

Mergesort with CUDA

Fabian Schuetze

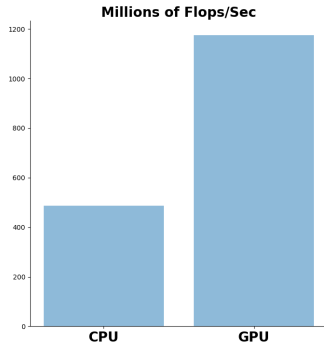
25.02.2020

Fork on Github:

<https://github.com/FabianSchuetze/mergesort>

GPU: fast, cheap, and approachable

Fast:



Cheap: CPU: \$400, GPU: \$200

Approachable: CUDA: C++ Syntax, called from C and C++

Intro I: Simple Program

Intro II: Nvidia's two big Architecture Decisions

Merge

Memory Hirachy of CUDA

Merging with local memory

Intro I: Simple Program

A Simple Cuda Program

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

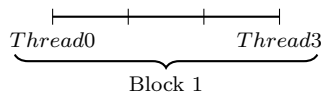
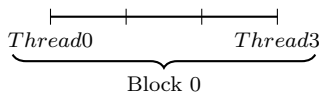
```
1  dim3 Grid(2)
2  dim3 Block(4)
3  square<<<Grid, Block>>>(...)
```

A Simple Cuda Program

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

```
1 dim3 Grid(2)
2 dim3 Block(4)
3 square<<<Grid, Block>>>(...)
```

2: Blocks & Grid spawn Threads:

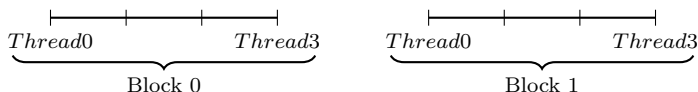


A Simple Cuda Program

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

```
1 dim3 Grid(2)
2 dim3 Block(4)
3 square<<<Grid, Block>>>(...)
```

2: Blocks & Grid spawn Threads:



3: Cuda Code follows C++ Syntax with Extension:

```
1 __global__ void square(int *array, int n) {
2     int tid = blockDim.x * blockIdx.x + threadIdx.x;
3     if (tid < n) {
4         array[tid] = array[tid] * array[tid];
5     }
6 }
```

How a full CUDA Program is Run:

```
1  __global__ void square(int *array, int n) {
2      int tid = blockDim.x * blockIdx.x + threadIdx.x;
3      if (tid < n) array[tid] = array[tid] * array[tid];
4  }
5
6  int main() {
7      int a[3]; fill_array(a, 3);
8      // 1. Initialize Memory on GPU and copy from RAM //
9      unsigned int sz = 3 * sizeof(int);
10     int* d_a;
11     cudaMalloc((int**)&d_a, sz);
12     cudaMemcpy(d_a, a, sz, cudaMemcpyHostToDevice);
13     // 2. Define Thread Structure and Launch Kernel //
14     dim3 block(4), grid(2);
15     square<<<grid, block>>>(d_a, 3);
16     // 2. Copy GPU memory back to RAM //
17     cudaMemcpy(a, d_a, sz, cudaMemcpyDeviceToHost);
```


Intro II: Nvidia's two big Architecture Decisions

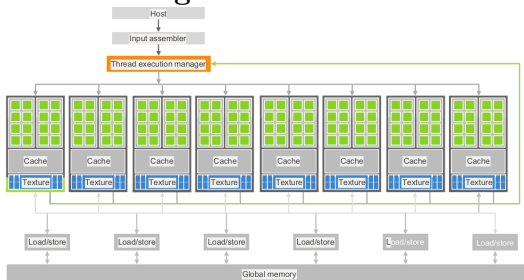
Software/Hardware interface

```
1  dim3 block(4);  
2  dim3 grid(2);  
3  square<<<grid, block>>>(...);
```

1. Software: Specify Blocks and Threads

Software/Hardware interface

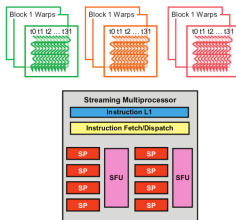
GPU: Rectangles are Sets of Processors



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

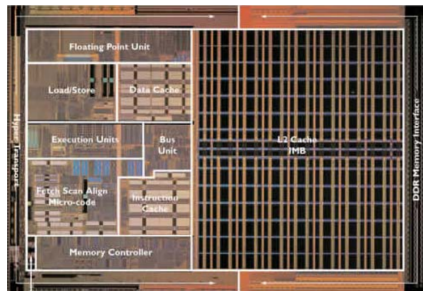


1. **Software:** Specify Blocks and Threads
2. **Hardware:**
 1. **Blocks to SM:** Block 1 \rightarrow SM1, Block 2 \rightarrow SM2 ...
 2. **Threads to Processors:** Wrapped threads are allocated

Reasons for strong GPU performance

Patterson and Hennessy (2009)

AMD Opteron: FP and Int Execution Units are small

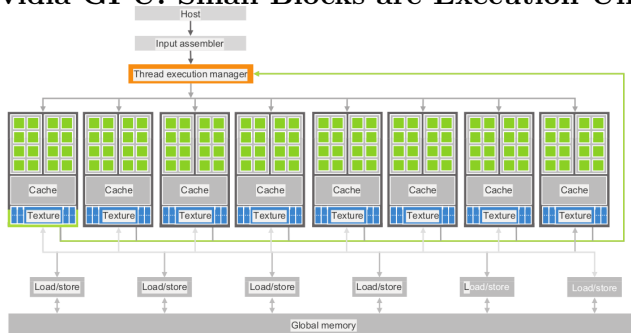


CPU: Chip full with fancy Branch Prediction, Caches, etc.

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.

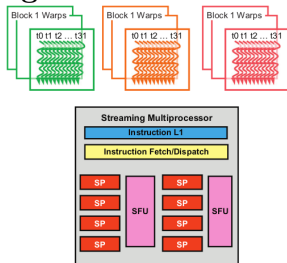
GPU:

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

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

```
1  void merge(T* a, T* b, T* c, int sz_a, int sz_b) {
2      int i = 0, j = 0, k = 0;
3      while (k < sz_a + sz_b)
4          if (i == sz_a)
5              c[k++] = b[j++];
6          else if (j == sz_b)
7              c[k++] = a[i++];
8          else if (a[i] <= b[j])
9              c[k++] = a[i++];
10         else
11             c[k++] = b[j++];
12 }
```

Problem: complexity, $\mathcal{O}(n)$

How to split A and B to spawn 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 spawn 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 spawn 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:

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

Thread 2: Merge:

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

How to split A and B to spawn 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:

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

Thread 2: Merge:

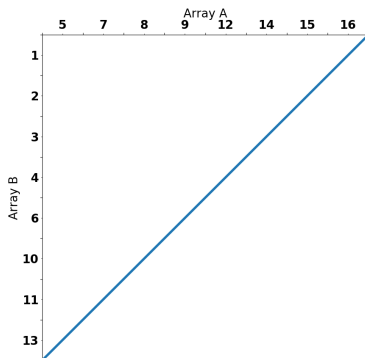
$$A_2 = \begin{bmatrix} 0 & 0 \end{bmatrix}; \quad B_2 = \begin{bmatrix} 1 & 1 \end{bmatrix}; \quad C_2 = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$$

Naive strategy: Doesn't work

Instead: How to split arrays

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

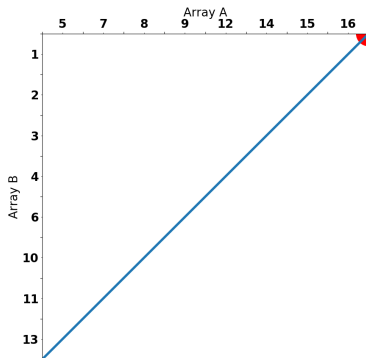
How to split two arrays in equal chunks?



Instead: How to split arrays

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

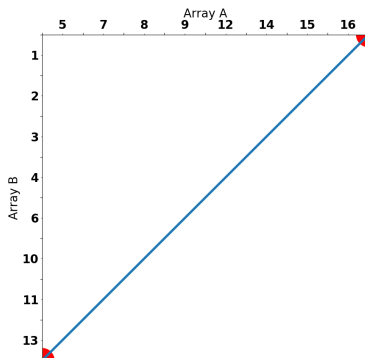
Feasible split: Array A to Thread 1, B to Thread 2



Instead: How to split arrays

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

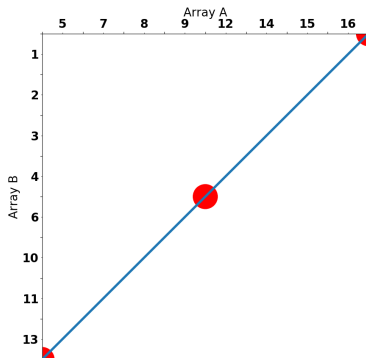
Another split: Array B to Thread 1, A to Thread 2



Instead: How to split arrays

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

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

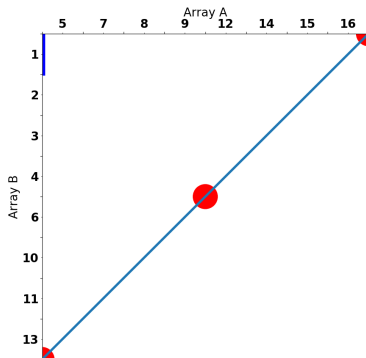


Summary: All allocations along vertical line split work equally!

Instead: How to split arrays

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

Mergepath: Optimal split: One Element of B



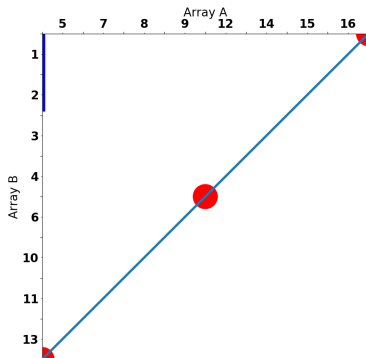
Summary: All allocations along vertical line split work equally!

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 Element of B



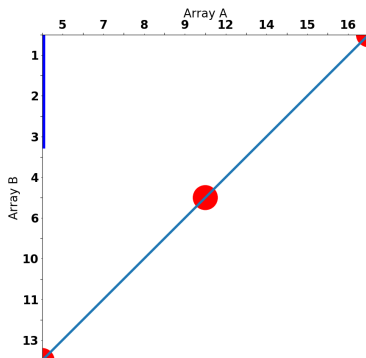
Summary: All allocations along vertical line split work equally!

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 Element of B



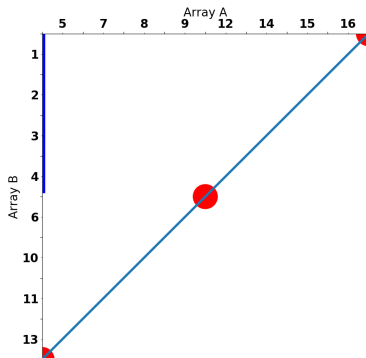
Summary: All allocations along vertical line split work equally!

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 Element of B



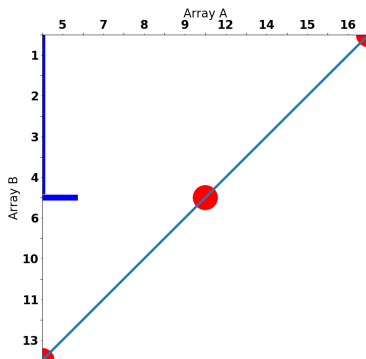
Summary: All allocations along vertical line split work equally!

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

Instead: How to split arrays

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

Mergepath: Optimal split: First element of A



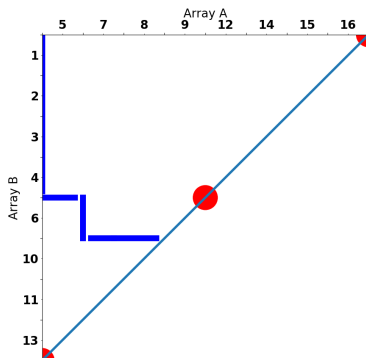
Summary: All allocations along vertical line split work equally!

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

Instead: How to split arrays

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

Mergepath: Optimal split!



Split index of A: 3

Split index of B: 5

Merge

Merge with Cuda (Divide-and-Conquer)

```
1  __global__
2  void parallelMerge(int* a, int sz_a, int* b, int sz_b,
3                    int* c, int length)
4  {
5      int diag = threadIdx.x * length;
6      int a_split = mergepath(a, sz_a, b, sz_b, diag);
7      int b_split = diag - a_split;
8      merge(a, a_split, sz_a, b, b_split, sz_b, c, diag,
9           length);
10 }
```

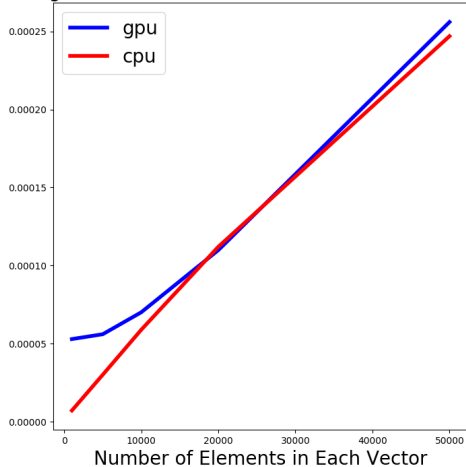
Each identifies split indices

Split indices suffice to merge two sub-arrays into c

Problem: Slow as a Snail

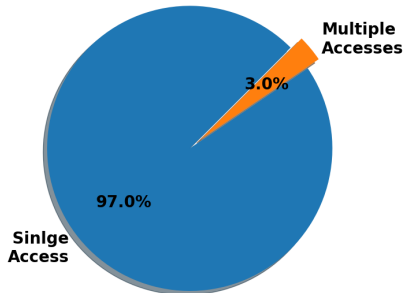
std::merge (with -O3 optimization) is as fast!

Merge time vs Number of Elements for CPU and GPU

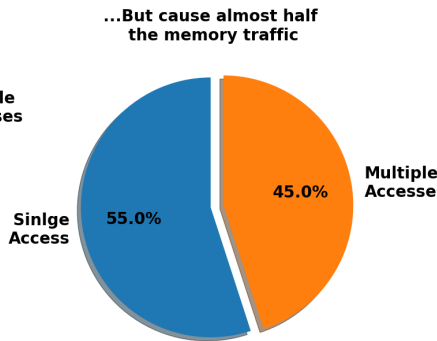
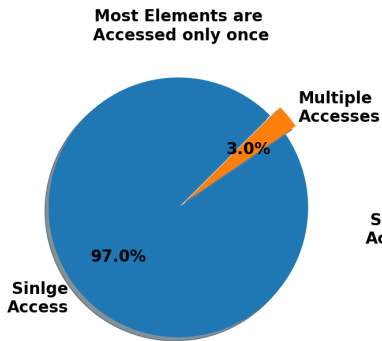


Reason: Too much global memory access

Most Elements are Accessed only once



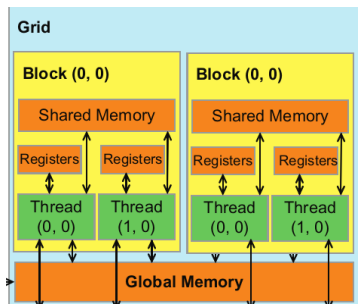
Reason: Too much global memory access



Memory Hirachy of CUDA

Shared memory: Tiny but Fast

Kirk and Hwu (2012)



Global \Leftrightarrow Shared:

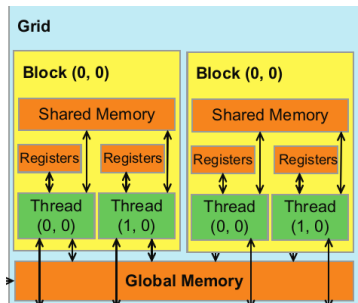
Visibility: All threads \Leftrightarrow Private for Block

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 \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

Conclusion: Load Subarrays in Shared \Rightarrow 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

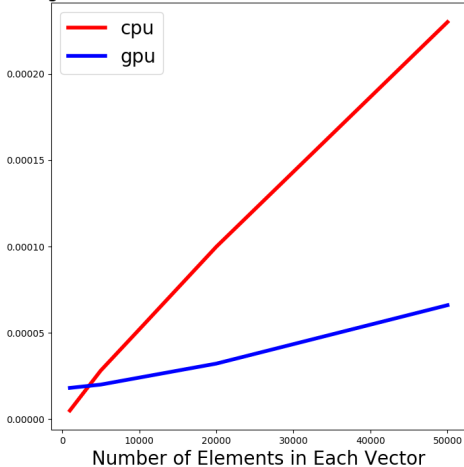
Load into shared memory

Threads use `mergepath` again (now from shared memory)

Threads merge subset (as before)

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

Merge time vs Number of Elements for CPU and GPU



-  Sean Baxter. *Intro - Modern GPU*. 2016 (cit. on pp. 22–31).
-  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. 37, 38).
-  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. 22–31).
-  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. 13–15).