An Introduction to CUDA and GPU Programming

Scientific Computing and Data Analysis

OKINAWA INSTITUTE OF SCIENCE AND TECHNOLOGY GRADUATE UNIVERSITY

沖縄科学技術大学院大学

GPU

- GPU is a ”Graphics Processing Unit”
  - Developed for games
  - High computational capability

- CUDA is NVIDIAs C/C++ language extension for HPC computing on GPUs
  - But other options exist: OpenCL, OpenACC...

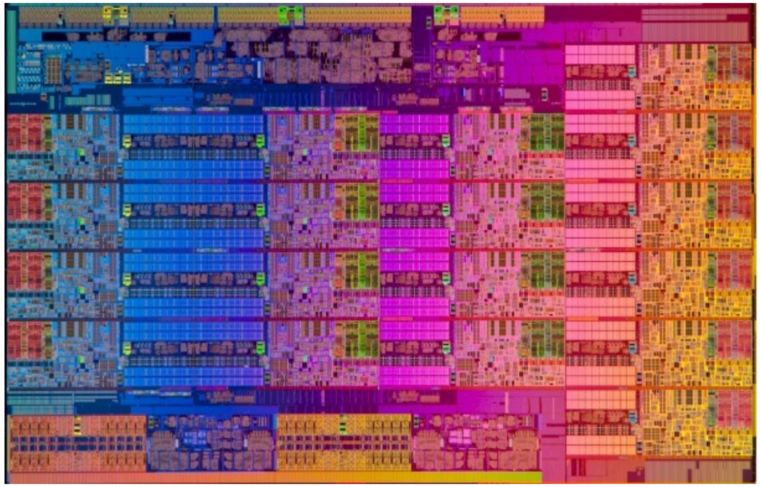
Trend: multiple cores, parallel execution

As transistor count grows, CPUs have gained more cores and more features.

A CPU is a collection of 2-20 independent, fully **general cores** connected in a local network, with local and shared memory, and I/O to the outside world.

Xeon E5 2699v3

5.5B transistors
18 cores
45 MB L3 cache



Meanwhile...

- PC graphics gained hardware support, 2D then 3D
- Early 2000s: GPUs with on-card programmable 3D transformation and lighting calculations (“shaders”)

A shader is a *small* program that:

  - Runs on many pixels or many vertices at once - (“SIMD”)
  - Is compute bound – only math with no I/O or conditionals
  - Is stream oriented – can process a stream of data points with little or no additional state

OIST

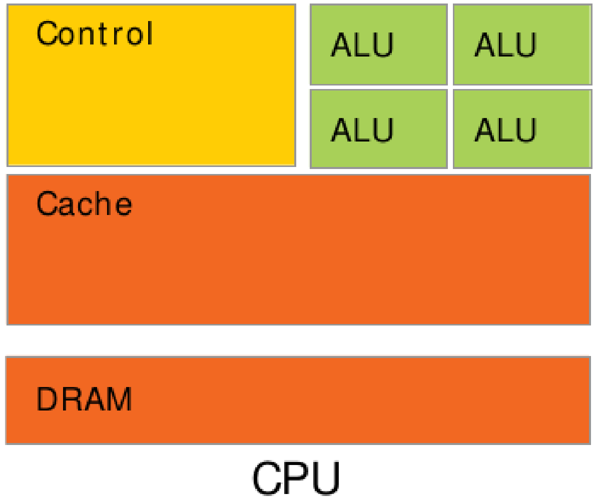
Meanwhile...

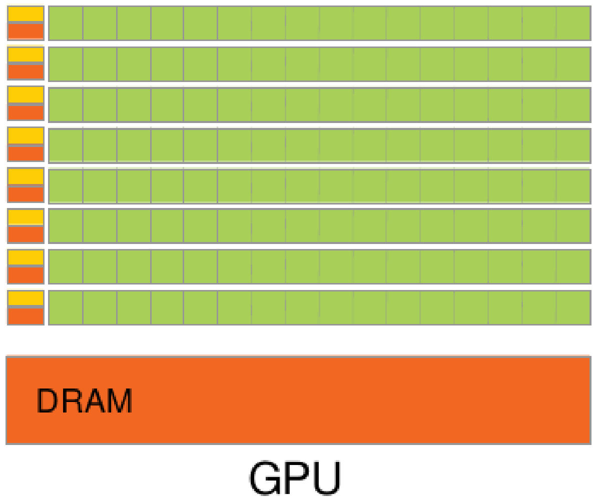
- Researchers realized that you can run any function as a shader
  First only a curiosity, but it became obvious that GPUs were potentially powerful general code accelerators.

- 2007: NVIDIA releases **CUDA**, Apple releases **OpenCL** (later standardized under Chronos).

  Both are direct means of harnessing the GPU compute power without hacks and workarounds.

OIST





CPU programs are **complex**:
- OS, simulators, games...

Cores are **independent**:
- Each core is a full computer

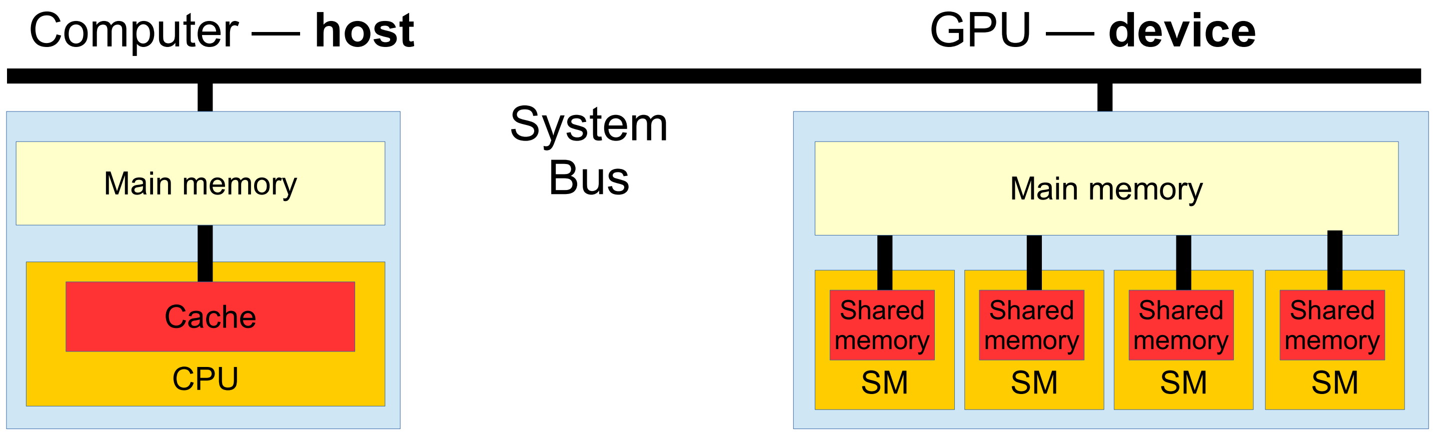
General computing tasks:
- Math, I/O, interactive applications.

GPU programs are **simple**:
- Small size, simple control flow

Cores are very **parallel**:
- Many execution threads share one control unit

Very **specialized** for math:
- floating point function evaluators.

OIST

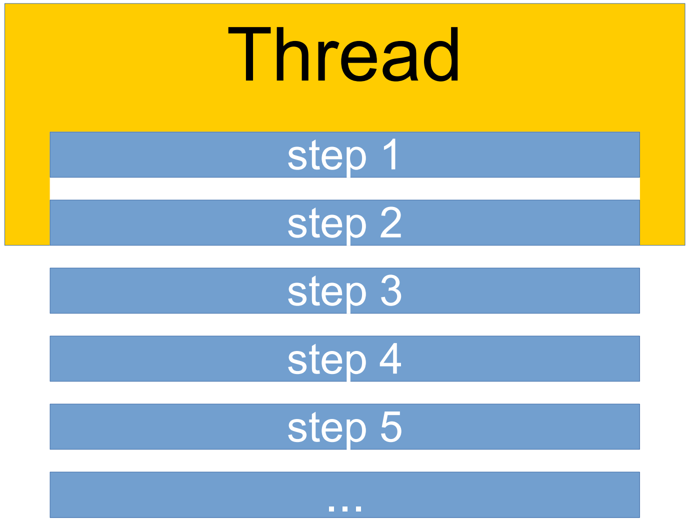


- The CPU ("*host*") and the GPU ("*device*") are separate
  $\longrightarrow$ you need to copy data to and from the GPU
- The GPU has "streaming multiprocessors" = processors that each can run hundreds of threads.
  - Each SM has 8-64 CUDA cores; with more threads they take turns on the SM

GPU concepts

A **thread** runs a single computation.

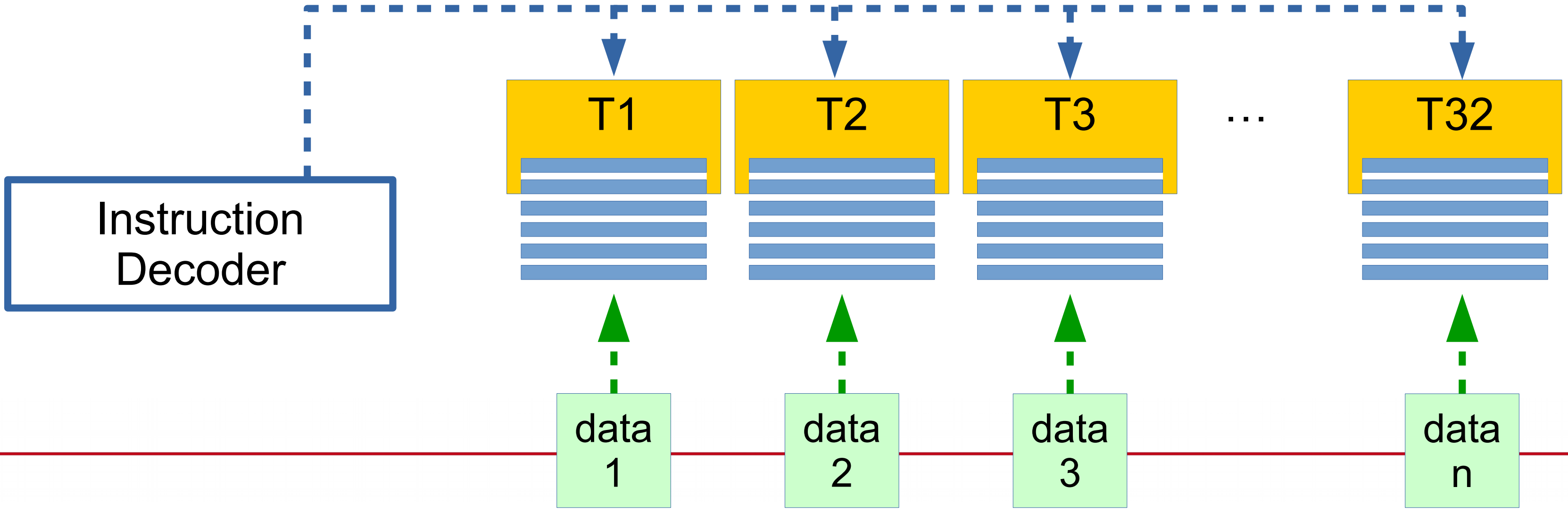
- Like CPU threads, they share memory and code with other threads.
- Much simpler, slower than CPU cores.
- Limited thread-local memory, registers.



A **warp** is a collection of 32 threads

- All threads in a warp run the *same code* at the *same time* → one thread takes less space and energy than a CPU core

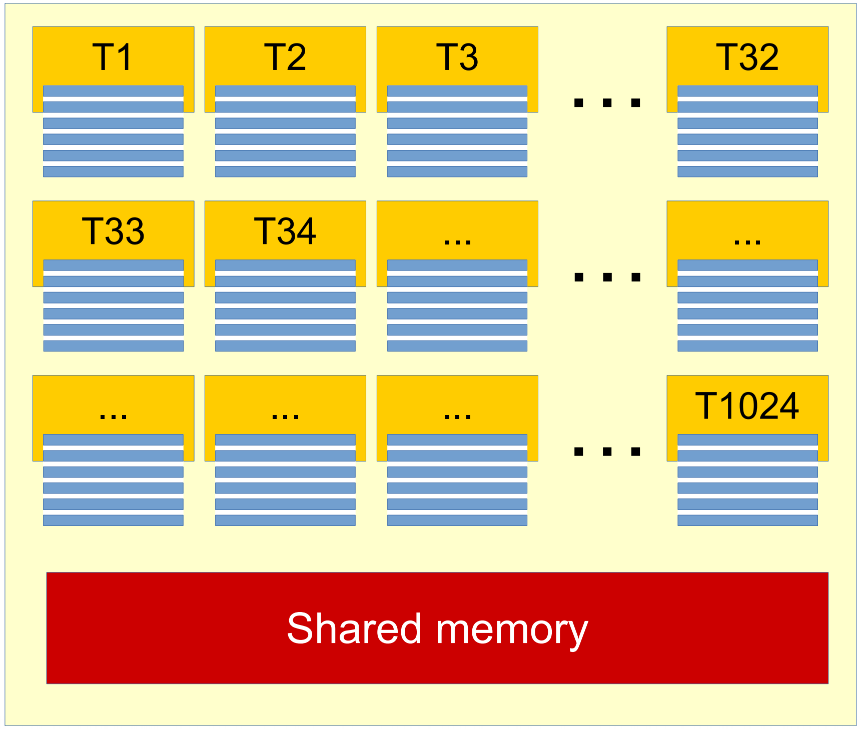
- Threads have their own registers and variables.
- Same instructions, but act on different data (“SIMD”)



Blocks

A **block** is a collection of up to 1024 threads

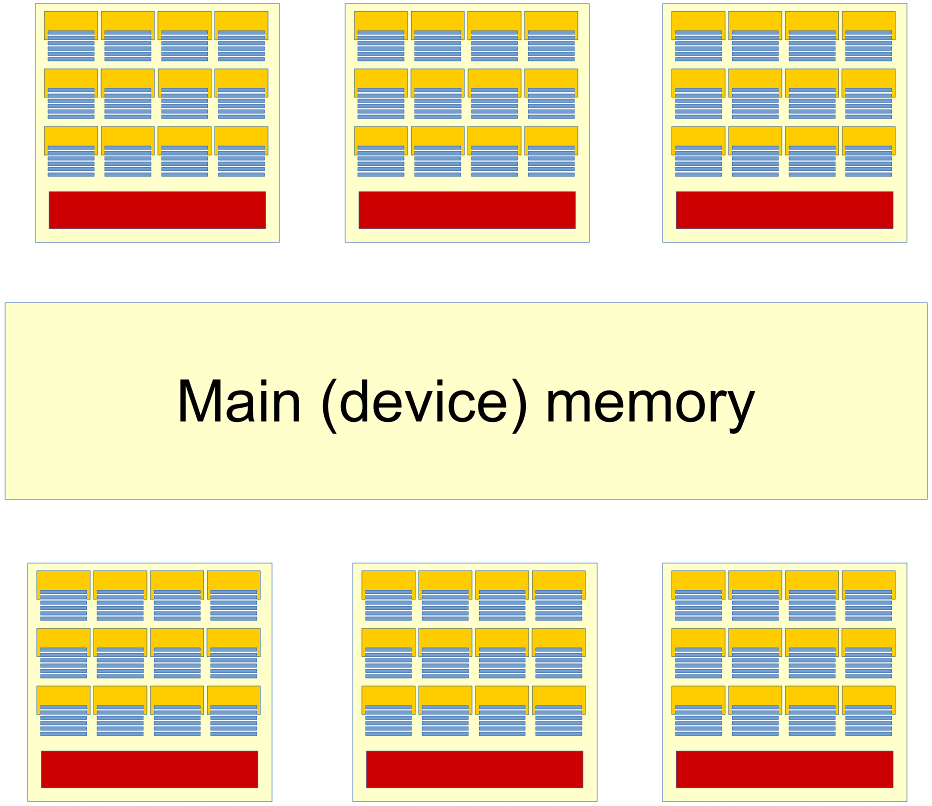
- All threads run the same code
- Has shared fast memory (48KB)
- You can organize threads in 1-D, 2-D or 3-D, but that's only programmer convenience



Grid

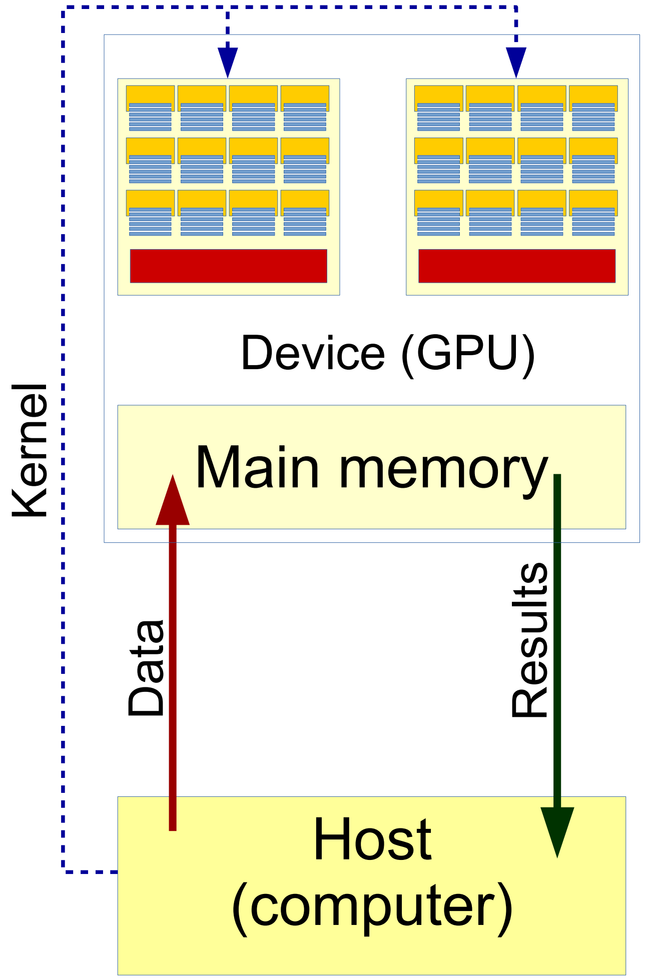
A **grid** is a set of blocks

- Blocks are **independent**
  - Can not access data in shared memory of other blocks
- Blocks run in **any order.**
- Threads in all blocks have main memory in common
- You can organize blocks in different dimensions, the same as threads in a block.



Workflow

1) Write a C function (a “*kernel*”) that will run on all **threads**
2) Copy your data from **host** to **device**
3) Run the **kernel** on the **device** with your data, using some number of **blocks** and **threads**
4) Copy results from **device** back to **host**
5) Repeat from 3); or finish up
6) Done!



Log In on Tombo!

- First, let’s all log in on Tombo:

```
$ ssh <your-ID>@tombo.oist.jp
```

- Copy the code from the common area:

```
$ cp -r /work/training/GPU .
```

- Go to the new directory:

```
$ cd GPU/code
```

GPU Resources at OIST

```
$ ssh <your-ID>@tombo.oist.jp
$ cp -r /work/share/training/GPU .
$ cd GPU/code
```

Tombo:

"gpu"	- 1 node 2* K40	Training

Sango:

"gpu"	- 2 node 4* K80	image analysis pipelines
"powernv"	- 4 node 4* P100	general, deep learning

Saion:

"gpu"	- 9 node 4* P100	general, deep learning
	- 8 node 4* V100	
"powernv"	- 4 node 4* V100	general, deep learning

OIST

```
$ ssh <your-ID>@tombo.oist.jp
$ cp -r /work/share/training/GPU .
$ cd GPU/code
```

- You need to ask for the GPU partition, *and* reserve the GPU resource:

```
$ srun --partition=gpu --gres=gpu <program>
```

- Use multiple cards with `--gres=gpu:N`
- For compilation, load the cuda module:

```
$ module load cuda/8.0.27
```

OIST

Program #1: GPU vector addition

We will adapt a simple program to run on the GPU:

```
void vec_add(float *a, float *b, float *c, int n) {

    int index;
    for (index=0; index<n; index++) {
        c[index] = a[index] + b[index];
    }
}
```

Vector addition is the “hello world” of parallel programming
The source is `GPU/code/vec_add.c`

The CPU-based version

- Allocate three arrays:

  a, b inputs
  c output

- Initialize all elems

  a,b = 5.0
  c = 0.0

- Call `vec_add()`

```
void vec_add(float *a, float *b, float *c, int n) {

    int index;
    for (index=0; index<n; index++) {
        c[index] = a[index] + b[index];
    }
}

int main(int argc, char **argv) {
    int i;
    float *c, *b, *a;

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

    for (i = 0; i<N; i++ ) {
        a[i] = b[i] = 5.0;
        c[i] = 0.0;
    }

    vec_add(a,b,c,N);
}
```

OIST

We have two extra things in our code. Timing:

```
double time_diff_nano(struct timespec *toc, struct timespec *tic) {
    return (1e9*(toc->tv_sec-tic->tv_sec)+
        (toc->tv_nsec-tic->tv_nsec));
}
...
clock_gettime(CLOCK_REALTIME, &tic);
...
clock_gettime(CLOCK_REALTIME, &toc);
time_diff_nano(&toc, &tic)/1000.0;
```

and sanity check:

```
float sum = 0.0;
for (index = 0; index<N; index++) {
    sum += c[index];
}

if (fabs((sum-CORRECT)/sum)>EPSILON) {
    printf("   correct sum: %.2f - fail\n", CORRECT);
}
```

OIST

Run the serial code

- Compile our example program:

```
$ gcc -O3 vec_add.c -o addcpu
```

On some systems you may need '`-lrt`' for the timer.

- Run it on Tombo:

```
$ srun -t 1:00 --mem=50m ./addcpu
calculated sum: 10485760.00
Time: 1441.77 μs
```

- Time: ~1425-1550μs

Allocate memory on the GPU

- Make a copy (or just use code/vec_add.cu) and open it:

```
$ cp vec_add.c vec_add.cu
```

- Allocate memory on the GPU (and free it at the end):

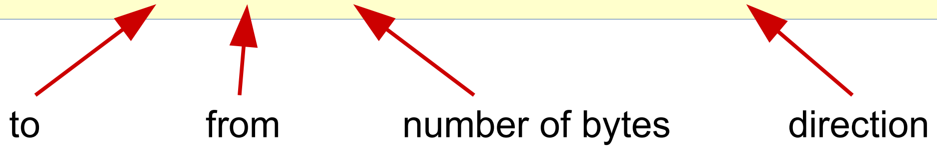
```
float *dc, *db, *da;

cudaMalloc( &da, size );
cudaMalloc( &db, size );
cudaMalloc( &dc, size );

cudaFree( da );
cudaFree( db );
cudaFree( dc );
```

  - All cuda functions start with “cuda”
  - You usually don’t need to add `#include` statements — the cuda compiler adds then for you.

Move our data to the GPU

- Copy our data over:

```
cudaMemcpy( da, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( db, b, size, cudaMemcpyHostToDevice );
cudaMemcpy( c, dc, size, cudaMemcpyDeviceToHost );
```



`cudaMemcpyHostToDevice` = copy to GPU from computer
`cudaMemcpyDeviceToHost` = copy to computer from GPU

also `cudaMemcpyHostToHost`,`cudaMemcpyDeviceToDevice`

Many function variations available: **`cudaMemcpyAsync`**`()`, **`cudaMemcpy2DTo[From]Array`**`()`, **`cudaMemcpyToSymbol`**`()` …

Create the kernel

Our original vector addition function:

```
void vec_add(double *a,  double *b, double *c, int n) {
    int index;

    for (index = 0; index<n; index++) {
        c[index] = a[index] + b[index];
    }
}
```

Add "`__global__`" specifier:

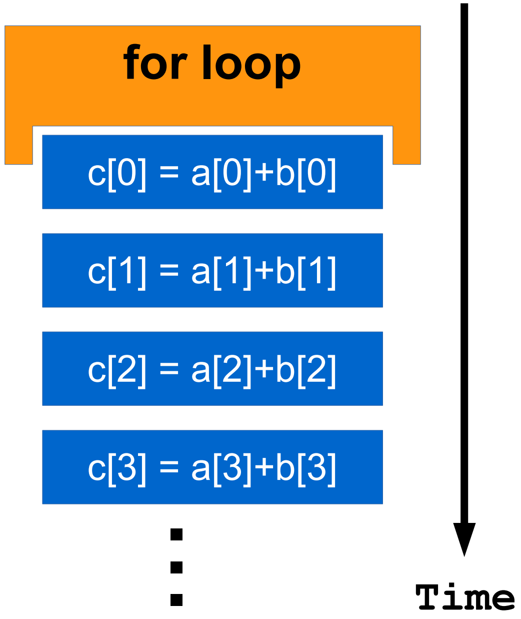
```
__global__ void vec_add(double *a,  double *b, double *c, int n) {
```

`__global__` = can be called from **host** and runs on **device**.

Also available: `__device__` and `__host__`.

Create the kernel — parallelize

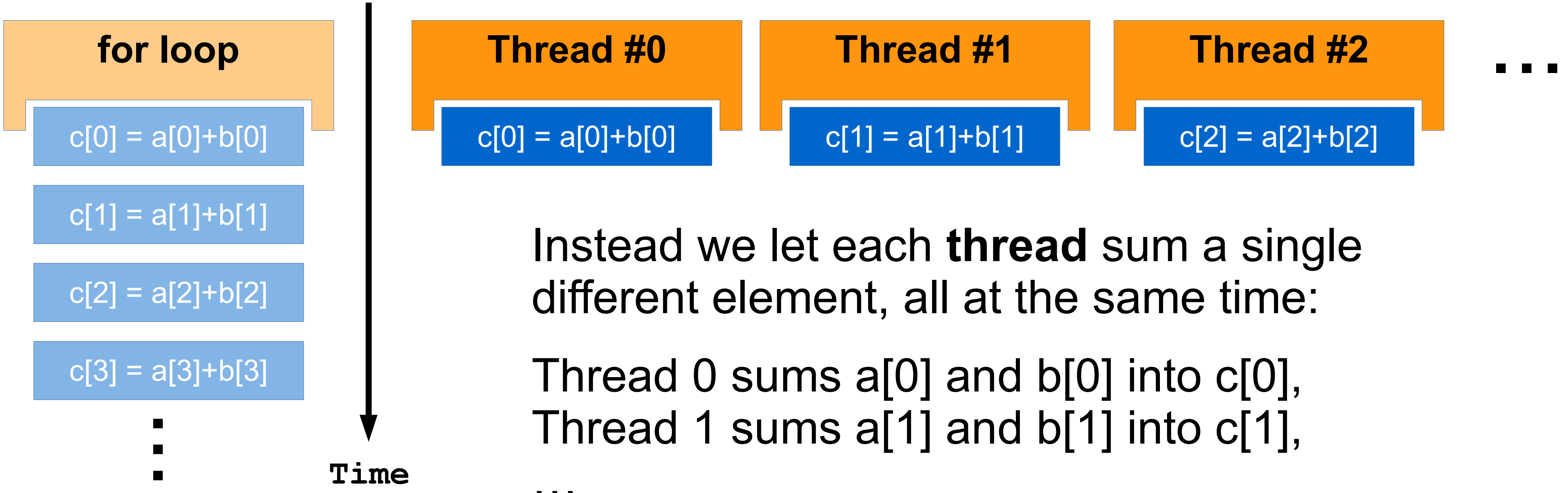
```
for (index = 0; index<n; index++) {
    c[index] = a[index] + b[index];
}
```



Our vector addition function steps through the loop over time, and sums a different set of elements at each time step.

Create the kernel — parallelize

```
for (index = 0; index<n; index++) {
    c[index] = a[index] + b[index];
}
```

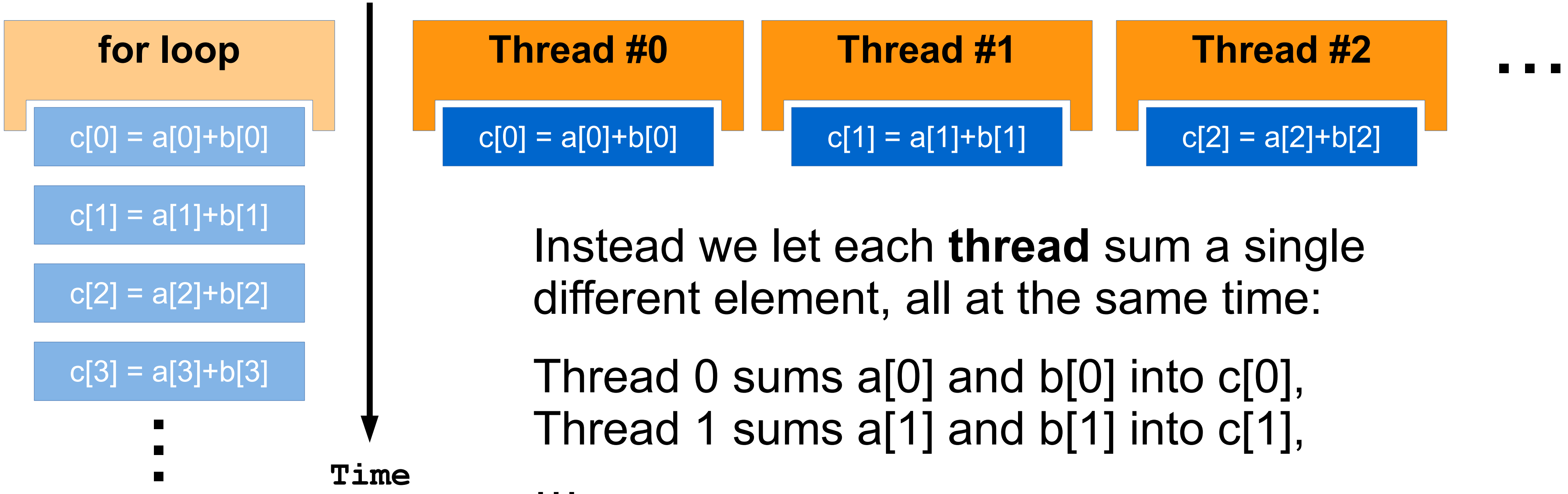


Instead we let each **thread** sum a single different element, all at the same time:

Thread 0 sums a[0] and b[0] into c[0],
Thread 1 sums a[1] and b[1] into c[1],
...

Create the kernel — parallelize

```
index = ???
c[index] = a[index] + b[index];
```

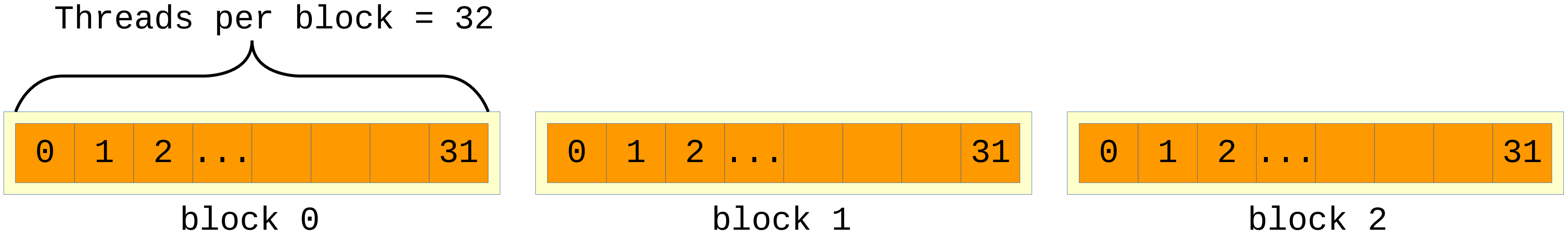


Instead we let each **thread** sum a single different element, all at the same time:

Thread 0 sums a[0] and b[0] into c[0],
Thread 1 sums a[1] and b[1] into c[1],
...

Create the kernel — parallelize

```
index = ???
c[index] = a[index] + b[index];
```



In kernel functions, CUDA automagically defines variables with the block and thread IDs.

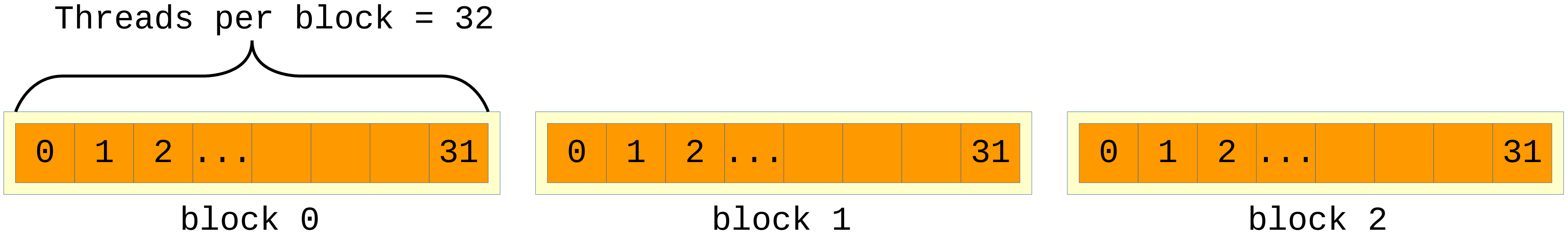
`blockDim.x` = number of threads/block

`blockIdx.x` = current block (0, ...)

`threadIdx.x` = thread in current block

Create the kernel — parallelize

```
int index = blockIdx.x*blockDim.x + threadIdx.x;
c[index] = a[index] + b[index];
```



In kernel functions, CUDA automagically defines variables with the block and thread IDs.

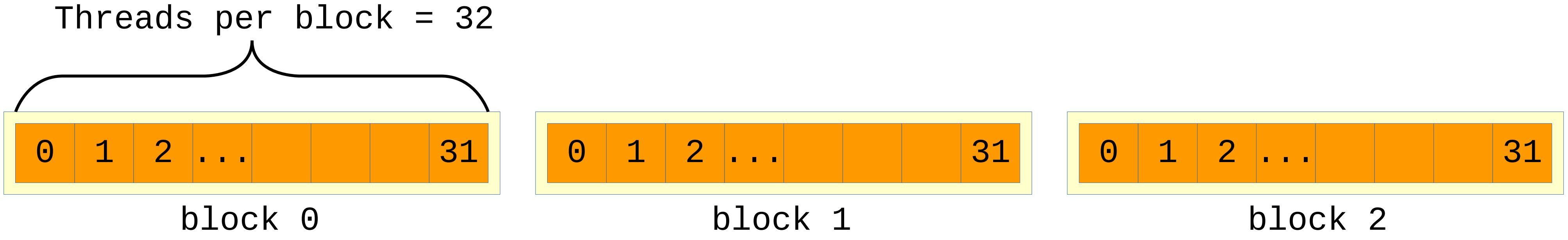
`blockDim.x` = number of threads/block

`blockIdx.x` = current block (0, ...)

`threadIdx.x` = thread in current block

Create the kernel — parallelize

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



In kernel functions, CUDA automagically defines variables with the block and thread IDs.

`blockDim.x` = number of threads/block

`blockIdx.x` = current block (0, ...)

`threadIdx.x` = thread in current block

Call the kernel

```
#define THREADS 512

vec_add<<< (N+THREADS-1)/THREADS, THREADS >>>(da, db, dc,N);
```

The notation is:

```
func<<< blocks, threads per block >>>();
```

“<<< , >>>” is a CUDA extension for calls to a CUDA kernel.

- allocates the blocks and threads per block that we specify;
- copy the code to the GPU card;
- set `blockDim`, `blockIdx` and `threadIdx` for the kernel function

Build and run!

- To build our CUDA program we need the compiler.
  Load the cuda module:

```
$ module load cuda/8.0.27
```

- Compile using nvcc:

```
$ nvcc -o addcuda vec_add.cu
```

- Run:

```
$ srun -p gpu --mem=1G --gres=gpu -t 1:00 ./addcuda
Time: 184.37 µs
```

- Runtime (this time): 184 µs
- Our CPU version took ~1450 µs — a **7×** speedup!

But...

We’re being a little unfair to the CPU. Let’s see:

```
    cudaMemcpy (da, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy (db, b, size, cudaMemcpyHostToDevice );

    clock_gettime(CLOCK_REALTIME, &tic);

    vec_add<<< (N+THREADS-1)/THREADS, THREADS >>>(da,db,dc,N);
    cudaDeviceSynchronize();
    clock_gettime(CLOCK_REALTIME, &toc);

    cudaMemcpy (c, dc, size, cudaMemcpyDeviceToHost );
```

- We measure the `vec_add()` time, but not the memory copying:

  Only `vec_add()`: ~1450 µs (CPU) ~180 µs (GPU)

OIST

But...

Include the `cudaMemcpy()` calls in our time measurement:

```
    clock_gettime(CLOCK_REALTIME, &tic);
    cudaMemcpy (da, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy (db, b, size, cudaMemcpyHostToDevice );

    vec_add<<< (N+THREADS-1)/THREADS, THREADS >>>(da,db,dc,N);

    cudaMemcpy (c, dc, size, cudaMemcpyDeviceToHost );
    clock_gettime(CLOCK_REALTIME, &toc);
```

- Measure the `vec_add()` time *and* memory transfers:

`vec_add()+copy`: ~1450 µs (CPU) ~**15000** µs (GPU)

Only `vec_add()`: ~1450 µs (CPU) ~**180** µs (GPU)

Lesson:

`vec_add()+copy`: ~1450 μs (CPU) ~**15000** μs (GPU)

Only `vec_add()`: ~1450 μs (CPU) ~**180** μs (GPU)

- GPU jobs *should be* compute-bound.
  - Lots of math, few memory transfers
- Memory transfers are **expensive**
  - Do as much as possible on the GPU without transferring data
  - You can run multiple kernels after one another without moving data.

Example #2: dot product

Dot product: p = a1*b1 + a2*b2 + a3*b3 + ...

Two operations:

- Elementwise product
  - We already (almost) did in the last example
- *Reduction*
  - combine all elements into a single value with some function
  - Very common operation, not always trivial.

Serial version

The serial version (dotprod.c):

- pairwise multiplication:

  `tmp = a*b`

- Pairwise summation of tmp:

  `dotp = sum(tmp)`

And in main():

- We just call `vec_dot()`.

```
void vec_dot(float *a,  float *b, float *dotp) {
     int index, s, i;
     float tmp[N];

     for (index = 0; index<N; index++) {
         tmp[index] = a[index] * b[index];
     }

     for(s = (N/2); s>0; s/=2) {
         for(i = 0; i<s; i++) {
             tmp[i] += tmp[i+s];
         }
     }
     *dotp = tmp[0];
}
```

```
     float dotp = 0.0;
     vec_dot(a, b, &dotp);
```

Pairwise Summation

Summing large data sets naively will cause a form of *catastrophic cancellation* — one term will become orders of magnitude larger than the other, and you lose significant digits.

```
    for(s = (N/2); s>0; s/=2) {
        for(i = 0; i<s; i++) {
            tmp[i] += tmp[i+s];
        }
    }
```

We sum elements in pairs:

`tmp[0] += tmp[N/2]; tmp[1] += tmp[N/2+1]`…

Then sum the pairs in pairs until we have a single element.
All terms now have the same order of magnitude.

Pairwise Summation

Summing large data sets naively will cause *catastrophic cancellation*

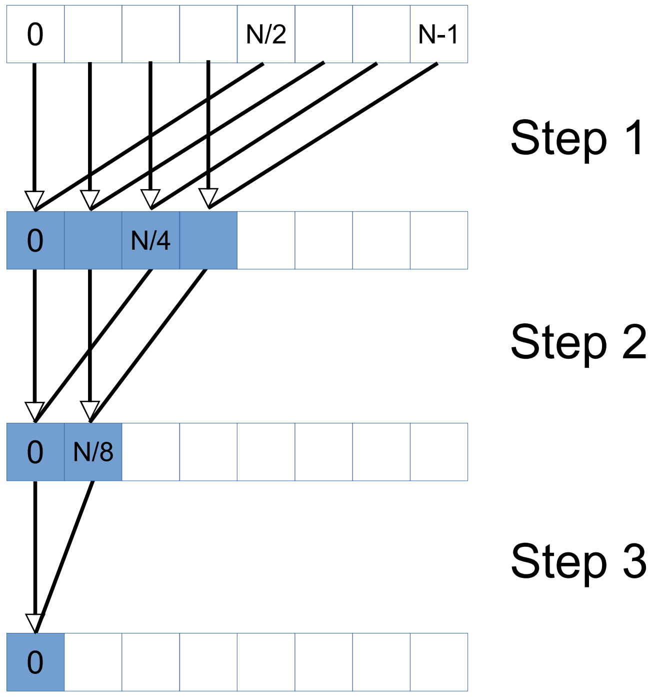
We avoid it by pairwise summation:

We sum elements in pairs:

```
tmp[0] = tmp[0] + tmp[N/2];
tmp[1] = tmp[1] + tmp[N/2+1]…
```

We recursively sum each pair in the same way, until we have a single element.

All summation terms will have about the same order of magnitude.



Run the serial code

Compile our example program:

```
$ gcc -O3 dotprod.c -o dotpcpu
```

Run it on Tombo:

```
$ srun -t 1:00 --mem=50m ./dotpcpu
calculated dot product: 26214400.00
time: 4899.19 µs
```

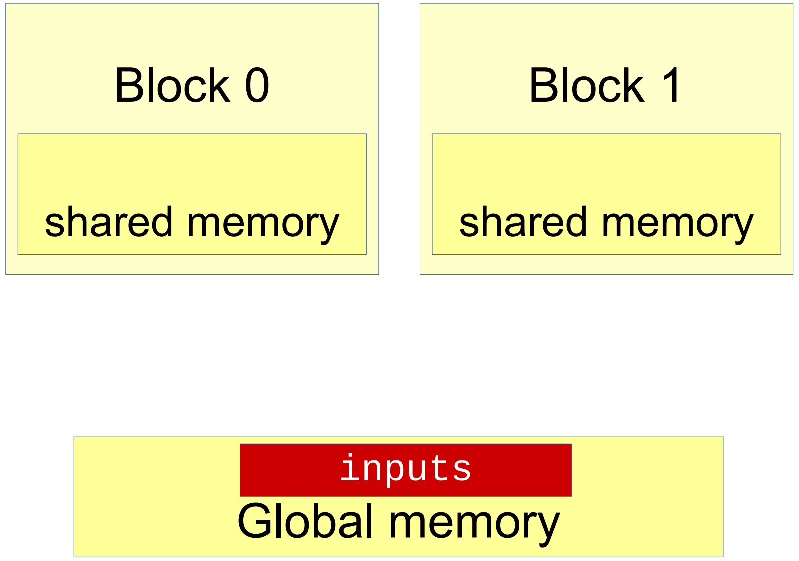
We get times in the 4800-5200µs range

Speed improvement: use shared memory

Blocks have private “shared” memory

- 48KB (can be changed)
- faster access than global memory
- memory accesses in different blocks are independent

  → no memory contention



Process:

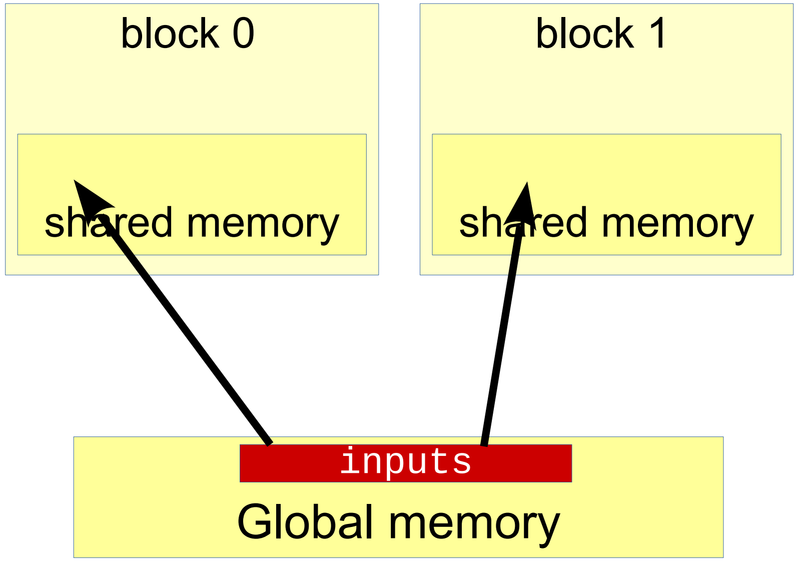
- Read input from global memory
- Calculate, using shared memory for intermediate values
- each block does a partial reduction - one partial value per block
- finally add its partial value to the final result in global memory

Speed improvement: use shared memory

Blocks have private "shared" memory

- 48KB (can be changed)
- faster access than global memory
- memory accesses in different blocks are independent

  → no memory contention



Process:

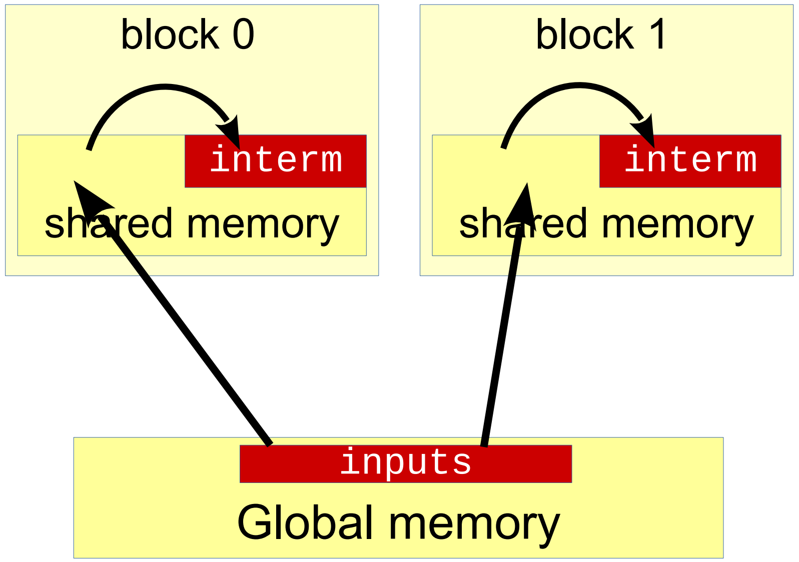
- Read input from global memory
- Calculate, using shared memory for intermediate values
- each block does a partial reduction - one partial value per block
- finally add its partial value to the final result in global memory

Speed improvement: use shared memory

Blocks have private “shared” memory

- 48KB (can be changed)
- faster access than global memory
- memory accesses in different blocks are independent

  → no memory contention



Process:

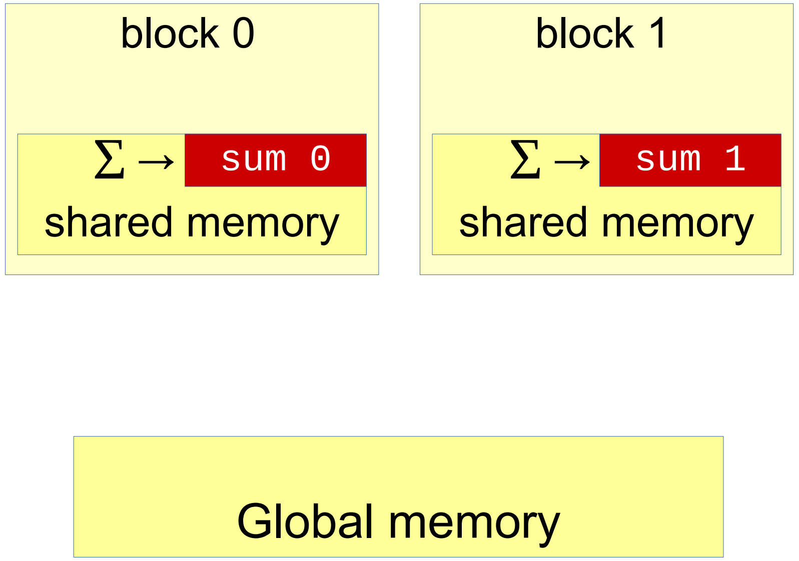
- Read input from global memory
- Calculate, using shared memory for intermediate values
- each block does a partial reduction - one partial value per block
- finally add its partial value to the final result in global memory

Speed improvement: use shared memory

Blocks have private “shared” memory

- 48KB (can be changed)
- faster access than global memory
- memory accesses in different blocks are independent

  → no memory contention



Process:

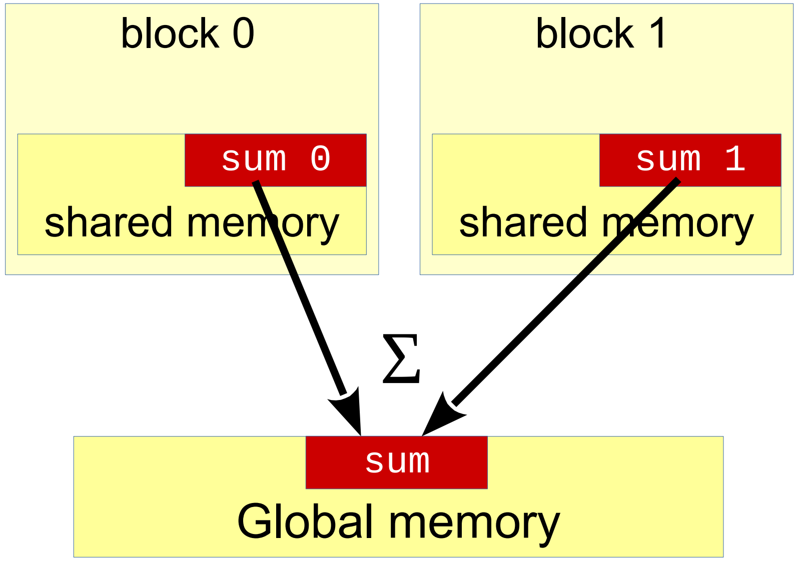
- Read input from global memory
- Calculate, using shared memory for intermediate values
- each block does a partial reduction - one partial value per block
- finally add its partial value to the final result in global memory

Speed improvement: use shared memory

Blocks have private “shared” memory

- 48KB (can be changed)
- faster access than global memory
- memory accesses in different blocks are independent

  → no memory contention



Process:

- Read input from global memory
- Calculate, using shared memory for intermediate values
- each block does a partial reduction - one partial value per block
- finally add its partial value to the final result in global memory

The GPU version (dotprod.cu), elementwise multiplication:

```
__global__ void vec_dotp(float *a,  float *b, float *dotp) {
    int s;
    int tid = threadIdx.x;
    int index = blockIdx.x*blockDim.x + threadIdx.x;

    // allocate block-local memory
    __shared__ float tmp[THREADS];

    tmp[tid] = a[index] * b[index];
    __syncthreads();
```

- Get local memory with “`__shared__`”.
  - Much faster than global memory, but limited size
- `__syncthreads()` synchronizes all threads.
  - Threads in a warp are synchronized, but threads in *different* warps are not.

The GPU version (dotprod.cu), reduction:

```
    tmp[tid] = a[index] * b[index];
    __syncthreads();

    for(s = (THREADS/2); s>0; s/=2) {
        if (tid < s) {
            tmp[tid] += tmp[tid+s];
        }
        __syncthreads();
    }
```

- Same pairwise summation as in the serial program
  - But inner loop is parallel
  - We must synchronize after each iteration so all threads really are finished.

The GPU version (dotprod.cu), reduction:

```
    tmp[tid] = a[index] * b[index];
    __syncthreads();

    for(s = (THREADS/2); s>0; s/=2) {
        if (tid < s) {
            tmp[tid] += tmp[tid+s];
        }
        __syncthreads();
    }
```

```
    for(s = (N/2); s>0; s/=2) {
        for(i = 0; i<s; i++) {
            tmp[i] += tmp[i+s];
        }
    }
```

serial
version

- Same pairwise summation as in the serial program
  - But inner loop is parallel
  - We must synchronize after each iteration so all threads really are finished.

The GPU version (dotprod.cu), reduction:

```
    tmp[tid] = a[index] * b[index];
    __syncthreads();

    for(s = (THREADS/2); s>0; s/=2) {
        if (tid < s) {
            tmp[tid] += tmp[tid+s];
        }
        __syncthreads();
    }
```

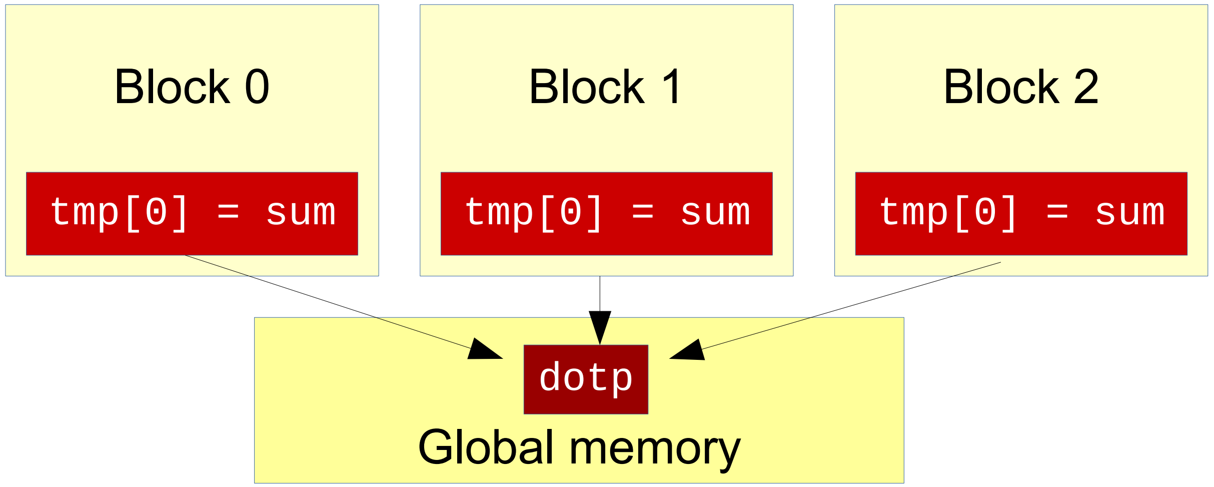
```
    for(s = (N/2); s>0; s/=2) {
        for(i = 0; i<s; i++) {
            tmp[i] += tmp[i+s];
        }
    }
```

serial version

- Same pairwise summation as in the serial program
  - But inner loop is parallel
  - We must synchronize after each iteration so all threads really are finished.

NOTE: this example works only when THREADS is a power of 2 and N is a multiple of THREADS

We have the partial results in each block:



- Add the partial sums together into the final result.
- But blocks are independent, and could access dotp at the same time (“*race condition*”)

  → need to use an “*atomic*” operation: `atomicAdd()`

The complete reduction:

```
    for(s = (THREADS/2); s>0; s/=2) {
        if (tid < s) {
            tmp[tid] += tmp[tid+s];
        }
        __syncthreads();
    }

    if (tid == 0) {
        atomicAdd(dotp, tmp[0]);
    }
```

- Pairwise sum all elements in the block
- Finally thread #0 atomically adds the result into the return parameter

- Compile:

```
$ nvcc -o dotpcuda dotprod.cu
```

- Run it on Tombo:

```
$ srun -p gpu --mem=50m --gres=gpu -t 1:00 ./dotpcuda
calculated dot product: 26214400.000000
time: 5258.94 µs
```

Result: ~4800-5200 µs (CPU) **~5100-5500 µs** (GPU)

OIST

Let’s try adding more work: run the kernel twice

CPU call:

```
    // calculate dot product of a and b, return in dotp
    vec_dot(a, b, &dotp);
    vec_dot(a, b, &dotp);
```

CUDA call:

```
    // calculate dot product of a and b, return in dotp
    vec_dot<<< (N+THREADS-1)/THREADS, THREADS>>>(da, db, ddotp);
    vec_dot<<< (N+THREADS-1)/THREADS, THREADS>>>(da, db, ddotp);
```

Result: ~7500 µs (CPU) **~5300 µs** (GPU)

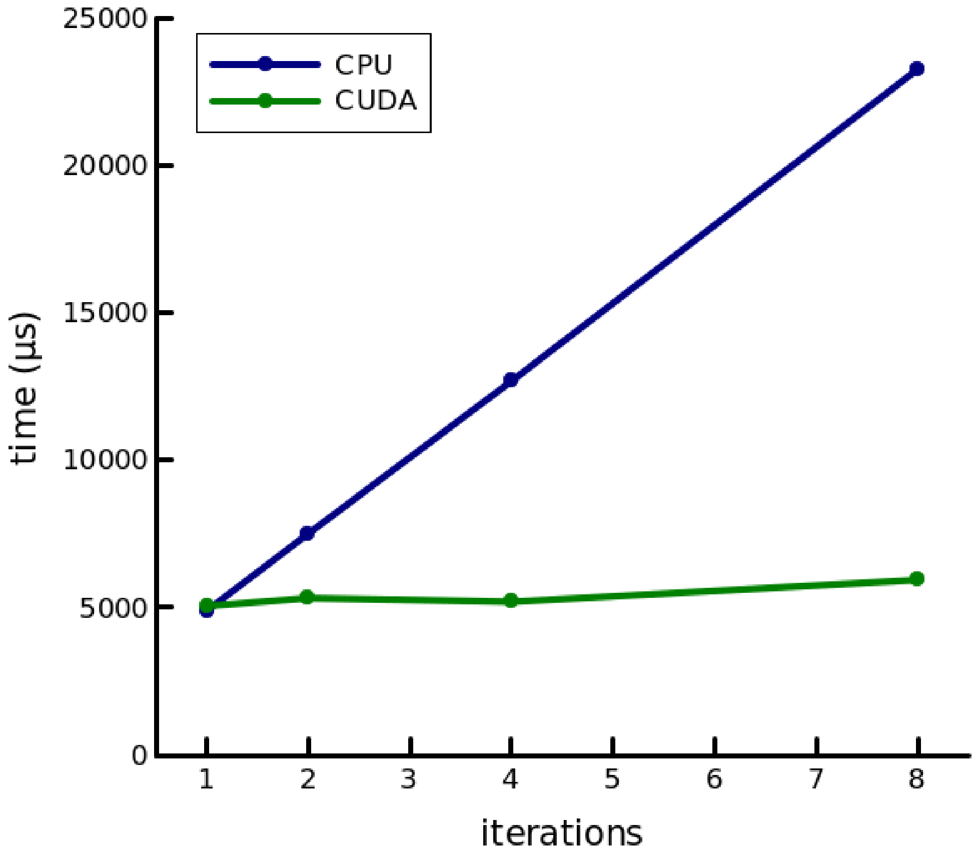
CPU and CUDA

Let's test for different number of iterations of our dot product:

- CUDA has large, fixed transfer cost

  → For small amounts of work, a single CPU core is faster

  - Even for 8 dot products, the time is dominated by the data transfer
- Tombo nodes have 16 cores. If we use them, the CPU will be faster.
- Practical GPU speed improvement is usually less than 3-4 times CPU.



Summary

- Use `__shared__` to allocate shared memory
  - Fast, but limited size (~48Kb)
  - Does not persist between different kernels
- `__syncthreads()` synchronizes all threads in a block
  - threads in a warp are synchronous, but different warps are not.
- Use atomic operations when multiple threads have to change the same data
  - Blocks are independent, so atomics are necessary
  - Atomic operations in global memory faster than in shared.

Final Points

- GPU *computation* is fast. GPU *data transfer* is slow.
  - To reduce transfer amount, filter data on the CPU.
  - Do as much calculation as possible on the data in the GPU.
  - Avoid storing intermediate values on the host.
  - Running kernels is cheap
- Memory organisation matters a lot
  - keep data in block shared memory or thread local memory
  - Access data sequentially
  - Global memory is persistent across kernels

Odds and Ends

You can write functions that run on both host and device:

```
__HOST__ __DEVICE__ float myfunc() {

#ifdef __CUDA_ARCH__
    // CUDA code, probably called from a kernel

#else
    // Host code, running without CUDA

#endif
}
```

__CUDA_ARCH__ defines the compute capability level, but is only defined in code that runs on the GPU.

Odds and Ends

- NVIDIA separates it's cards by "compute capability". Each newer capability version is a superset of previous ones.

  Set the desired capability level with "--arch" parameter:

```
$ nvcc --arch=sm_35 -o dotpcuda dotprod.cu
```

- For example: atomicAdd() for integers appeared in 1.1 (sm_11); floating point version in 2.0 (sm_20) and double precision only in version 6.0 (sm_60).

- Wikipedia has a great page: https://en.wikipedia.org/wiki/CUDA

OIST