Course Introduction

CS315B

Administrivia

- Syllabus on cs315.stanford.edu
    - Assignments will be managed through Canvas

- Instructor: Alex Aiken
    - 9-10:20 TT

- Structure
    - 6 (smallish) programming assignments
    - A course project
    - Some readings (papers and notes)
    - No exams

- Office hours 11-12 Wed and 3-4 Fri
- Ed discussion group
- No lecture recordings

Course Topic

- How do we program modern supercomputers?
- Assumption 1: Current supercomputers are tomorrow’s ordinary computers.
- Assumption 2: We need new ways to program contemporary machines.

Course Approach

- Lectures on programming supercomputers
  - Past, present and future
- Focus on task-based parallel programming
  - And specifically on cuNumeric & Regent
  - Developed at Stanford/SLAC, NVIDIA, and LANL
- Programming assignments and the project will use cuNumeric & Regent

Amdahl's Law

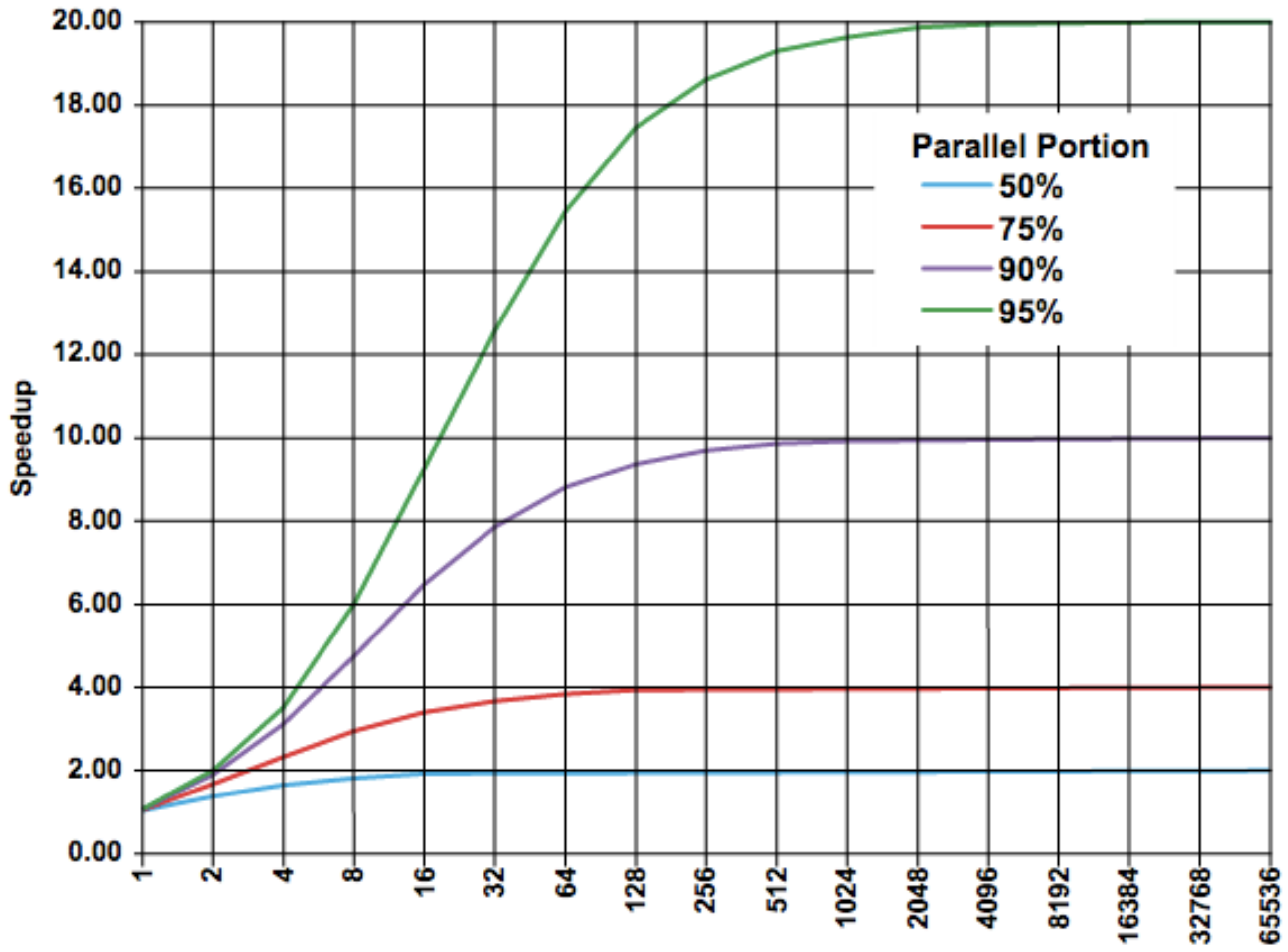
$$\text{Speedup} = \frac{1}{(1 - p) \; + \; (p / s)}$$

where

p = portion of the program sped up

s = factor improvement of that portion

Speed vs. # of Processors for Values of p



Discussion

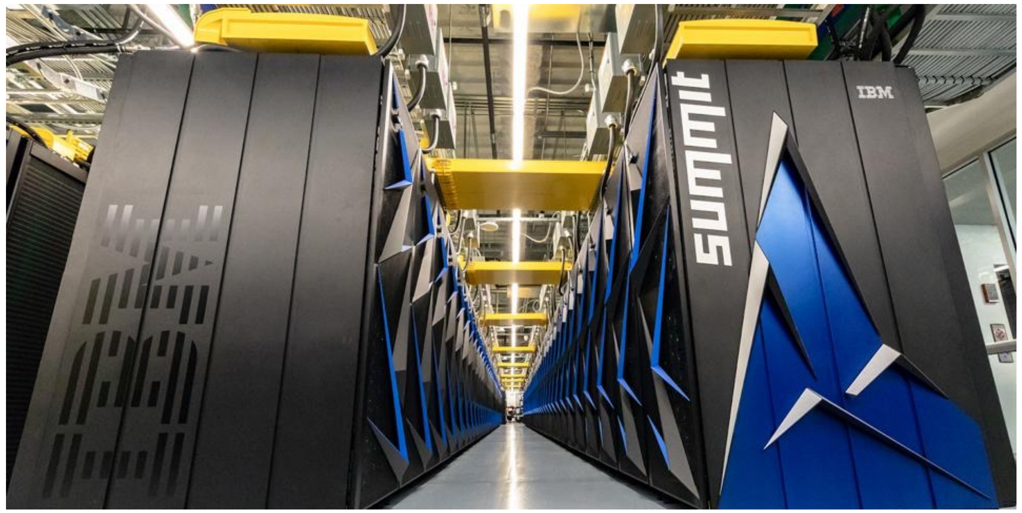
- Amdahl’s law is simple and general
  - Not about a specific machine or program

- And unforgiving
  - To speed up by 1000x, must parallelize 99.9%
  - To reach 10,000x, must parallelize 99.99%
  - And these are not very aggressive targets!

Summit

- 4,000+ nodes
- 6 GPUS/node
- 84 stream multiprocessors (SMs)/GPU
- 64-way processing in each SM

- ~120M ”threads”
- 200 PF/sec



Consequences

- Even tiny sequential bottlenecks can matter
    - *None* can remain

- Each order of magnitude improvement requires additional work

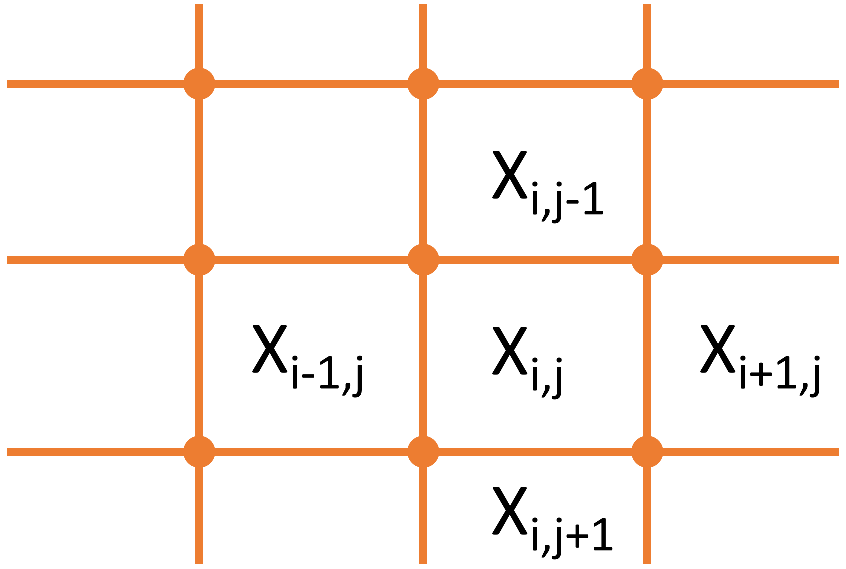
- And the temptation to customize to a particular machine is great

Beyond Amdahl’s Law

- But Amdahl’s Law is only one reason why parallel programming is hard

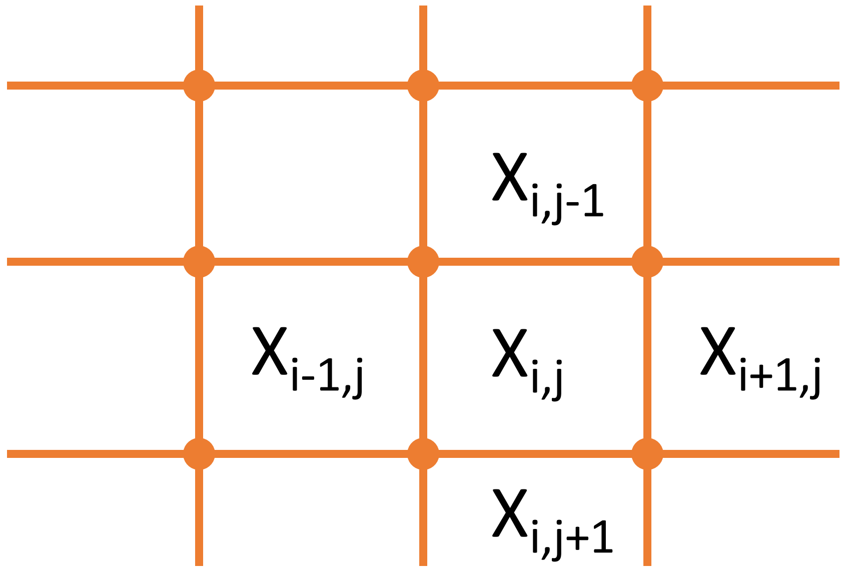
- Resource management is also hard

An Example



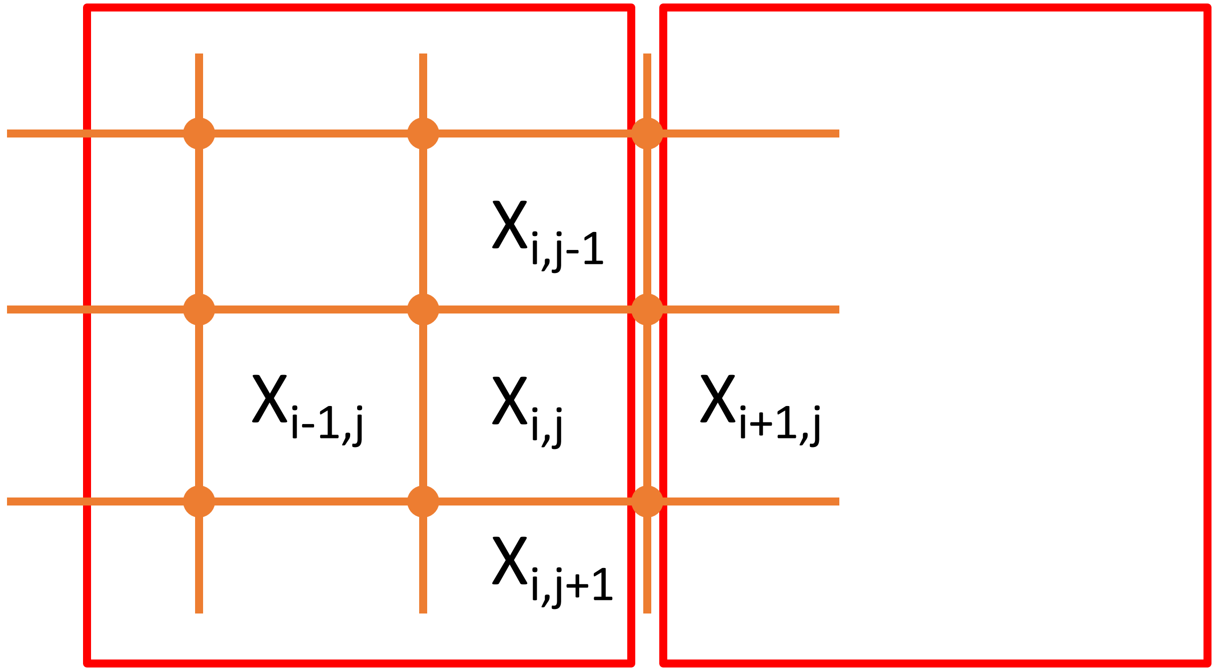
Goal: For all i,j compute $x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1})$

Issue 1



Can I refer to $x_{i,j}$, $x_{i-1,j}$, $x_{i,j-1}$, $x_{i+1,j}$, $x_{i,j+1}$ *at the same time?*

Why Not?



Can I refer to $x_{i,j}$, $x_{i-1,j}$, $x_{i,j-1}$, $x_{i+1,j}$, $x_{i,j+1}$ at the same time?

Resource: Memory (Hardware Level)

Distributed Memory

- Hardware exposes physically disjoint memories

Shared Memory

- Hardware provides a single hardware address space

Resource: Memory (Program Level)

Global Address Space

- Programming language allows any piece of data to be named anywhere in the machine

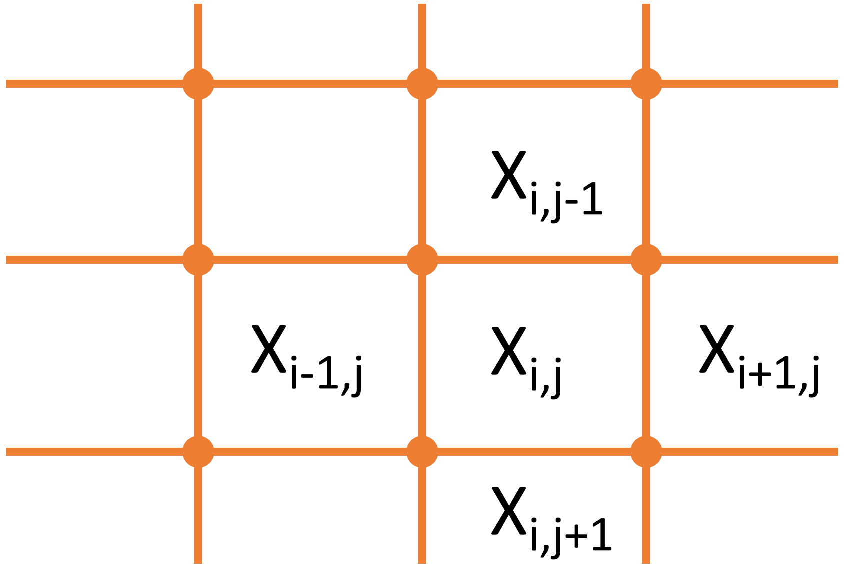
Local Address Space

- Programming language only allows data to be named that is “near” the processor

Software vs. Hardware

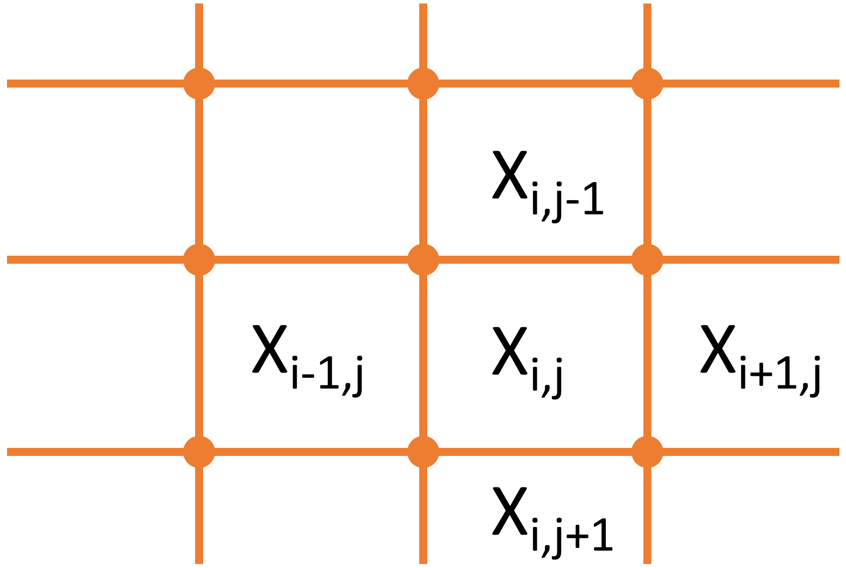
- Global address space is easy to implement on shared memory hardware
  - Hardware is complex
- Global address space is much more complex to implement on distributed memory hardware
  - Language system is complex

The Example Again



Goal: For all i,j compute $x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1})$

Issue 1b



What is the cost of referring to $x_{i,j}$, $x_{i-1,j}$, $x_{i,j-1}$, $x_{i+1,j}$, $x_{i,j+1}$?

Locality

- Is the data “close” to the processor?

- Local address space
  - Yes, memory references are always cheap
  - Programmer structures program for locality

- Global address space
  - Memory references may have greatly varying cost
  - E.g., on distributed memory machines
  - Or machines with caches

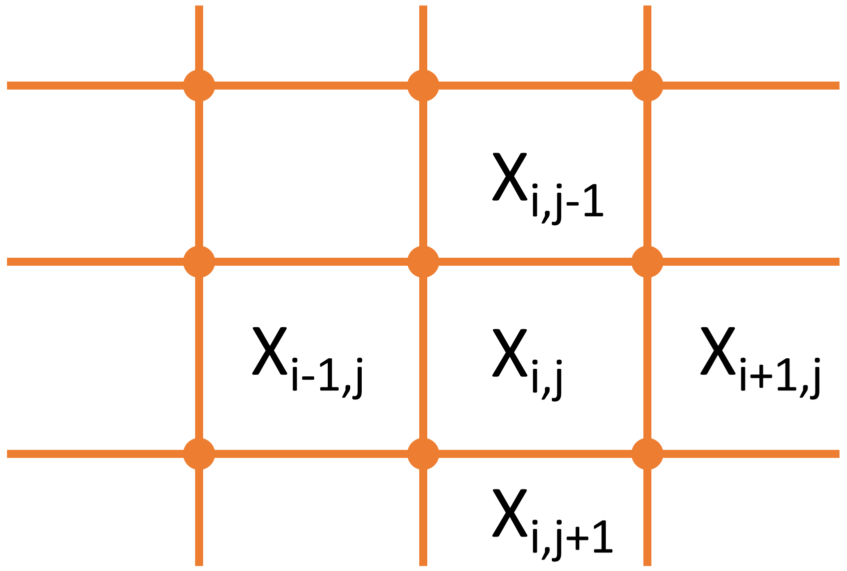
Summary: Memory

- Memory is a critical resource

- Who deals with the reality that memory is physically distributed?
  - *Shared memory:* the hardware does it
  - *Global address space:* the compiler/runtime does it
  - *Local address space:* the programmer does it

- Programs can exhibit good or bad locality

Issue 2

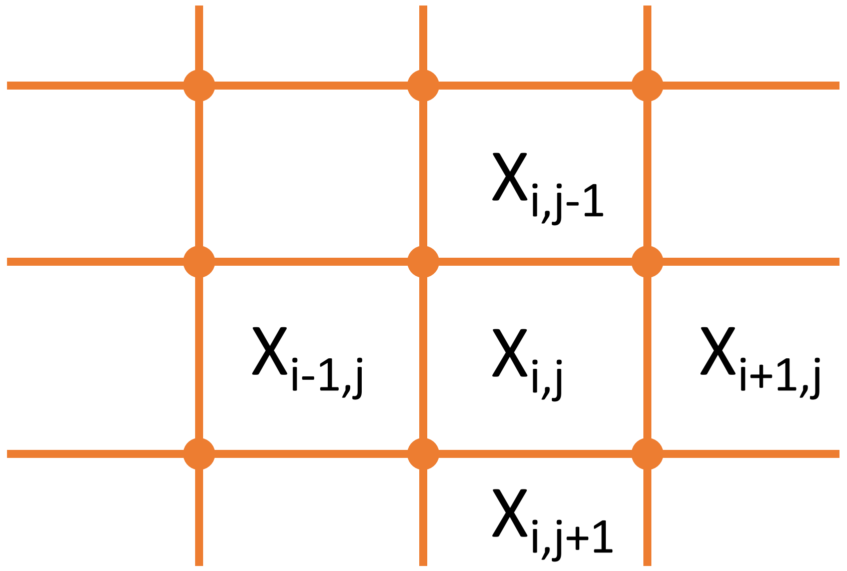


Goal: For all i,j compute $x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1})$
In parallel for each i,j.
How many copies of the program do I need?

Control

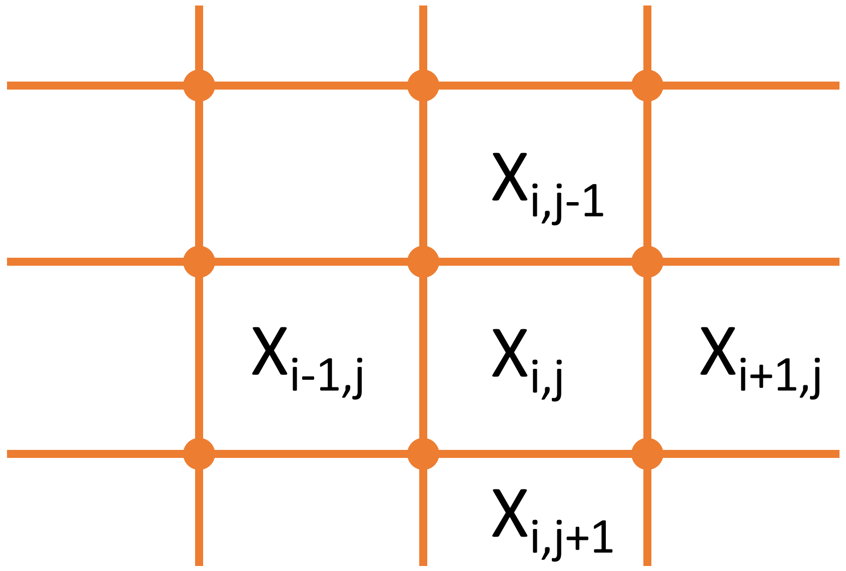
- Control is a resource
- Parallel copies of the program require state
  - At least a *program counter,* but usually more
  - This state must be stored somewhere and managed
- Note this is different from the question of how many processors there are
  - Number of executing “jobs” not necessarily the same as number of processors

Answer 1



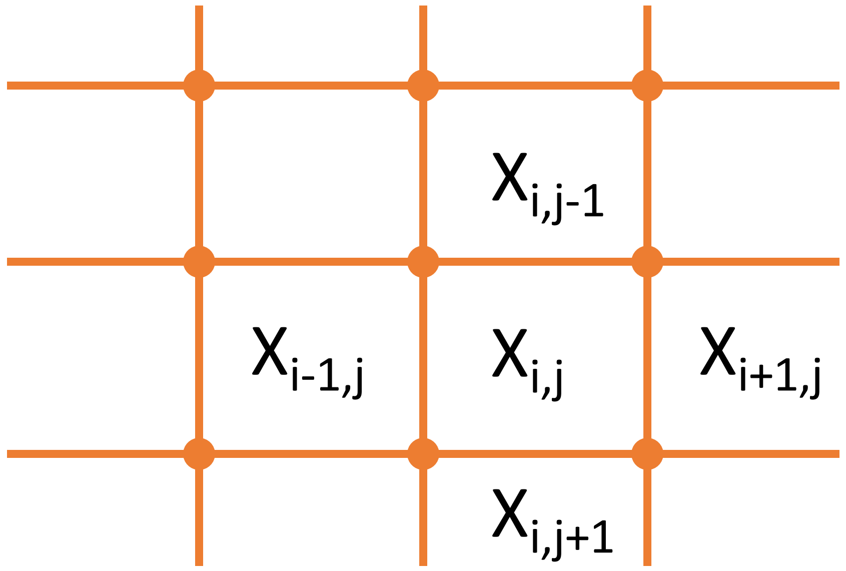
One control context for each i,j

Answer 2



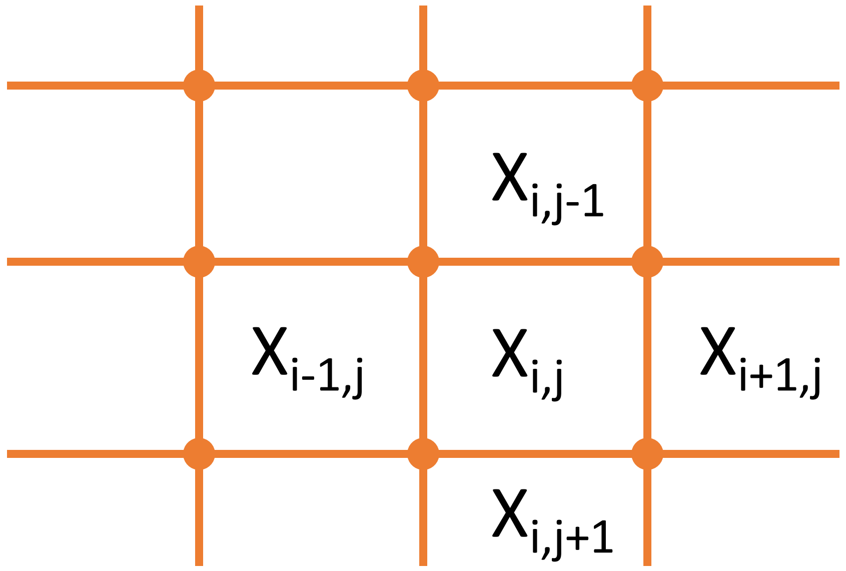
One control context for all *i,j*

Answer 3



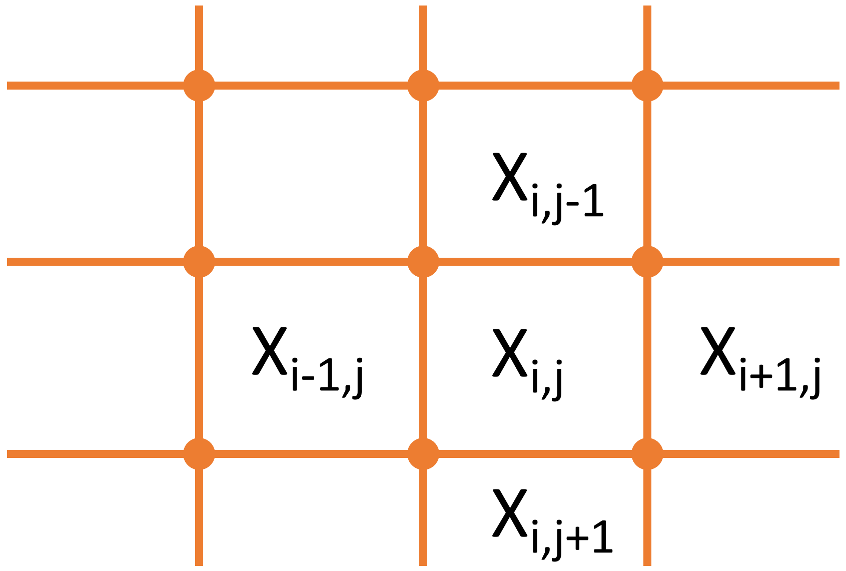
One control context for each processor.

Question



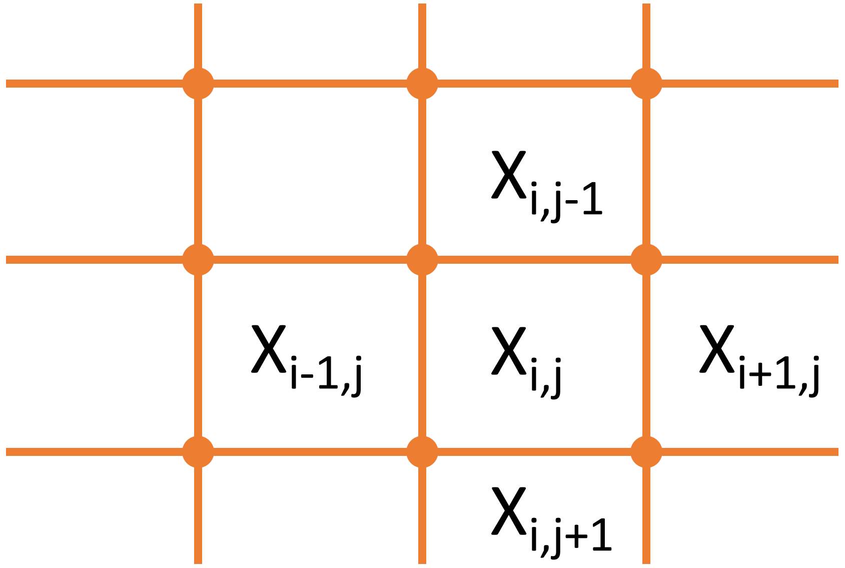
- What is the output for
  - For all i,j compute $x_{i,j} = \text{AVG}(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1})$

Issue 2b



Goal: For all i,j compute $x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1})$
In parallel for each i,j.
In what order do reads and writes happen?

Issue 2b



Goal: For all i,j compute $x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1})$

Does $x_{i-1,j}$ use the old or new value of $x_{i,j}$?

Synchronization

- Many read/write orders are possible

- To ensure a particular order, must use *synchronization*
  - Multiple control contexts must coordinate their actions

- Large variety of synchronization abstractions
  - Locks, semaphores, condition variables, barriers, …

Summary: Control

- Control is a resource
    - Replicating control is expensive
- Many control contexts
    - Parallel jobs run asynchronously
    - Synchronization required
- One/few control contexts
    - Can still execute on many data elements
    - Synchronization built-in

Summary: Control (Cont.)

- Who deals with the fact that the hardware provides a limited number of control contexts?
  - Compiler/runtime system may provide more contexts than physically available
  - Or not: Let the programmer deal with it

- Who deals with synchronization?
  - Many strategies from hardware, compiler, programmer, to combinations of all three

Discussion

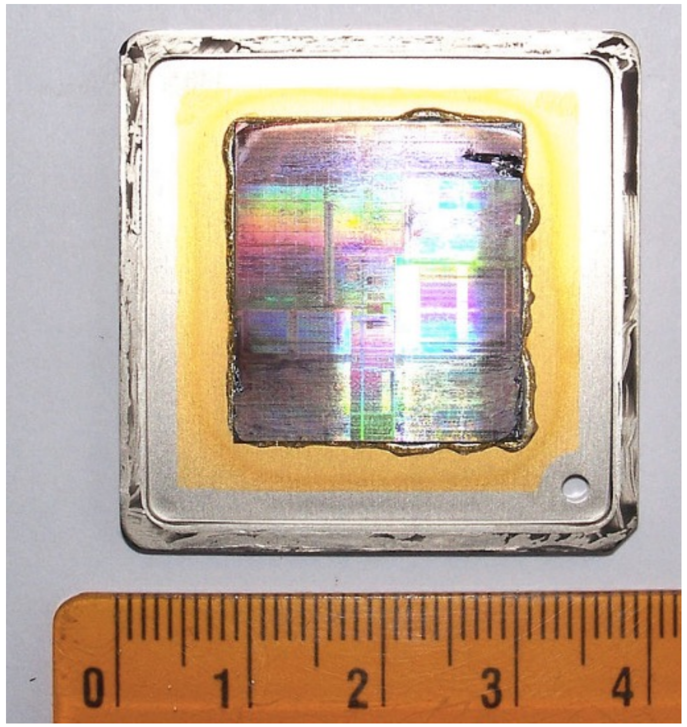
- Two fundamental resources

- Memory
  - locality

- Control
  - synchronization

Hardware

Hardware



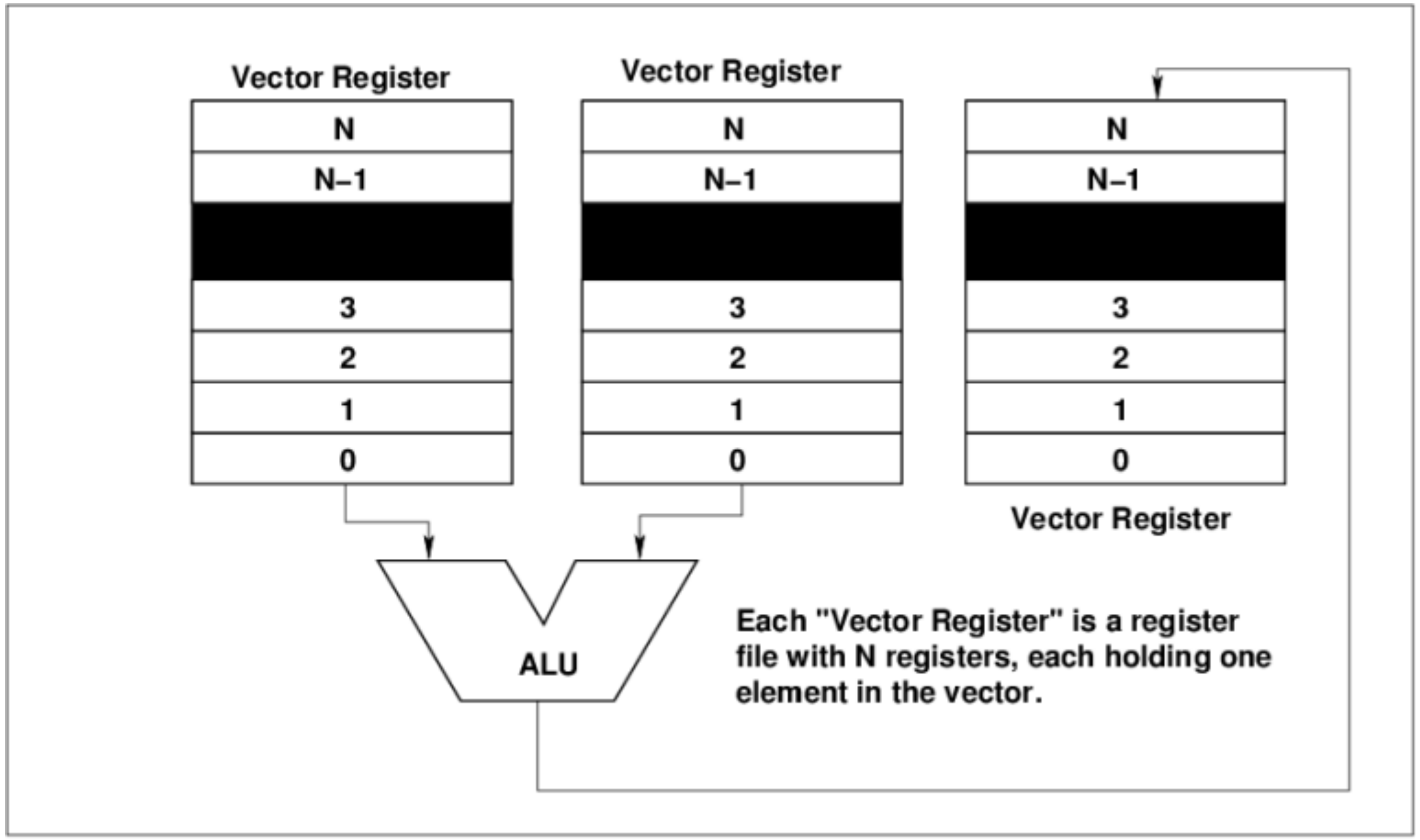
Characteristics

- Operations within a die/chip are fast
    - Off-chip operations are much slower
- The transistor budget for any chip is fixed
    - But is still increasing over time
- Do we spend the transistors on memory or control?

Four Examples

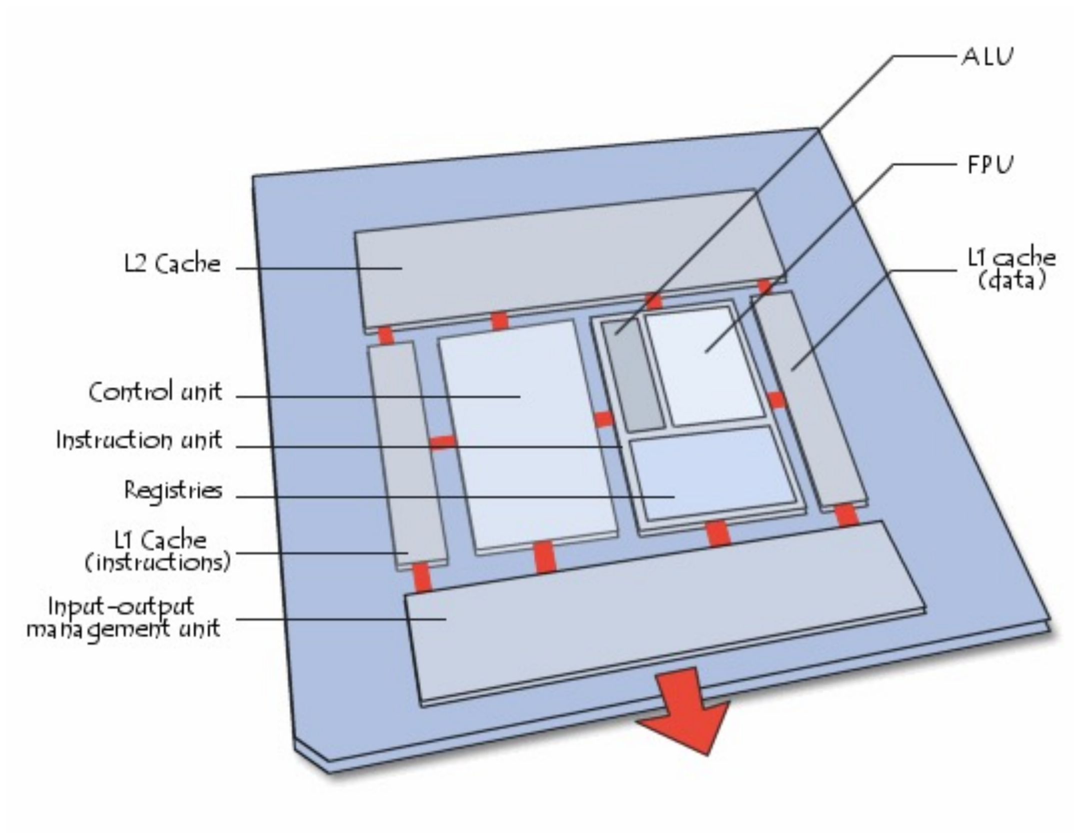
- Vector Processors
- CPUs
- Multicore
- GPUs

Vector Processor

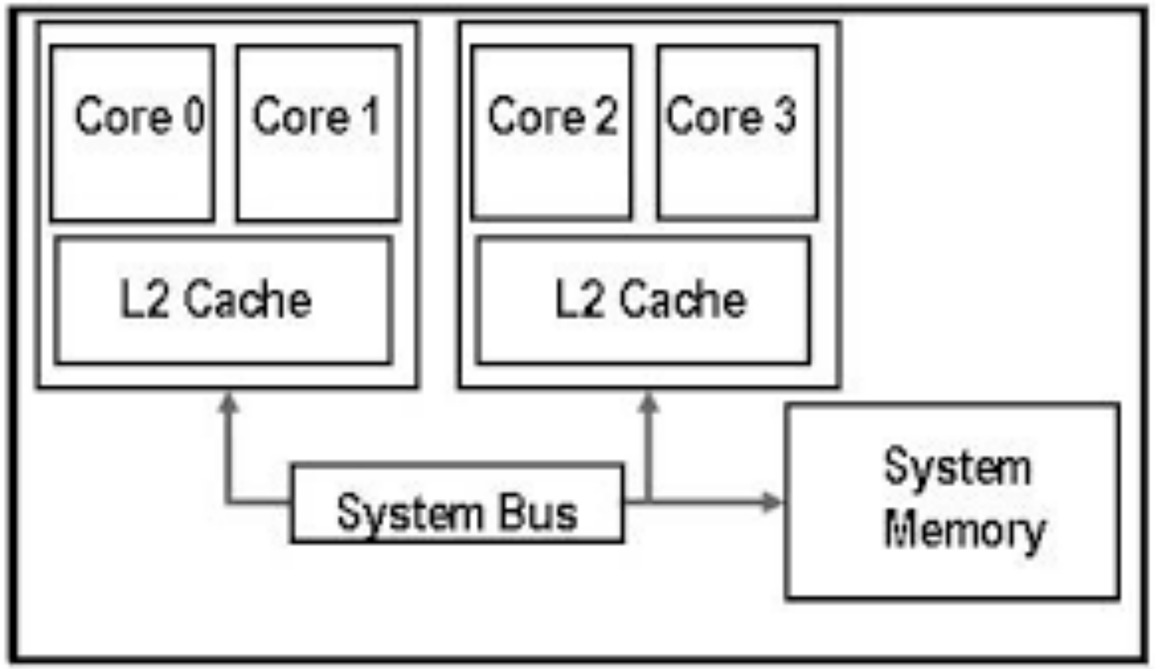


Each "Vector Register" is a register file with N registers, each holding one element in the vector.

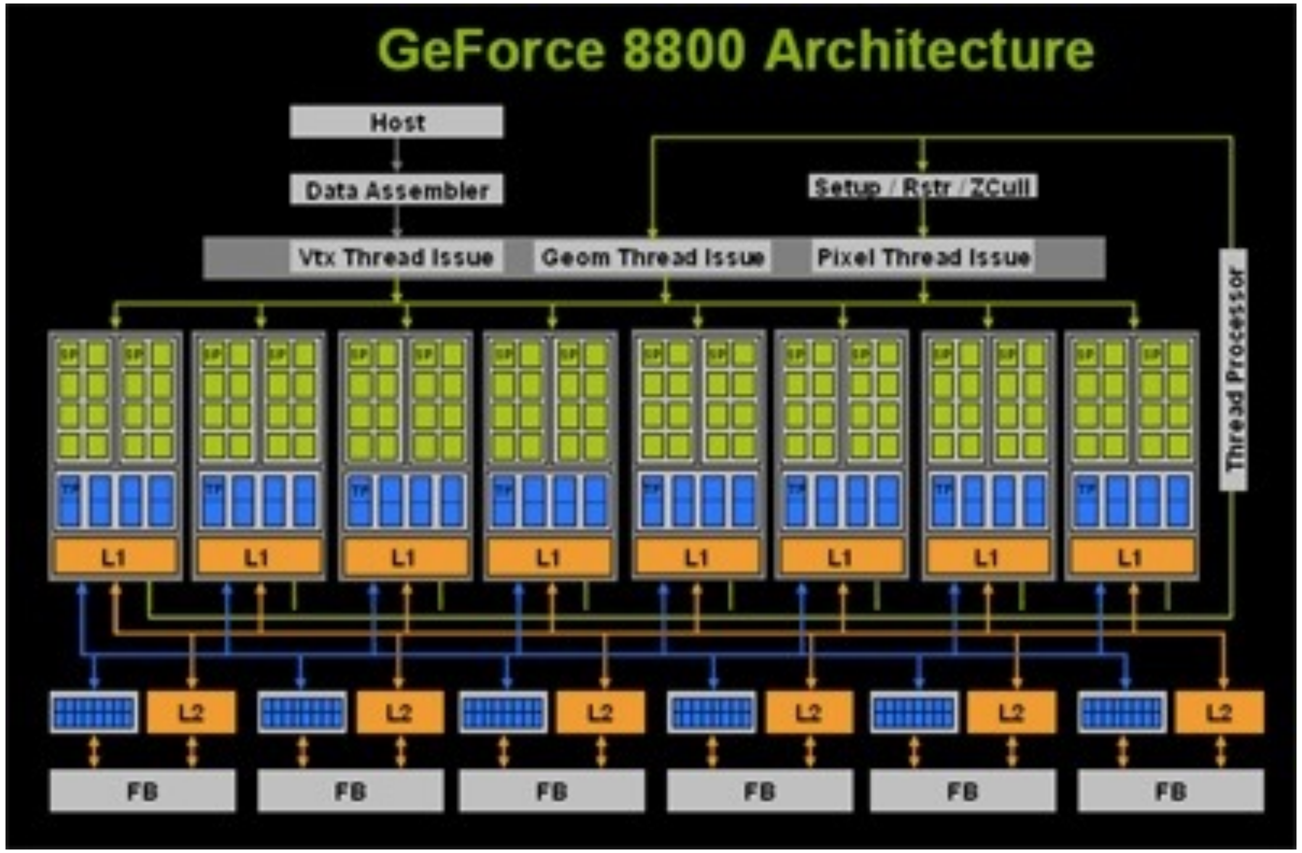
CPUs


ALU
FPU
L1 cache (data)
L2 Cache
Control unit
Instruction unit
Registries
L1 Cache (instructions)
Input-output management unit

Multicore


Core 0
Core 1
Core 2
Core 3
L2 Cache
L2 Cache
System Bus
System Memory

GPUs


GeForce 8800 Architecture
Host
Data Assembler
Setup / Rstr / ZCull
Vtx Thread Issue
Geom Thread Issue
Pixel Thread Issue
Thread Processor
SP
TF
L1
L2
FB

Summary

- Control and memory are fundamental resources
- At the hardware-level, different designs make different tradeoffs

The Memory Hierarchy

- Individual cores
  - GPU or CPU w/vector units
- NUMA domains
- Multicore chips
- Boards
- Boxes
- Racks

- Operations within a level are generally faster than operations at the next higher level
- But a level has much less memory than the next level up

Modern Supercomputers

- Consist of
  - CPUs
  - Multicore
  - Vector processors
  - GPUs

- Strongly hierachical

Summary

- Parallel programming limited by
  - Amdahl’s Law
  - 2D resource management problem
    - Memory & control
  - Different technologies at different scales
    - And the roles they play

- Next time: The Way Things Were
  - How we’ve programmed these machines for 20+ yrs