

Optimization Techniques for Large Data Structures on CUDA

Jakob Siegel, Juergen Ributzka
and Xiaoming Li

Overview

- Introduction to CUDA
- Memory Optimizations
- Loop unrolling revisited
- Conclusion

CUDA Real "Hardware"

	Intel Core 2 Extreme QX9650	NVIDIA GeForce GTX 280
Transistors	820 million	1.4 billion
Processor frequency	3 GHz	1296 MHz
Cores	4	240
Cache/Shared Memory	6 MB x 2	16 KB x 30
Threads executed per cycle	4	240
Active hardware threads	4	30720
Peak FLOPS	96 GFLOPS	933 GFLOPS
Memory controllers	off-die	8 x 64 bit
Memory bandwidth	12.8 GBps	141.7 GBps

CUDA Programming Model

- The GPU is seen as a *compute device* to execute a portion of an application that
 - Has to be executed many times
 - Can be isolated as a function
 - Works independently on different data
- Such a function can be compiled to run on the *device*. The resulting program is called a Kernel

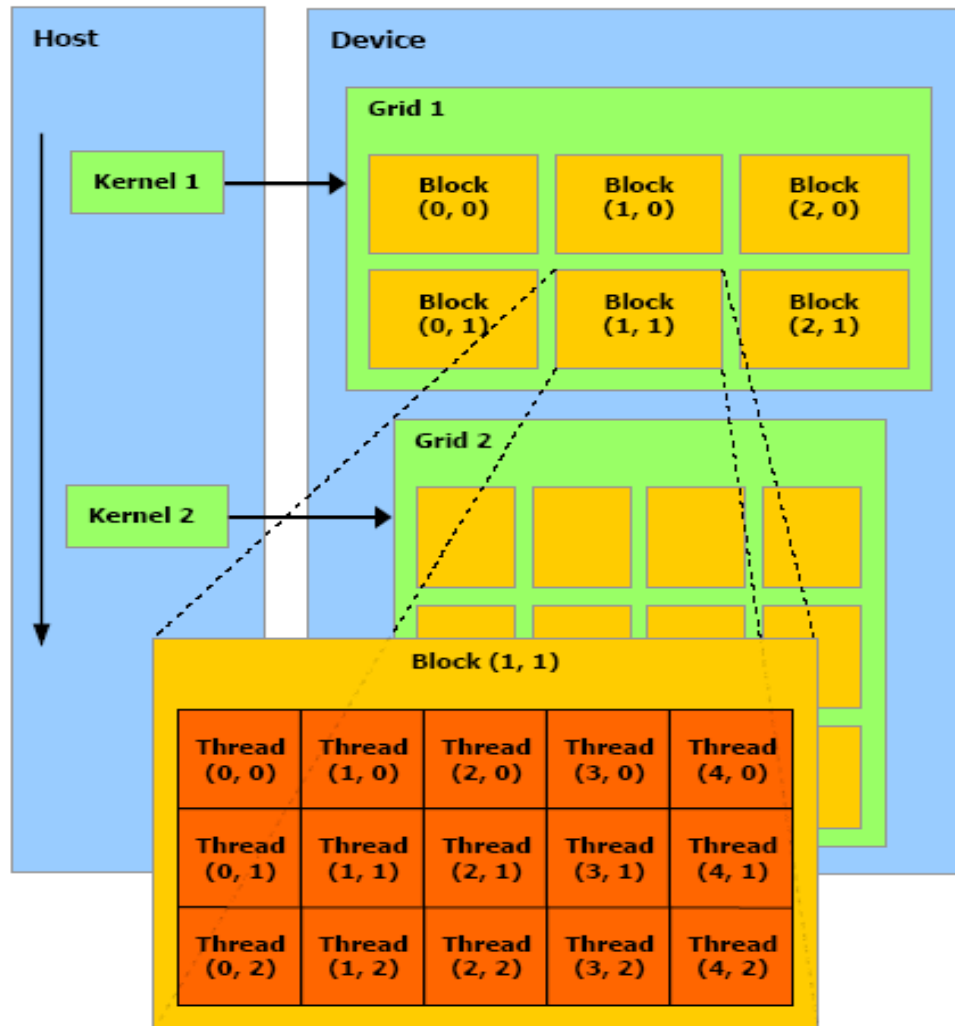
CUDA Programming Model

- The batch of threads that executes a kernel is organized as a grid of thread blocks

CUDA Programming Model

- Thread Block
 - Batch of threads that can cooperate together
 - Fast shared memory
 - Synchronizable
 - Thread ID
 - Limited number of threads in a block
- Threads in a block are executed in batches of SIMD threads
 - The SIMD thread batch is called Warp

CUDA Programming Model



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

Physical Limits for G80

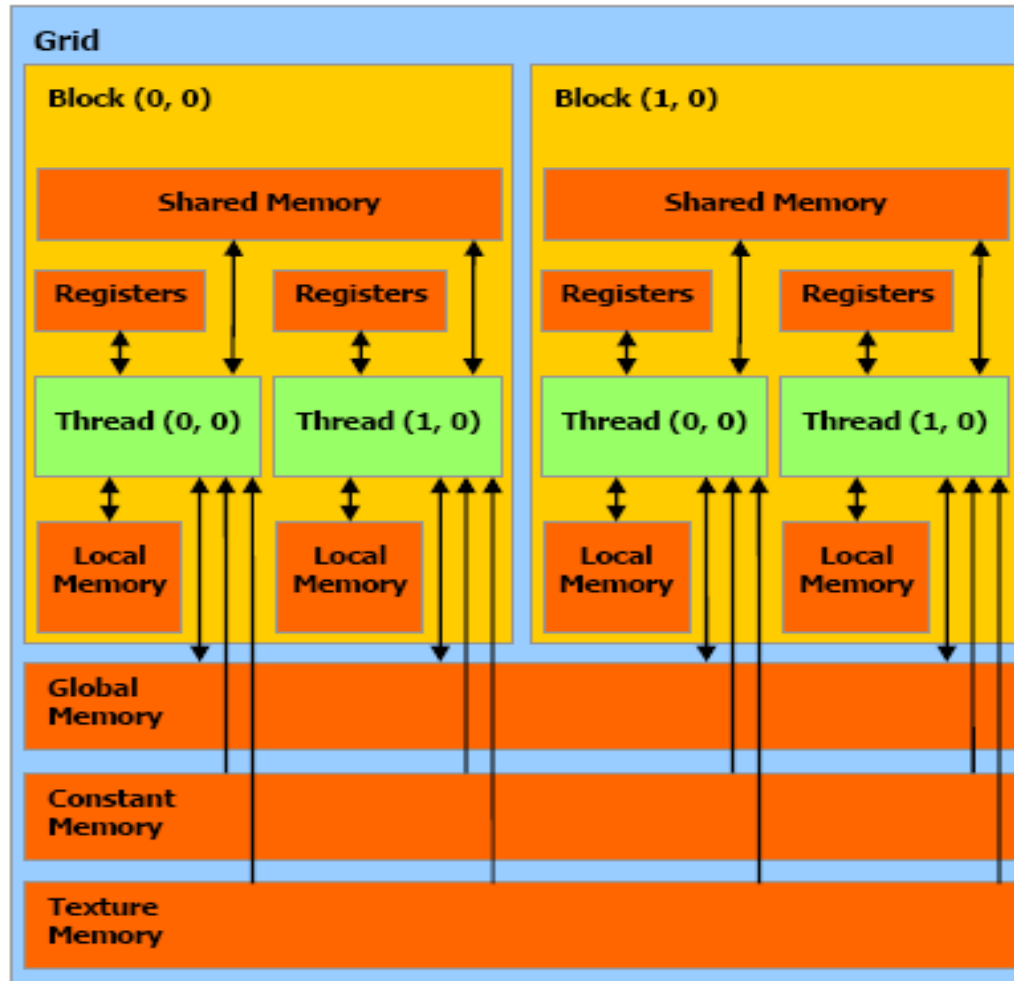
Maximum Thread Blocks Per Multiprocessor

Limited by Max Warps / Multiprocessor

Limited by Registers / Multiprocessor

Limited by Shared Memory / Multiprocessor

CUDA Memory Model



CUDA Memory Model

- A thread has only access to the device's DRAM and on-chip memory
 - Read-write per-thread *registers*,
 - Read-write per-thread *local memory*,
 - Read-write per-block *shared memory*,
 - Read-write per-grid *global memory*,
 - Read-only per-grid *constant memory*,
 - Read-only per-grid *texture memory*.

CUDA Memory Model

- global, constant, and texture memory
 - Can be read from or written to by the host
 - persistent across kernel calls by one application
 - optimized for different memory usages

CUDA Memory Model

- Global Memory
 - Is not cached
 - Important to follow the right access pattern
 - Read 64-bit or 128-bit words with single instruction
 - Alignment requirements automatically fulfilled for built-in types
 - Alignment requirements can be enforced by alignment specifiers.

CUDA Memory Model

- Memory Instructions
 - 2 clock cycles to issue memory instruction
 - Global memory additional 200 to 300 clock cycles of memory latency
 - Global memory latency can be hidden by thread scheduler → sufficient independent arithmetic instructions

CUDA Memory Model

- Memory Bandwidth
 - Depends on the memory access pattern
 - Device memory is of much higher latency and lower bandwidth than on-chip memory
- Device memory accesses should be minimized

CUDA Memory Model

- Shared Memory
 - Is on-chip:
 - much faster than the local and global memory,
 - as fast as a register when no bank conflicts,
 - divided into equally-sized memory banks.
 - Successive 32-bit words are assigned to successive banks,
 - Each bank has a bandwidth of 32 bits per clock cycle.

CUDA Memory Model

- Reduce access to the device memory by staging data into shared memory.

Each thread of a block

- loads data from device to shared memory,
- synchronizes with other threads that each thread can safely read shared memory,
- processes the data in shared memory,
- synchronizes if necessary to make sure shared memory is up to date,
- writes results back to device memory.

CUDA Memory Model

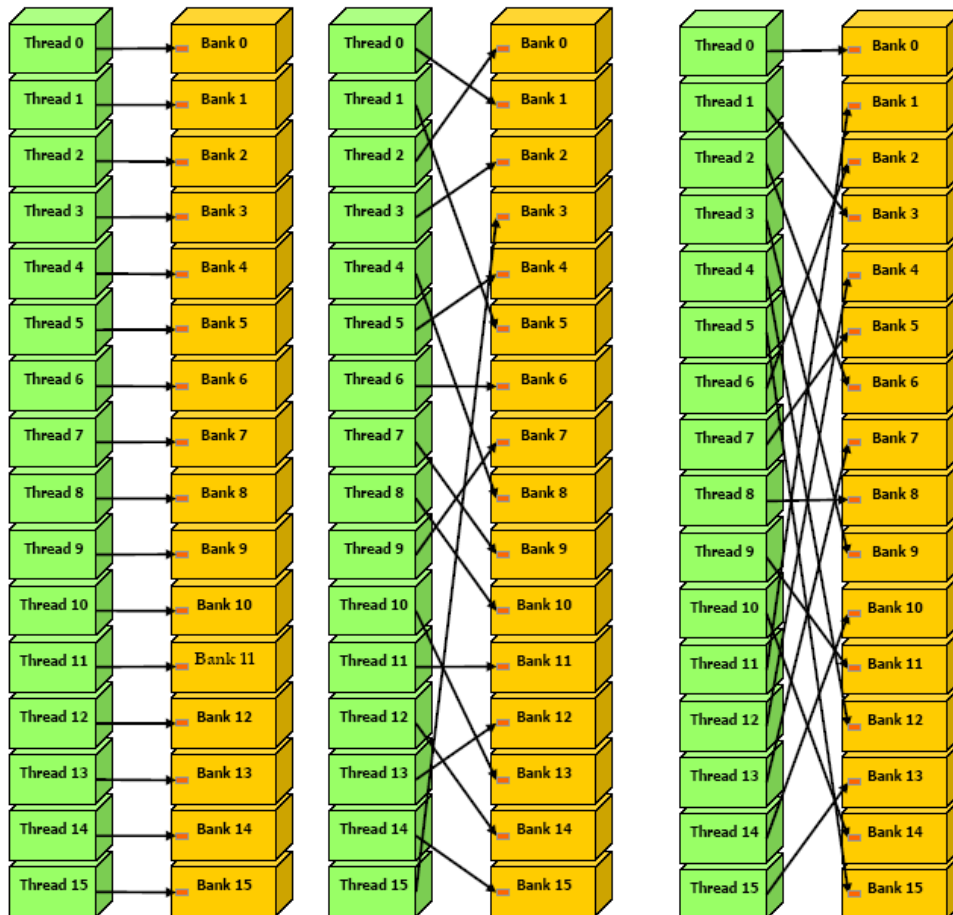
- Shared Memory

Reminder: warp size is 32, number of banks is 16

- memory request requires two cycles for a warp
 - One for the first half, one for the second half of the warp
 - No conflicts between threads from first and second half

CUDA Memory Model

- Shared Memory



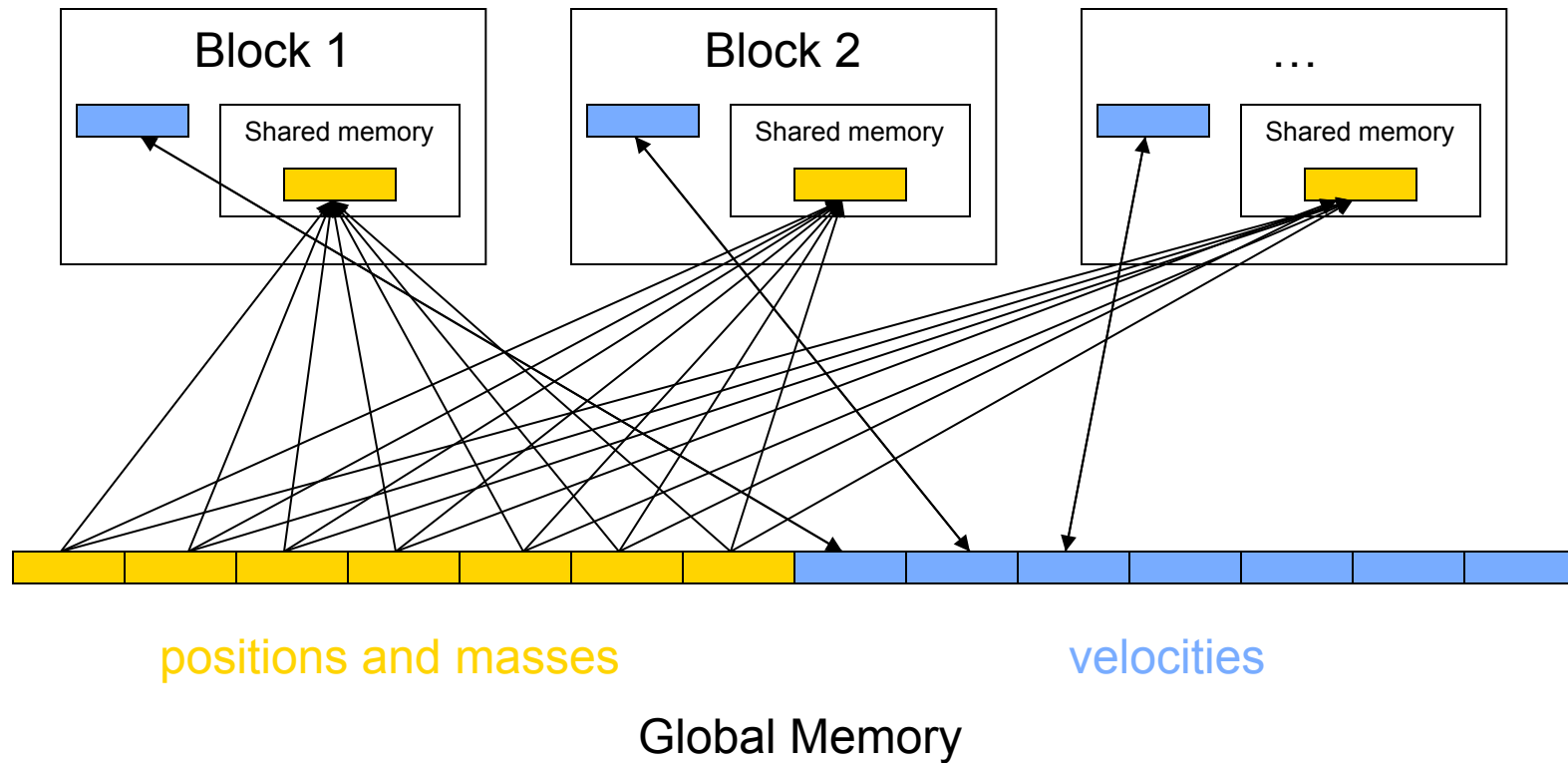
Overview

- Introduction to CUDA
- **Memory Optimizations**
- Loop unrolling revisited
- Conclusion

Case Study: Gravit – a Gravity Simulator

- Each thread calculates the forces on one single particle
 - Simple n^2 algorithm
 - Set of particles can easily be divided into blocks
 - Each block steps through all particles in slices and mirrors them into shared memory
 - No communication needed between blocks
 - Synchronization between threads only needed to guarantee shared memory consistency

Basic Implementation



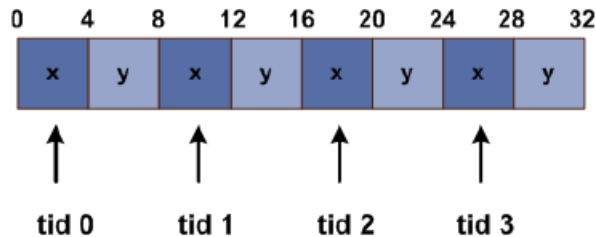
AoS vs SoA
on the G80 architecture

AoS vs SoA

(review)

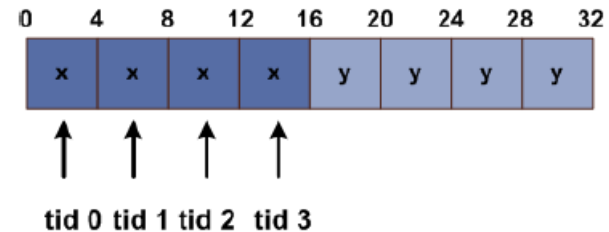
```
struct S{
    float x;
    float y;
};
struct S myData[N]
```

- preventing coalesced reads



```
struct S{
    float x[N];
    float y[N];
};
struct S myData;
```

- Leading to coalesced reads



In this case a SoA seems preferable to an AoS

Why is an AoS preferable on the G80 architecture

- Alignment specifiers and automatically aligned build in types allow for 64 or 128 bit reads from global memory.
- Reduction of number of memory operations.
- By adding an alignment specifier to the SoA from the previous slide and reading from global memory into registers...

```
struct __align__(8) S {  
    float x;  
    float y;  
};
```

... we can improve the performance drastically

- SoA: contiguous reads for x and y (up to 600 cycles)
- AoS: one still contiguous 64bit read to get x and y (up to 300 cycles)
- Even more obvious for 128 bit structures. SoA ~1200 vs AoS ~300 cycles

AoS and shared memory

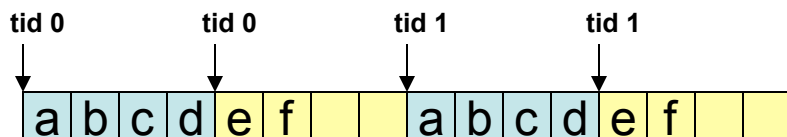
- When reading AoS from global memory always think about the shared memory layout.
 - Shared memory only supports 32 bit reads/writes
 - AoS that allow for good access to global memory will result in bank conflicts in shared memory.
 - Global memory: 64 bit or 128 bit
 - Shared memory: multiples of a stride of 3 → 96bit
- Changing the layout of the data from an AoS in global memory to a SoA in shared memory might be beneficial.

One step further: SoAoS

For structures that exceed the 128bit alignment boundary

```
struct __align__(16) S {  
    float a;  
    float b;  
    float c;  
    float d;  
    float e;  
    float f;  
};
```

```
struct S myData[N];
```



- Each thread will have to perform 2 128 bit reads
- The single reads are no longer contiguous
- Idea: a Structure of Arrays of Structures (SoAoS)

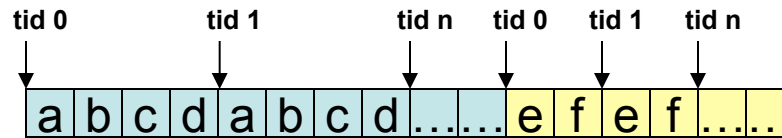
One step further: SoAoS

```
struct __align__(16) S16 {  
    float a;  
    float b;  
    float c;  
    float d;  
};
```

```
struct __align__(8) S8 {  
    float e;  
    float f;  
};
```

```
struct S {  
    struct S16 x[N];  
    struct S8 y[N];  
};
```

```
struct S myData;
```



- The single reads are now again contiguous across threads
- This is just an idea to show that there are many things to try that might lead to better performance for global memory access

Overview

- Introduction to CUDA
- Memory Optimizations
- **Loop unrolling revisited**
- Conclusion

Loop unrolling

- A simple transformation
 - Enabling aggressive instruction scheduling
- Kernels on GPU are not good candidates for unrolling if using CPU standards
 - Inner loops are small
 - Few iterations
 - Small possibilities for instruction reordering

Revisit loop unrolling for GPU

- Patterns of CUDA kernel code
- Thread setup code S that sets up the environment for one single thread.
 - Fetch data needed just by this single thread.
 - Executed once for every thread.
- Thread block setup code B
 - Executed N/K times
- Inner loop P
 - Executed N^2 times

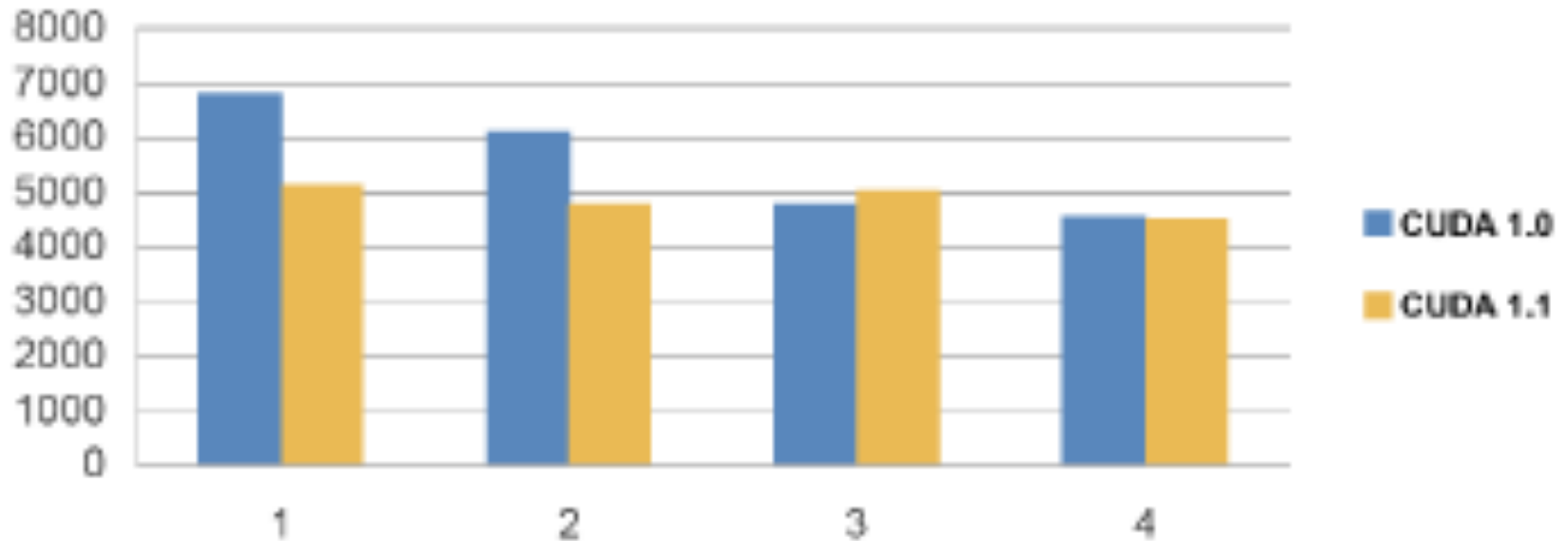
Performance modeling for unrolling

- Assume $S+B+P = 1$
- Potential speedup

$$\text{Speedup} = \frac{N * (S_1 + \frac{N}{K} * B_1 + N * P_1)}{N * (S_2 + \frac{N}{K} * B_2 + N * P_2)} \approx \frac{P_1}{P_2}$$

Results and conclusion

Runtime



Runtime

