

# A CODELESS INTRODUCTION TO GPU PARALLELIZATION

Will Landau, Prof. Jarad Niemi

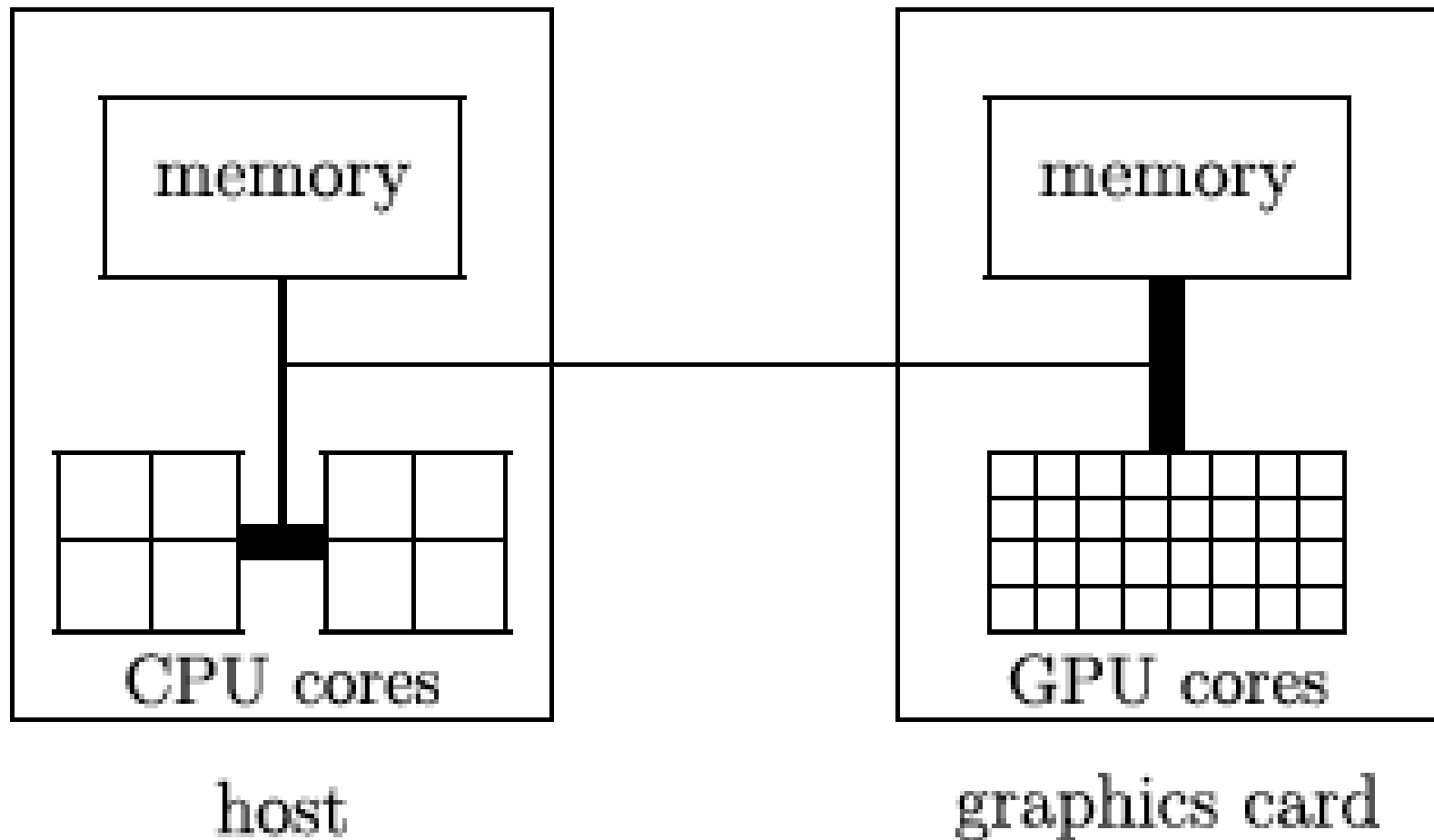
# HOW THE CPU AND GPU WORK TOGETHER

A GPU can't run a whole computer on its own because it can't do control flow and it doesn't have access to all the system hardware.

In a GPU-capable computer, the CPU is the main processor, and the GPU is an optional hardware add-on.

The CPU is the “master” of the computer, and it can delegate its highest-throughput parallelizable arithmetic load to the GPU “minion”.

Another analogy: the CPU uses the GPU in the same way that a human uses a hand-held calculator.



# GPUS AND PARALLELIZATION

**Parallelization:** Running multiple calculations simultaneously. It speeds up calculations dramatically, and GPUs are much better at it than CPUs.

**Kernel:** An instruction set executed on the GPU. (All others are executed on the CPU.)

In CUDA C, a kernel is any function prefixed with the keyword, `__global__`. (More on that in a later talk.)

# REMINDER

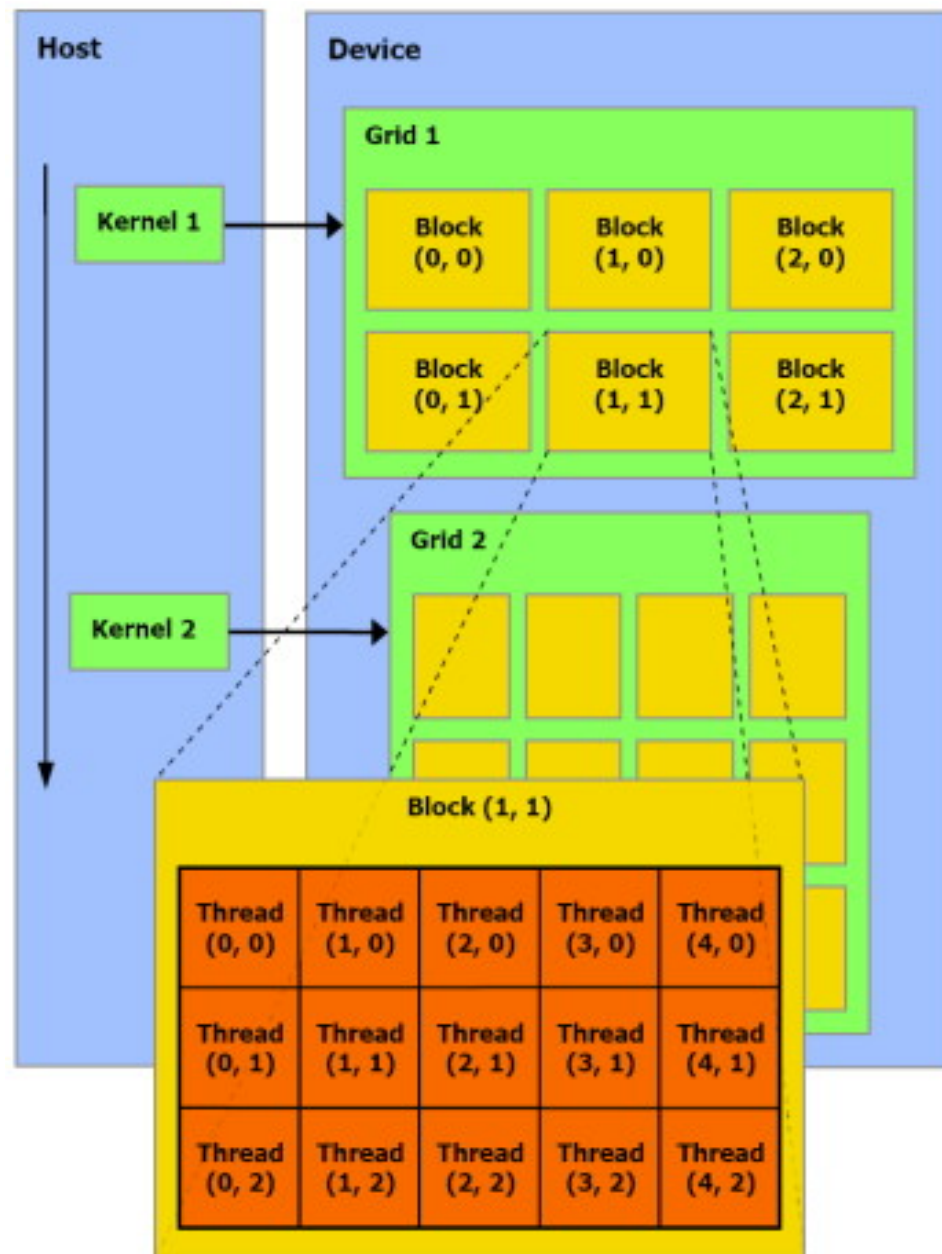
There are several kinds of parallelization, all implemented differently:

1. CPU parallelization
2. GPU parallelization
3. parallel cloud computing
4. parallelization for openMP

I will only focus on GPU parallelization, which does not completely generalize to other kinds of parallelization.

# PARALLELIZING A WORKLOAD ON A GPU

1. The CPU sends a CPU-to-GPU command called a **kernel** to a single GPU core.
2. The GPU core multitasks to execute the command:
  - a. The GPU makes  $N \cdot M$  **copies** of the kernel's code, and then runs all those copies simultaneously. Those parallel copies are called **threads**.
  - b. The  $N \cdot M$  threads are partitioned into  $N$  groups, called **blocks**, of  $M$  threads each.
  - c. The sum total of all the threads from a kernel call is a **grid**.



## IMPORTANT REMARKS:

- With one grid per kernel, GRIDS are executed SEQUENTIALLY.
- Blocks within the same grid are executed SIMULTANEOUSLY (unless otherwise specified).
- Threads within the same grid are executed SIMULTANEOUSLY (unless otherwise specified), whether they share a block or not.



## **NOTE: PARALLELIZATION HAS TWO EQUIVALENT DEFINITIONS**

1. Running different calculations simultaneously.
2. Breaking up a calculation into grids, then into blocks, and then into threads.

When I say “parallelization” in practice, I will most likely be referring to definition 2.

# WHEN TO PARALLELIZE

Calculations you want to parallelize:

- Repeated floating point arithmetic procedures that can all be done simultaneously.
- Anything that can be broken down into or framed as such.

Calculations you don't want to parallelize:

- Inherently sequential calculations, such as recursions.
- Control flow: if-then statements, etc.
- CPU system routines, such as printing to the console.

# EXAMPLES OF EASILY PARALLELIZABLE ALGORITHMS

Linear algebraic algorithms are particularly amenable to GPU computing because they involve a high volume of simple arithmetic.

I will showcase:

1. vector addition
2. the pairwise (cascading) sum
3. matrix multiplication
4. the QR factorization

# VECTOR ADDITION

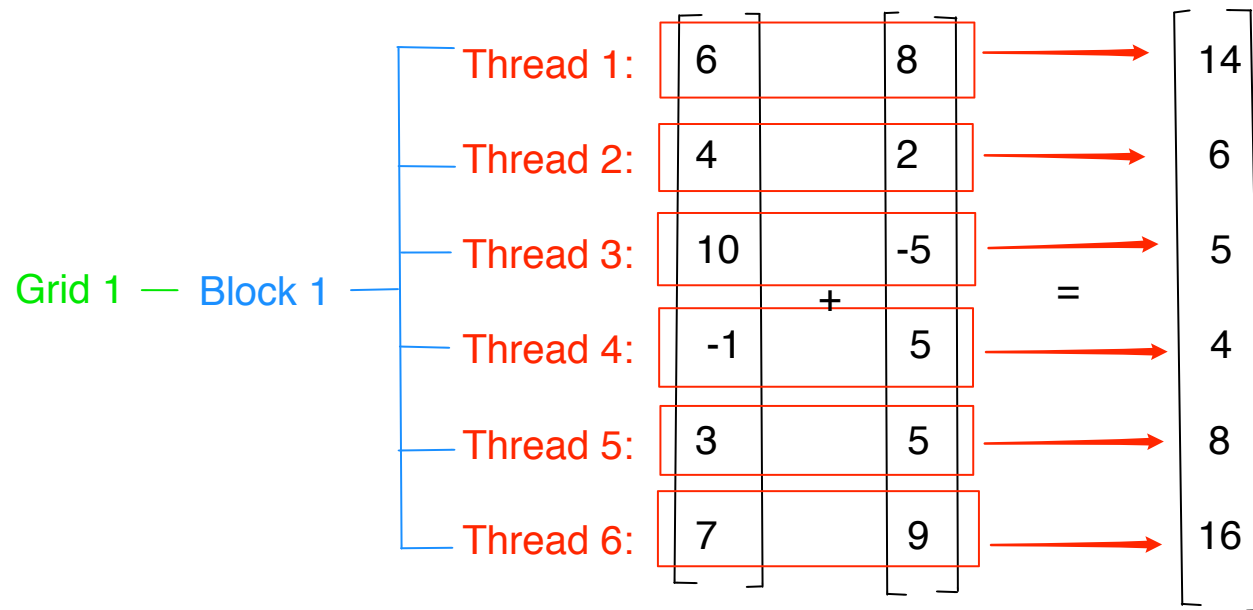
Say I have two vectors:

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

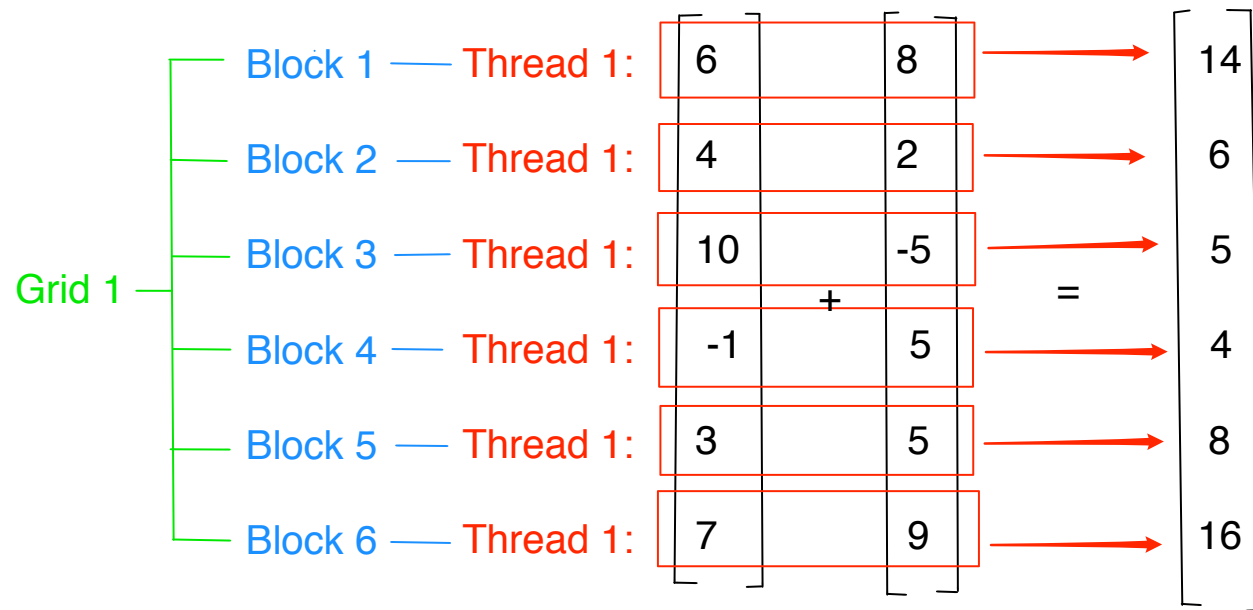
I compute their sum,  $c = a + b$ , by:

$$c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{bmatrix}$$

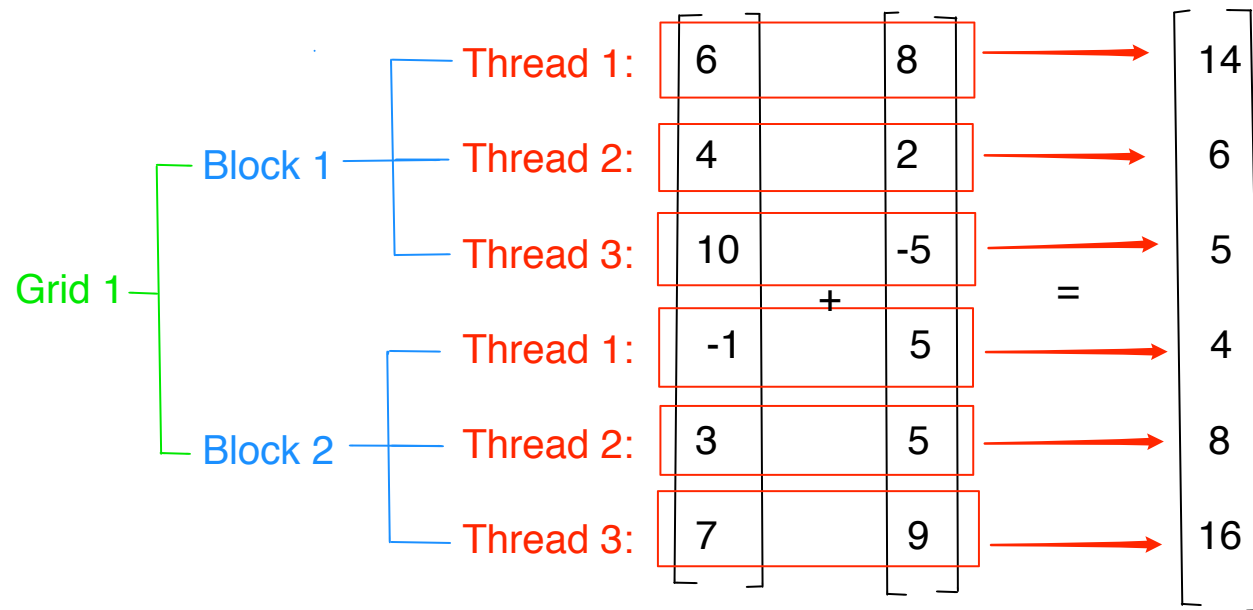
# PARALLELIZING VECTOR ADDITION: METHOD 1 OF 3



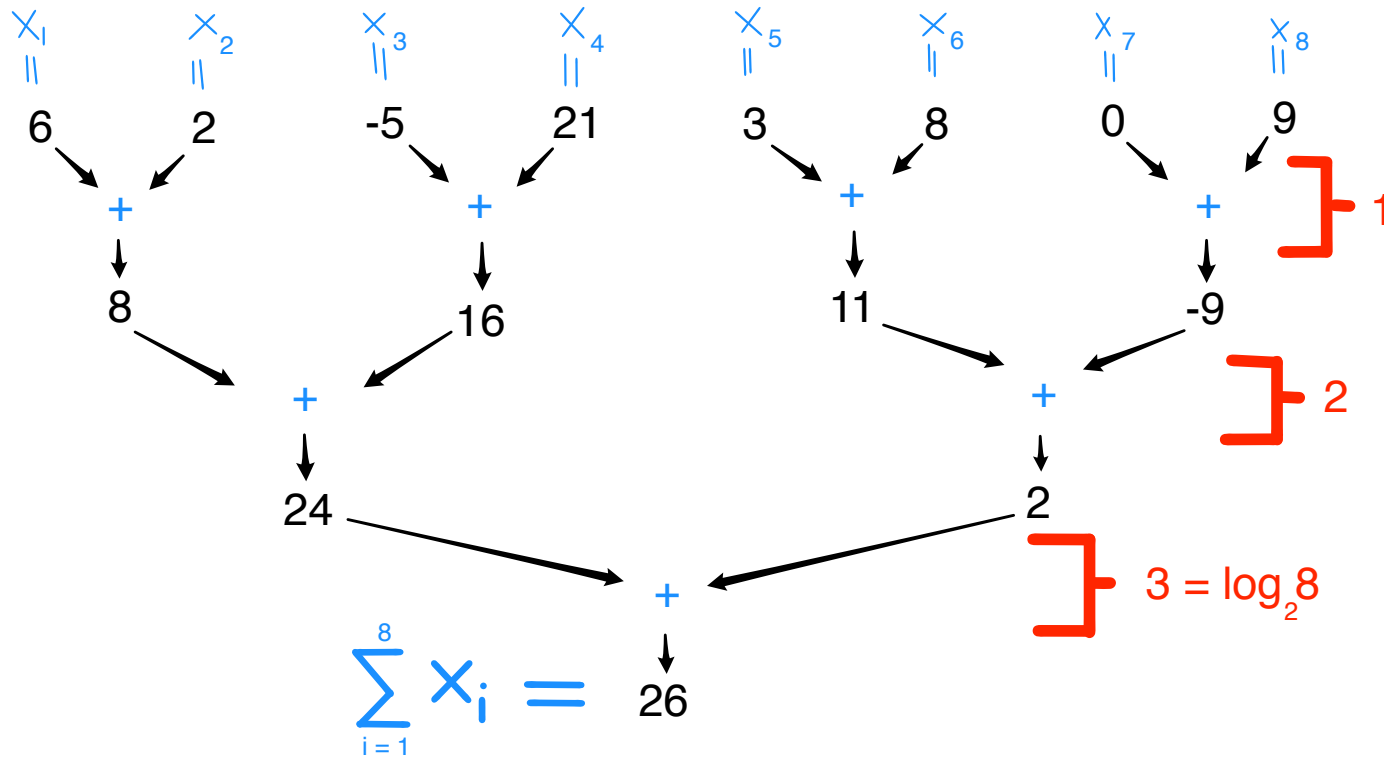
# PARALLELIZING VECTOR ADDITION: METHOD 2 OF 3



# PARALLELIZING VECTOR ADDITION: METHOD 3 OF 3



## 2. THE PAIRWISE (CASCADING) SUM





# A RIGOROUS DESCRIPTION

Suppose you have a vector  $X_0 = (x_{(0,1)}, x_{(0,2)}, \dots, x_{(0,n)})$ , where  $n = 2^m$  for some  $m > 0$ .

Compute  $\sum_{i=1}^n x_{(0,i)}$  in the following way:

1. Create a new vector:

$$X_1 = (\underbrace{x_{(0,1)} + x_{(0,2)}}_{x_{(1,1)}}, \underbrace{x_{(0,3)} + x_{(0,4)}}_{x_{(1,2)}}, \dots, \underbrace{x_{(0,n-1)} + x_{(0,n)}}_{x_{(1,n/2)}})$$

2. Create another new vector:

$$X_2 = (\underbrace{x_{(1,1)} + x_{(1,2)}}_{x_{(2,1)}}, \underbrace{x_{(1,3)} + x_{(1,4)}}_{x_{(2,2)}}, \dots, \underbrace{x_{(1,n/2-1)} + x_{(1,n/2)}}_{x_{(2,n/4)}})$$

3. Continue this process until you get a singleton vector:

$$X_m = (\underbrace{x_{(m-1,1)}, x_{(m-1,2)}}_{x_{(m,1)}})$$

Notice:  $\sum_{i=1}^n x_{(0,i)} = x_{(m,1)}$

# PARALLELIZING THE PAIRWISE SUM

Spawn one grid with a single block and  $n = 2^m$  threads.  
For each iteration  $i = 1, 2, \dots, m$ , do the following:

1. Assign thread  $j$  to compute:

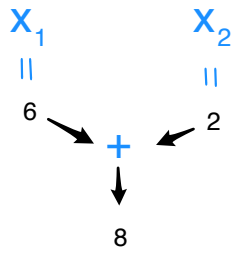
$$x_{(i,j)} = x_{(i-1, 2j-1)} + x_{(i-1, 2j)}$$

for  $j = 1, 2, \dots, \frac{n}{2^i}$ .

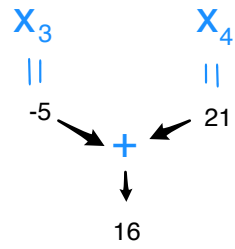
2. Wait until all the above  $\frac{n}{2^i}$  threads have completed step 1.
3. Set  $i = i + 1$  and repeat.

# EXAMPLE

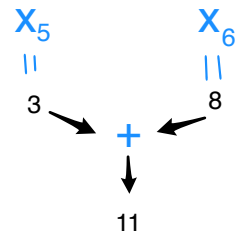
Thread 1:



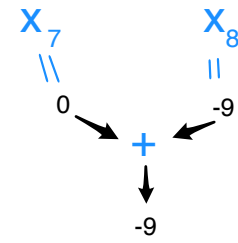
Thread 2:

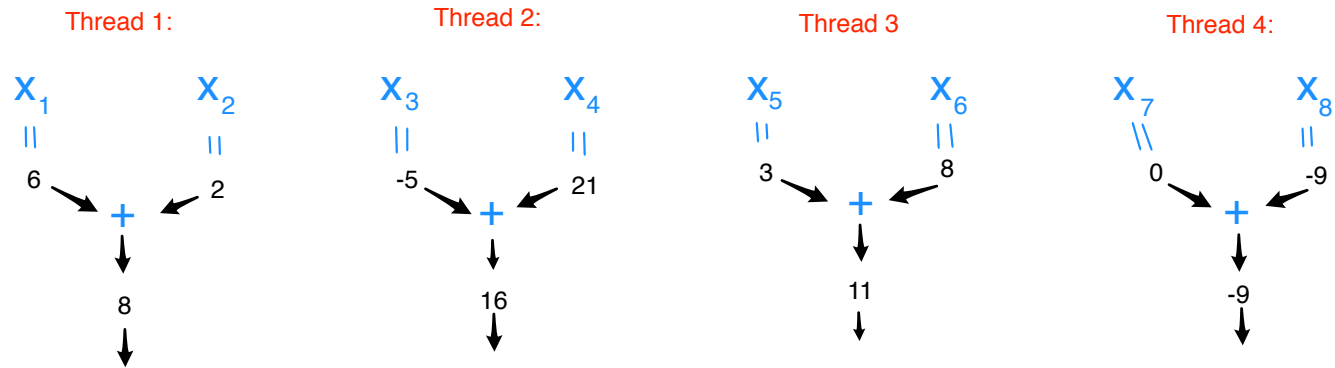


Thread 3



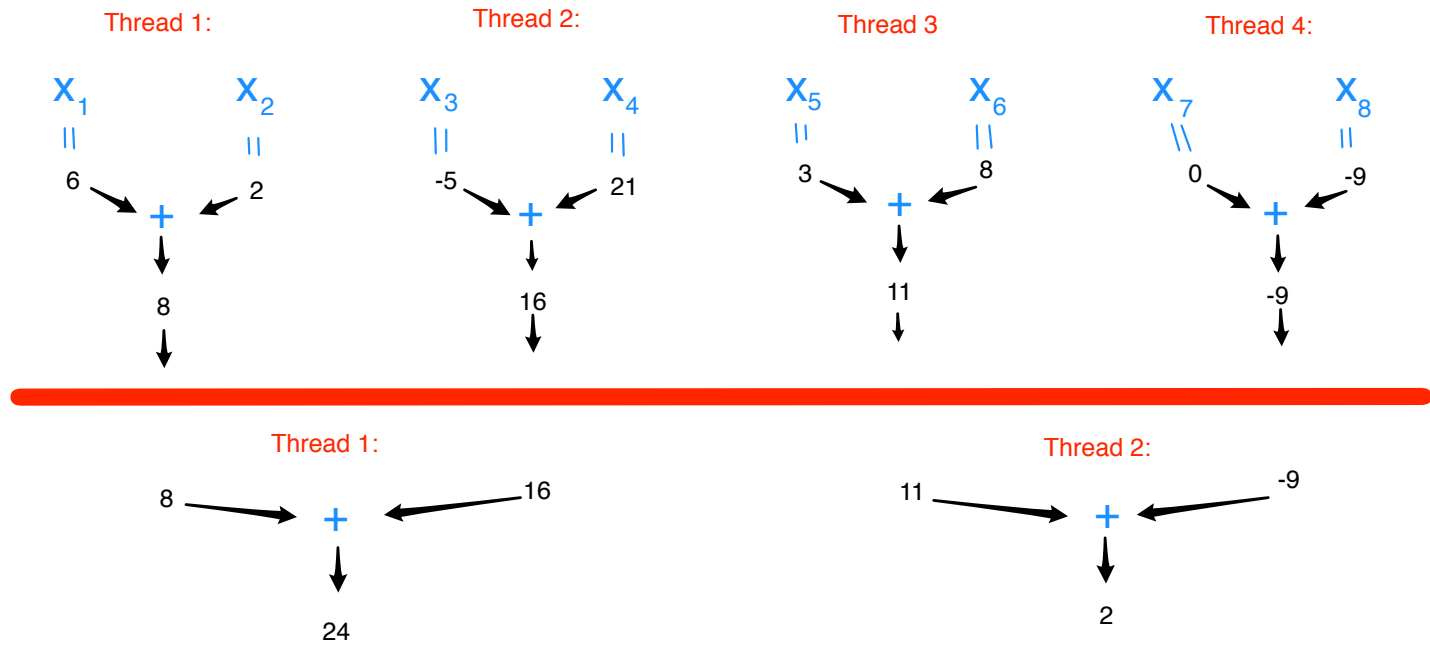
Thread 4:

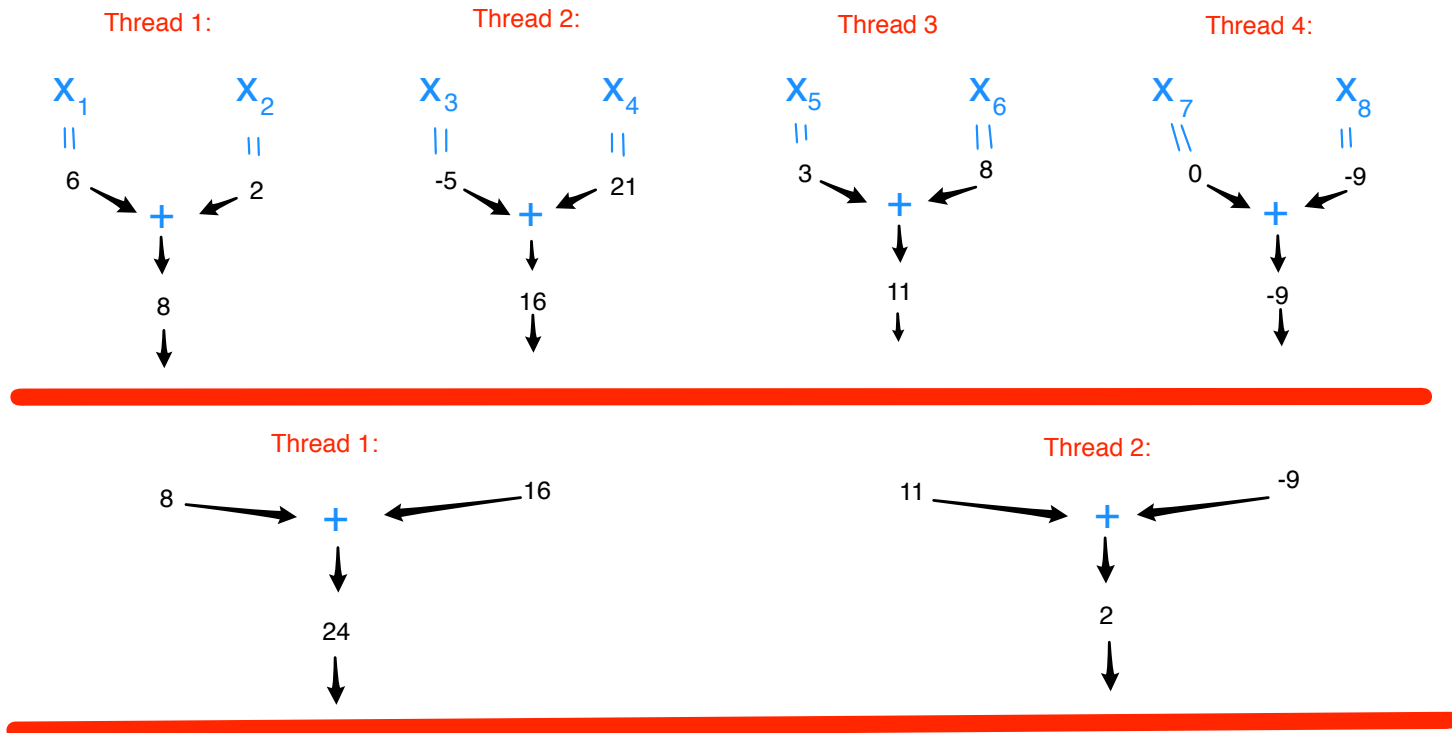




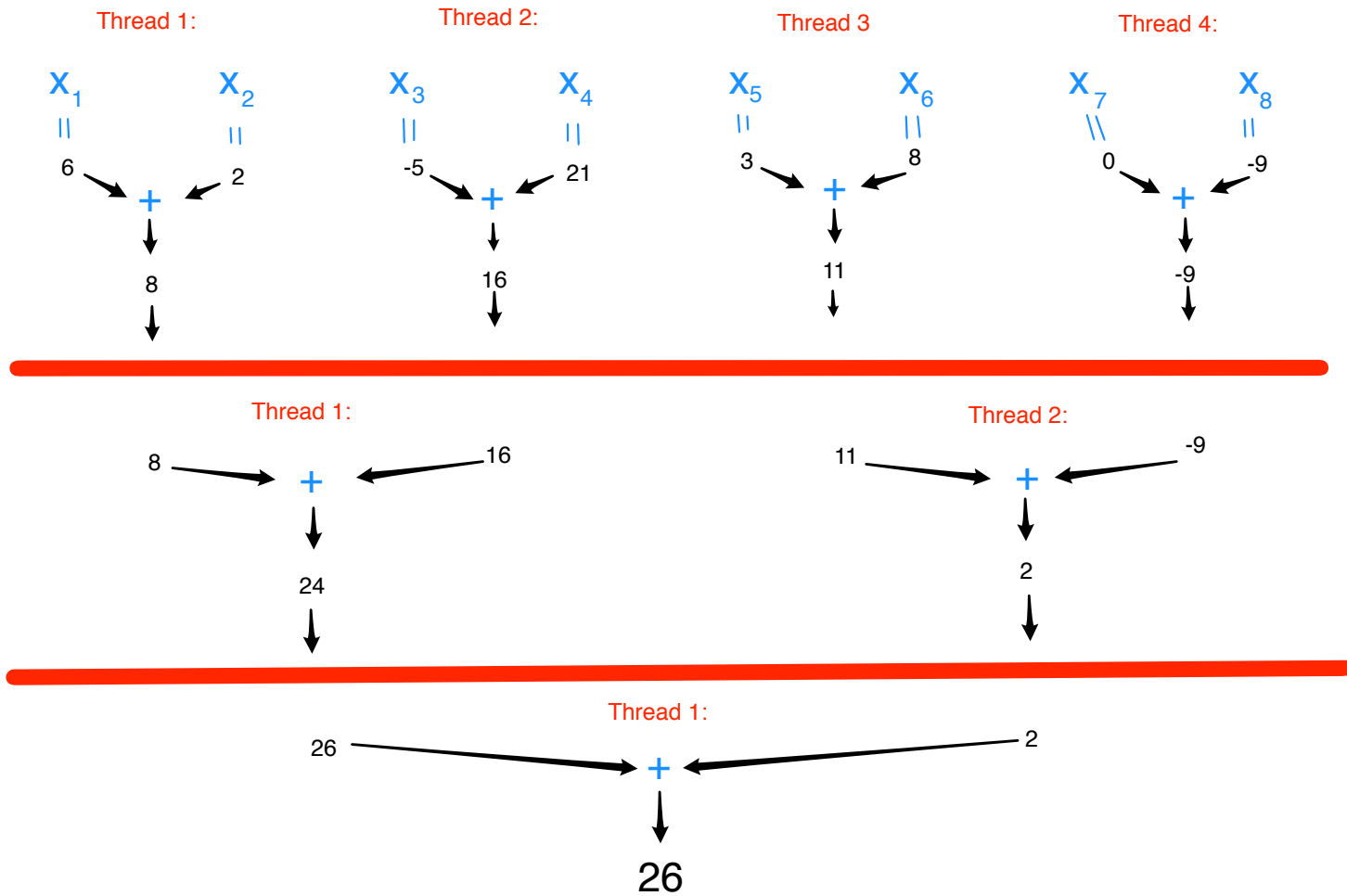
---

STOP HERE AND WAIT FOR ALL THREADS TO REACH THIS POINT!

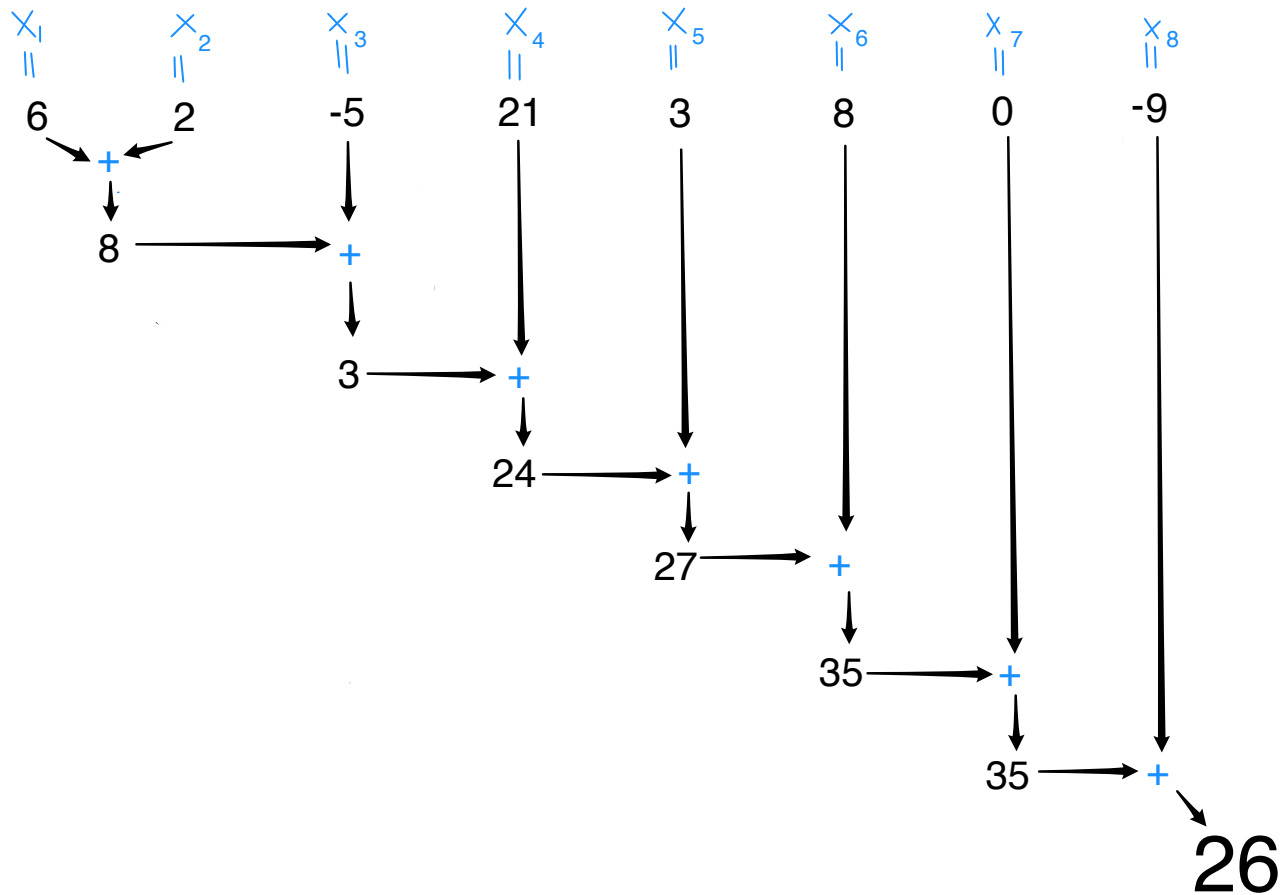




STOP AND WAIT FOR THREADS 1 AND 2 TO REACH THIS POINT!



## COMPARE TO THE SEQUENTIAL VERSION



The pairwise sum requires only  $\log_2(n)$  sequential steps, whereas the sequential sum requires  $n - 1$  steps.



## AN ASIDE: SYNCHRONIZING THREADS

**Synchronization:** Waiting for all parallel tasks to reach a checkpoint before allowing any of those tasks to proceed beyond that checkpoint.

NOTE:

- Threads from the same block can be synchronized easily.
- In general, do not try to synchronize threads from different blocks. It's possible, but extremely inefficient.

### 3. MATRIX MULTIPLICATION

Consider an  $m \times n$  matrix,  $A = (a_{ij})$ , and an  $n \times p$  matrix,  $B = (b_{ij})$ . Compute  $A \cdot B$ :

1. Break apart  $A$  into its rows:  $A = \begin{bmatrix} a_{1.} \\ a_{2.} \\ \vdots \\ a_{m.} \end{bmatrix}$ , where each  $a_{i.} = [a_{i1} \ a_{i2} \ \cdots \ a_{in}]$
2. Break apart  $B$  into its columns:  $B = [b_{.1} \ b_{.2} \ \cdots \ b_{.p}]$ , where each  $b_{.j} = \begin{bmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ b_{nj} \end{bmatrix}$
3. Compute  $C = A \cdot B$  elementwise, using the usual matrix multiplication rules to find each  $a_{i.} \cdot b_{.j}$ :

$$C = A \cdot B = \begin{bmatrix} (a_{1.} \cdot b_{.1}) & (a_{1.} \cdot b_{.2}) & \cdots & (a_{1.} \cdot b_{.p}) \\ (a_{2.} \cdot b_{.1}) & (a_{2.} \cdot b_{.2}) & & (a_{2.} \cdot b_{.p}) \\ \vdots & & \ddots & \vdots \\ (a_{m.} \cdot b_{.1}) & (a_{m.} \cdot b_{.2}) & \cdots & (a_{m.} \cdot b_{.p}) \end{bmatrix}$$

i.e.:

$$C_{(i,j)} = a_{i.} \cdot b_{.j}$$

# PARALLELIZING MATRIX MULTIPLICATION

Spawn one grid with  $m \cdot p$  blocks. Assign block  $(i, j)$  to compute  $C_{(i,j)} = a_{i.} \cdot b_{.j}$  using the following steps:

1. Spawn  $n$  threads.
2. Tell the  $k$ 'th thread to compute  $c_{ijk} = a_{ik}b_{kj}$ .
3. Synchronize the  $n$  threads to make sure we have finished calculating all of  $c_{ij1}, c_{ij2}, \dots, c_{ijn}$  before proceeding.
4. Compute  $C_{(i,j)} = \sum_{k=1}^n c_{ijk}$  as a pairwise sum.

## EXAMPLE

Say I want to compute  $A \cdot B$ , where:

$$A = \begin{bmatrix} 1 & 2 \\ -1 & 5 \\ 7 & -9 \end{bmatrix} \quad B = \begin{bmatrix} 8 & 8 & 7 \\ 3 & 5 & 2 \end{bmatrix}$$

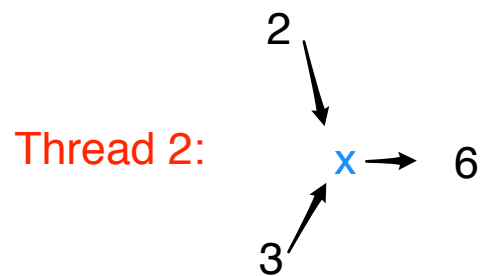
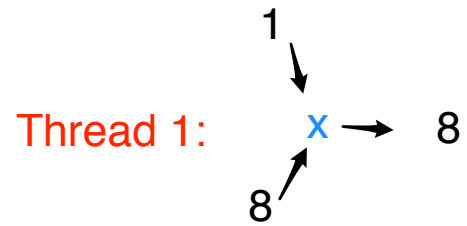
which I'm setting up as:

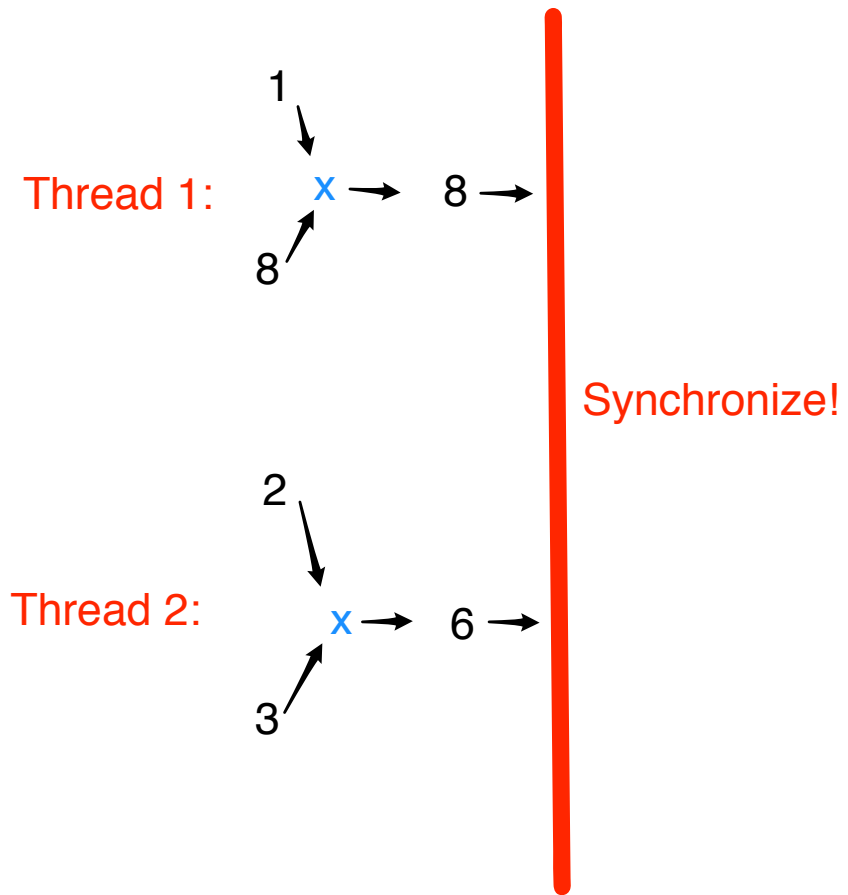
$$C = A \cdot B = \begin{bmatrix} \left( \begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \right) & \left( \begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix} \right) & \left( \begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 2 \end{bmatrix} \right) \\ \left( \begin{bmatrix} -1 & 5 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \right) & \left( \begin{bmatrix} -1 & 5 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix} \right) & \left( \begin{bmatrix} -1 & 5 \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 2 \end{bmatrix} \right) \\ \left( \begin{bmatrix} 7 & -9 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \right) & \left( \begin{bmatrix} 7 & -9 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix} \right) & \left( \begin{bmatrix} 7 & -9 \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 2 \end{bmatrix} \right) \end{bmatrix}$$

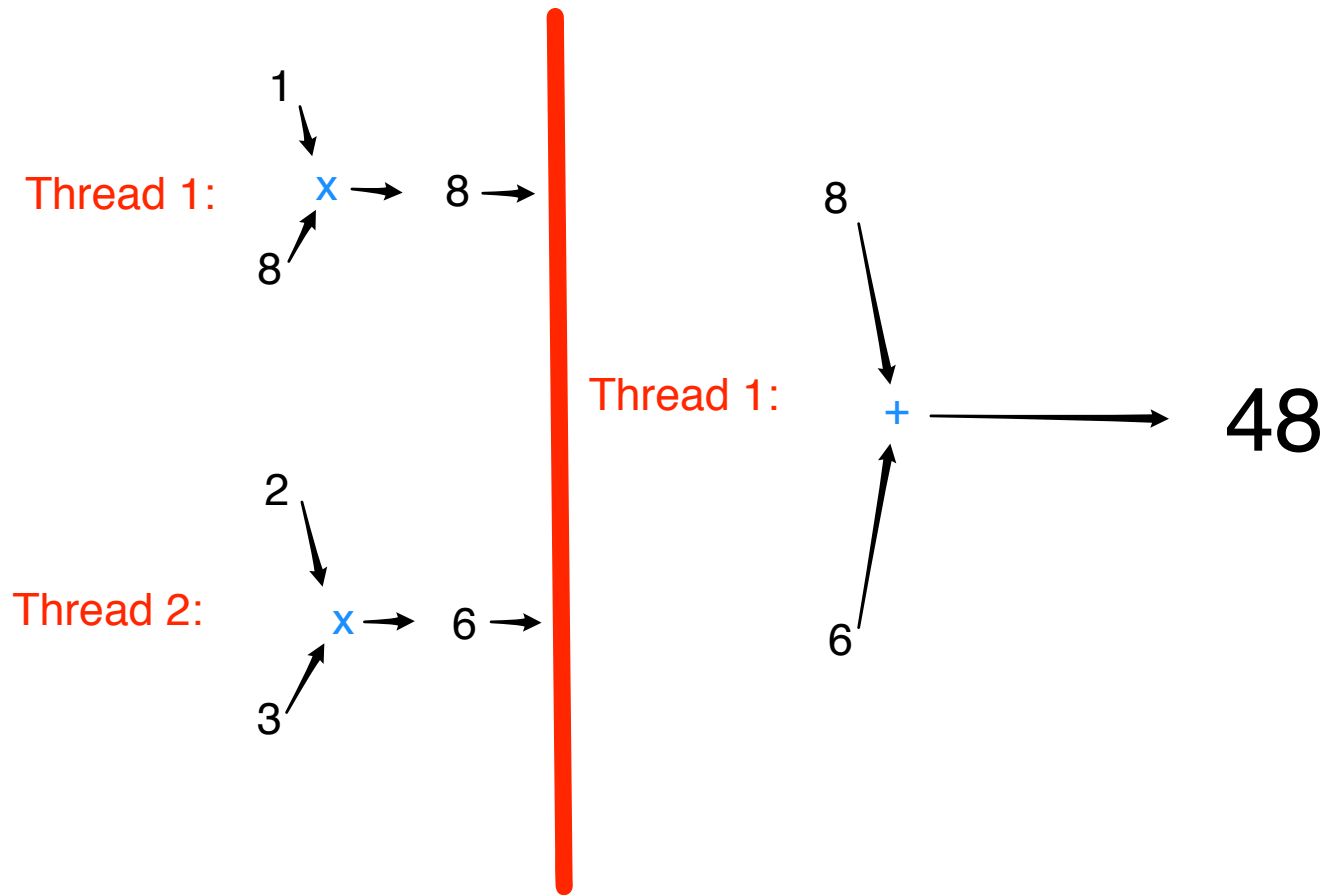
$$\begin{bmatrix} \overset{\text{Block (1, 1)}}{\left( \begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \right)} & \overset{\text{Block (1, 2)}}{\left( \begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix} \right)} & \overset{\text{Block (1, 3)}}{\left( \begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 2 \end{bmatrix} \right)} \\ \overset{\text{Block (2, 1)}}{\left( \begin{bmatrix} -1 & 5 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \right)} & \overset{\text{Block (2, 2)}}{\left( \begin{bmatrix} -1 & 5 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix} \right)} & \overset{\text{Block (2, 3)}}{\left( \begin{bmatrix} -1 & 5 \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 2 \end{bmatrix} \right)} \\ \overset{\text{Block (3, 1)}}{\left( \begin{bmatrix} 7 & -9 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \right)} & \overset{\text{Block (3, 2)}}{\left( \begin{bmatrix} 7 & -9 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix} \right)} & \overset{\text{Block (3, 3)}}{\left( \begin{bmatrix} 7 & -9 \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 2 \end{bmatrix} \right)} \end{bmatrix}$$

We don't need to synchronize the blocks because we can compute them independently.

Consider Block (1,1), which computes  $\begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix}$ :









# LECTURE SERIES MATERIALS

These lecture slides, a tentative syllabus for the whole lecture series, and code are available at:

<https://github.com/wlandau/gpu>.

After logging into your home directory on impact1, type:

```
git clone https://github.com/wlandau/gpu
```

into the command line to download all the course materials.

## REFERENCES

Lay, David C. *Linear Algebra and Its Applications*. 3rd Ed. Addison Wesley, 2006.

J. Sanders and E. Kandrot. *CUDA by Example*. Addison-Wesley, 2010.