Mergesort with CUDA

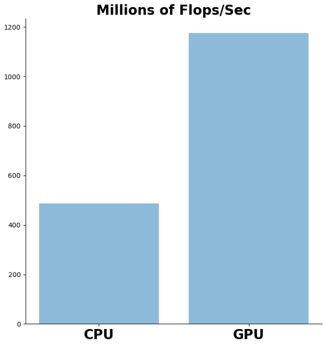
Fabian Schuetze

25.02.2020

Fork on Github:
https://github.com/FabianSchuetze/mergesort

GPU: fast, cheap, and approachable

Fast:



Cheap: CPU: $400, GPU: $200

Approachable: CUDA: Extension to C, called from C/C++

Intro I: Simple Program

A Simple Cuda Programm

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

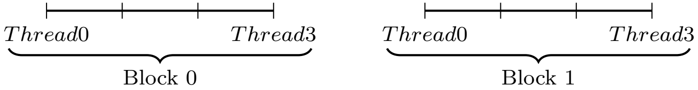
```
dim3 Grid(2)
dim3 Block(4)
add_arrays<<<Grid, Block>>>(...)
```

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 Threads:

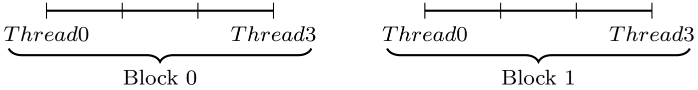


A Simple Cuda Programm

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

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

2: Blocks & Grid spawn Threads:



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];
    }
```

Intro II: Nvidia's two big Architecture Decisions

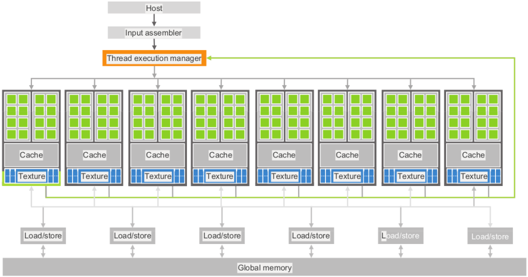
Software/Hardware interface

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

1. Software: Specify Blocks and Threads

Software/Hardware interface

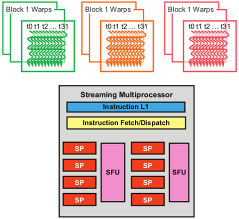
GPU: Rectangles are Sets of Processors



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

Software/Hardware interface

GPU: Group of Threads allocated to Processors

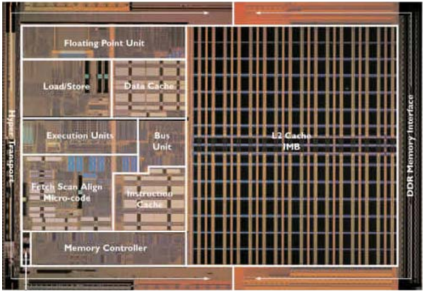


1. Software: Specify Blocks and Threads
2. Hardware:
    1. Blocks to SM: Block 1 $\rightarrow$ SM1, Block 2 $\rightarrow$ SM2 ...
    2. Threads to Processors: Wrappend threads are allocated

Reasons for strong GPU performance

Patterson and Hennessy (2009)

AMD Opteron: FP and Int Execution Units are small

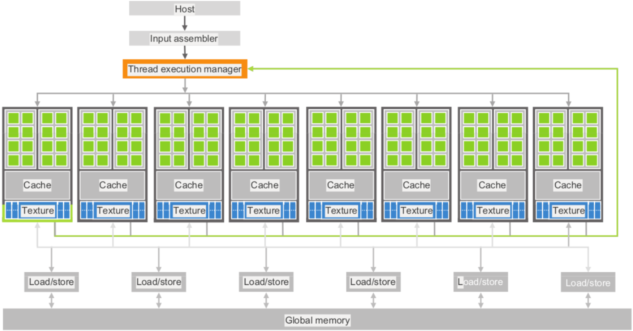


CPU: Chip full with fancy Branch Prediction, Caches, etc.

Reasons for strong GPU performance

Patterson and Hennessy (2009)

Nvidia GPU: Small Blocks are Execution Units



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

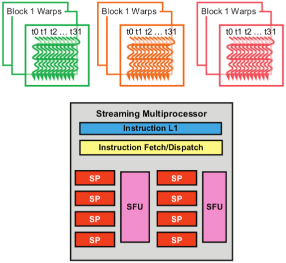
GPU:

1. Simplicity: Just Cores on chip

Reasons for strong GPU performance

Patterson and Hennessy (2009)

Multithreading: Context Switches are Cheap



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

GPU:

1. Simplicity: Just Cores on chip
2. Multithreading: Wrap1 in long-latency op: Start Wrap2

Merge

Serial Merge

$$A = \begin{bmatrix} 5 & 7 & 8 & 12 \end{bmatrix}; \quad B = \begin{bmatrix} 3 & 4 & 6 & 10 \end{bmatrix}$$

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

```
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, $\mathcal{O}(n)$

How to split A and B to spwan many threads?

Example:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}; \quad B = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$$
$$C = \begin{bmatrix} ? & ? & ? & ? & ? & ? & ? \end{bmatrix}$$

How to split A and B to spwan many threads?

Example:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}; \quad B = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$$
$$C = \begin{bmatrix} ? & ? & ? & ? & ? & ? & ? \end{bmatrix}$$

Naive: 2 Threads, half A and B

How to split A and B to spwan many threads?

Example:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}; \quad B = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$$

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

Naive: 2 Threads, half A and B

Result:

Thread 1: Merge:

$$A_1 = \begin{bmatrix} 0 & 0 \end{bmatrix}; \quad B_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}; \qquad C_1 = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$$

Thread 2: Merge:

$$A_2 = \begin{bmatrix} 0 & 0 \end{bmatrix}; \quad B_2 = \begin{bmatrix} 1 & 1 \end{bmatrix}; \qquad C_2 = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$$

How to split A and B to spwan many threads?

Example:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}; \quad B = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$$
$$C = \begin{bmatrix} ? & ? & ? & ? & ? & ? & ? \end{bmatrix}$$

Naive: 2 Threads, half A and B

Result:

Thread 1: Merge:

$$A_1 = \begin{bmatrix} 0 & 0 \end{bmatrix}; \quad B_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}; \qquad C_1 = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$$

Thread 2: Merge:

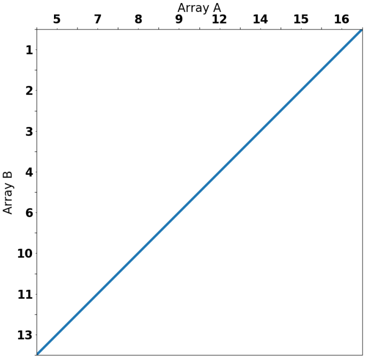
$$A_2 = \begin{bmatrix} 0 & 0 \end{bmatrix}; \quad B_2 = \begin{bmatrix} 1 & 1 \end{bmatrix}; \qquad C_2 = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$$

Naive strategy: Doesn't work

Instead: How to split arrays

Odeh et al. (2012); Baxter (2016)

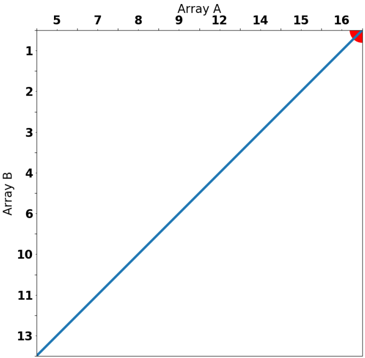
How to split two arrays in equal chuncks?



Instead: How to split arrays

Odeh et al. (2012); Baxter (2016)

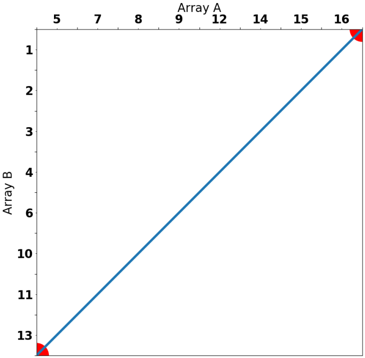
Feasible split: Array A to Thread 1, B to Thread 2



Instead: How to split arrays

Odeh et al. (2012); Baxter (2016)

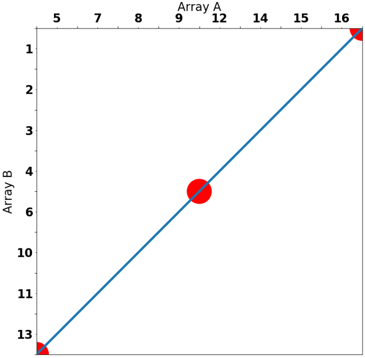
Another split: Array B to Thread 1, A to Thread 2



Instead: How to split arrays

Odeh et al. (2012); Baxter (2016)

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



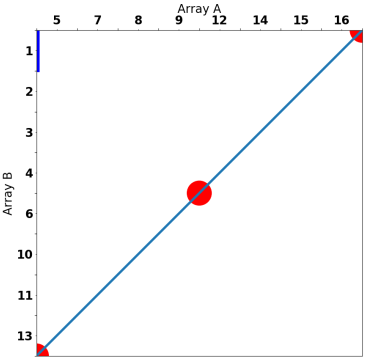
Summary: All allocations along vertical line split work equally!

Merge

Instead: How to split arrays

Odeh et al. (2012); Baxter (2016)

Mergepath: Optimal split: One Elemet of B



Summary: All allocations along vertical line split work equally!

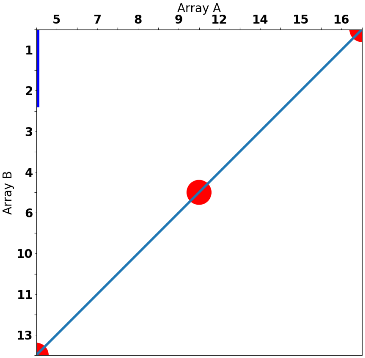
Mergpath: Vertical move: pick array B; horizontal move: Array

Merge

Instead: How to split arrays

Odeh et al. (2012); Baxter (2016)

Mergepath: Optimal split: Another Elemet of B



Summary: All allocations along vertical line split work equally!

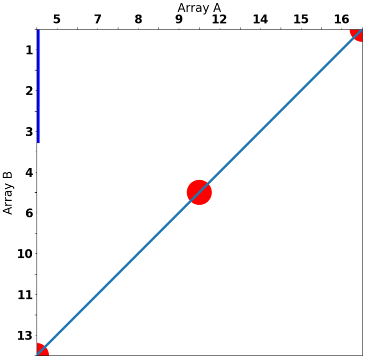
Mergpath: Vertical move: pick array B; horizontal move: Array

Merge

Instead: How to split arrays

Odeh et al. (2012); Baxter (2016)

Mergepath: Optimal split: Another Elemet of B



Summary: All allocations along vertical line split work equally!

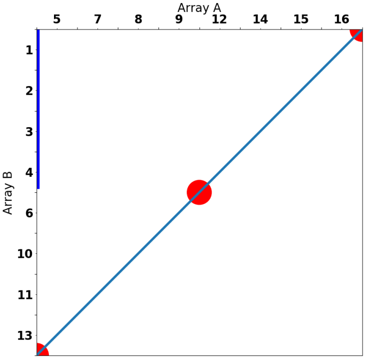
Mergpath: Vertical move: pick array B; horizontal move: Array

Merge

Instead: How to split arrays

Odeh et al. (2012); Baxter (2016)

Mergepath: Optimal split: Another Elemet of B



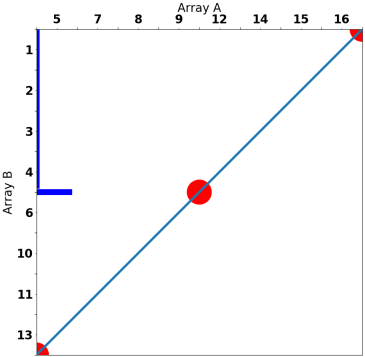
Summary: All allocations along vertical line split work equally!

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

Instead: How to split arrays

Odeh et al. (2012); Baxter (2016)

Mergepath: Optimal split: First elment of A



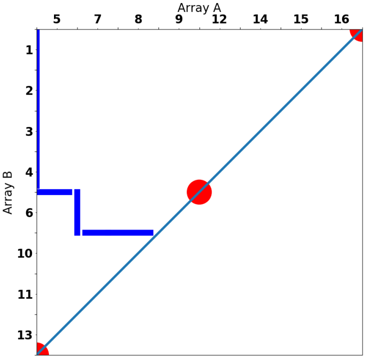
Summary: All allocations along vertical line split work equally!

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

Instead: How to split arrays

Odeh et al. (2012); Baxter (2016)

Mergepath: Optimal split!



Split index of A: 3

Split index of B: 5

Merge

Merge with Cuda (Divide-and-Conquer)

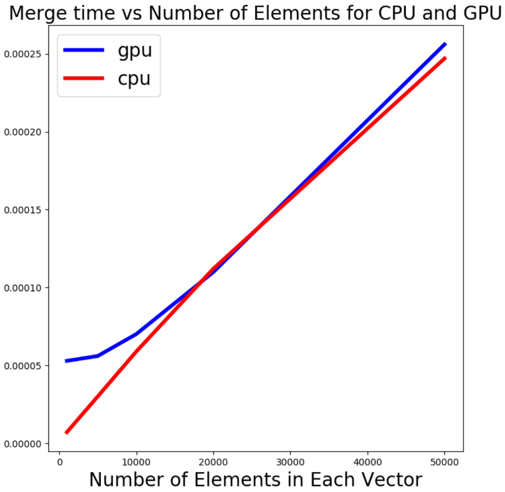
```
__global__
void paralleMerge(int* a, int sz_a, int* b, int sz_b,
                  int* c, int length)
{
    int diag = threadIdx.x * length;
    int a_split = mergepath(a, sz_a, b, sz_b, diag);
    int b_split = diag - a_split;
    merge(a, a_split, sz_a, b, b_split, sz_b, c, diag, leng
}
```

Each identifies split indices

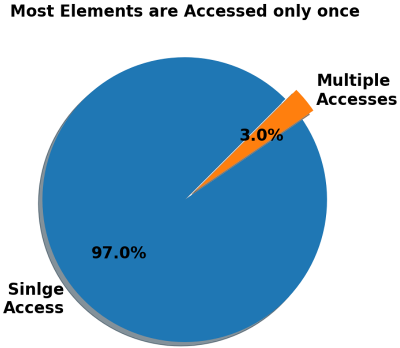
Split indices suffice to merge two sub-arrays into c

Problem: Slow as a Snail

`std::merge` (with -O3 optimization) is as fast!

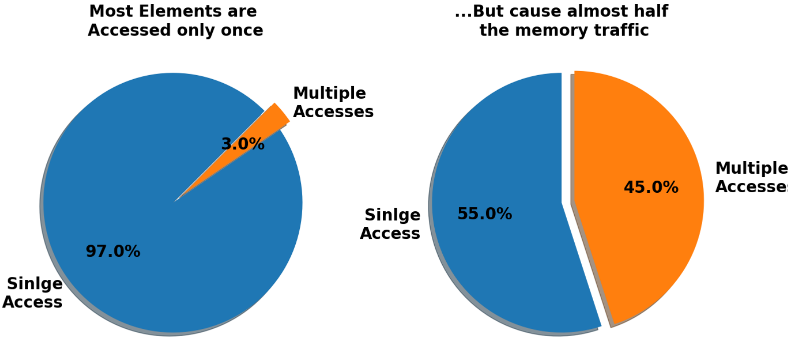


Reason: Too much global meory access



Merge

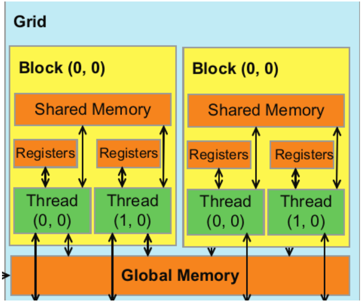
Reason: Too much global meory access



Memory Hirachy of CUDA

Shared memory: Tiny but Fast

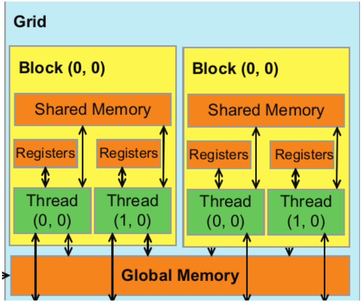
Kirk and Hwu (2012)



	Global $\Leftrightarrow$ Shared:
Visibility:	All threads $\Leftrightarrow$ Private for Block
Size:	2 GB $\Leftrightarrow$ Total/Block 192/48 KB
Latency:	8 GB/s $\Leftrightarrow$ 80 GB/s

Shared memory: Tiny but Fast

Kirk and Hwu (2012)



	Global ⇔ Shared:
Visibility:	All threads ⇔ Private for Block
Size:	2 GB ⇔ Total/Block 192/48 KB
Latency:	8 GB/s ⇔ 80 GB/s

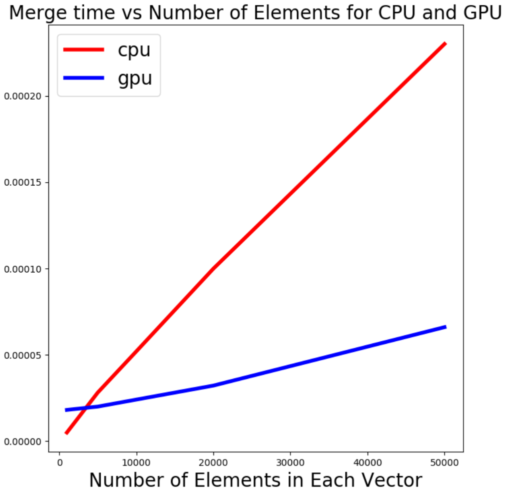
Conclusion: Load Subarrays in Shared ⇒ faster memory access

Merging with local memory

Merging with shared memory: Use MergePath twice

1. Problem: Shared memory tiny: 48 Kb
2. Solution: Split arrays into subarrays by block
3. Approach (Divide-and-Conquer twice):
   Identify subarrays with `mergepath`; smaller than 48KB
   Load into shared memeory
   Threads use `mergepath` again (now from shared memory)
   Threads merge subset (as before)

With Shared Memory: GPU faster than `std::merge`



Sean Baxter. *Intro - Modern GPU.* 2016 (cit. on pp. 21–30).

David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* English. 2 edition. Amsterdam: Morgan Kaufmann, Dec. 2012 (cit. on pp. 36, 37).

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. 21–30).

David A. Patterson and John L. Hennessy. *Computer organization and design: the hardware/software interface.* 4th ed. Burlington, MA: Morgan Kaufmann Publishers, 2009 (cit. on pp. 12–14).