 6 of 8 on euler99
Hello from process 7 of 8 on euler99
pi is approximately 3.14159, Error is 8.33333e-08
euler02 ~/CodeBits> |
```

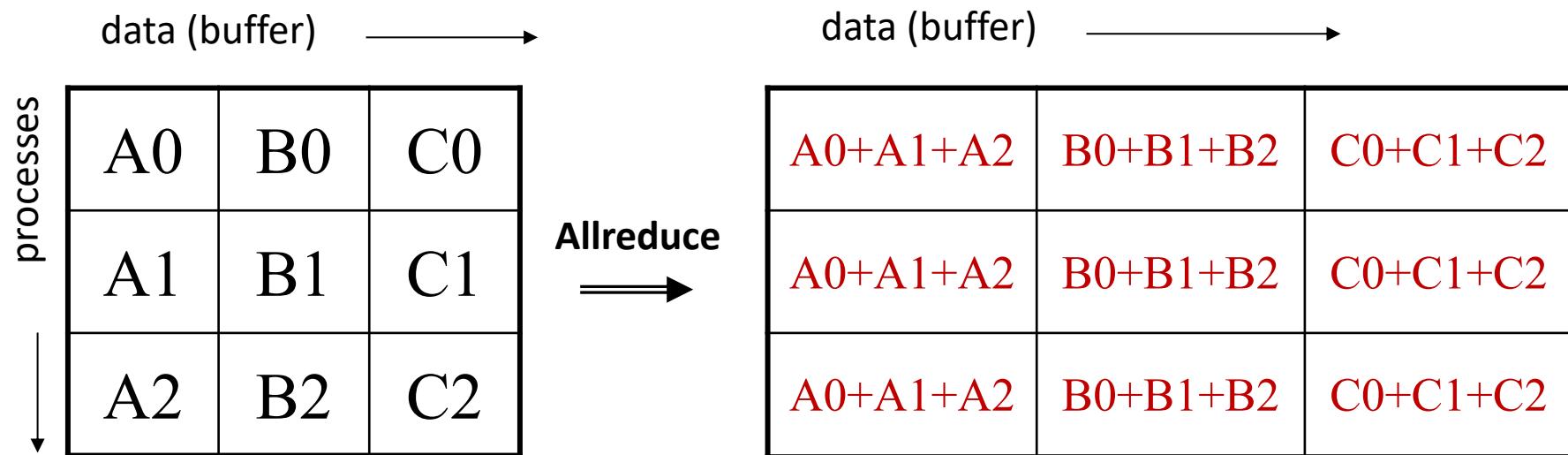
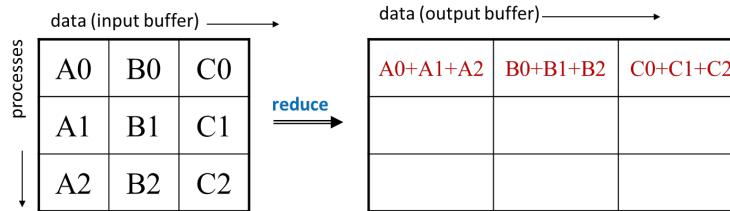
# MPI\_Reduce, MPI\_Allreduce

- **MPI\_Reduce**: result is collected by the root only
  - The operation is applied element-wise for each element of the input arrays on each processor
- **MPI\_Allreduce**: result is sent out to all ranks in the communicator



# MPI\_Allreduce

Here's what MPI\_Reduce does



# MPI\_Allreduce

- Function prototype

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

- IN **sendbuf** (address of send buffer)
- OUT **recvbuf** (address of receive buffer)
- IN **count** (number of elements in send buffer)
- IN **datatype** (data type of elements in send buffer)
- IN **op** (reduce operation)
- IN **cmm** (communicator)

# Example, MPI\_Allreduce

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

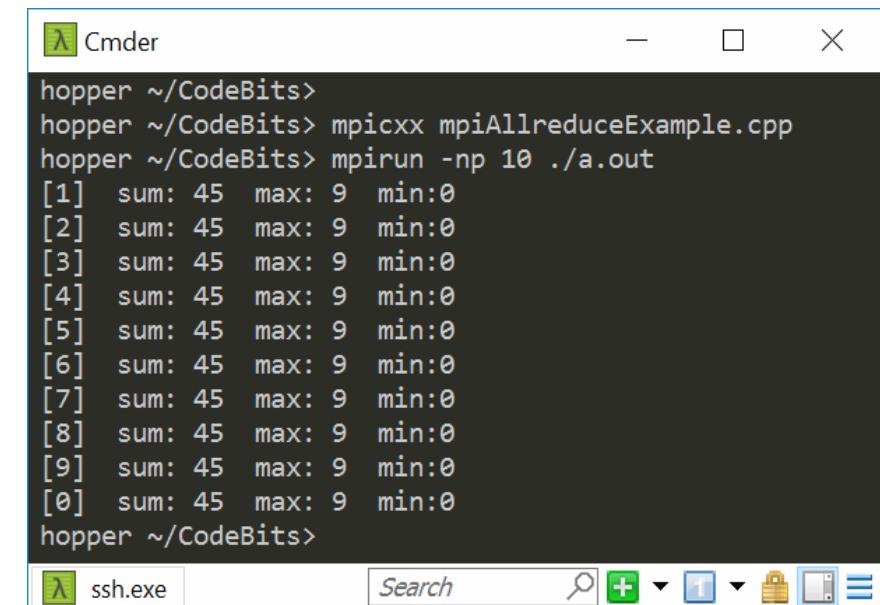
int main(int argc, char **argv) {
    int myRank, nprocs, gsum, gmax, gmin;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    MPI_Allreduce(&myRank, &gsum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&myRank, &gmax, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    MPI_Allreduce(&myRank, &gmin, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

    printf("[%d] sum: %d max: %d min:%d\n", myRank, gsum, gmax, gmin);
    MPI_Finalize();

    return 0;
}
```



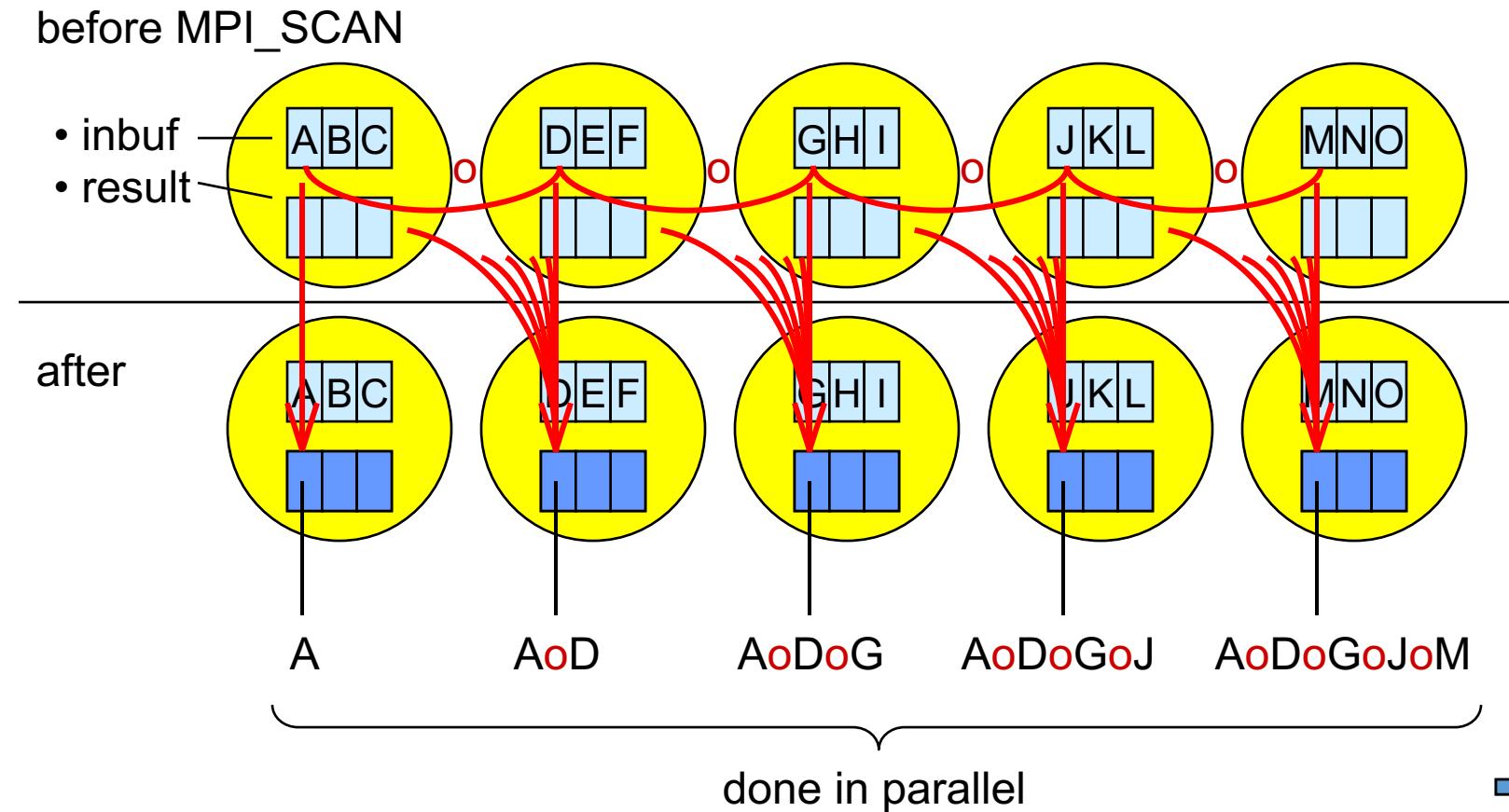
The terminal window titled 'Cmder' displays the execution of a MPI application. It shows the command mpicxx mpiAllreduceExample.cpp followed by mpirun -np 10 ./a.out. The output consists of 10 lines, each containing the rank (0-9), the sum (45), the maximum value (9), and the minimum value (0).

```
hopper ~/CodeBits>
hopper ~/CodeBits> mpicxx mpiAllreduceExample.cpp
hopper ~/CodeBits> mpirun -np 10 ./a.out
[1] sum: 45 max: 9 min:0
[2] sum: 45 max: 9 min:0
[3] sum: 45 max: 9 min:0
[4] sum: 45 max: 9 min:0
[5] sum: 45 max: 9 min:0
[6] sum: 45 max: 9 min:0
[7] sum: 45 max: 9 min:0
[8] sum: 45 max: 9 min:0
[9] sum: 45 max: 9 min:0
[0] sum: 45 max: 9 min:0
hopper ~/CodeBits>
```

# MPI\_Scan

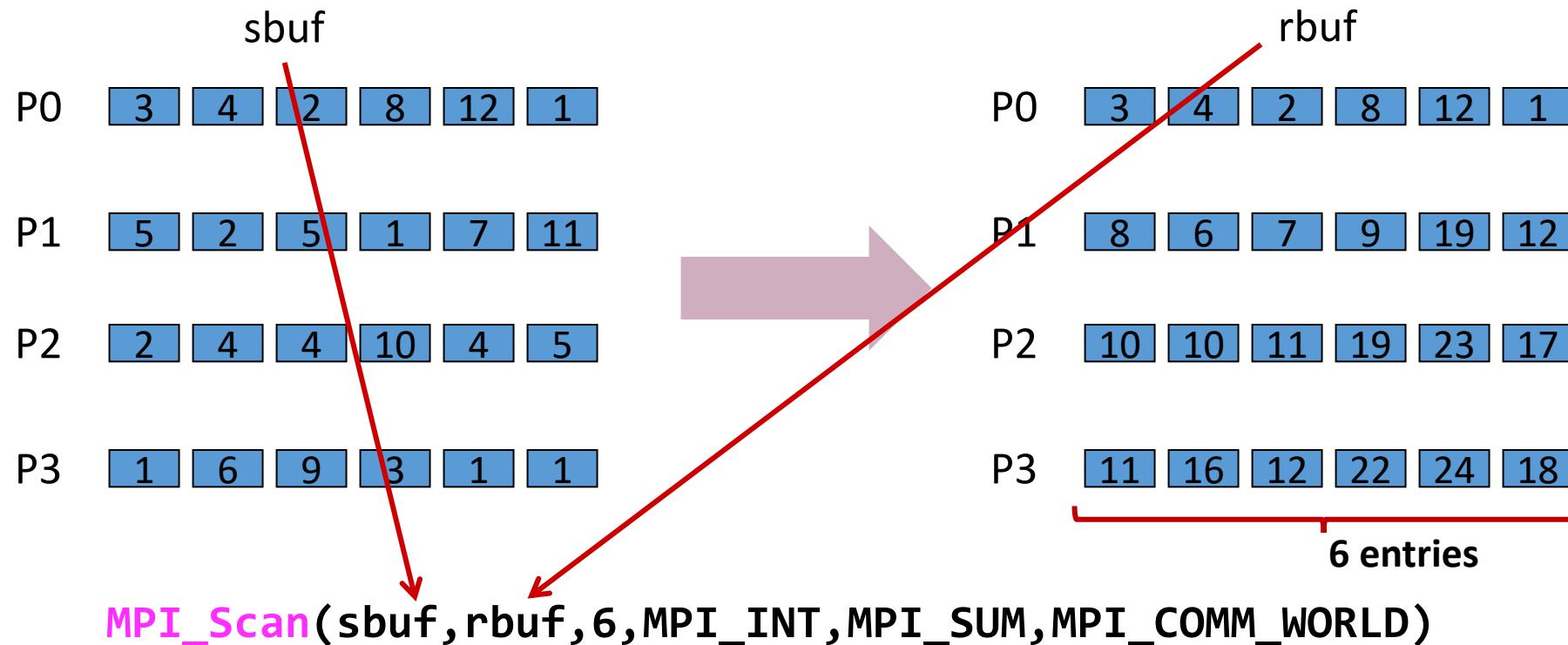
- Performs a prefix reduction on data distributed across a communicator
- The operation returns, in the receive buffer of the process with rank  $i$ , the reduction of the values in the send buffers of processes with ranks  $0, \dots, i$  (inclusive)
- Type of operations supported, their semantics, and constraints on send/receive buffers are as for [MPI\\_REDUCE](#)

# MPI\_Scan

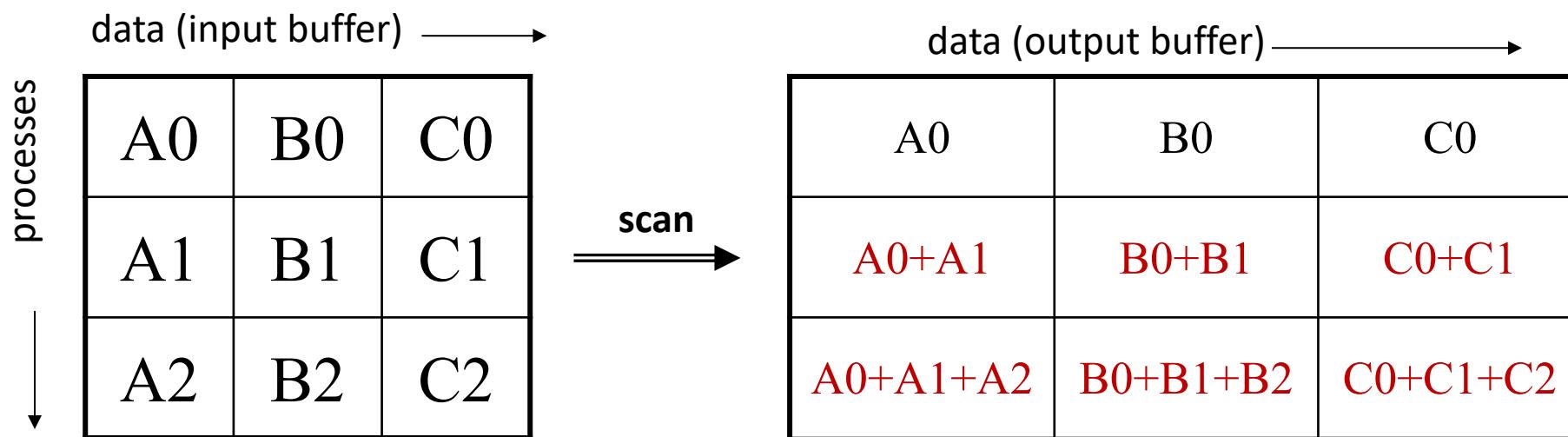


# MPI\_Scan: Quick example

- Process  $i$  receives data reduced on process 0 through  $i$



# Scan Operation [inclusive flavor of it; there's also an exclusive flavor]



# MPI\_Scan

- Function prototype

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

- IN **sendbuf** (address of send buffer)
  - OUT **recvbuf** (address of receive buffer)
  - IN **count** (number of elements in send buffer)
  - IN **datatype** (data type of elements in send buffer)
  - IN **op** (reduce operation)
  - IN **comm** (communicator)
- 
- Note: **count** refers to total number of elements that will be received into **recvbuf** after operation is complete

# Example, MPI\_Scan

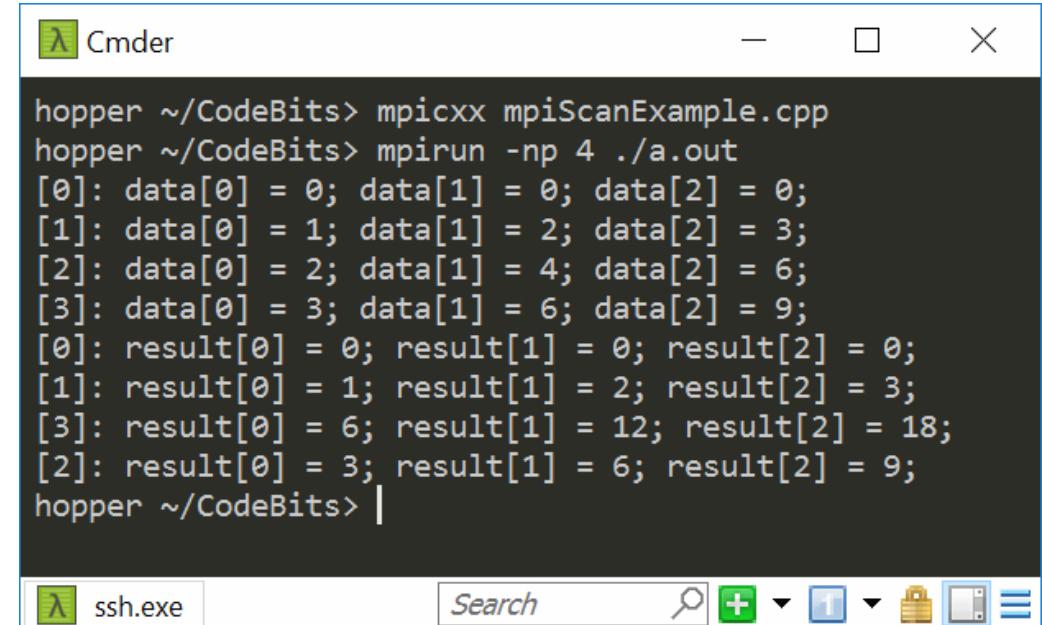
```
#include <mpi.h>
#include <cstdio>
#include <stdio.h>
void main(int argc, char **argv) {
    int myRank, nprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    const int dimArray = 3;
    int* data_1 = new int[dimArray];
    for (int i = 0; i < dimArray; i++) data_1[i] = (i+1)*myRank;
    for (int n = 0; n < nprocs; n++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (myRank == n) {
            printf("[%d]: ", myRank);
            for(int i = 0; i < dimArray; i++) printf("data[%d] = %d; ", i, data_1[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    int* result = new int[dimArray];
    MPI_Scan(data_1, result, dimArray, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (int n = 0; n < nprocs; n++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (myRank == n) {
            printf("[%d]: ", myRank);
            for(int i = 0; i < dimArray; i++) printf("result[%d] = %d; ", i, result[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    delete[] result; delete[] data_1;
}
```



The screenshot shows a terminal window titled "Cmder" running on a Linux system. The command `mpicxx mpiScanExample.cpp` was run to compile the MPI program. Then, `mpirun -np 4 ./a.out` was executed to run the program with 4 processes. The terminal displays the initial data array and the resulting scanned array across the four processes.

```
hopper ~/CodeBits> mpicxx mpiScanExample.cpp
hopper ~/CodeBits> mpirun -np 4 ./a.out
[0]: data[0] = 0; data[1] = 0; data[2] = 0;
[1]: data[0] = 1; data[1] = 2; data[2] = 3;
[2]: data[0] = 2; data[1] = 4; data[2] = 6;
[3]: data[0] = 3; data[1] = 6; data[2] = 9;
[0]: result[0] = 0; result[1] = 0; result[2] = 0;
[1]: result[0] = 1; result[1] = 2; result[2] = 3;
[3]: result[0] = 6; result[1] = 12; result[2] = 18;
[2]: result[0] = 3; result[1] = 6; result[2] = 9;
hopper ~/CodeBits> |
```

The terminal also shows the current working directory as `~/CodeBits` and the command `ssh.exe` in the search bar.

# User-Defined Reduction Operations

- Operator handles
  - Predefined: MPI\_SUM, MPI\_MAX, etc.
  - User-defined
- User-defined operation  $\oplus$ :
  - User-defined function must implement the operation  $\text{valueA} \oplus \text{valueB}$
  - NOTE: the op must be associative; i.e.,  $(\text{valueA} \oplus \text{valueB}) \oplus \text{valueC} = \text{valueA} \oplus (\text{valueB} \oplus \text{valueC})$
- Here's how you should register a user-defined reduction function:

```
MPI_Op_create( MPI_User_function *func, int commute, MPI_Op *op);
```

- **commute** tells the MPI library whether **func** is commutative or not
  - Allows the MPI runtime to do tricks and improve efficiency of the op

# Example, MPI\_Op\_create

```
#include <cmath>
#include <cstdio>
#include <mpi.h>
#include <stdio.h>

void norm1(double *a, double *b, int *len, MPI_Datatype *type) {
    *b = std::abs(*a) + std::abs(*b);
}

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

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

    MPI_Op my_op;
    MPI_Op_create((MPI_User_function *)norm1, 1, &my_op);

    double sendbuf = (my_rank + 1.0) * std::pow(-1.0, my_rank);
    printf("PE %d  val = %f\n", my_rank, sendbuf);

    double recvbuf;
    MPI_Reduce(&sendbuf, &recvbuf, 1, MPI_DOUBLE, my_op, 0, MPI_COMM_WORLD);

    if (my_rank == 0) printf("L1 norm = %f\n", recvbuf);

    MPI_Finalize();
}
```

The terminal window shows the following output:

```
hopper ~/CodeBits> mpicxx mpiCustomOpExample.cpp
hopper ~/CodeBits> mpirun -np 8 ./a.out
PE 5  val = -6.000000
PE 7  val = -8.000000
PE 0  val = 1.000000
PE 1  val = -2.000000
PE 2  val = 3.000000
PE 3  val = -4.000000
PE 4  val = 5.000000
L1 norm = 36.000000
PE 6  val = 7.000000
hopper ~/CodeBits>
```

# thrust code typically much simpler

```
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <cmath>

template <typename T> struct absval {
    __host__ __device__
    T operator()(const T& x) const {
        return fabs(x);
    }
};

int main(void) {
    // initialize host array
    float x[4] = {1.0, -2.0, 3.0, -4.0};

    // transfer to device
    thrust::device_vector<float> d_x(x, x + 4);

    absval<float> unary_op;
    float res = thrust::transform_reduce(d_x.begin(), d_x.end(), unary_op, 0.f, thrust::plus<float>());

    std::cout << res << std::endl;
    return 0;
}
```

# MPI Derived Types

[Describing Non-contiguous and Heterogeneous Data]

# The Relevant Question

- The relevant question that we want to be able to answer:
  - “What’s in your buffer?”
- Communication mechanisms discussed so far allow send/recv of a contiguous buffer of identical elements of predefined data types
- Often one wants to send non-homogenous elements (structure) or chunks that are **not contiguous** in memory
- MPI enables you to define **derived datatypes** to answer the question “What’s in your buffer?”

# MPI Datatypes

- MPI Primitive Datatypes
  - `MPI_CHAR`, `MPI_FLOAT`, `MPI_INTEGER`, etc.
- Derived Data types - can be constructed by four methods:
  - contiguous
  - vector
  - indexed
  - structwhich can be subsequently used in all point-to-point and collective communication
- The motivation: create your own types to suit your needs
  - More convenient
  - More efficient

# Typemaps

- An MPI derived type can be described with a **Typemap**, which specifies:
  - A sequence of **primitive data types**
  - A sequence of integers that represent the **byte displacements**, measured from the beginning of the buffer

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$$

- Displacements are not required to be positive, distinct, or in increasing order (however, negative displacements will precede the buffer)
- Order of items need not coincide with their order in memory, and an item may appear more than once

# Typemaps

Primitive data type	Displacement from buf
$type_0$	$disp_0$
$type_1$	$disp_1$
...	...
$type_i$	$disp_i$
...	...
$type_{n-1}$	$disp_{n-1}$

Entry  $i$  has type  $type_i$  and displacement  $buf + disp_i$

# Extent

- Extent: distance, in bytes, from beginning to end of type
- More specifically, the **extent** of a data type is defined as:  
... the span from the first byte to the last byte occupied by entries in this data type rounded up to satisfy alignment requirements
- Example:
  - Typemap={{**double**,0},{**char**,8}} i.e. offsets of 0 and 8 respectively.
  - Now assume that doubles are aligned strictly at addresses that are multiples of 8
  - Therefore, extent = 16  
(9 rounds to next multiple of 8, which is where the next double in this derived datatype would land)

# Typemaps: Examples

- What is extent of type {(char, 0), (double, 8)}?
- Answer: 16
- Is {(double, 8), (char, 0)} a valid type for the above?
- Answer: yes, since order does not matter
- What are the Typemaps of `MPI_INT`, `MPI_DOUBLE`, etc.?
- Answer:
  - {(int,0)}
  - {(double, 0)}
  - etc.

# Type Signature

- The **sequence** of primitive data types (i.e. displacements ignored) is the **type signature** of the data type
- Example: a typemap of  
 $\{(double, 0), (int, 8), (char, 12)\}$
- ...has a type signature of  
 $\{double, int, char\}$

# Datatype Interrogators

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent);
```

- **datatype** - primitive or derived **datatype**
- **extent** - returns extent of **datatype** in bytes

MPI\_Aint - C type that  
holds any valid address

```
int MPI_Type_size(MPI_Datatype datatype, int *size);
```

- **datatype** - primitive or derived **datatype**
- **size** - returns size in bytes of the entries in the *type signature* of **datatype**
  - Gaps don't contribute to size
  - This is the total size of the data in a message that would be created with this datatype
  - Entries that occur multiple times in the datatype are counted with their multiplicity

# Committing Datatypes

- Each derived data type constructor returns an *uncommitted* data type. Think of the commit process as a compilation of data type description into efficient internal form

```
int MPI_Type_commit (MPI_Datatype *datatype);
```

- **Required** for any derived data type before it can be used in communication
- Subsequently can use in any function call where an **MPI\_Datatype** is specified

# MPI\_Type\_free

- Call to `MPI_Type_free` sets the value of an MPI data type to `MPI_Datatype_NULL`

```
int MPI_Type_free(MPI_Datatype *datatype);
```

- Data types that were derived from the defined data type are unaffected.

# MPI Type-Definition Functions [“constructors”]

- `MPI_Type_contiguous`: a replication of data type into contiguous locations
- `MPI_Type_vector`: replication of data type into locations that consist of equally spaced blocks
- `MPI_Type_create_hvector`: like vector, but successive blocks are not multiple of base type extent
- `MPI_Type_indexed`: non-contiguous data layout where displacements between successive blocks need not be equal
- `MPI_Type_create_struct`: most general – each block may consist of replications of different data types

Inconsistent naming convention is unfortunate but carries no deeper meaning. It is a compatibility issue between old and new versions of MPI.

# MPI\_Type\_contiguous

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

- IN count (replication count)
  - IN oldtype (base data type)
  - OUT newtype (handle to new data type)
- 
- Creates a new type which is simply a replication of old type into contiguous locations

```

#include <stdio.h>
#include <mpi.h>
/* !!! Should be run with at least four processes !!! */
int main(int argc, char *argv[]) {
    int rank;
    MPI_Status status;
    struct {
        int x;
        int y;
        int z;
    } point;
    MPI_Datatype ptype;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Type_contiguous(3,MPI_INT,&ptype);
    MPI_Type_commit(&ptype);
    if( rank==3 ){
        point.x=15; point.y=23; point.z=6;
        MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
    }
    else if( rank==1 ) {
        MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,&status);
        printf("P:%d received coords are (%d,%d,%d) \n",rank,point.x,point.y,point.z);
    }
    MPI_Type_free(&ptype);
    MPI_Finalize();
    return 0;
}

```

[serban@lagrange:~/CODES/MPI\$

[serban@lagrange:~/CODES/MPI\$ mpicc -o type\_cntg type\_contiguous.c  
[serban@lagrange:~/CODES/MPI\$ mpiexec -np 10 ./type\_cntg  
P:1 received coords are (15,23,6)  
[serban@lagrange:~/CODES/MPI\$ ]

# Motivation: MPI\_Type\_vector

- Assume you have a 2D array of integers, and want send the last column

```
int x[4][8];
```

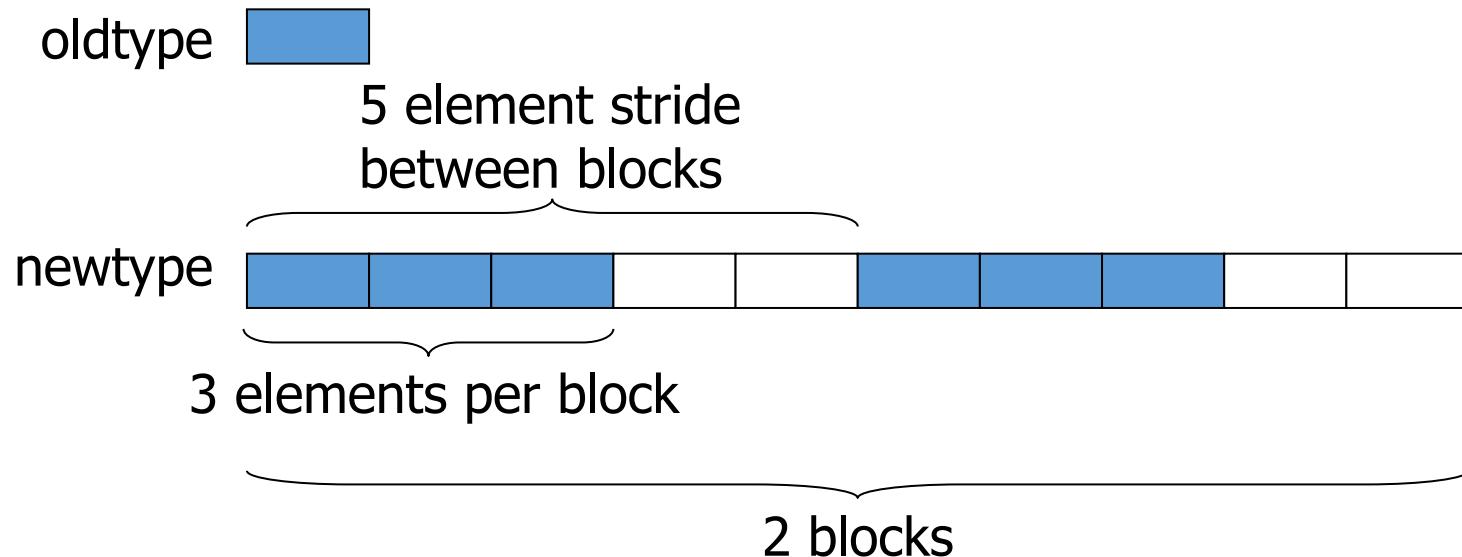
Content of x:

10	11	12	13	14	15	16	
100	101	102	103	104	105	106	
1000	1001	1002	1003	1004	1005	1006	
10000	10001	10002	10003	10004	10005	10006	

17
107
1007
10007

- There should be a way to say that I want to transfer integers, 4 of them, and they are stored in array x, 8 integers apart (the stride)

# MPI\_Type\_vector: Example



- `count = 2`
- `blocklength = 3`
- `stride = 5`

# MPI\_Type\_vector

- **MPI\_Type\_vector** is a constructor that allows replication of a data type into locations that consist of equally spaced blocks.
- Each block is obtained by concatenating the same number of copies of the old data type
- Spacing between blocks is a multiple of the extent of the old data type
- One way to look at it:
  - You want some entries but don't care about other entries in an array
  - There is a repeatability to this pattern of "wanted" and "not wanted" entries

# MPI\_Type\_vector

```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

- IN **count** (number of blocks)
- IN **blocklength** (number of elements per block)
- IN **stride** (spacing between start of each block, measured as #elements)
- IN **oldtype** (base datatype)
- OUT **newtype** (handle to new type)
- Allows replication of old type into locations of equally spaced blocks. Each block consists of same number of copies of **oldtype** with a stride that is multiple of extent of old type

```

#include <mpi.h>
#include <math.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rank,i,j;
    MPI_Status status;
    double x[4][8];
    MPI_Datatype coltype;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Type_vector(4, 1, 8, MPI_DOUBLE, &coltype);
    MPI_Type_commit(&coltype);

    if(rank==3){
        for(i=0;i<4;++i)
            for(j=0;j<8;++j) x[i][j]=pow(10.0,i+1)+j;
        MPI_Send(&x[0][7],1,coltype,1,52,MPI_COMM_WORLD);
    }
    else if(rank==1) {
        MPI_Recv(&x[0][2],1,coltype,3,52,MPI_COMM_WORLD,&status);
        for(i=0;i<4;++i)printf("P:%d my x[%d][2]=%1f\n",rank,i,x[i][2]);
    }

    MPI_Type_free(&coltype);
    MPI_Finalize();
    return 0;
}

```

Annotations for MPI\_Type\_vector parameters:

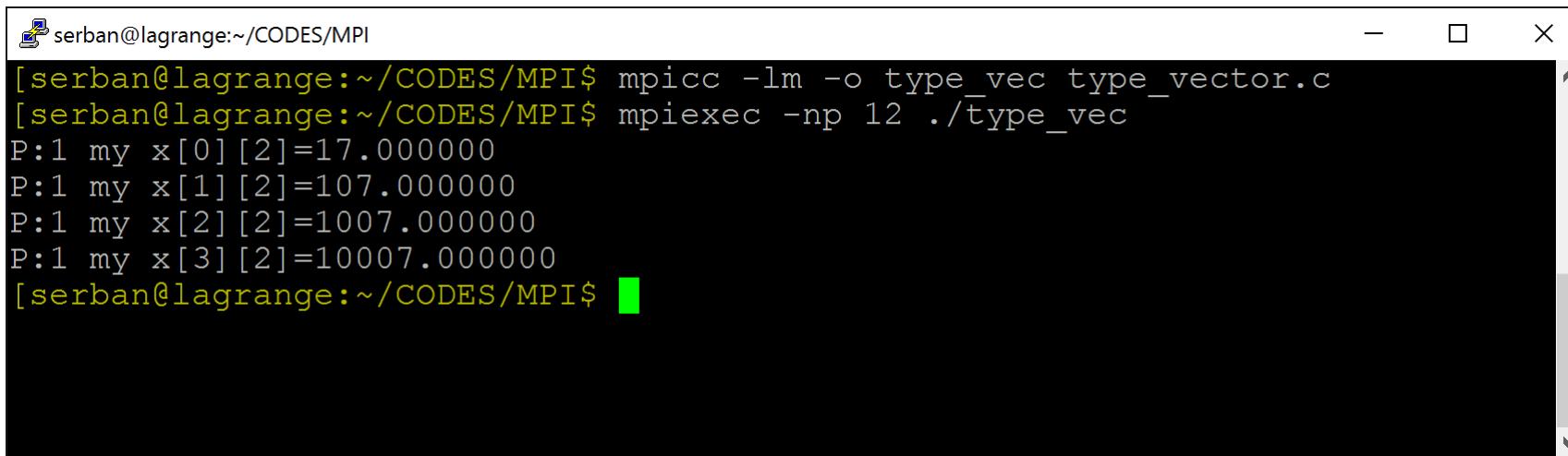
- count:** number of blocks
- blocklength:** number of elements per block
- stride:** offset (in number of elements)
- oldtype:** base type
- newtype:** handle to derived type

# Output, Example

Content of x:

10	11	12	13	14	15	16	
100	101	102	103	104	105	106	
1000	1001	1002	1003	1004	1005	1006	
10000	10001	10002	10003	10004	10005	10006	17
							107
							1007
							10007

17  
107  
1007  
10007

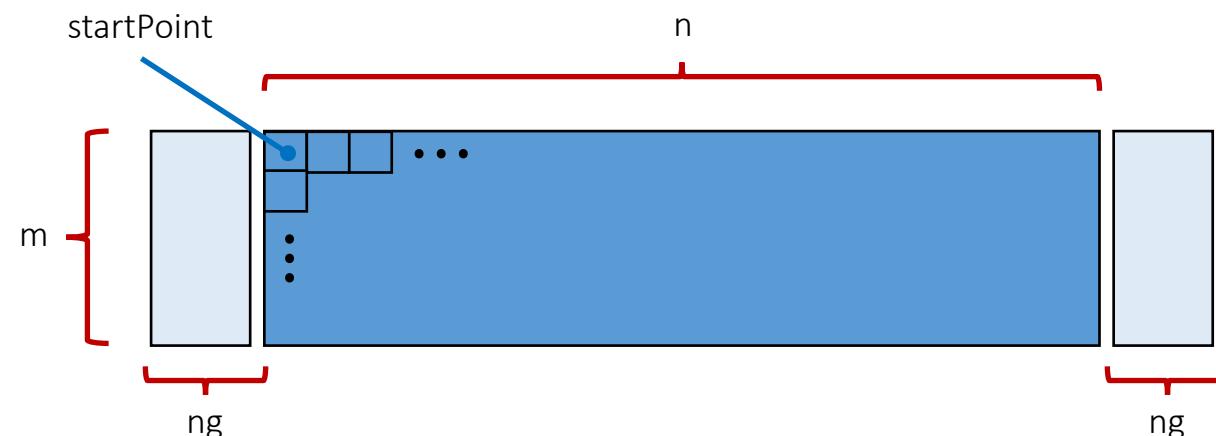


A terminal window titled "serban@lagrange:~/CODES/MPI" displaying MPI output. The window shows the compilation of "type\_vec.c" and "type\_vector.c" using mpicc and mpiexec respectively, followed by four lines of output from process P:1 showing the value of "x[2]" as 17.000000, 107.000000, 1007.000000, and 10007.000000.

```
[serban@lagrange:~/CODES/MPI$ mpicc -lm -o type_vec type_vector.c
[serban@lagrange:~/CODES/MPI$ mpiexec -np 12 ./type_vec
P:1 my x[0][2]=17.000000
P:1 my x[1][2]=107.000000
P:1 my x[2][2]=1007.000000
P:1 my x[3][2]=10007.000000
[serban@lagrange:~/CODES/MPI$
```

# Exercise: MPI\_Type\_vector

- Given: Local 2D array of doubles with interior size  $m \times n$  and  $ng$  ghostcells at each edge
- You wish to send the interior (non ghostcell) portion of the array
- How would you describe the data type to do this in a single MPI call?

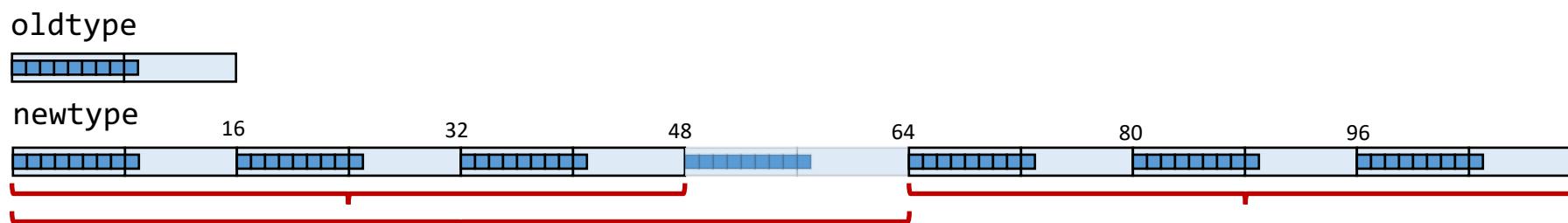


```
MPI_Type_vector (m, n, n+2*ng, MPI_DOUBLE, &interior);
MPI_Type_commit (&interior);
MPI_Send (startPoint, 1, interior, dest, tag, MPI_COMM_WORLD);
```

```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

# Exercise: Typemap

- Start with `oldtype` for which  
`Typemap = {(double, 0), (char, 8)}`
- What is Typemap of `newtype` if defined as below?  
`MPI_Type_vector(2, 3, 4, oldtype, &newtype)`



```
{ {(double, 0),(char, 8)}, {(double,16),(char,24)}, {(double,32),(char, 40)},  
{(double,64),(char,72)}, {(double,80),(char,88)}, {(double,96),(char,104)} }
```

```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

# Exercise: MPI\_Type\_vector

- How can you express

```
MPI_Type_contiguous(count, old, &new);
```

as a call to `MPI_Type_vector`?

```
MPI_Type_vector (count, 1, 1, old, &new);
```

```
MPI_Type_vector (1, count, count, old, &new);
```

```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

# Outline, Parallel Computing w/ MPI

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks

# MPI – We’re Scratching the Surface

- In some MPI implementations there are more than 300 MPI functions
  - Not all of them part of the MPI standard though, some vendor specific

MPI\_Abort, MPI\_Accumulate, MPI\_Add\_error\_class, MPI\_Add\_error\_code, MPI\_Add\_error\_string, MPI\_Address, MPI\_Allgather, MPI\_Algather, MPI\_Alloc\_mem, MPI\_Allreduce, MPI\_Altoall, MPI\_Altoallw, MPI\_Attr\_delete, MPI\_Attr\_get, MPI\_Attr\_put, MPI\_Barrier, MPI\_Bcast, MPI\_Bsend, MPI\_Bsend\_init, MPI\_Buffer\_attach, MPI\_Buffer\_detach, MPI\_Cancel, MPI\_Cart\_coords, MPI\_Cart\_create, MPI\_Cart\_get, MPI\_Cart\_map, MPI\_Cart\_rank, MPI\_Cart\_shift, MPI\_Cart\_sub, MPI\_Cartdim\_get, MPI\_Comm\_call\_errhandler, MPI\_Comm\_compare, MPI\_Comm\_create, MPI\_Comm\_create\_errhandler, MPI\_Comm\_create\_keyval, MPI\_Comm\_delete\_attr, MPI\_Comm\_dup, MPI\_Comm\_free, MPI\_Comm\_free\_keyval, MPI\_Comm\_get\_attr, MPI\_Comm\_get\_errhandler, MPI\_Comm\_get\_name, MPI\_Comm\_group, MPI\_Comm\_rank, MPI\_Comm\_remote\_group, MPI\_Comm\_remote\_size, MPI\_Comm\_set\_attr, MPI\_Comm\_set\_errhandler, MPI\_Comm\_set\_name, MPI\_Comm\_size, MPI\_Comm\_split, MPI\_Comm\_test\_inter, MPI\_Dims\_create, MPI\_Errhandler\_create, MPI\_Errhandler\_free, MPI\_Errhandler\_get, MPI\_Errhandler\_set, MPI\_Error\_class, MPI\_Error\_string, MPI\_Exscan, MPI\_File\_call\_errhandler, MPI\_File\_close, MPI\_File\_create\_errhandler, MPI\_File\_delete, MPI\_File\_get\_amode, MPI\_File\_get\_atomicity, MPI\_File\_get\_byte\_offset, MPI\_File\_get\_errhandler, MPI\_File\_get\_group, MPI\_File\_get\_info, MPI\_File\_get\_position, MPI\_File\_get\_position\_shared, MPI\_File\_get\_size, MPI\_File\_get\_type\_extent, MPI\_File\_get\_view, MPI\_File\_iread, MPI\_File\_iread\_at, MPI\_File\_iread\_shared, MPI\_File\_iwrite, MPI\_File\_iwrite\_at, MPI\_File\_iwrite\_shared, MPI\_File\_open, MPI\_File\_pallocate, MPI\_File\_read, MPI\_File\_read\_all, MPI\_File\_read\_all\_begin, MPI\_File\_read\_all\_end, MPI\_File\_read\_at\_all, MPI\_File\_read\_at\_all\_begin, MPI\_File\_read\_at\_all\_end, MPI\_File\_read\_ordered, MPI\_File\_read\_ordered\_begin, MPI\_File\_read\_ordered\_end, MPI\_File\_read\_sharing, MPI\_File\_seek, MPI\_File\_seek\_shared, MPI\_File\_set\_atomicity, MPI\_File\_set\_errhandler, MPI\_File\_set\_info, MPI\_File\_set\_size, MPI\_File\_set\_view, MPI\_File\_sync, MPI\_File\_write, MPI\_File\_write\_all, MPI\_File\_write\_all\_begin, MPI\_File\_write\_all\_end, MPI\_File\_write\_at\_all, MPI\_File\_write\_at\_all\_begin, MPI\_File\_write\_at\_all\_end, MPI\_File\_write\_ordered, MPI\_File\_write\_ordered\_begin, MPI\_File\_write\_ordered\_end, MPI\_File\_write\_shared, MPI\_Finalize, MPI\_Finalized, MPI\_Free\_mem, MPI\_Gather, MPI\_Gatherv, MPI\_Get, MPI\_Get\_address, MPI\_Get\_count, MPI\_Get\_elements, MPI\_Get\_processor\_name, MPI\_Get\_version, MPI\_Graph\_create, MPI\_Graph\_get, MPI\_Graph\_map, MPI\_Graph\_neighbors, MPI\_Graph\_neighbors\_count, MPI\_Graphdims\_get, MPI\_Grequest\_complete, MPI\_Grequest\_start, MPI\_Group\_compare, MPI\_Group\_difference, MPI\_Group\_excl, MPI\_Group\_incl, MPI\_Group\_intersection, MPI\_Group\_range\_excl, MPI\_Group\_range\_incl, MPI\_Group\_rank, MPI\_Group\_size, MPI\_Group\_translate\_ranks, MPI\_Group\_union, MPI\_Ibsend, MPI\_Info\_create, MPI\_Info\_delete, MPI\_Info\_dup, MPI\_Info\_free, MPI\_Info\_get, MPI\_Info\_get\_nkeys, MPI\_Info\_get\_nthkey, MPI\_Info\_get\_valuenlen, MPI\_Info\_set, MPI\_Init, MPI\_Init\_thread, MPI\_Initialized, MPI\_Intercomm\_create, MPI\_Intercomm\_merge, MPI\_Iprobe, MPI\_Irecv, MPI\_Isend, MPI\_Issend, MPI\_Keyval\_create, MPI\_Keyval\_free, MPI\_Op\_create, MPI\_Op\_free, MPI\_Pack, MPI\_Pack\_external, MPI\_Pack\_external\_size, MPI\_Pack\_size, MPI\_Pcontrol, MPI\_Probe, MPI\_Put, MPI\_Query\_thread, MPI\_Recv, MPI\_Recv\_init, MPI\_Reduce, MPI\_Reduce\_scatter, MPI\_Register\_datarep, MPI\_Request\_free, MPI\_Request\_get\_status, MPI\_Rsend, MPI\_Rsend\_init, MPI\_Scan, MPI\_Scatter, MPI\_Scatterv, MPI\_Send, MPI\_Send\_init, MPI\_Sendrecv, MPI\_Sendrecv\_replace, MPI\_Ssend, MPI\_Ssend\_init, MPI\_Start, MPI\_Startall, MPI\_Status\_set\_cancelled, MPI\_Status\_set\_elements, MPI\_Test, MPI\_Test\_cancelled, MPI\_Testall, MPI\_Testany, MPI\_Testsome, MPI\_Topo\_test, MPI\_Type\_commit, MPI\_Type\_contiguous, MPI\_Type\_create\_darray, MPI\_Type\_create\_f90\_complex, MPI\_Type\_create\_f90\_integer, MPI\_Type\_create\_f90\_real, MPI\_Type\_create\_hindexed, MPI\_Type\_create\_hvector, MPI\_Type\_create\_indexed\_block, MPI\_Type\_create\_keyval, MPI\_Type\_create\_resized, MPI\_Type\_create\_struct, MPI\_Type\_create\_subarray, MPI\_Type\_delete\_attr, MPI\_Type\_dup, MPI\_Type\_extent, MPI\_Type\_free, MPI\_Type\_free\_keyval, MPI\_Type\_get\_attr, MPI\_Type\_get\_contents, MPI\_Type\_get\_envelope, MPI\_Type\_get\_extent, MPI\_Type\_get\_name, MPI\_Type\_get\_true\_extent, MPI\_Type\_hindexed, MPI\_Type\_hvector, MPI\_Type\_indexed, MPI\_Type\_ib, MPI\_Type\_match\_size, MPI\_Type\_set\_attr, MPI\_Type\_set\_name, MPI\_Type\_size, MPI\_Type\_struct, MPI\_Type\_ub, MPI\_Type\_vector, MPI\_Unpack, MPI\_Unpack\_external, MPI\_Wait, MPI\_Waitall, MPI\_Waitany, MPI\_Waitsome, MPI\_Win\_call\_errhandler, MPI\_Win\_complete, MPI\_Win\_create, MPI\_Win\_create\_errhandler, MPI\_Win\_create\_keyval, MPI\_Win\_delete\_attr, MPI\_Win\_fence, MPI\_Win\_free, MPI\_Win\_free\_keyval, MPI\_Win\_get\_attr, MPI\_Win\_get\_errhandler, MPI\_Win\_get\_group, MPI\_Win\_get\_name, MPI\_Win\_lock, MPI\_Win\_post, MPI\_Win\_set\_attr, MPI\_Win\_set\_errhandler, MPI\_Win\_set\_name, MPI\_Win\_start, MPI\_Win\_test, MPI\_Win\_unlock, MPI\_Win\_wait, MPI\_Wtick, MPI\_Wtime

- Recall the 80/20 rule: six calls is probably what you need to implement a decent MPI code...
  - MPI\_Init, MPI\_Comm\_Size, MPI\_Comm\_Rank, MPI\_Send, MPI\_Recv, MPI\_Finalize

# MPI: Textbooks, Further Reading...

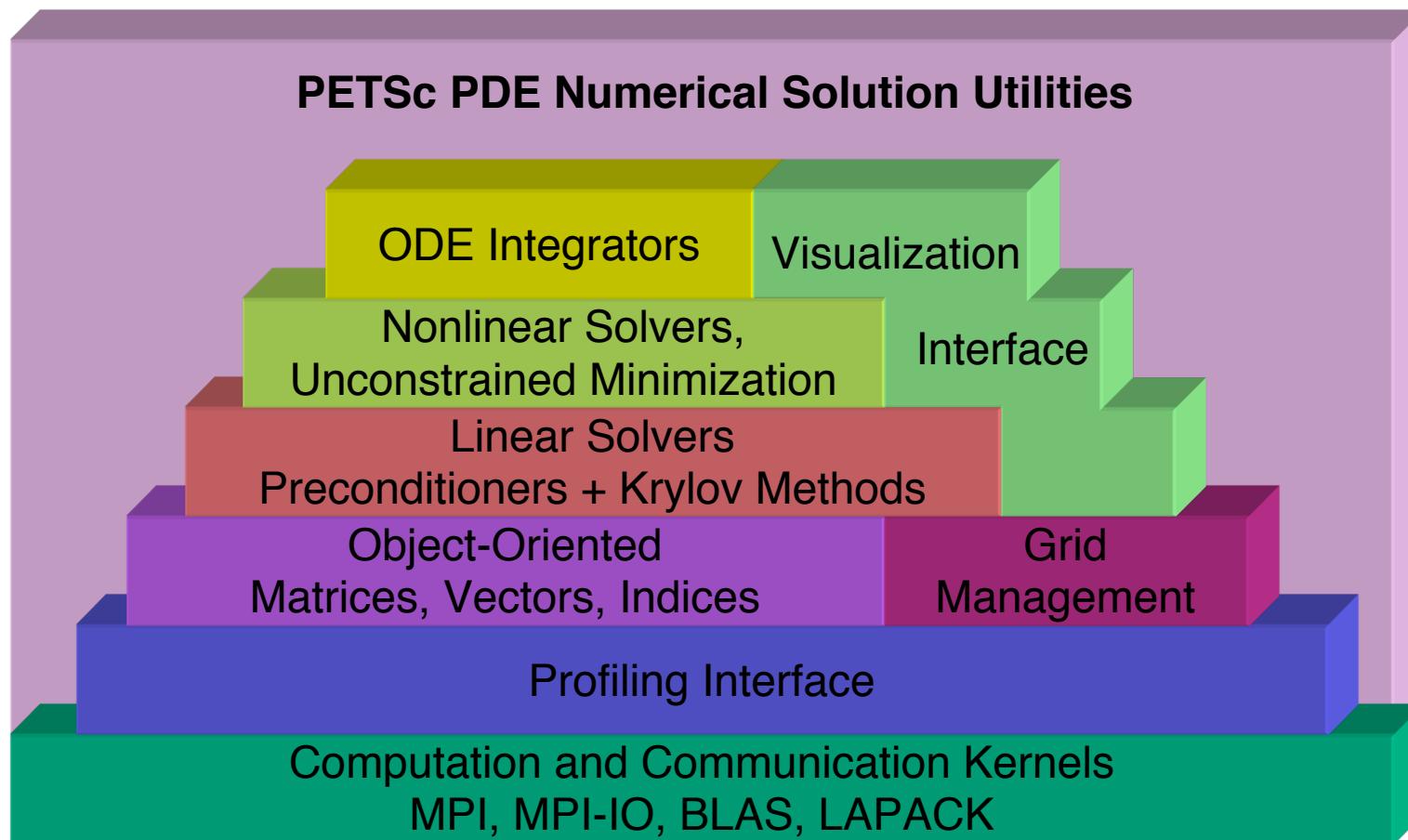
- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July 18, 1997)
- **MPI: The Complete Reference**, Marc Snir and William Gropp et al., The MIT Press, 1998 (2-volume set)
- **Using MPI: Portable Parallel Programming With the Message-Passing Interface** and **Using MPI-2: Advanced Features of the Message-Passing Interface**. William Gropp, Ewing Lusk and Rajeev Thakur, MIT Press, 1999 – also available in a single volume ISBN 026257134X.
- **Parallel Programming with MPI**, Peter S. Pacheco, Morgan Kaufmann Publishers, 1997 - very good introduction.
- **Parallel Programming with MPI**, Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti. Training handbook from EPCC
  - [http://www.epcc.ed.ac.uk/computing/training/document\\_archive/mpi-course/mpi-course.pdf](http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.pdf)

# The PETSc Library

[The message: Use libraries whenever possible]

- PETSc: Portable, Extensible Toolkit for Scientific Computation
  - One of the most successful libraries built on top of MPI
  - Intended for use in large-scale application projects,
  - Developed at Argonne National Lab (Barry Smith)
  - Open source, available for download at <http://www.mcs.anl.gov/petsc/petsc-as/>
- PETSc: routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution
- PETSc also provides routines for
  - Sparse matrix assembly
  - Distributed arrays
  - General scatter/gather (e.g., for unstructured grids)

# Structure of PETSc



# PETSc Numerical Components

Nonlinear Solvers	
Newton-based Methods	Other
Line Search	Trust Region

Time Steppers			
Euler	Backward Euler	Pseudo Time Stepping	Other

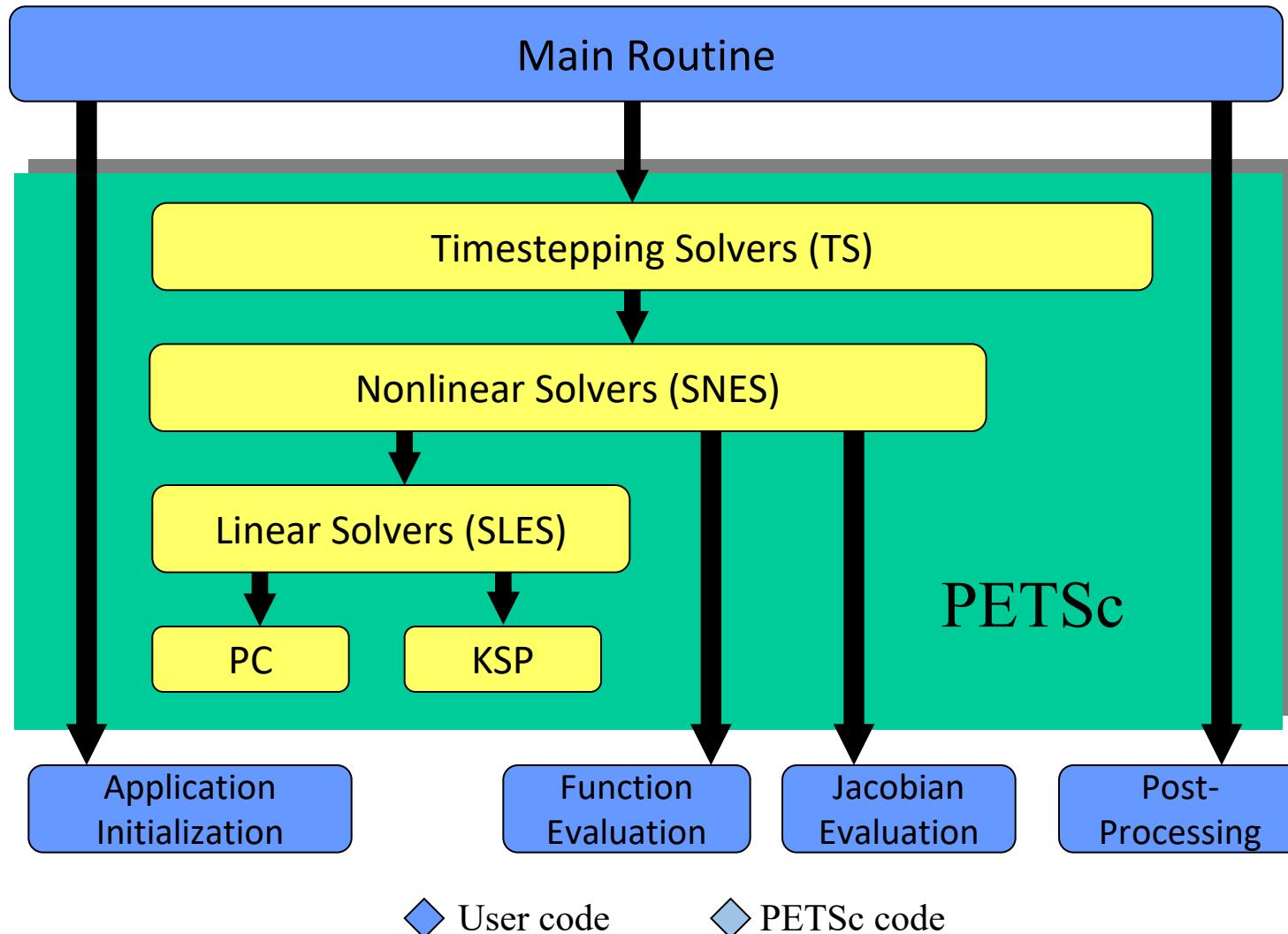
Krylov Subspace Methods							
GMRES	CG	CGS	Bi-CG-STAB	TFQMR	Richardson	Chebychev	Other

Preconditioners						
Additive Schwartz	Block Jacobi	Jacobi	ILU	ICC	LU (Sequential only)	Others

Matrices					
Compressed Sparse Row (AIJ)	Blocked Compressed Sparse Row (BAIJ)	Block Diagonal (BDIAG)	Dense	Matrix-free	Other

Distributed Arrays	Index Sets			
Vectors	Indices	Block Indices	Stride	Other

# Flow Control for PDE Solution



## Additional Material

# MPI\_Exscan

- `MPI_Exscan` is like `MPI_Scan`, except that the contribution from the calling process is not included in the result at the calling process (it is contributed to the subsequent processes)
- The value in `recvbuf` on the process with rank 0 is undefined, and `recvbuf` is not significant on process 0
- The value in `recvbuf` on the process with rank 1 is defined as the value in `sendbuf` on the process with rank 0
- For processes with rank  $i > 1$ , the operation returns, in the receive buffer of the process with rank  $i$ , the reduction of the values in the send buffers of processes with ranks  $0, \dots, i-1$  (inclusive)
- The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for `MPI_REDUCE`

# MPI\_Exscan

- Function prototype

```
int MPI_Exscan (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm cmm);
```

- IN `sendbuf` (address of send buffer)
- OUT `recvbuf` (address of receive buffer)
- IN `count` (number of elements in send buffer)
- IN `datatype` (data type of elements in send buffer)
- IN `op` (reduce operation)
- IN `cmm` (communicator)

```

#include <mpi.h>
#include <cstdio>

void main(int argc, char **argv) {
    int myRank, nprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    const int dimArray = 2;
    int* data_l = new int[dimArray];
    for (int i = 0; i < dimArray; i++) data_l[i] = (i+1)*myRank;
    for (int n = 0; n < nprocs; n++) {
        if (myRank == n) {
            printf("[%d]: ", myRank);
            for(int i = 0; i < dimArray; i++) printf("data[%d] = %d; ", i, data_l[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    int* result = new int[dimArray];
    MPI_Exscan(data_l, result, dimArray, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (int n = 0; n < nprocs; n++) {
        if (myRank == n) {
            printf("[%d]: ", myRank);
            for(int i = 0; i < dimArray; i++) printf("result[%d] = %d; d", i, data_l[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    MPI_Finalize();
    delete[] result;
    delete[] data_l;
}

```

## Example: MPI\_Exscan

[serban@euler24:~/CODES/MPI]

```

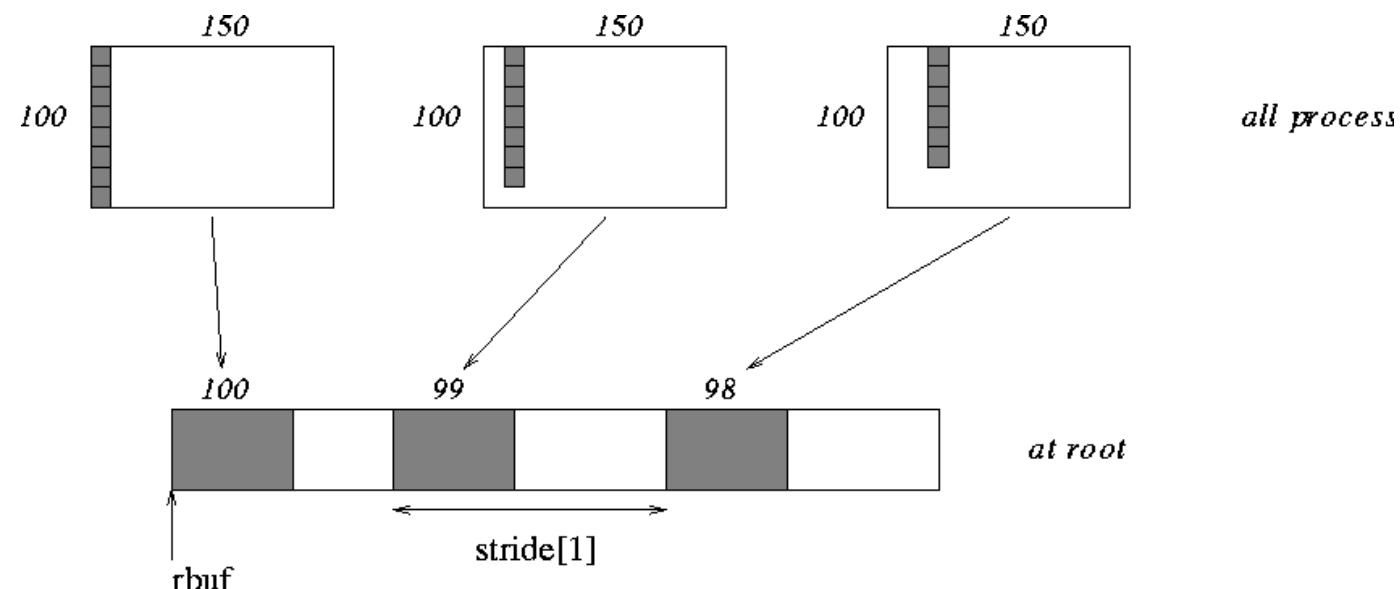
[serban@euler24:~/CODES/MPI$ mpiCC exscan.cpp
[serban@euler24:~/CODES/MPI$ mpiexec -np 5 ./a.out
[0]: data[0] = 0; data[1] = 0;
[1]: data[0] = 1; data[1] = 2;
[2]: data[0] = 2; data[1] = 4;
[3]: data[0] = 3; data[1] = 6;
[4]: data[0] = 4; data[1] = 8;
[0]: result[0] = -1378015304; result[1] = 32705;
[4]: result[0] = 6; result[1] = 12;
[1]: result[0] = 0; result[1] = 0;
[2]: result[0] = 1; result[1] = 2;
[3]: result[0] = 3; result[1] = 6;
[serban@euler24:~/CODES/MPI$ █

```

# MPI\_Gatherv

- Function Prototype

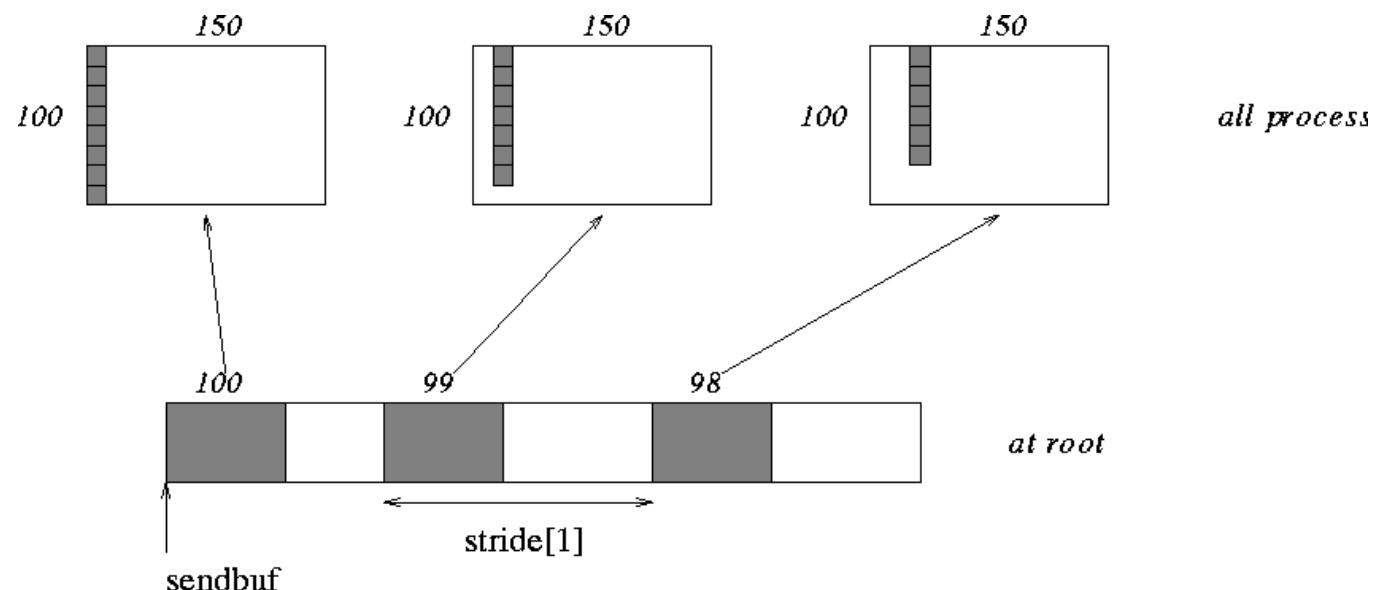
```
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
                 void *recvbuf, const int *recvcounts, const int *displs,  
                 MPI_Datatype recvtype, int root, MPI_Comm comm)
```



# MPI\_Scatterv

- Function Prototype

```
int MPI_Scatterv(const void *sendbuf, const int *sendcounts, const int *displs,  
                  MPI_Datatype sendtype, void *recvbuf, int recvcount,  
                  MPI_Datatype recvtype,  
                  int root, MPI_Comm comm)
```



# ME759 Wrap Up

# Course Objectives [slide of January 22, 2020]

- Get familiar with today's software and hardware that can speed up your code
  - Mostly done through parallel computing, at multiple levels
- Recognize applications/problems that can draw on advanced computing
- Gain basic skills that will help you map these applications onto a parallel computing hardware/software stack
  - Write code, build, link, run, debug, profile
- Introduce basic software design patterns for parallel computing
- Expand your mindset when it comes to writing software for scientific computing

# ME759: Opportunities for Efficiency Gains Discussed

Efficiency Gain Opport.	Hardware asset coming into play	Software Enabler
Cluster (distributed mem.)	Group of nodes communicate through fast interconnect	MPI, Charm++, Chapel
Multi-Socket Node	Group of CPUs on the same node, talk through main mem.	OpenMP, MPI
Acceleration (GPU/Phi)	Compute devices accelerating parallel computation on one node	CUDA, OpenCL, OpenMP
Multi-Core Processor	Communication through shared caches and main mem.	OpenMP, TBB, pthreads
Vectorization	Higher operation throughput via special/fat registers	AVX, SSE
Pipelining	Sequence of instruction sharing functional units	Assembly
Superscalar	Non-sequence instructions sharing functional units	Assembly

We have full control → 

We have little to no control → 

# Skills I hope You Picked Up in ME759

- I think of these as items that you can claim to be familiar with during an interview:
  - Basic understanding of what happens on one core
  - Basic understanding of hardware for parallel computing
  - CUDA programming
  - OpenMP Programming
  - MPI Programming, scratched the surface
  - Understanding of the parallel computing model induced by each solution; i.e., MPI, OpenMP, or CUDA
  - Understanding that data movement dictates performance

# ME759: Everything compressed on one slide

- Know your hardware
- Don't move data around
  - Energy and time expensive
- Seek solutions that expose concurrency/parallelism
- Use the tools of the trade to work like a pro, and then get paid like a pro
  - Use debuggers, profilers, CMake, memory checkers, compiler flags, git, etc.

# Quote of the day, yet again

“Human beings only use ten percent of their brains. Ten percent! Can you imagine how much we could accomplish if we used the other sixty percent?”

Ellen DeGeneres, American comedian, television host, actress, writer, and producer [1958 - ]

# Perhaps the most important thing: keep thinking & exploring

- ME759 was about convincing you that there is “the other 60%” that you can tap into
- What “the other 60%” is continuously changing
- To keep using “the other 60%,” you’ll have to keep reading, thinking, and exploring

# Don't look at ME759 as a fish. Pick up the fishing

**"If you give a man a fish he is hungry again in an hour. If you teach him to catch a fish you do him a good turn."**

Anne Isabella Thackeray Ritchie, English writer [1837 –1919]  
(also in the Chinese, Jewish, Navajo, Italian, etc., folklore)

# Perhaps the most important thing: keep thinking & exploring

- ME759 was about convincing you that there is “the other 60%” that you can tap into
- What “the other 60%” is continuously changing
- To keep using “the other 60%,” you’ll have to keep reading, thinking, and exploring
- In an ideal world, ME759 taught you how to fish. Not hand you a fish

# Email, April 2, 2019

[shown with permission]

Dear Dan,

I took your HPC course in Spring 2018, and really enjoyed it. I did an internship with ANSYS in Summer 2018 and I was able to significantly improve their code performance because of the course.

Initially, their code did not scale as expected. I found a line in the code where the size of an array was being changed inside a nested loop. I changed this and fixed the slowdown. The performance went from  $O(N^2)$  to the expected  $O(N \log N)$ . Now that the code was scaling as expected, we were able to publish it (along with some other material) as an SAE paper. I have attached the paper [here](#). I will be presenting it next week in Detroit.

I just wanted to let you know of this, and thank you for the great course!

Regards,

Arpit Agarwal

4th year Ph.D. student

Mario Trujillo's group

End of 759.

Thank you.

Don't forget to write.