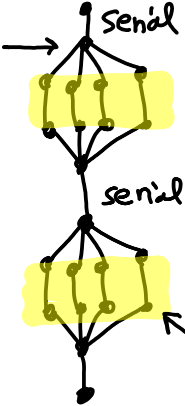
Data Migration (cont.)

Host Programming. Serial
CPU
activity for GPU
Copy data
execute gpu

Parallel
Kernel programming.



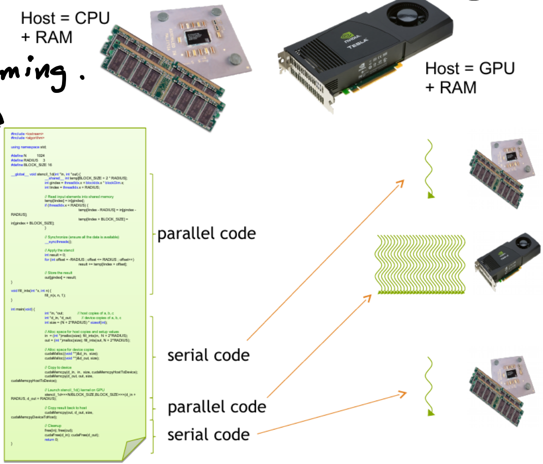
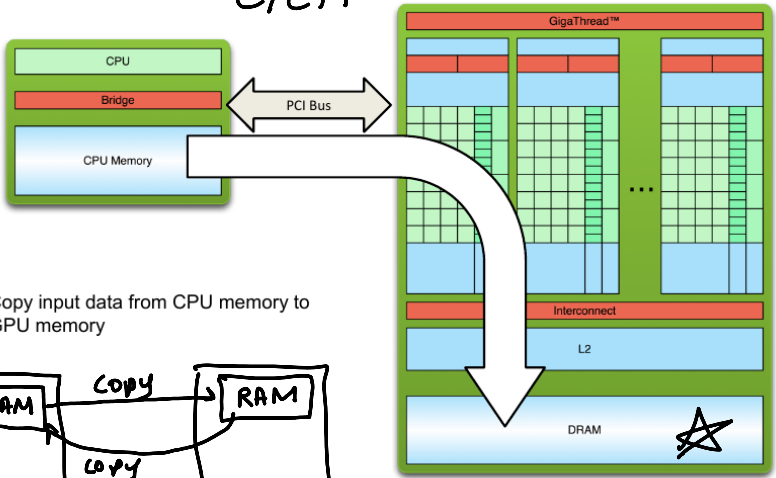


Figure 4: Porting portions of your code to GPU

Data Migration (cont.)

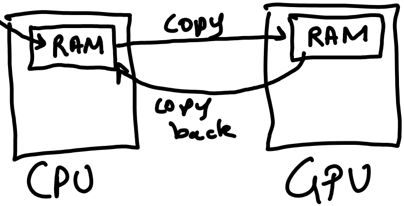
Serial Programing.

C/C++



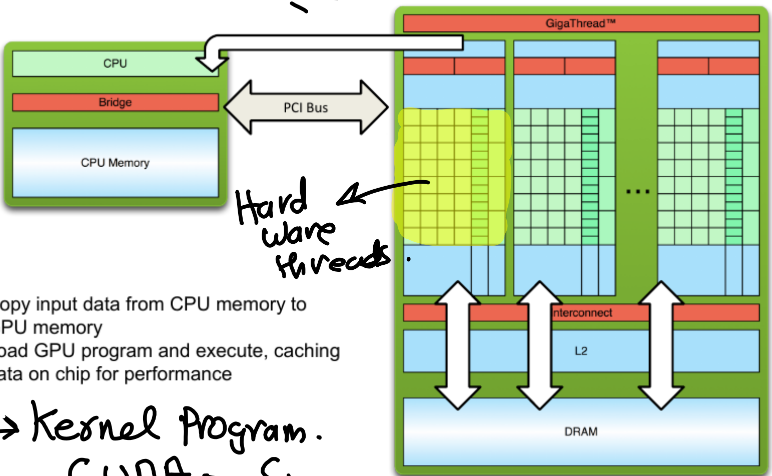
Disk

1. Copy input data from CPU memory to GPU memory



Data Migration (cont.)

1 : M

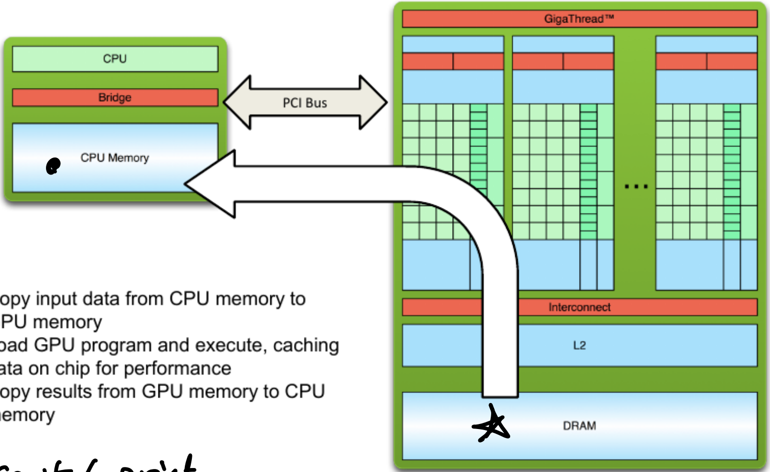


Hard ware threads.

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

→ Kernel Program.
CUDA — C

Data Migration (cont.)



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

cout / print.

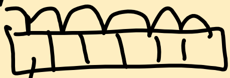
Squaring an Array

Kernel.

C Code

```
void squareCPU(float *a, int size) {
    for (int x = 0; x < size; x++)
        a[x] *= a[x];
}
```

for



a[0] *= a[0]

OpenCL Kernel Code

```
__kernel void squareCPU(__global float *a) {
    int x = get_global_id(0);
    a[x] *= a[x];
}
```

flag.

~~for~~ x

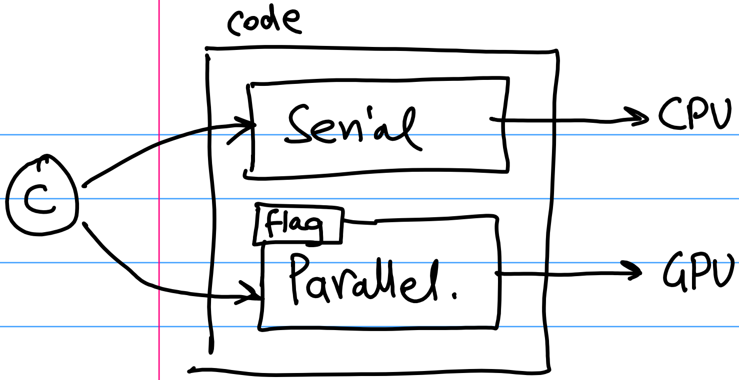
CUDA Kernel Code

```
__global void hello(float *a) {
    int x = threadIdx.x;
    a[x] *= a[x];
}
```

x = thread id.

Object.member;

a[0] *= a[0]



Compiler. (Help needed)

Building up towards Hello World

Code

```
int main()
{
   printf("hello world!\n");
   return 0;
}
```

CPU

GPU

Compilation

- nvcc hello_world.cu
- ./a.out

Building up towards Hello World: Inserting GPU Code

Serial

function call.

Parallel

```
__global__ void myKernel(void)
{
}

int main()
{
  myKernel<<<1,1>>>();

  printf("hello world!\n");
  return 0;
}
```

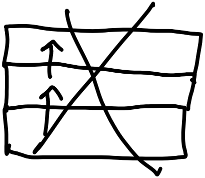
```
// indicates function runs on
// device and is called from
// host code
```

retrieve.

fn name();

```
// like function call, but
// with more information
// 1st digit number of blocks
// 2nd digit threads per block
```

80% ← <<<1 ,1 >>>



- nvcc will separate source code into host and device components
  - Device functions processed by NVIDIA compiler
  - Host functions processed by standard host compiler

Building up towards Hello World: Inserting GPU Code

~~1~~ 2 ~~3~~
copy exec copy

```
__global__ void myKernel(void)    // indicates function runs on
{                                 // device and is called from
}                                 // host code
```

empty.

```
int main()
{
  myKernel<<<1,1>>>();            // like function call, but
                                  // with more information
                                  // 1st digit number of blocks
                                  // 2nd digit threads per block
  printf("hello world!\n");
  return 0;
}
```

Serial
2
1 thread
Serial.

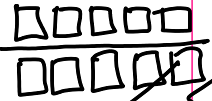
- nvcc will separate source code into host and device components
  - Device functions processed by NVIDIA compiler
  - Host functions processed by standard host compiler

H.Ware.		Developer
μProcessor	—	Block
Core	—	thread.

1 block + 1 thread.

1 μP. + 1 core.

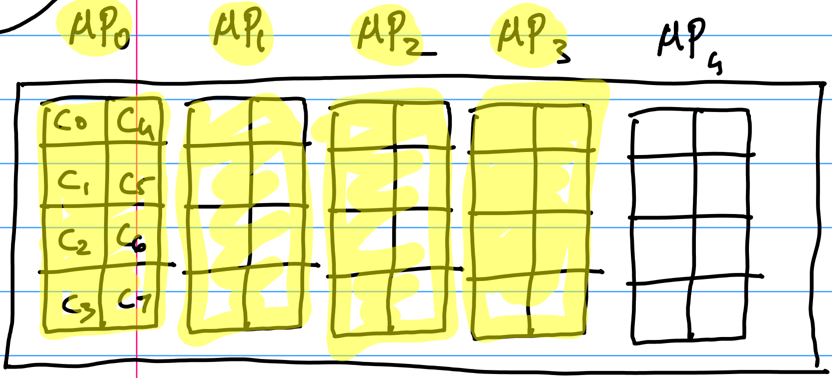
5 MP → 8 Cores. ⇒ 8x5 = 40 cores.



→ 2 time units.

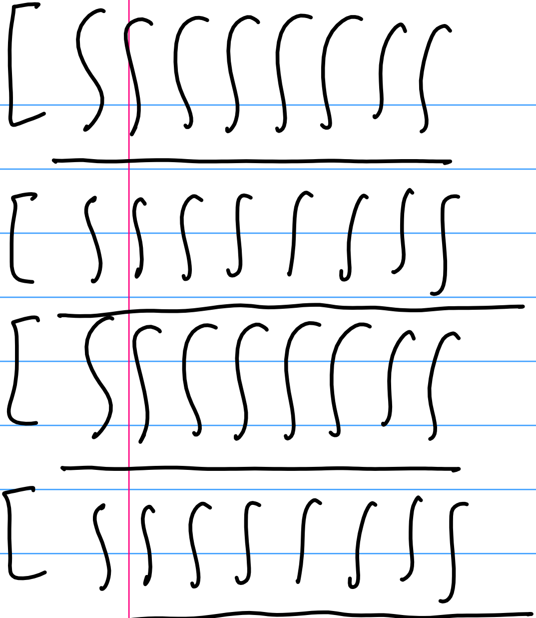
→ 10 MP.

<<< 10 , 8 >>>



parallel → executed somewhat sequentially 32

1 MP.

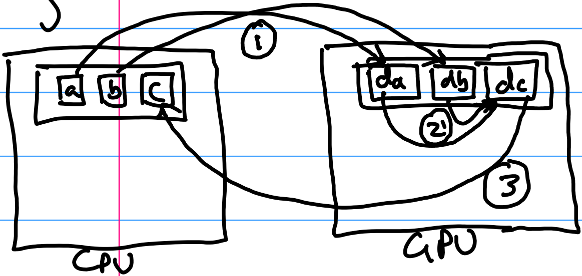


4 time units

32 threads.

```
int main( )
{
    int a = 5
    int b = 6
(1)
(2) int c = a + b;
(3)
}
```

0. Create memory location on gpu.
1. copy (a,b)
2. exec
3. copy



```
/* a, b, c are pointers to memory location on the device */
__global__ void add(int *a, int *b, int *c)
{  *c = *a + *b;
}

int main()
{  int a=2, b=7, c, *da, *db, *dc;

   cudaMalloc((void **)&da, sizeof(int)); // allocate
   cudaMalloc((void **)&db, sizeof(int)); // on device
   cudaMalloc((void **)&dc, sizeof(int));

   // Copying in blocking mode
   cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
   add<<<1,1>>>(da, db, dc);
   cudaMemcpy(&c, dc, sizeof(int), cudaMemcpyDeviceToHost);

   printf("a + b = %d\n", c);
   cudaFree(da); cudaFree(db); cudaFree(dc);
   return 0;
}
```

int *da = &a;

destination

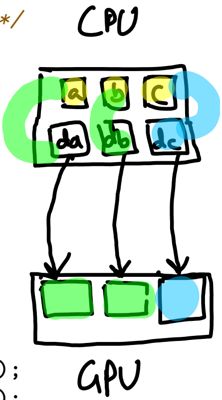
source

0

1

2

3

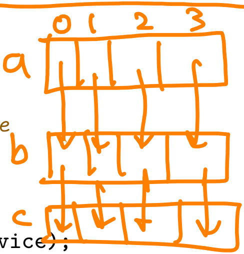


*c = *a + *b

```
/* a, b, c are pointers to memory location on the device */
__global__ void add(int *a, int *b, int *c)
{  *c = *a + *b;
}
int main()
{  int a=2, b=7, c, *da, *db, *dc;

   cudaMalloc((void **)&da, sizeof(int)); // allocate
   cudaMalloc((void **)&db, sizeof(int)); // on device
   cudaMalloc((void **)&dc, sizeof(int));

   // Copying in blocking-mode
   cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
   add<<<1,1>>>(da, db, dc);
   cudaMemcpy(&c, dc, sizeof(int), cudaMemcpyDeviceToHost);

   printf("a + b = %d\n", c);
   cudaFree(da); cudaFree(db); cudaFree(dc);
   return 0;
}
```

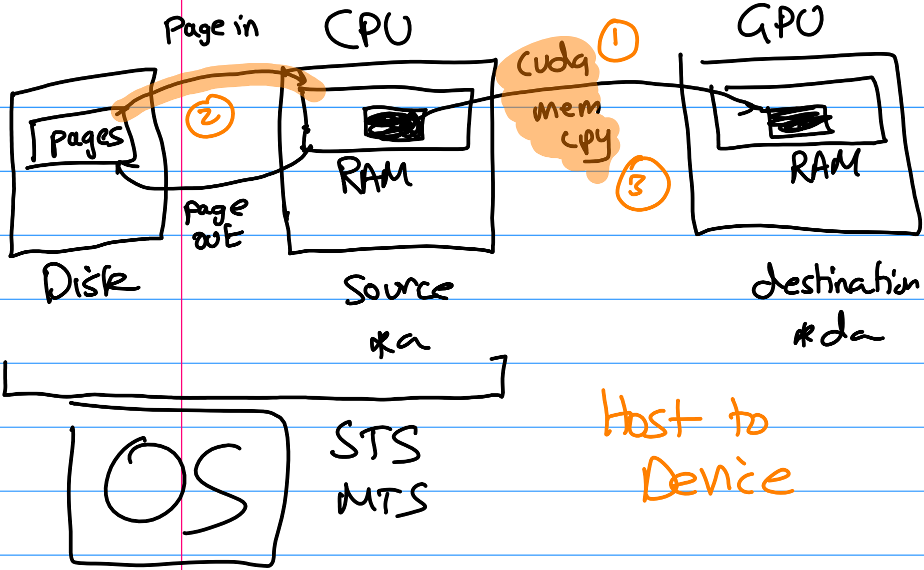
```
/* a, b, c are pointers to memory location on the device */
__global__ void add(int *a, int *b, int *c)
{  *c = *a + *b;
}
int main()
{  int a=2, b=7, c, *da, *db, *dc;

   cudaMalloc((void **)&da, sizeof(int)); // allocate
   cudaMalloc((void **)&db, sizeof(int)); // on device
   cudaMalloc((void **)&dc, sizeof(int));

   // Copying in blocking-mode
   cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
   add<<<1,1>>>(da, db, dc);
   cudaMemcpy(&c, dc, sizeof(int), cudaMemcpyDeviceToHost);

   printf("a + b = %d\n", c);
   cudaFree(da); cudaFree(db); cudaFree(dc);
   return 0;
}
```

c[x] = a[x] + b[x]



4 threads

<<< 1, 4 >>>



Host to
Device

```
/* a, b, c are pointers to memory location on the device */
__global__ void add(int *a, int *b, int *c)
{  *c = *a + *b;
}
int main()
{  int a=2, b=7, c, *da, *db, *dc;

   cudaMalloc((void **)&da, sizeof(int)); // allocate
   cudaMalloc((void **)&db, sizeof(int)); // on device
   cudaMalloc((void **)&dc, sizeof(int));

   // Copying in blocking-mode
   cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
   add<<<1,1>>>(da, db, dc);
   cudaMemcpy(&c, dc, sizeof(int), cudaMemcpyDeviceToHost);

   printf("a + b = %d\n", c);
   cudaFree(da); cudaFree(db); cudaFree(dc);
   return 0;
}
```

Running in Parallel

- so far; pointing parameters to GPU, and running single thread on GPU.
- Time to look at how to run things in parallel.

Code Change

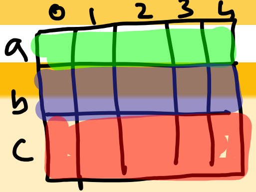
```
// add<<<1, 1>>>(da, db, dc);
   add<<<N, 1>>>(da, db, dc);
```

- Instead of running add() once, execute it N times in parallel

Changes in Kernel Code

<<<N, 1>>>

```
__global__ void add(int *a, int *b, int *c)
{
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

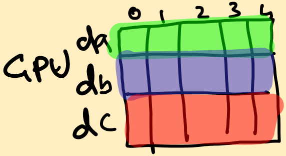


Changes in Host Code

```
int main()
{  int *a, *b, *c, *da, *db, *dc, N=5, i;
```

0

```
   a = (int*)malloc(sizeof(int)*N); // allocate host mem
   b = (int*)malloc(sizeof(int)*N); // and assign random
   c = (int*)malloc(sizeof(int)*N); // memory

   cudaMalloc((void **)&da, sizeof(int)*N);
   cudaMalloc((void **)&db, sizeof(int)*N);
   cudaMalloc((void **)&dc, sizeof(int)*N);
```



1

```
   cudaMemcpy(da, &a, sizeof(int)*N, cudaMemcpyHostToDevice);
   cudaMemcpy(da, &a, sizeof(int)*N, cudaMemcpyHostToDevice);
```

2

```
   add<<<N,1>>>(da, db, dc);
```

3

```
   cudaMemcpy(&c, dc, sizeof(int)*N, cudaMemcpyDeviceToHost);
```

N MP - 1 core .

```
   for (i = 0; i < N; i++)
       printf("a[%d] + b[%d] = %d\n", i, i, c[i]);
}
```

1 MP

<<<1, N>>>

N cores

That was Blocks in Parallel. What about Threads in Parallel

<<<N, 1>>> → c[blockIdx.x]

Function Call Change

```
// add<<<1, 1>>>(da, db, dc); // single thread GPU
// add<<<N, 1>>>(da, db, dc); // N blocks on GPU
   add<<<1, N>>>(da, db, dc); // N threads on GPU
```

Changes in Kernel Code

```
__global__ void add(int *a, int *b, int *c)
{
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- Rest of Host code would be the same

c[x] = a[x] + b[x]

<<<1, N>>> → c[threadIdx.x]