Mergesort with CUDA

25.02.2020

GPUs is fast and easily programmable

Fast: See number of flops Easy: CUDA: Extension to C-language, called from C/C++

Simple Program and Big Architecture Ideas

Merge

Memory Hirachy of CUDA

Merging with local memory

Simple Program and Big Architecture Ideas

A Simple Cuda Programm

1: Call Cuda Program from C/C++:

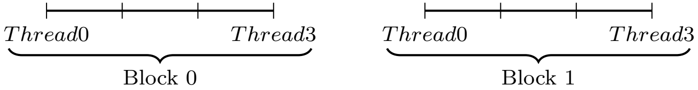
```
dim3 Grid(2)
dim3 Block(4)
add_arrays<<<Grid, Block>>>(...)
```

A Simple Cuda Programm

1: Call Cuda Program from C/C++:

```
dim3 Grid(2)
dim3 Block(4)
add_arrays<<<Grid, Block>>>(...)
```

2: Blocks & Grid spawn many threads on GPU:

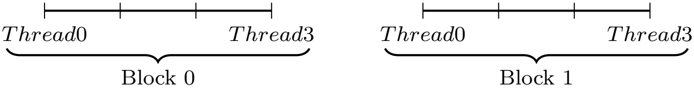


A Simple Cuda Programm

1: Call Cuda Program from C/C++:

```
dim3 Grid(2)
dim3 Block(4)
add_arrays<<<Grid, Block>>>(...)
```

2: Blocks & Grid spawn many threads on GPU:



3: Cuda Code is a C-Extension:

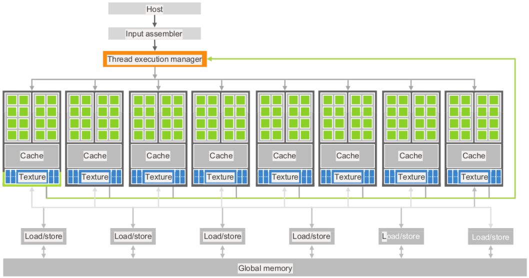
```
__global__
add_arrays(float* A, float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
```

Software/Hardware interface

```
dim3 Grid(2)
dim3 Block(4)
add_arrays<<<Grid, Block>>>(...)
```

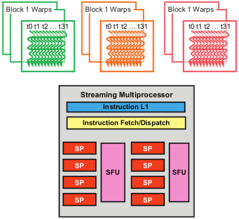
1. Software: Specify Blocks and Threads

Software/Hardware interface



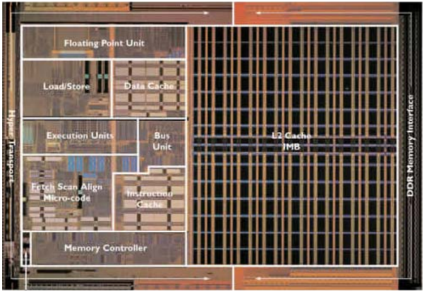
1. Software: Specify Blocks and Threads
2. Hardware:
1. Blocks to SM: Block 1 $\rightarrow$ SM1, Block 2 $\rightarrow$ SM2 ...

Software/Hardware interface



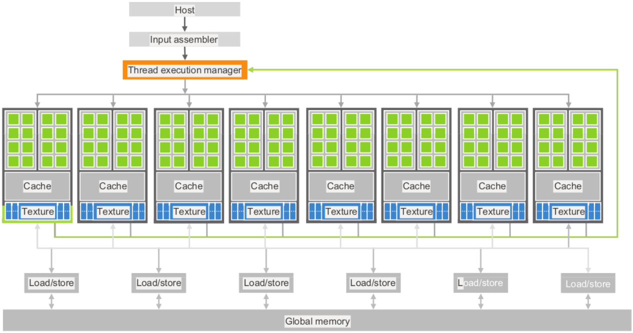
1. Software: Specify Blocks and Threads
2. Hardware:
1. Blocks to SM: Block 1 $\rightarrow$ SM1, Block 2 $\rightarrow$ SM2 …
2. Threads to Processors: Wrappend threads are allocated

Reasons for strong GPU performance



CPU: Chip full with fancy Branch Prediction, Caches, etc.

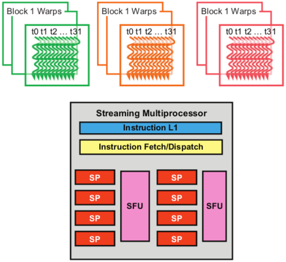
Reasons for strong GPU performance



CPU: Chip full with fancy Branch Prediction, Caches, etc.
GPU:

1. Simplicity: Just Cores on chip

Reasons for strong GPU performance



CPU: Chip full with fancy Branch Prediction, Caches, etc.

GPU:

1. Simplicity: Just Cores on chip
2. Multithreading: Wrap 1 in a long-latency op: Start Wrap 2

Merge

Serial Merge

$$A = \begin{bmatrix} 5 & 7 & 8 & 12 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 & 4 & 6 & 10 \end{bmatrix}$$

$$C = \begin{bmatrix} ? & ? & ? & ? & ? & ? & ? & ? \end{bmatrix}$$

```
void merge(T* a, T* b, T* c, int sz_a, int sz_b) {
    int i = 0, j = 0, k = 0;
    while (k < sz_a + sz_b)
        if (i == sz_a)
            c[k++] = b[j++];
        else if (j == sz_b)
            c[k++] = a[i++];
        else if (a[i] <= b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
}
```

Problem: complexity $O(n)$

How to split A and B to spwan many threads?

Example:

$$A = 0 \quad 0 \quad 0 \quad 0$$

$$B = 1 \quad 1 \quad 1 \quad 1$$

$$C = ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad ?$$

How to split A and B to spwan many threads?

Example:

$$A = 0 \quad 0 \quad 0 \quad 0$$

$$B = 1 \quad 1 \quad 1 \quad 1$$

$$C = ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad ?$$

Naive: 2 Threads, half A and B

How to split A and B to spwan many threads?

Example:

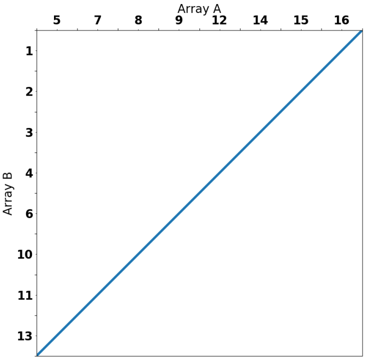
$$A = 0 \quad 0 \quad 0 \quad 0$$
$$B = 1 \quad 1 \quad 1 \quad 1$$
$$C = ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad ?$$

Naive: 2 Threads, half A and B Result:

Instead: How to split arrays

Odeh et al. (2012)

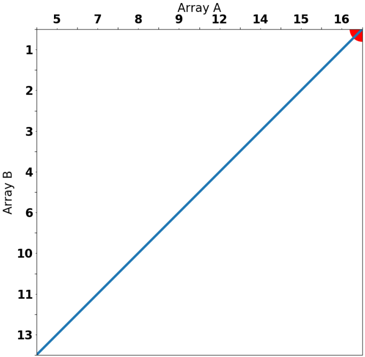
How to split two arrays in equal chuncks?



Instead: How to split arrays

Odeh et al. (2012)

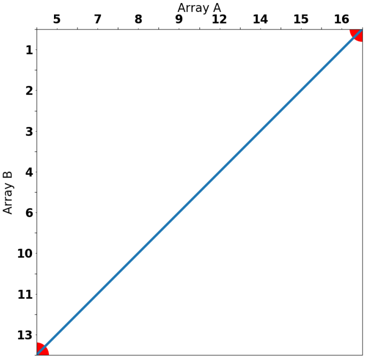
Feasible split: Array A to Thread 1, B to Thread 2



Instead: How to split arrays

Odeh et al. (2012)

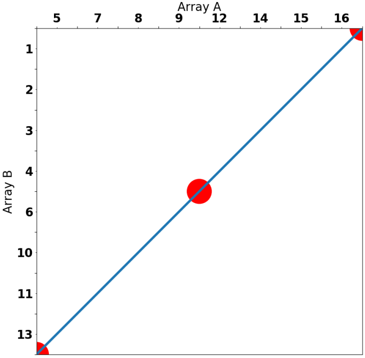
Another split: Array B to Thread 1, A to Thread 2



Instead: How to split arrays

Odeh et al. (2012)

Another (as before): Thread 1 gets half of A and B

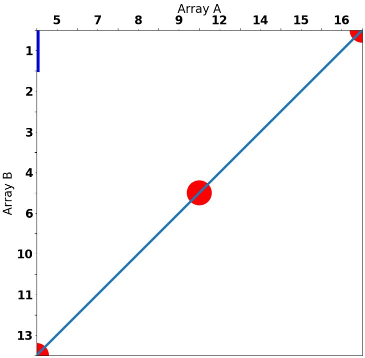


Summary: All allocations along vertical line split work equally!

Instead: How to split arrays

Odeh et al. (2012)

Mergepath: Optimal split: One Elemet of B



Summary: All allocations along vertical line split work equally!

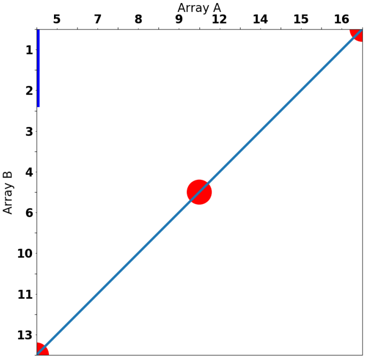
Mergpath: Vertical move: pick array B; horizontal move: Array

Merge

Instead: How to split arrays

Odeh et al. (2012)

Mergepath: Optimal split: Another Elemet of B



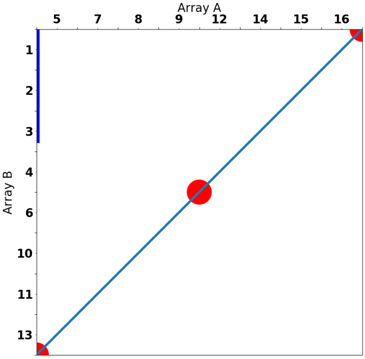
Summary: All allocations along vertical line split work equally!

Mergpath: Vertical move: pick array B; horizontal move: Array

Instead: How to split arrays

Odeh et al. (2012)

Mergepath: Optimal split: Another Elemet of B



Summary: All allocations along vertical line split work equally!

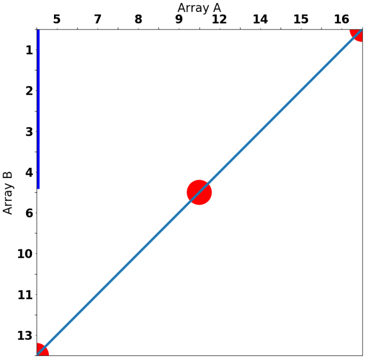
Mergpath: Vertical move: pick array B; horizontal move: Array

Merge

Instead: How to split arrays

Odeh et al. (2012)

Mergepath: Optimal split: Another Elemet of B



Summary: All allocations along vertical line split work equally!

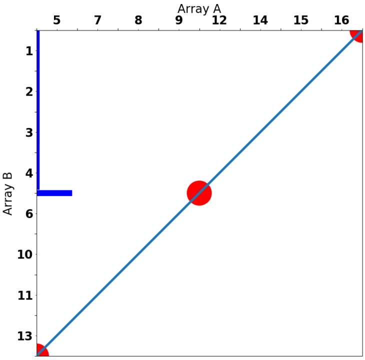
Mergpath: Vertical move: pick array B; horizontal move: Array

Merge

Instead: How to split arrays

Odeh et al. (2012)

Mergepath: Optimal split: First elment of A



Summary: All allocations along vertical line split work equally!

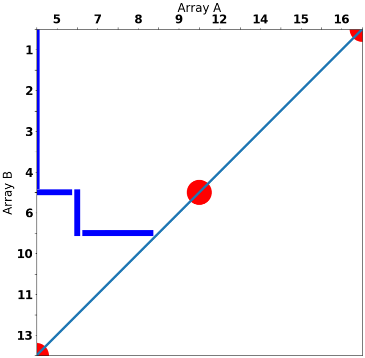
Mergpath: Vertical move: pick array B; horizontal move: Array

Merge

Instead: How to split arrays

Odeh et al. (2012)

Mergepath: Optimal split!



Split index of A: 3

Split index of B: 5

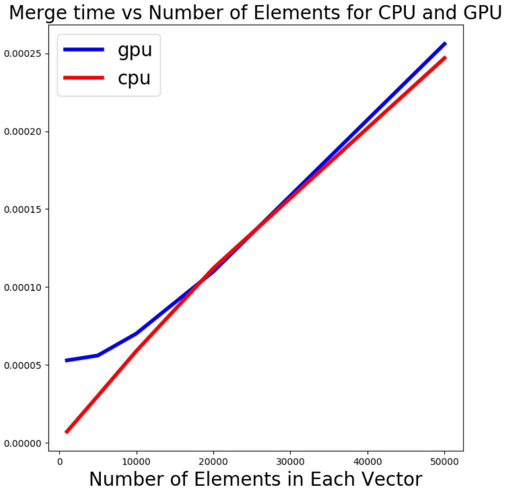
Merge with Cuda

```
__global__
void paralleMerge(int* a, int sz_a, int* b, int sz_b,
                  int* c, int length)
{
    int diag = threadIdx.x * length;
    int a_split = mergepath(a, sz_a, b, sz_b, diag);
    int b_split = diag - a_split;
    merge(a, a_split, sz_a, b, b_split, sz_b, c, diag, leng
}
```

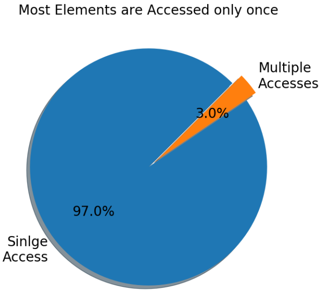
Each identifies split indices

Split indices suffice to merge two sub-arrays into c

Problem: Slow as a Snail

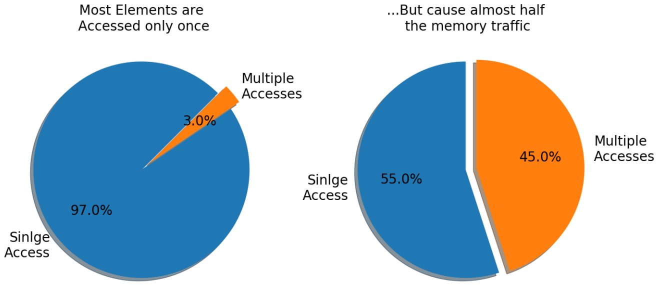


Reason: So much global meory access



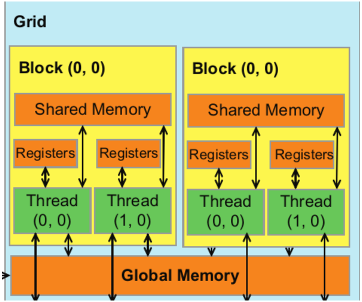
Merge

Reason: So much global meory access



Memory Hirachy of CUDA

Different Memories and Size



Visibility:

Global ⇔ Shared: All threads ⇔ Private for Block

Size:

Global ⇔ Shared: 2 GB ⇔ Total: 192 KB, per block: 48 KB

Latency:

Global ⇔ Shared: 8 GB/s ⇔ 80 GB/s

Merging with local memory

General Idea

Problem: Block shared memory: 48 Kb -> need to make blocks small enough

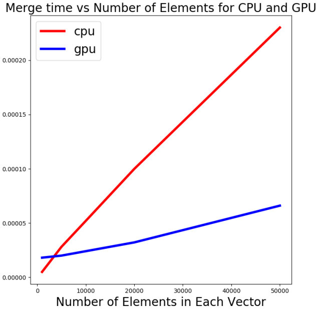
Solution:

- Determine large but small enough chuncks of arrays to load into blocks
- Load into shared memeory
- Each thread determines its range (as before), from shared memory
- Each thread merges subset (as before)

```
__global__
void paralleMerge(const int* a, int sz_a, const int* b, int
                  int* c, int* boundaries, int length,
                  int size_shared) {
    extern __shared__ int shared[];
    __shared__ int block_ranges[4];
    ranges(block_ranges, sz_a, sz_b, boundaries);
    loadtodevice(a, sz_a, b, sz_b, block_ranges, shared);
    int diag = threadIdx.x * length;
    if (diag < block_ranges[2] + block_ranges[3]) {
        int a_start =
            mergepath(shared, block_ranges[2], &shared[bloc
                      block_ranges[3], diag);
        int b_start = diag - a_start;
        merge(shared, a_start, block_ranges[2], &shared[blo
              b_start, block_ranges[3], c, diag + blockIdx.
              length);
    }
}
```

Merging with local memory

Show the results



Saher Odeh et al. “Merge Path - Parallel Merging Made Simple”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum.* ISSN: null. May 2012, pp. 1611–1618 (cit. on pp. 19–28).