A CODELESS INTRODUCTION TO GPU PARALLELISM

Will Landau, Prof. Jarad Niemi

OUTLINE

A. A review of GPU parallelism

B. How to GPU-parallelize the following:

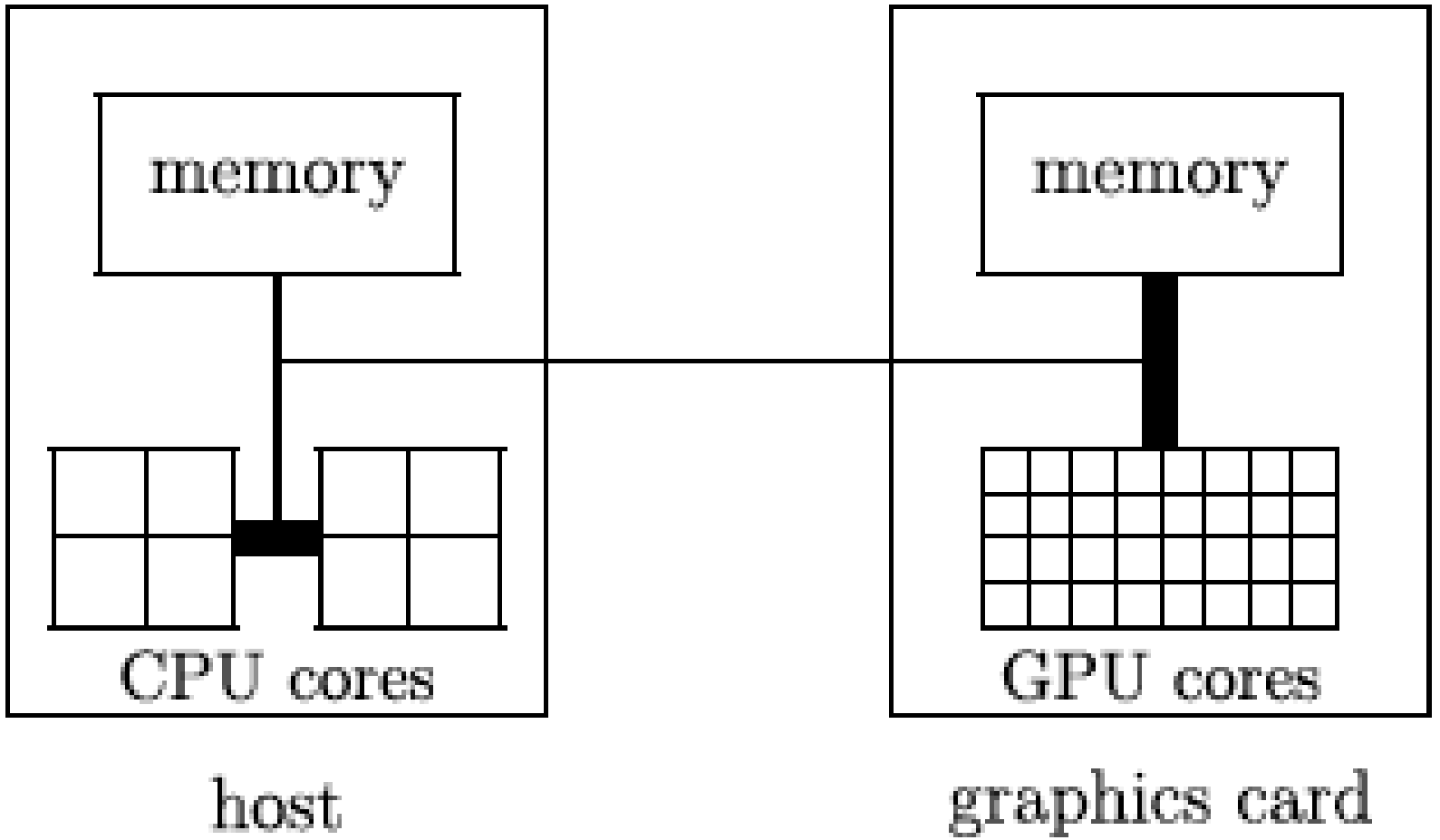
   1. vector addition
   2. the pairwise (cascading) sum
   3. matrix multiplication

HOW THE CPU AND GPU WORK TOGETHER

A GPU can't run a whole computer on its own because it doesn't have access to all the computer's hardware.

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

The CPU uses the GPU like a human would use a hand-held calculator: the CPU does all the main thinking and the GPU does the most cumbersome bits and pieces of number-crunching.


memory
memory
CPU cores
GPU cores
host
graphics card

GPU PARALLELISM

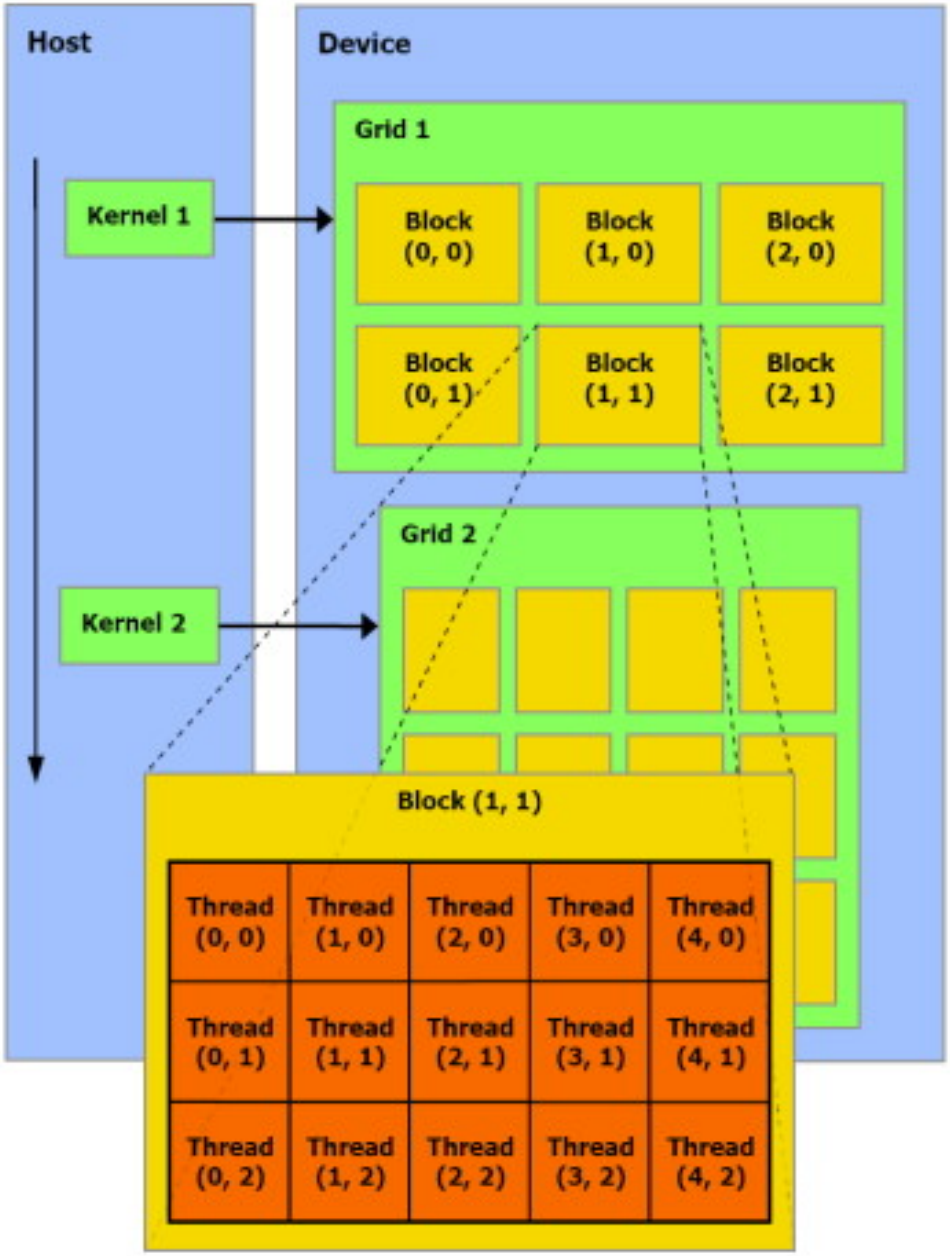
1. The CPU sends a CPU-to-GPU command called a **kernel** to a single GPU core.

2. The GPU core multitasks to execute the command:

   a. The GPU makes $B \cdot T$ copies of the kernel's code, and then runs all those copies simultaneously. Those parallel copies are called **threads**.

   b. The $B \cdot T$ threads are partitioned into B groups, called **blocks**, of T threads each.

   c. The sum total of all the threads from a kernel call is a **grid**.


Host
Device
Grid 1
Kernel 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kernel 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

WHEN TO PARALLELIZE

Calculations you want to parallelize:

- Highly repetitive floating point arithmetic procedures that can all be done simultaneously.

Calculations you don't want to parallelize:

- Inherently sequential calculations, such as recursions.
- Lengthy 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

VECTOR ADDITION

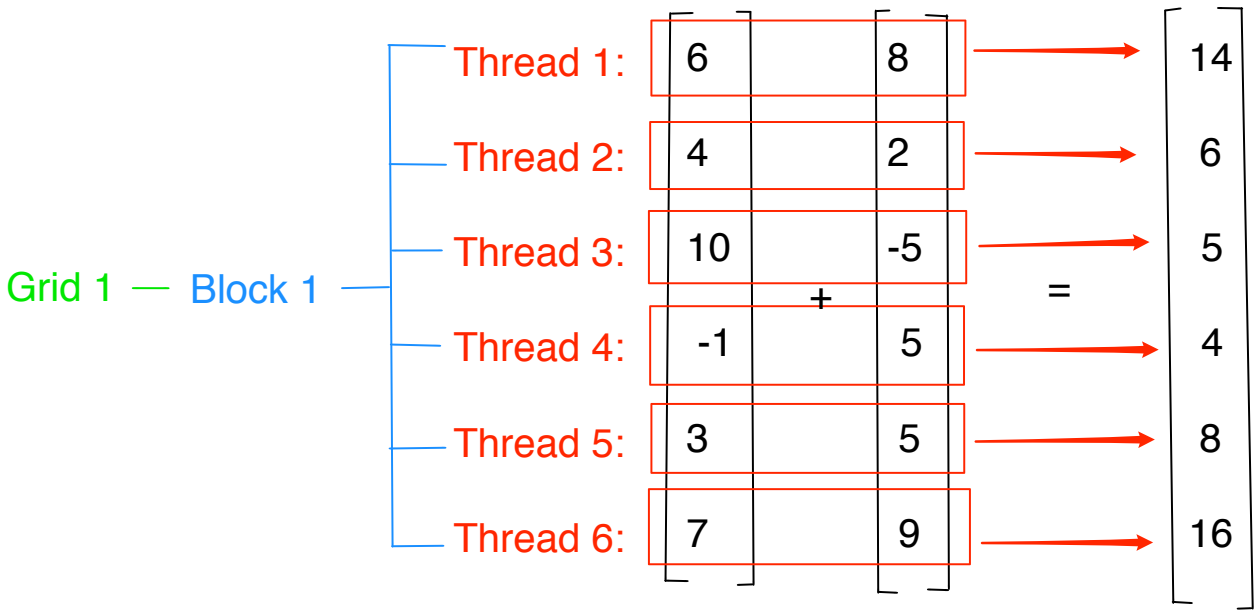
Say I have two vectors:

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

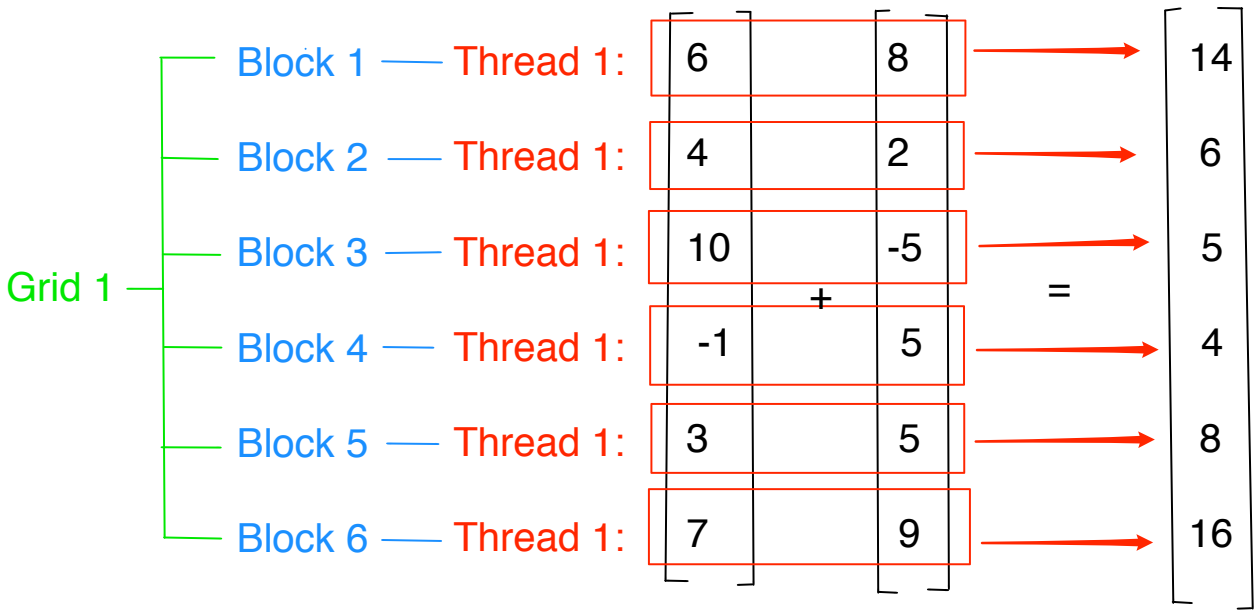
I compute their sum, $c = a + b$, by:

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

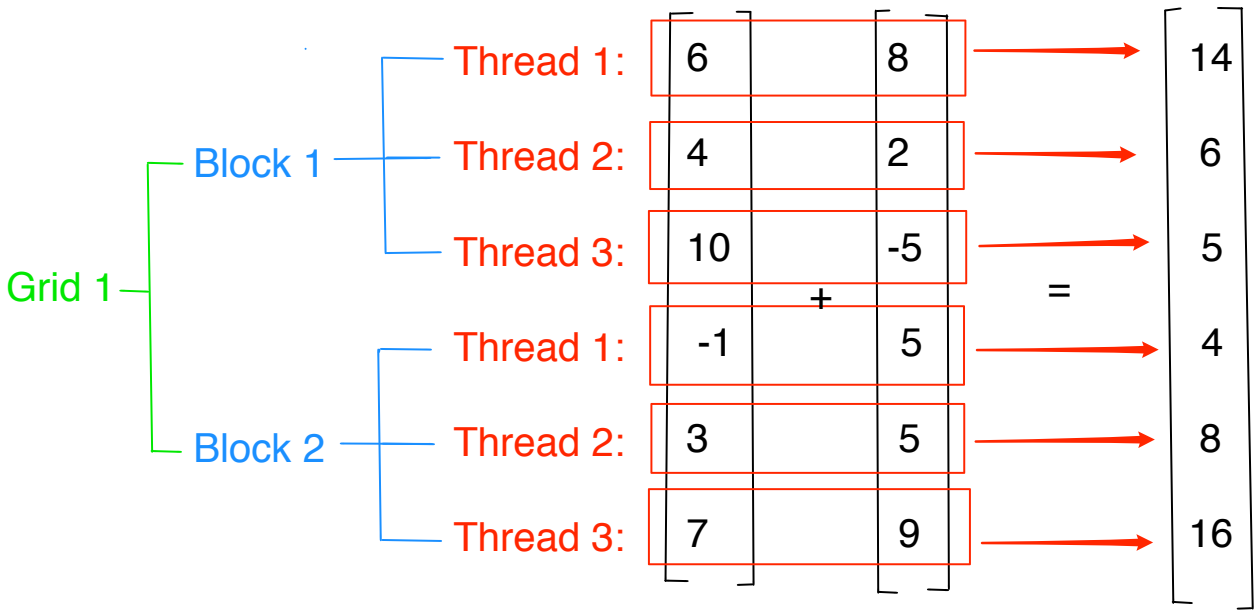
PARALLELIZING VECTOR ADDITION: METHOD 1 OF 3



PARALLELIZING VECTOR ADDITION: METHOD 2 OF 3



PARALLELIZING VECTOR ADDITION: METHOD 3 OF 3

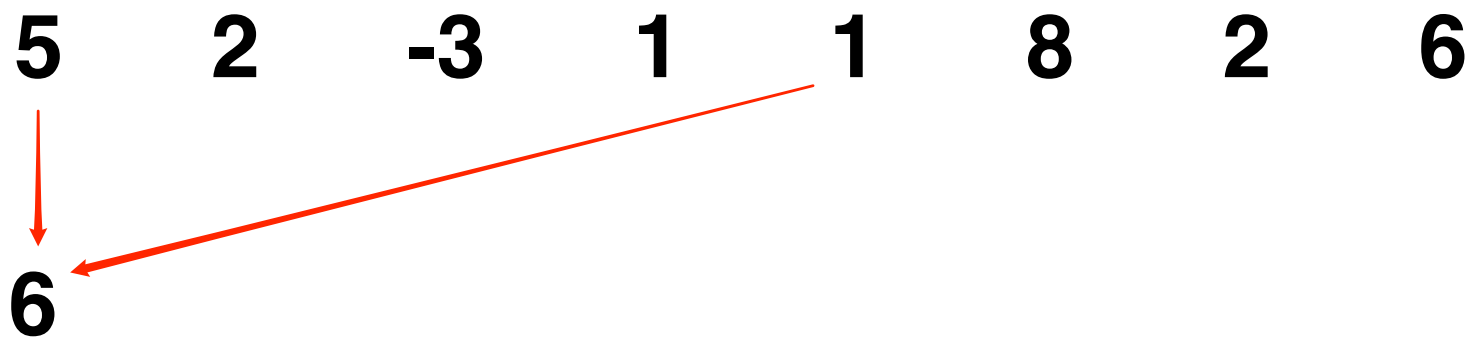


THE PAIRWISE SUM

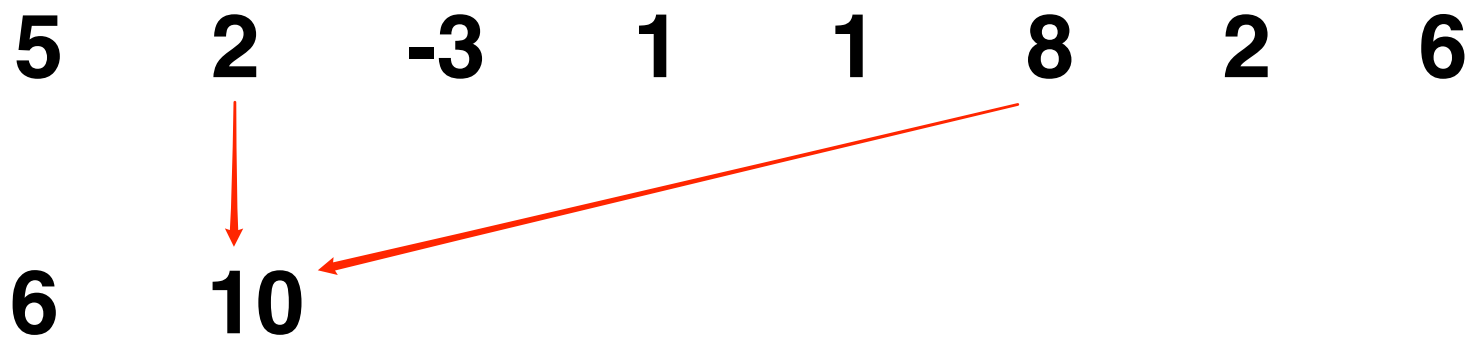
Let's take the pairwise sum of the vector:

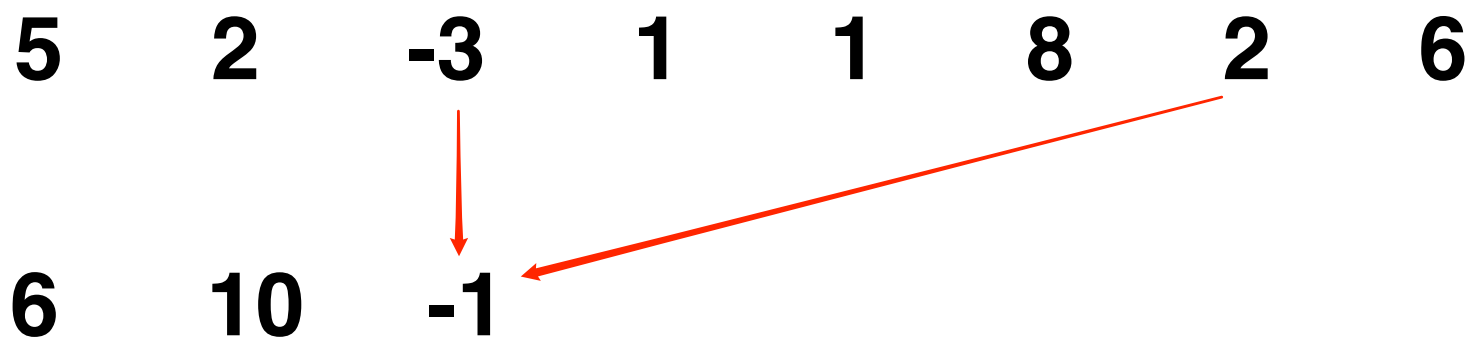
$$(5,\ 2,\ -3,\ 1,\ 1,\ 8,\ 2,\ 6)$$

Using one block of 4 threads.

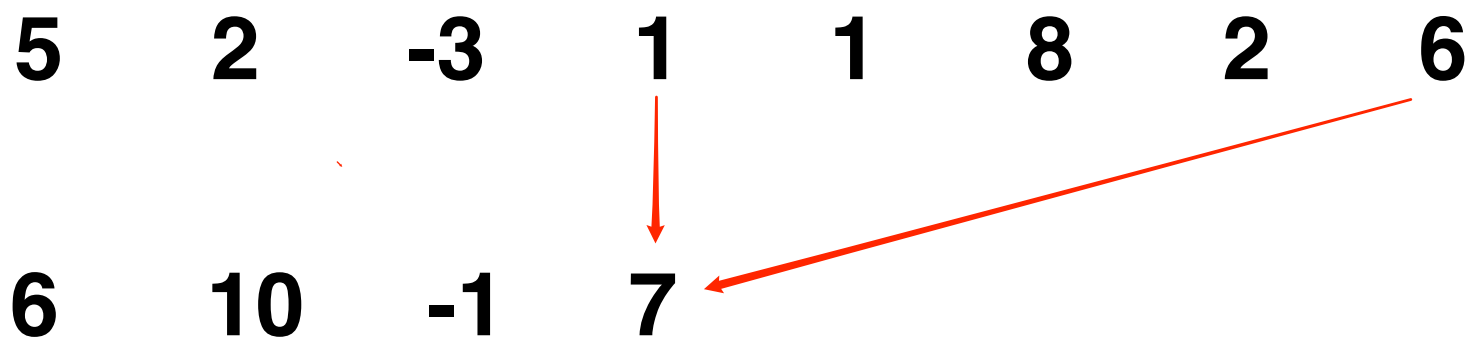


Thread 0


5 2 -3 1 1 8 2 6
6 10
Thread 1



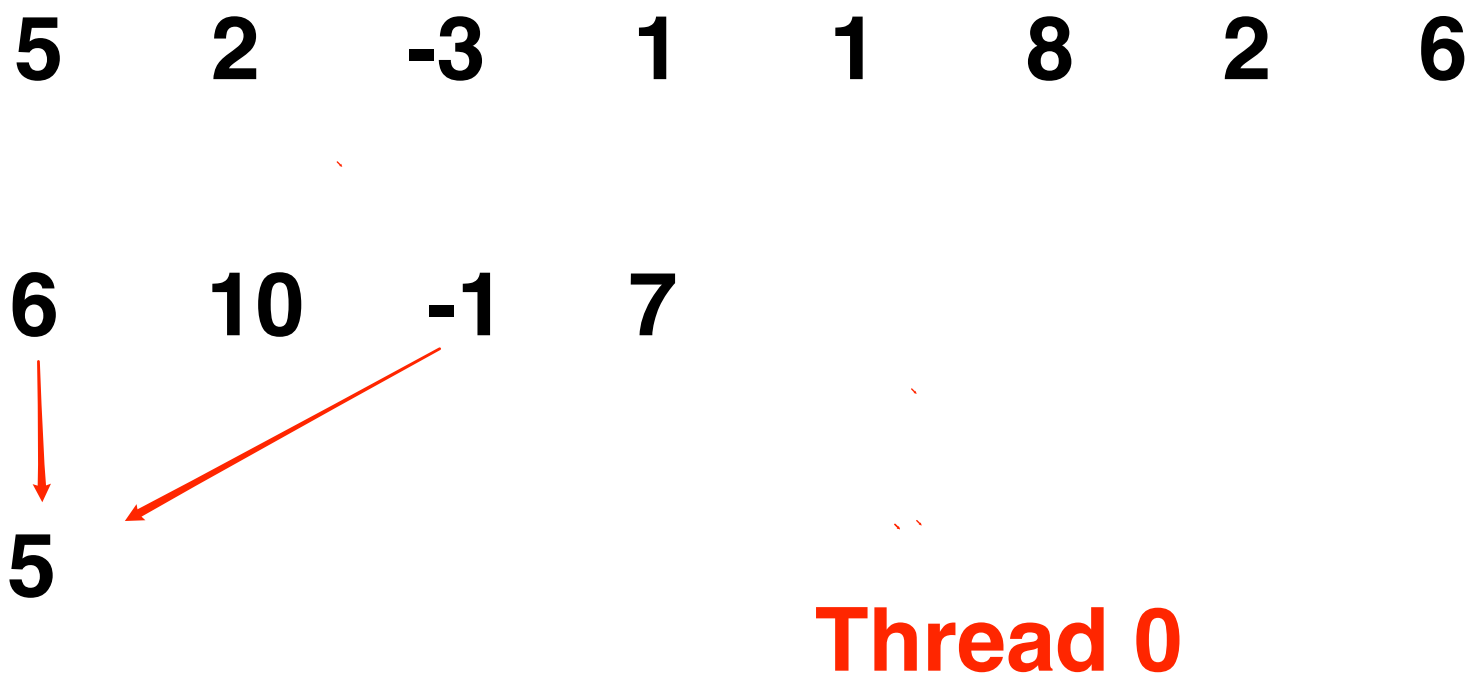
Thread 2

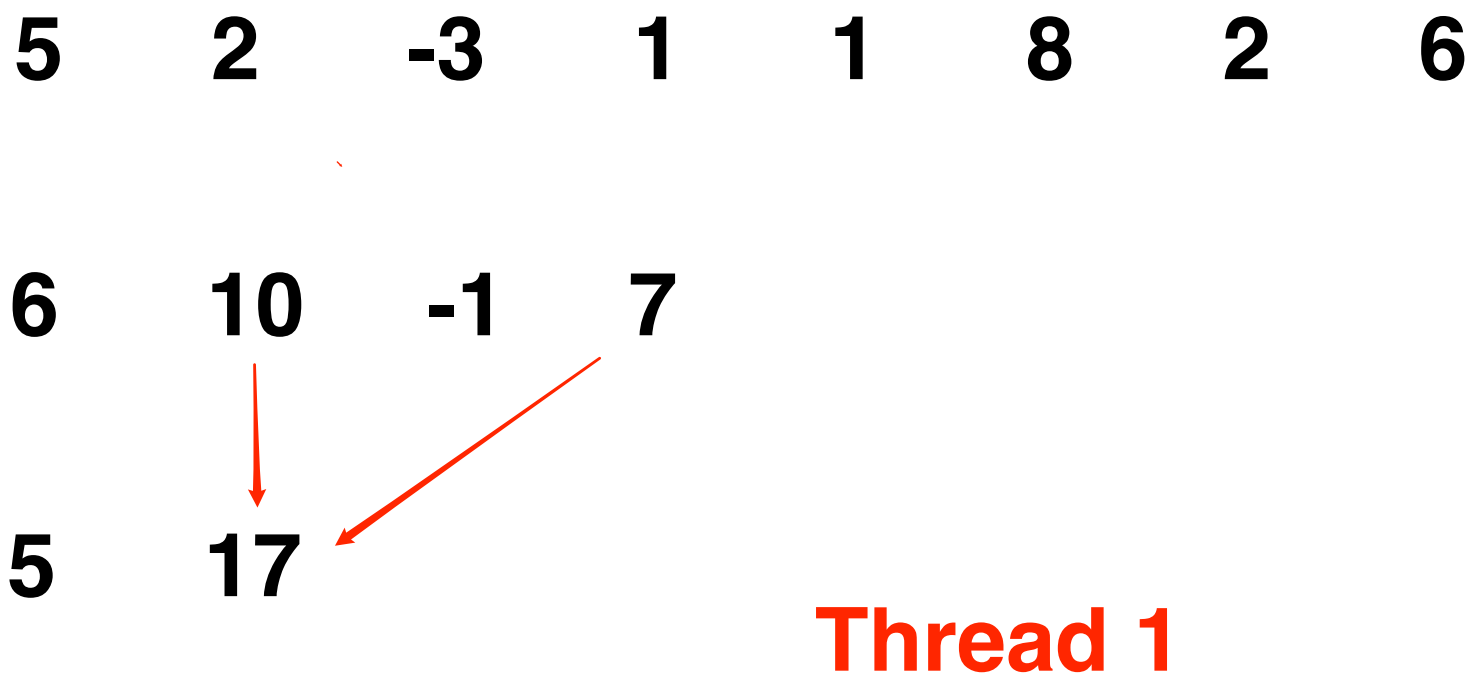

5 2 -3 1 1 8 2 6
6 10 -1 7
Thread 3

5 2 -3 1 1 8 2 6

6 10 -1 7

Synchronize threads


5 2 -3 1 1 8 2 6
6 10 -1 7
5
Thread 0


5 2 -3 1 1 8 2 6
6 10 -1 7
5 17
Thread 1

5 2 -3 1 1 8 2 6

6 10 -1 7

5 17

Synchronize Threads

5 2 -3 1 1 8 2 6

6 10 -1 7

5 17

22

Thread 0

AN ASIDE: SYNCHRONIZING THREADS

Synchronization: Waiting for all parallel tasks to reach a checkpoint before allowing any of them to proceed.

- 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.

A RIGOROUS DESCRIPTION OF THE PAIRWISE SUM

Suppose you have a vector $X_0 = (x_{(0,0)}, x_{(0,2)}, \ldots, x_{(0,n-1)})$, 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:
   $X_1 = (\underbrace{x_{(0,\ 0)} + x_{(0,\ n/2)}}_{x_{(1,\ 0)}},\ \underbrace{x_{(0,1)} + x_{(0,\ n/2+1)}}_{x_{(1,\ 1)}},\ \ldots, \underbrace{x_{(0,\ n/2-1)} + x_{(0,\ n-1)}}_{x_{(1,\ n/2-1)}})$
2. Create another new vector:
   $X_2 = (\underbrace{x_{(1,\ 0)} + x_{(1,\ n/4)}}_{x_{(2,\ 0)}},\ \underbrace{x_{(1,\ 1)} + x_{(1,\ n/4+1)}}_{x_{(2,\ 1)}},\ \ldots, \underbrace{x_{(1,n/4-1)} + x_{(1,\ n/2-1)}}_{x_{(2,\ n/4-1)}})$
3. Continue this process until you get a singleton vector:
   $X_m = (\underbrace{x_{(m-1,0)} + x_{(m-1,1)}}_{x_{(m,0)}})$

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

PARALLELIZING THE PAIRWISE SUM

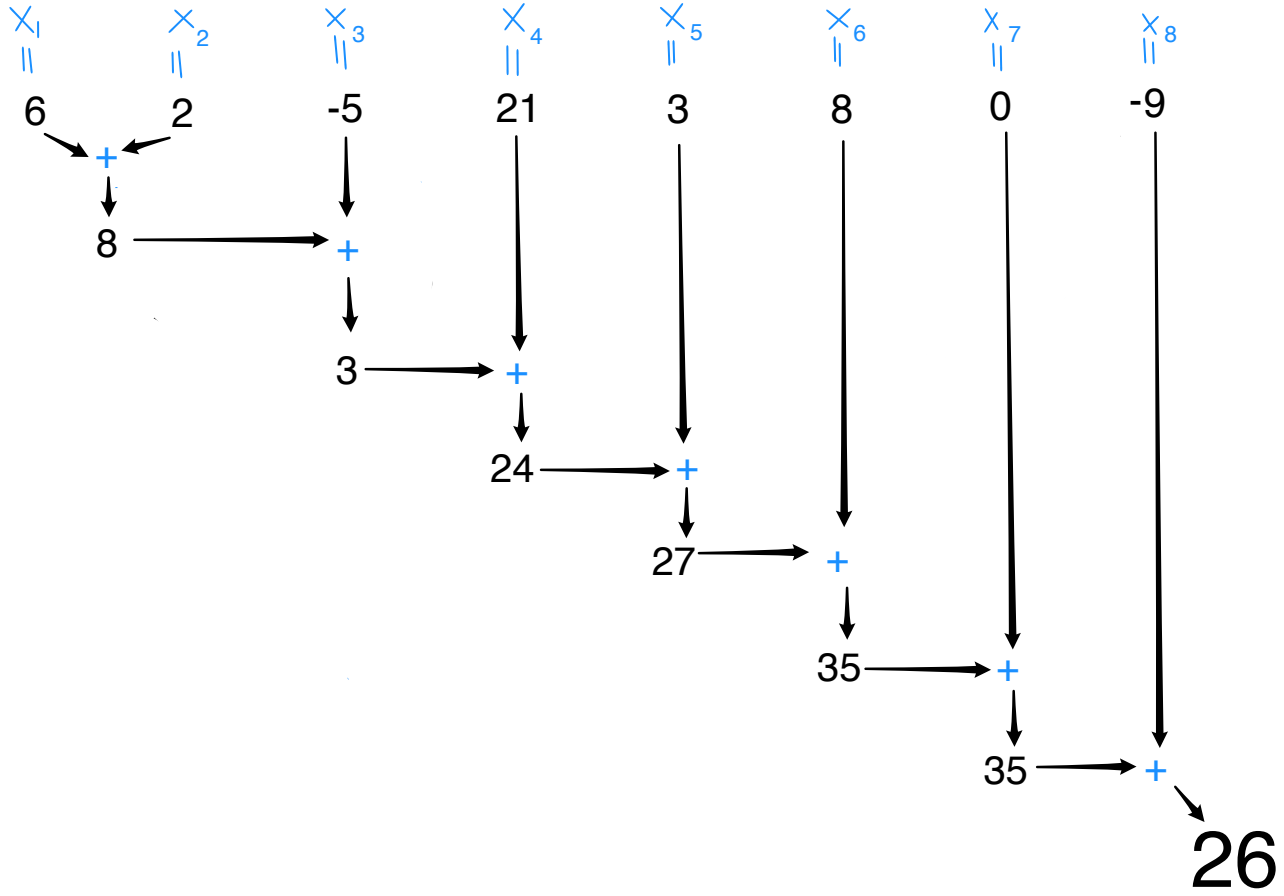
Spawn one grid with a single block and $n/2$ threads ($n = 2^m$). Starting with $i = 1$, do the following:

1. Set offset $= n/2^i$.
2. Assign thread j to compute:

   $$x_{(i,j)} = x_{(i-1,\ j)} + x_{(i-1,\ j+\text{offset})}$$

   for $j = 0, 2, \cdots, \text{offset} - 1$.
3. Wait until all the above $\frac{n}{2^i}$ threads have completed step 2.
4. Integer divide offset by 2. Repeat if offset > 0.

ASIDE: COMPARE TO THE SEQUENTIAL VERSION



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

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: $A = \begin{bmatrix} a_{1\cdot} \\ a_{2\cdot} \\ \vdots \\ a_{m\cdot} \end{bmatrix}$, where each $a_{i\cdot} = \begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{in} \end{bmatrix}$

2. Break apart B into its columns: $B = \begin{bmatrix} b_{\cdot 1} & b_{\cdot 2} & \cdots & b_{\cdot p} \end{bmatrix}$, where each $b_{\cdot j} = \begin{bmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ b_{nj} \end{bmatrix}$

3. Compute $C = A \cdot B$ elementwise, using the usual matrix multiplication rules to find each $a_i \cdot b_j$:

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

i.e.:

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

PARALLELIZING MATRIX MULTIPLICATION

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

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 $c_{ij1}, c_{ij2}, \ldots, c_{ijn}$ before proceeding.
4. Compute $C_{(i,j)} = \sum_{k=1}^{n} c_{ijk}$ as a pairwise sum.

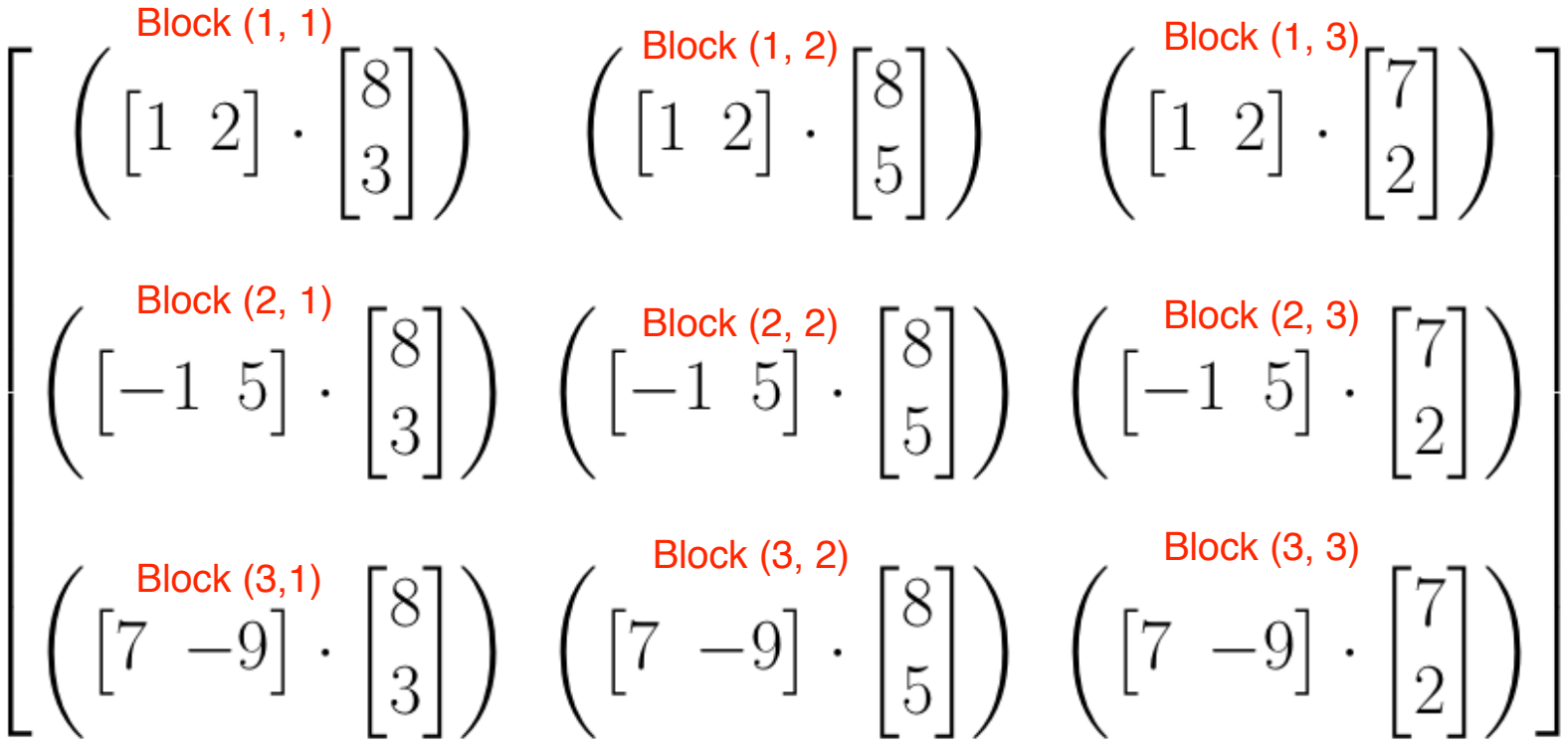
EXAMPLE

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

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

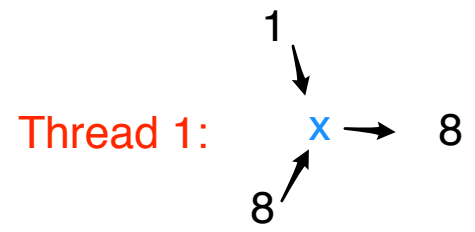
which I'm setting up as:

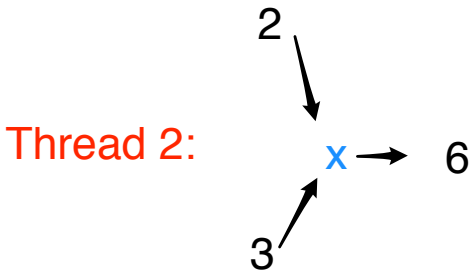
$$C = A \cdot B = \begin{bmatrix} \left([1 \ \ 2] \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \right) & \left([1 \ \ 2] \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix} \right) & \left([1 \ \ 2] \cdot \begin{bmatrix} 7 \\ 2 \end{bmatrix} \right) \\ \left([-1 \ \ 5] \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix}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

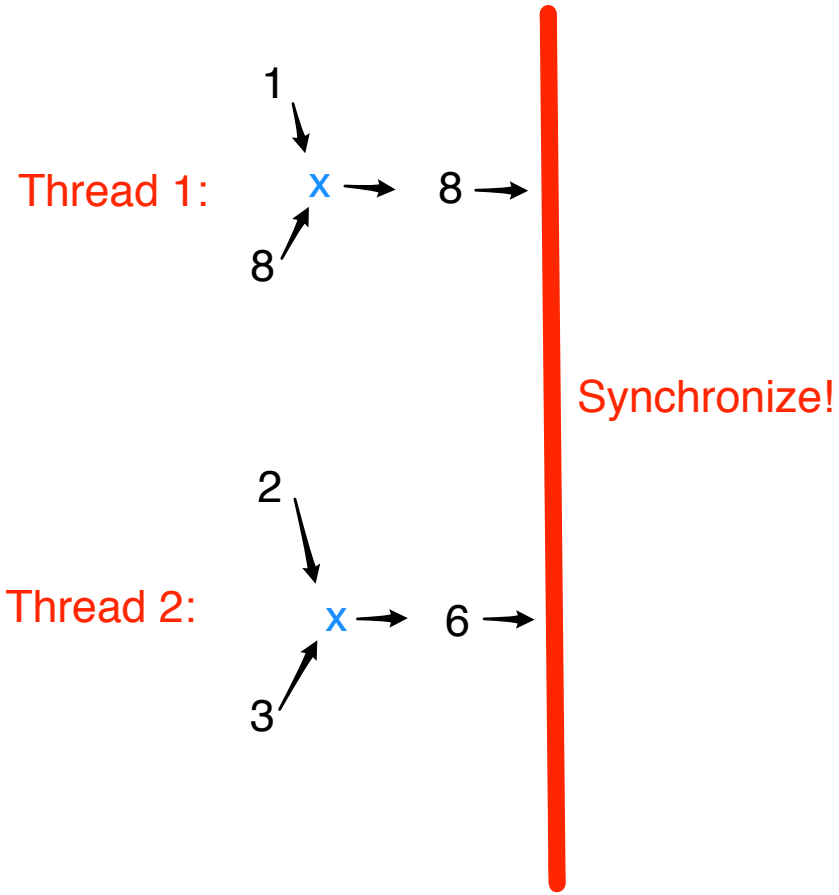


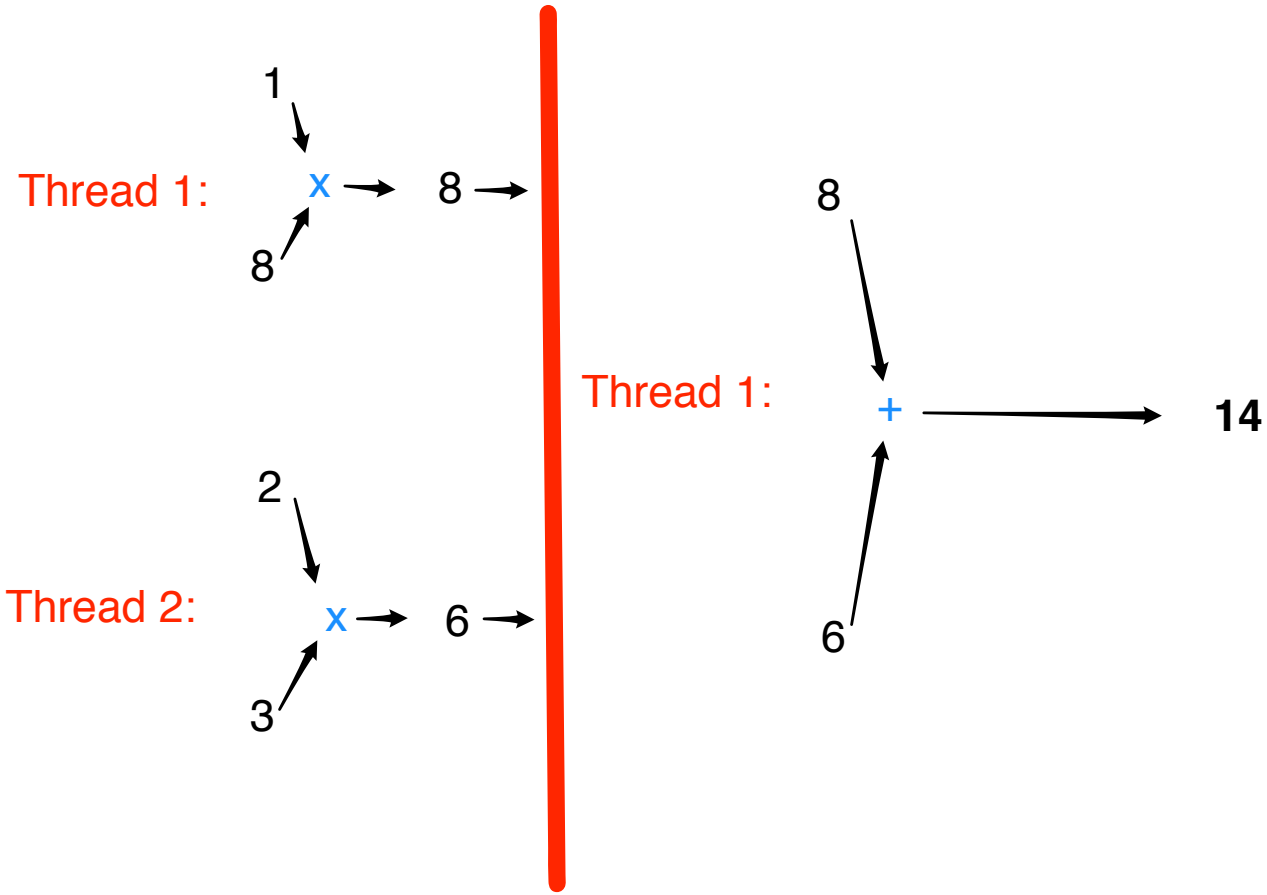
We don't need to synchronize the blocks because they can do their jobs independently.

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






Thread 1:
1
x
8
8
Synchronize!
Thread 2:
2
x
6
3


Thread 1:
1
x
8
8
Thread 2:
2
x
6
3
Thread 1:
8
+
6
14

OUTLINE

A. A review of GPU parallelism

B. How to GPU-parallelize the following:

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

PREVIEW: skeleton.cu, A BARE BONES CUDA C WORKFLOW

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

__global__ void some_kernel(...){...}

int main (void){
  // Declare all variables.
  ...
  // Dynamically allocate host memory.
  ...
  // Dynamically allocate device memory.
  ...
  // Write to host memory.
  ...
  // Copy host memory to device memory.
  ...
  // Execute kernel on the device.
  some_kernel<<< num_blocks, num_theads_per_block >>>(...);

  // Write device memory back to host memory.
  ...
  // Free dynamically-allocated host memory
  ...
  // Free dynamically-allocated device memory
  ...
}
```

MATERIALS

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

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

After logging into you home directory on impact1, type:

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

into the command line to download all the materials.

REFERENCES

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

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