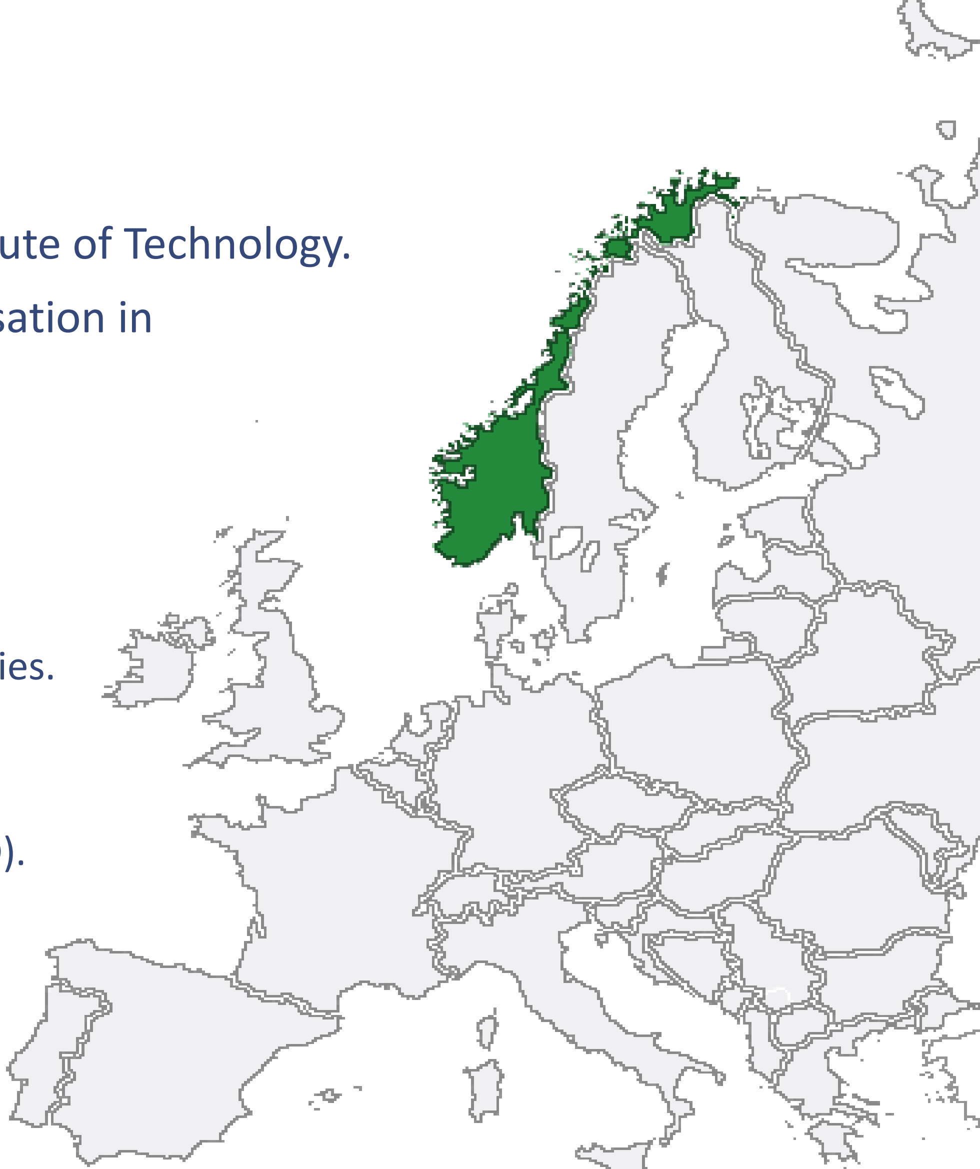
VeRoLog
EURO Working Group
on Vehicle Routing and
Logistics Optimization

VeRoLog 2014

Tutorial:
GPU and Heterogeneous Computing in Discrete Optimization

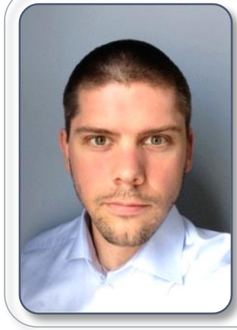
SINTEF

- Established 1950 by the Norwegian Institute of Technology.
- The largest independent research organisation in Scandinavia.
- A non-profit organisation.
- Motto: “Technology for a better society”.
- Key Figures*
  - 2100 Employees from 70 different countries.
  - 73% of employees are researchers.
  - 3 billion NOK in turnover (about 360 million EUR / 490 million USD).
  - 9000 projects for 3000 customers.
  - Offices in Norway, USA, Brazil, Chile, and Denmark.



[Map CC-BY-SA 3.0 based on work by Hayden120 and NuclearVacuum, Wikipedia]

About tutorial



André R. Brodtkorb, PhD
Andre.Brodtkorb@sintef.no

- Worked with GPUs for simulation since 2005
- Interested in scientific computing

Christian Schulz, PhD
Christian.Schulz@sintef.no

- Worked with GPUs in optimization since 2010
- Interested in optimization and geometric modelling

Part of tutorial based on:

GPU Computing in Discrete Optimization

- **Part I: Introduction to the GPU**
  (Brodtkorb, Hagen, Schulz, Hasle)
- **Part II: Survey Focused on Routing Problems**
  (Schulz, Hasle, Brodtkorb, Hagen)

EURO Journal on Transportation and Logistics, 2013.

Schedule

12:45 – 13:00	Coffee
13:00 – 13:45	Part 1: Intro to parallel computing
13:45 – 14:00	Coffee
14:00 – 14:45	Part 2: Programming examples
14:45 – 15:15	Coffee & cake
15:15 – 16:15	Part 3: Profiling and state of the art
16:15 – 16:30	Break
16:30 – 18:30	Get together & registration

VeRoLog
EURO Working Group
on Vehicle Routing and
Logistics Optimization

VeRoLog 2014

Tutorial:
GPU and Heterogeneous Computing in Discrete Optimization

Session 1: Introduction to Parallel Computing

Outline

- Part 1:
  - Motivation for going parallel
  - Multi- and many-core architectures
  - Parallel algorithm design
- Part 2
  - Example: Computing π on the GPU
  - Optimizing memory access

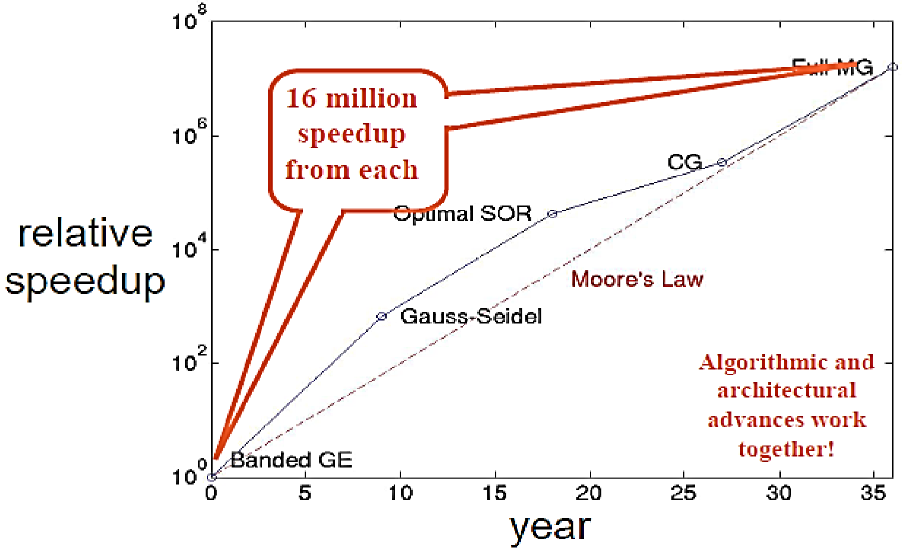
Motivation for going parallel

Why care about computer hardware?

- The key to increasing performance, is to consider the full algorithm and architecture interaction.

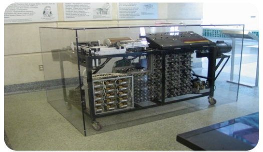
- A good knowledge of both the algorithm and the computer architecture is required.

- Our aim is to equip you with some key insights on how to design algorithms for todays and tomorrows parallel architectures.



Graph from David Keyes, Scientific Discovery through Advanced Computing, Geilo Winter School, 2008

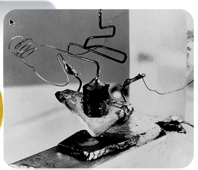
History lesson: development of the microprocessor 1/2



1942: Digital Electric Computer

(Atanasoff and Berry)

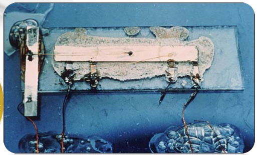
1956



1947: Transistor

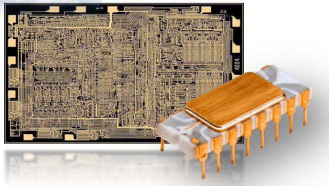
(Shockley, Bardeen, and Brattain)

2000



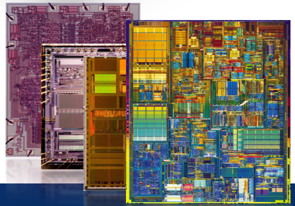
1958: Integrated Circuit

(Kilby)



1971: Microprocessor

(Hoff, Faggin, Mazor)



1971- Exponential growth

(Moore, 1965)

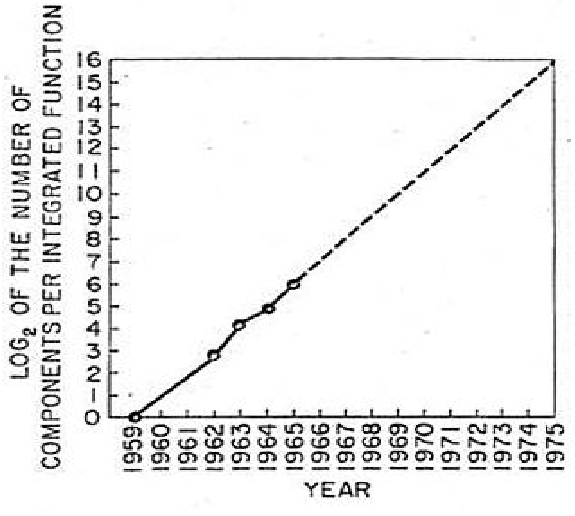
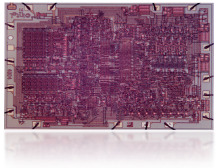


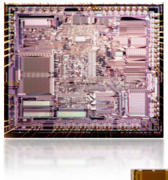
Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

History lesson: development of the microprocessor 2/2



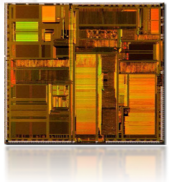
1971: 4004,

2300 trans, 740 KHz



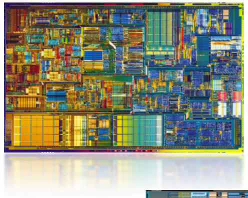
1982: 80286,

134 thousand trans, 8 MHz



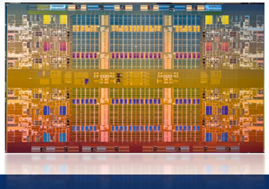
1993: Pentium P5,

1.18 mill. trans, 66 MHz



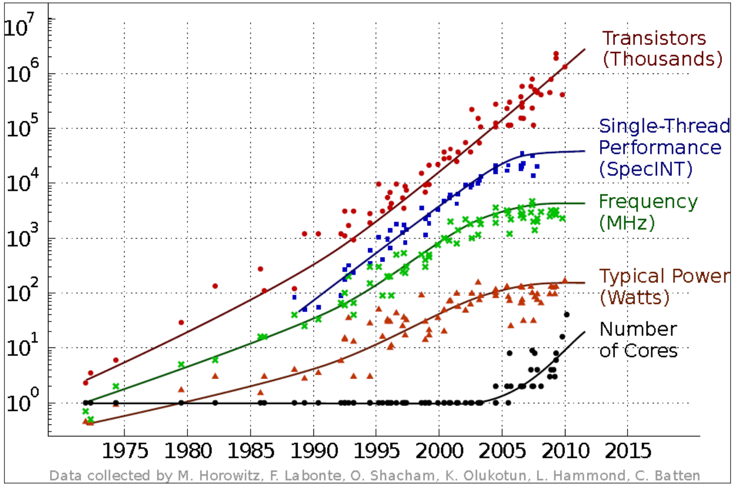
2000: Pentium 4,

42 mill. trans, 1.5 GHz



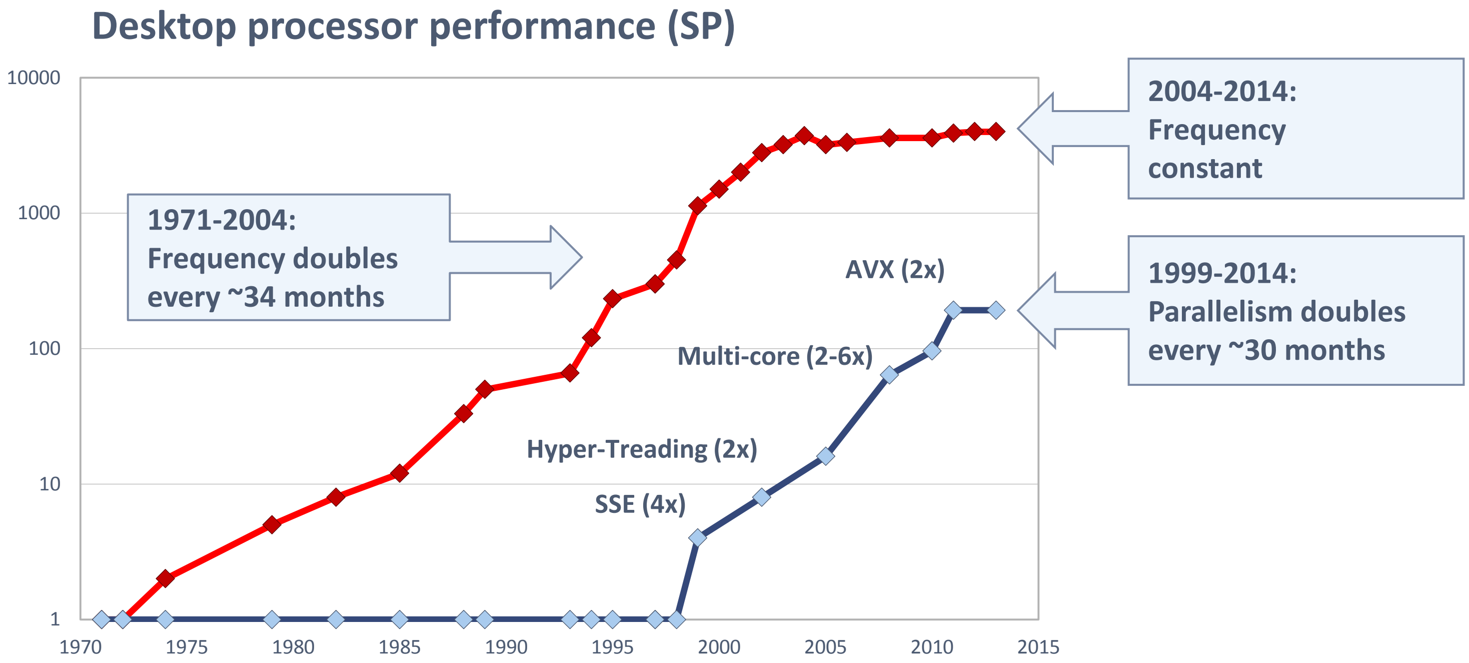
2010: Nehalem

2.3 bill. Trans, 8 cores, 2.66 GHz



Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten

End of frequency scaling



1971-2004: Frequency doubles every ~34 months

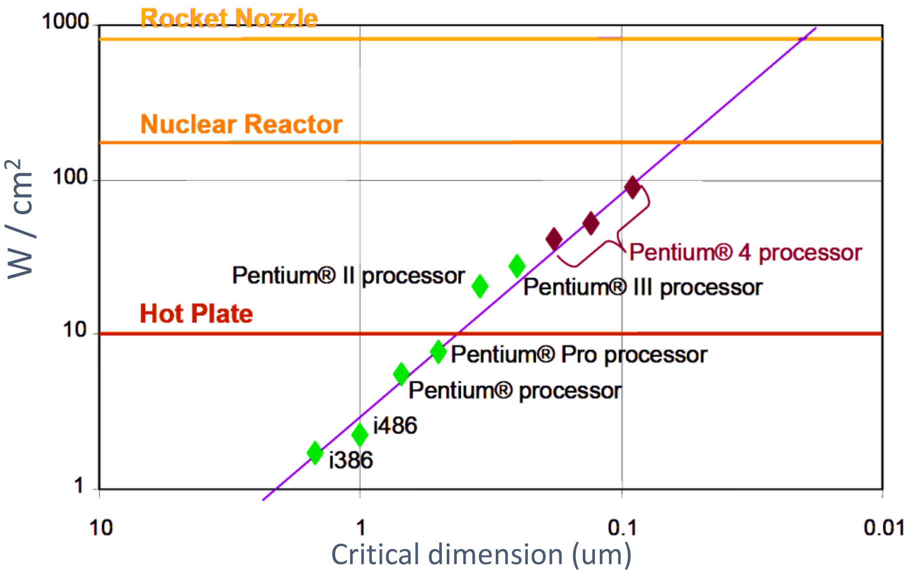
2004-2014: Frequency constant

1999-2014: Parallelism doubles every ~30 months

- 1970-2004: Frequency doubles every 34 months (Moore’s law for performance)
- 1999-2014: Parallelism doubles every 30 months

What happened in 2004?

- Heat density approaching that of nuclear reactor core: Power wall
  - Traditional cooling solutions (heat sink + fan) insufficient
- Industry solution: multi-core and parallelism!

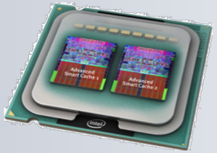


Original graph by G. Taylor, “Energy Efficient Circuit Design and the Future of Power Delivery” EPEPS’09

Why Parallelism?

The power density of microprocessors
is proportional to the clock frequency cubed:[1]

$$P_d \propto f^3$$

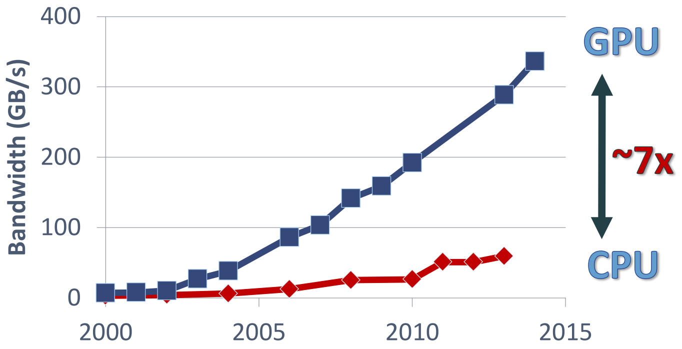
	Single-core	Dual-core
		
Frequency	100%	85%
Performance	100%	90% 90%
Power	100%	100%

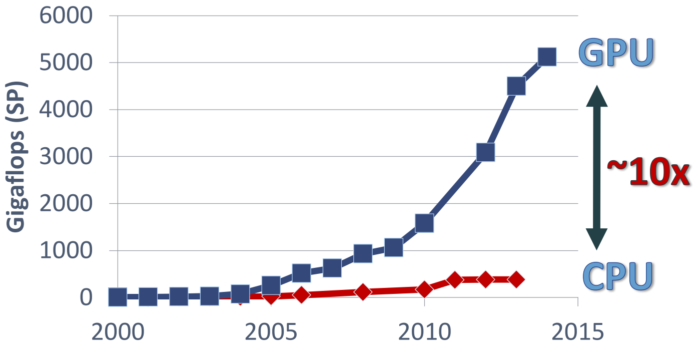
[1] Brodtkorb et al. State-of-the-art in heterogeneous computing, 2010

Massive Parallelism: The Graphics Processing Unit

- Up-to 5760 floating point operations in parallel!

- 5-10 times as power efficient as CPUs!





Multi- and many-core architectures

A taxonomy of parallel architectures

- A taxonomy of different parallelism is useful for discussing parallel architectures
  - 1966 paper by M. J. Flynn: Some Computer Organizations and Their Effectiveness
  - Each class has its own benefits and uses

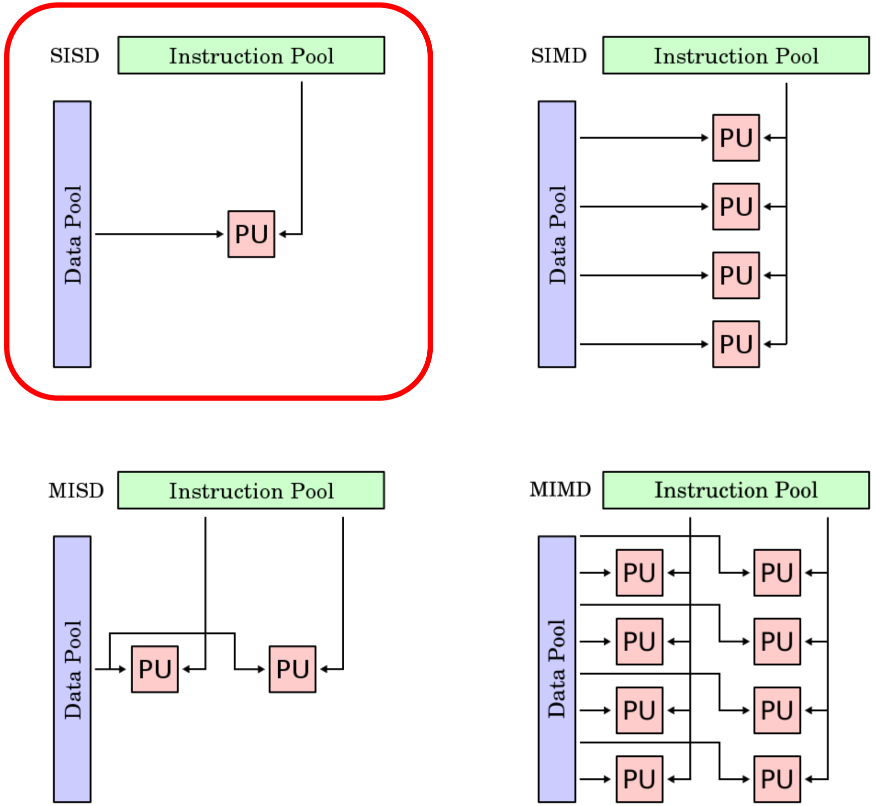
	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instructions	MISD	MIMD

M. J. Flynn, Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., 1966

Single instruction, single data

- Traditional serial mindset:
  - Each instruction is executed after the other
  - One instruction operates on a single element
  - The typical way we write C / C++ computer programs

- Example:
  - c = a + b

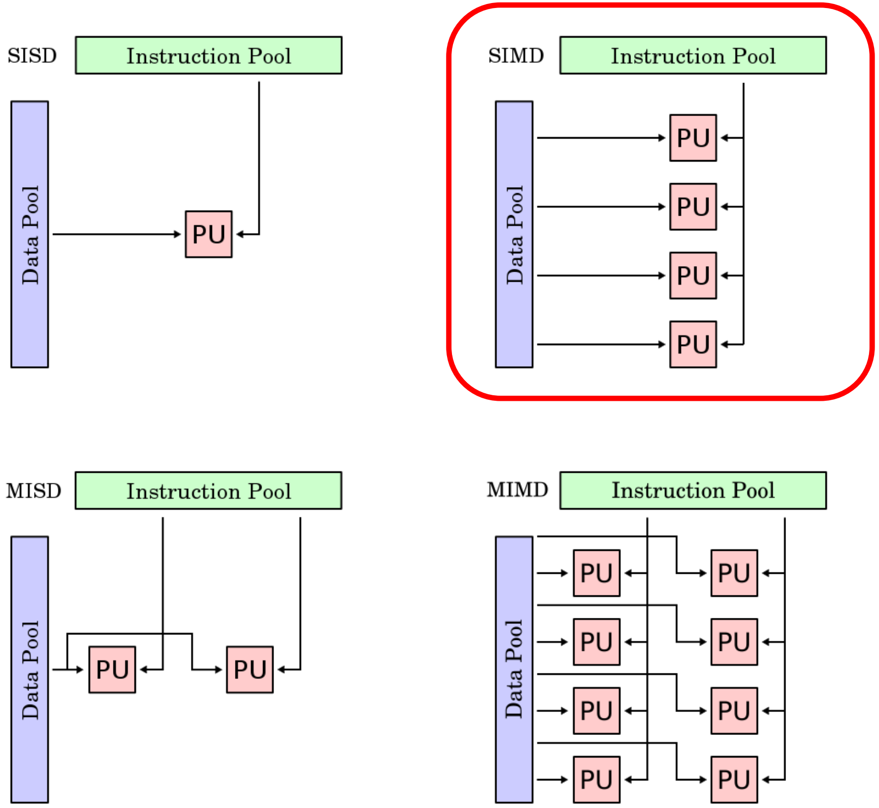


Images from Wikipedia, user Cburnett, CC-BY-SA 3.0

Single instruction, multiple data

- Traditional vector mindset:
  - Each instruction is executed after the other
  - Each instruction operates on multiple data elements simultaneously
  - The way vectorized MATLAB programs often are written

- Example:
  - c[i] = a[i] + b[i]   i=0…N
  - a, b, and c are vectors of fixed length (typically 2, 4, 8, 16, or 32)

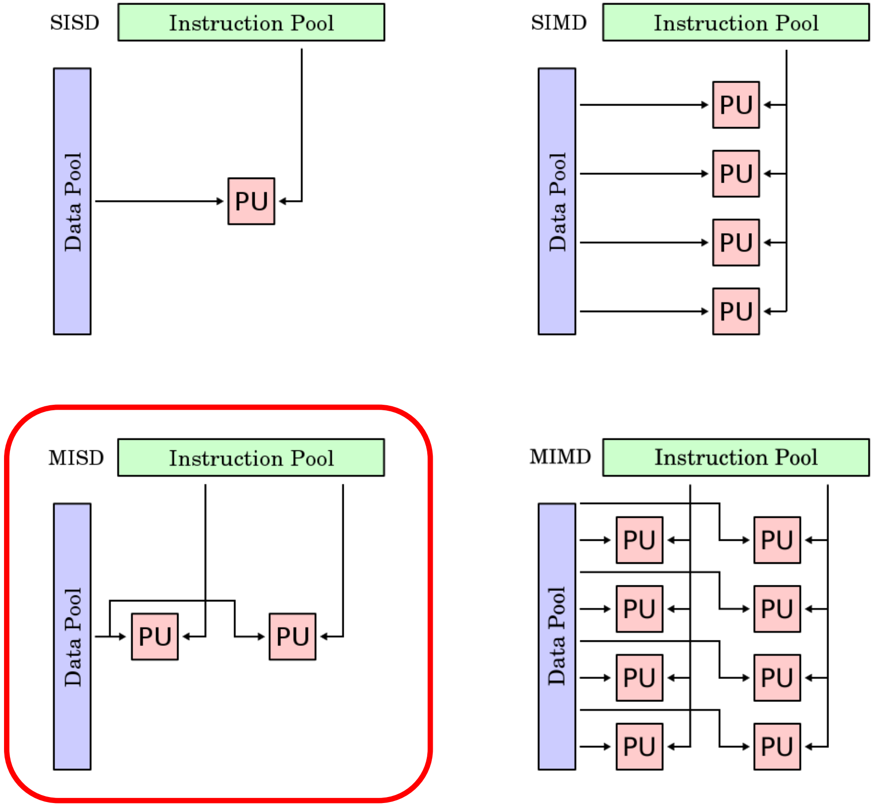


Images from Wikipedia, user Cburnett, CC-BY-SA 3.0

Multiple instruction, single data

- Only for special cases:
  - Multiple instructions are executed simultaneously
  - Each instruction operates on a single data element
  - Used e.g., for fault tolerance, or pipelined algorithms implemented on FPGAs

- Example (naive detection of catastrophic cancellation):
  - PU1: z1 = x*x – y*y
    PU2: z2 = (x-y) * (x+y)
    if (z1 – z2 > eps) { ... }

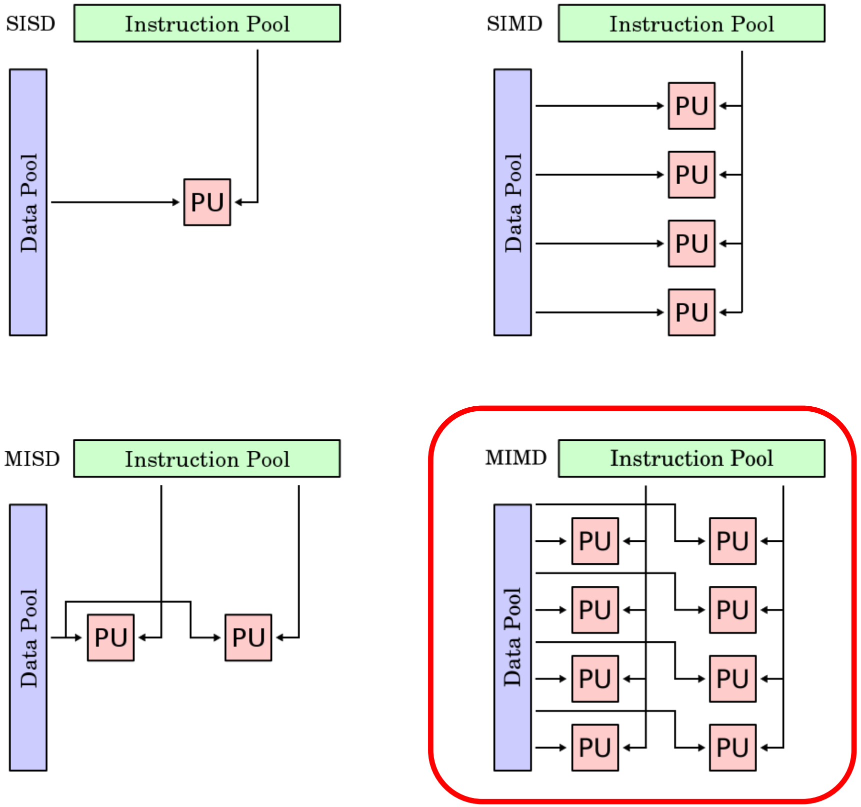


Images from Wikipedia, user Cburnett, CC-BY-SA 3.0

Multiple instruction, multiple data

- Traditional cluster computer
  - Multiple instructions are executed simultaneously
  - Each instruction operates on multiple data elements simultaneously
  - Typical execution pattern used in task-parallel computing

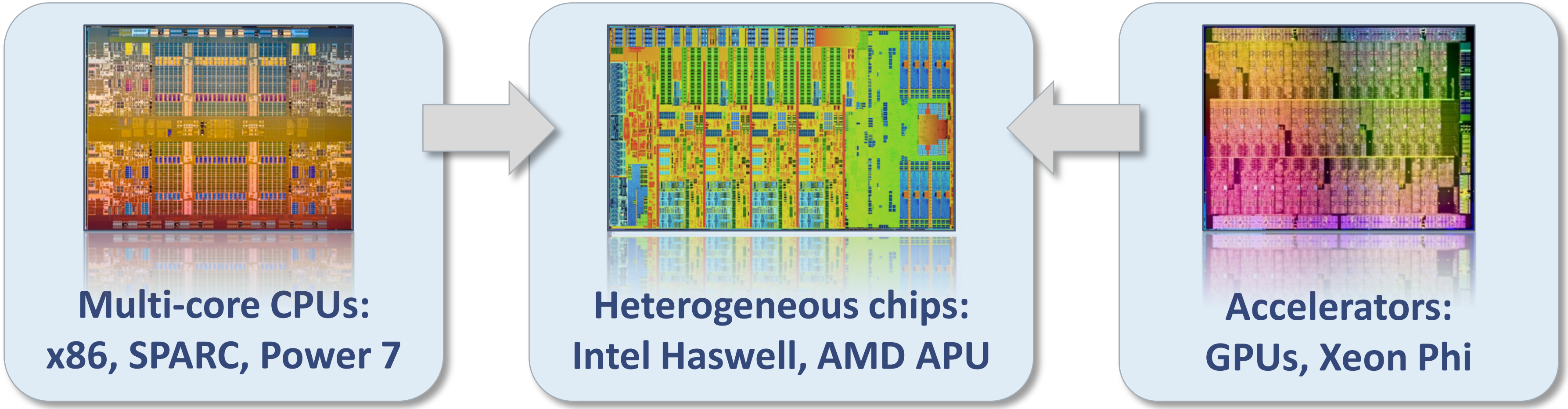
- Example:
  - PU1: c = a + b
    PU2: z = (x-y) * (x+y)
    variables can also vectors of fixed length (se SIMD)



Images from Wikipedia, user Cburnett, CC-BY-SA 3.0

Multi- and many-core processor designs

- 6-60 processors per chip
- 8 to 32-wide SIMD instructions
- Combines both SISD, SIMD, and MIMD on a single chip
- Heterogeneous cores (e.g., CPU+GPU on single chip)



Multi-core CPUs:
x86, SPARC, Power 7

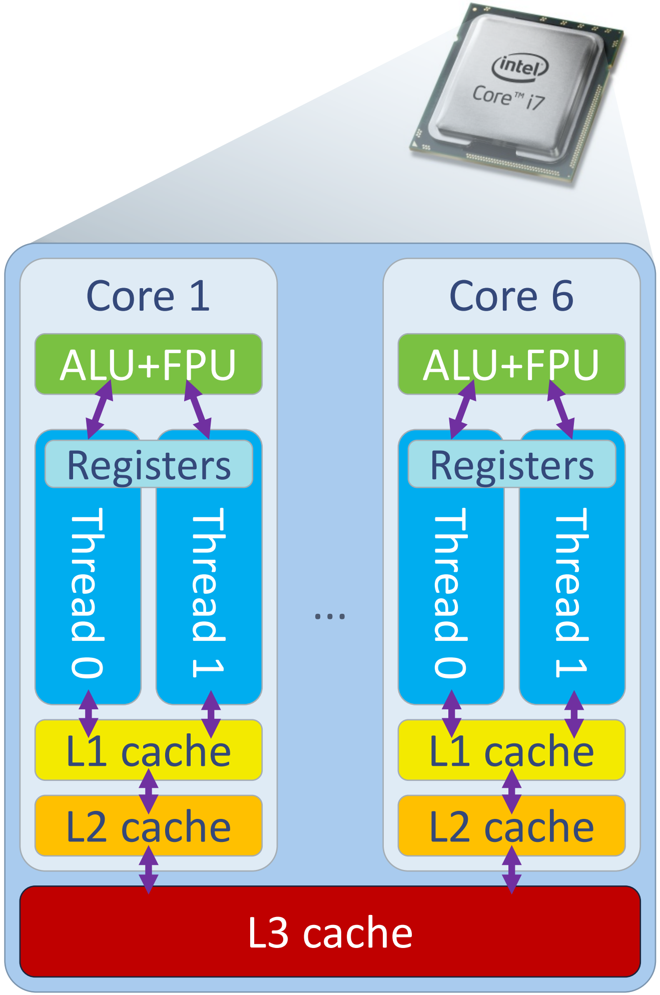
Heterogeneous chips:
Intel Haswell, AMD APU

Accelerators:
GPUs, Xeon Phi

Multi-core CPU architecture

- A single core
  - L1 and L2 caches
  - 8-wide SIMD units (AVX, single precision)
  - 2-way Hyper-threading (<u>hardware</u> threads)
    When thread 0 is waiting for data,
    thread 1 is given access to SIMD units
  - Most transistors used for cache and logic

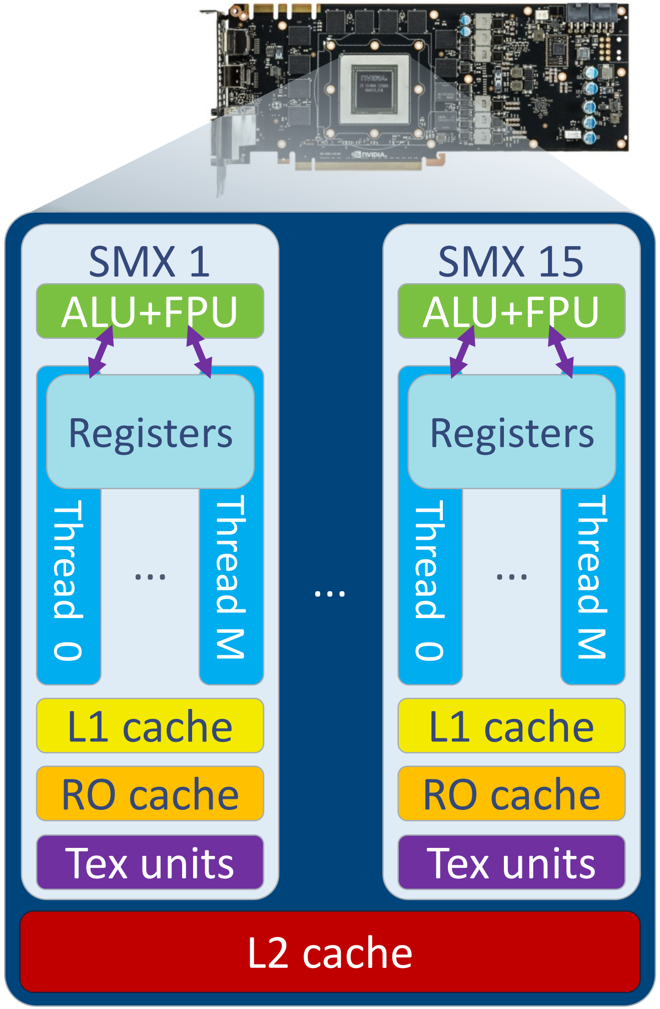
- Optimal number of FLOPS per clock cycle:
  - 8x: 8-way SIMD
  - 6x: 6 cores
  - 2x: Dual issue (fused mul-add / two ports)
  - <u>Sum: 96!</u>



Simplified schematic of CPU design

Many-core GPU architecture

- A single core (Called streaming multiprocessor, SMX)
  - L1 cache, Read only cache, texture units
  - Six 32-wide SIMD units (192 total, single precision)
  - Up-to 64 warps simultaneously (hardware warps)
    Like hyper-threading, but a warp is 32-wide SIMD
  - Most transistors used for floating point operations
- Optimal number of FLOPS per clock cycle:
  - 32x: 32-way SIMD
  - 2x: Fused multiply add
  - 6x: Six SIMD units per core
  - 15x: 15 cores
  - Sum: 5760!

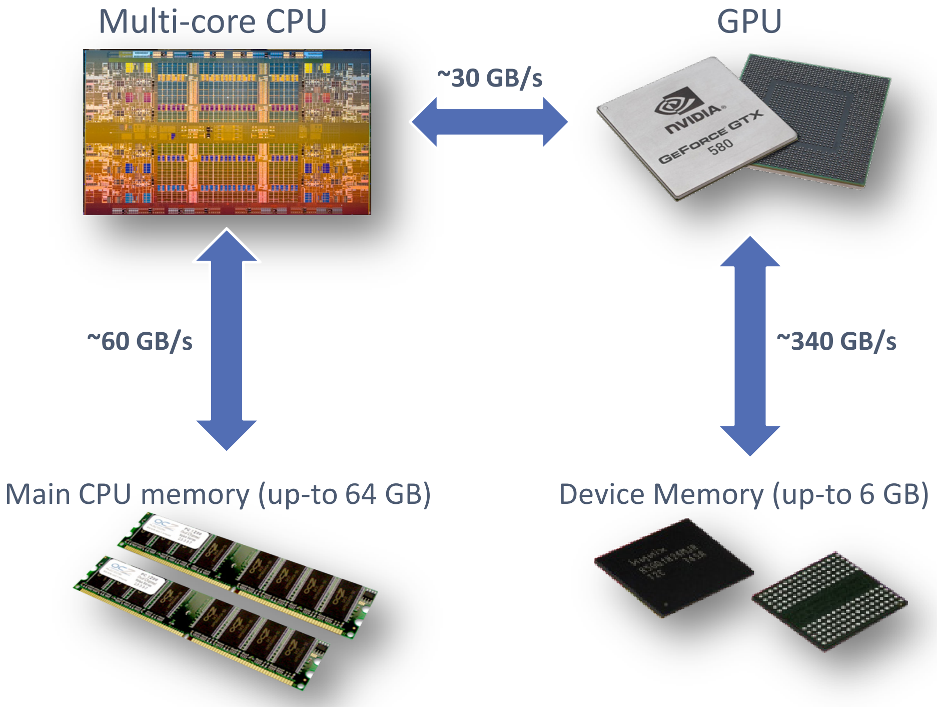


Simplified schematic of GPU design

Heterogeneous Architectures

- Discrete GPUs are connected to the CPU via the PCI-express bus
  - Slow: 15.75 GB/s each direction
  - On-chip GPUs use main memory as graphics memory

- Device memory is limited but fast
  - Typically up-to 6 GB
  - Up-to 340 GB/s!
  - Fixed size, and cannot be expanded with new dimm’s (like CPUs)



Multi-core CPU

GPU

~30 GB/s

~60 GB/s

~340 GB/s

Main CPU memory (up-to 64 GB)

Device Memory (up-to 6 GB)

Parallel algorithm design

Parallel computing

- Most algorithms are like baking recipies, Tailored for a single person / processor:
  - First, do A,
  - Then do B,
  - Continue with C,
  - And finally complete by doing D.

- How can we utilize an army of chefs?
  - Let's look at one example: computing Pi

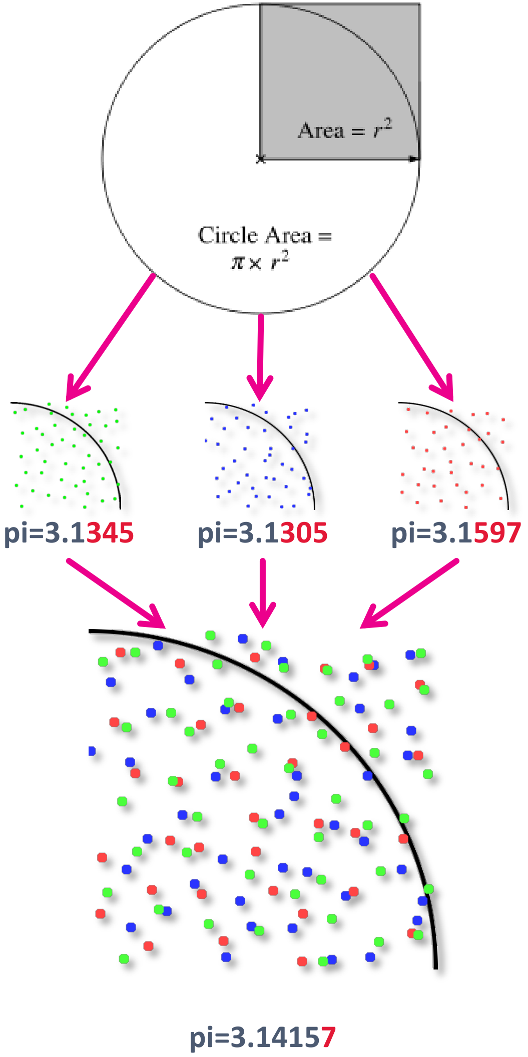
Picture: Daily Mail Reporter , www.dailymail.co.uk

Estimating π (3.14159...) in parallel

- There are many ways of estimating Pi. One way is to estimate the area of a circle.

- Sample random points within one quadrant
- Find the ratio of points inside to outside the circle
    - Area of quarter circle: $A_c = \pi r^2/4$
      Area of square: $A_s = r^2$
    - $\pi = 4\ A_c/A_s \approx 4$ #points inside / #points outside
- Increase accuracy by sampling more points
- Increase speed by using more nodes

- This can be referred to as a data-parallel workload: All processors perform the same operation.



Disclaimer: this is a naïve way of calculating PI, only used as an example of parallel execution

Data parallel workloads

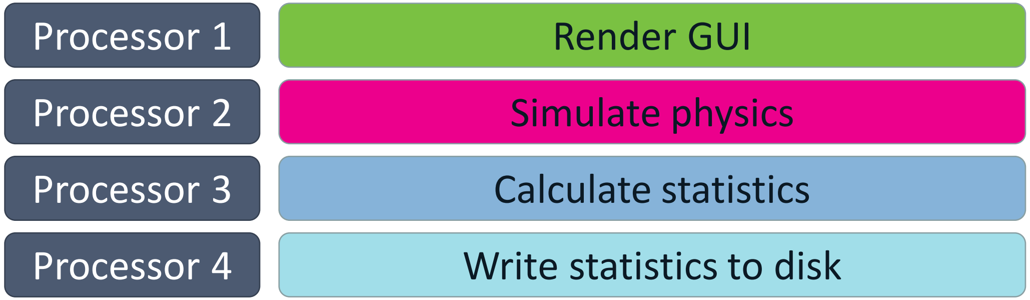
- Data parallelism performs the same operation for a set of different input data
- Scales well with the data size:
  The larger the problem, the more processors you can utilize
- Trivial example:
  Element-wise multiplication of two vectors:
  - c[i] = a[i] * b[i]   i=0…N
  - Processor i multiplies elements i of vectors a and b.

Task parallel workloads 1/3

- Task parallelism divides a problem into subtasks which can be solved individually

- Scales well for a large number of tasks:
  The more parallel tasks, the more processors you can use

- Example: A simulation application:



Processor 1	Render GUI
Processor 2	Simulate physics
Processor 3	Calculate statistics
Processor 4	Write statistics to disk

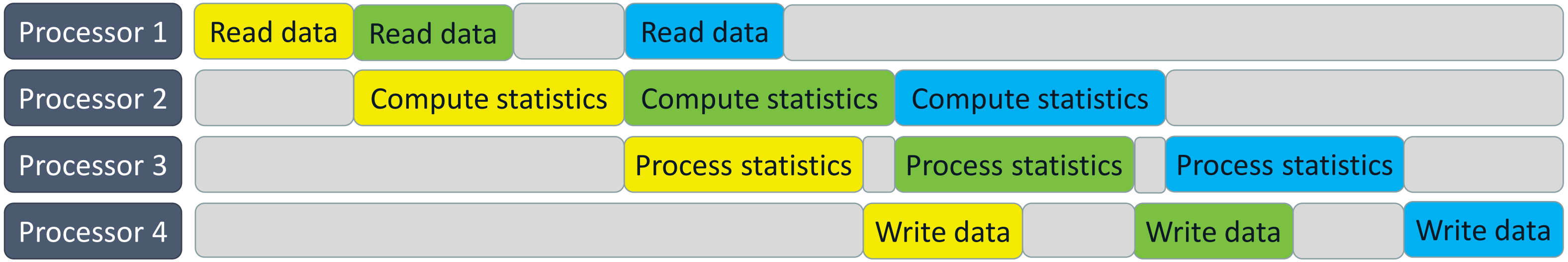
- Note that not all tasks will be able to fully utilize the processor

Task parallel workloads 2/3

- Another way of using task parallelism is
  to execute dependent tasks on different processors

- Scales well with a large number of tasks, but performance limited by slowest stage

- Example: Pipelining dependent operations



Processor 1 Read data Read data Read data

Processor 2 Compute statistics Compute statistics Compute statistics

Processor 3 Process statistics Process statistics Process statistics

Processor 4 Write data Write data Write data

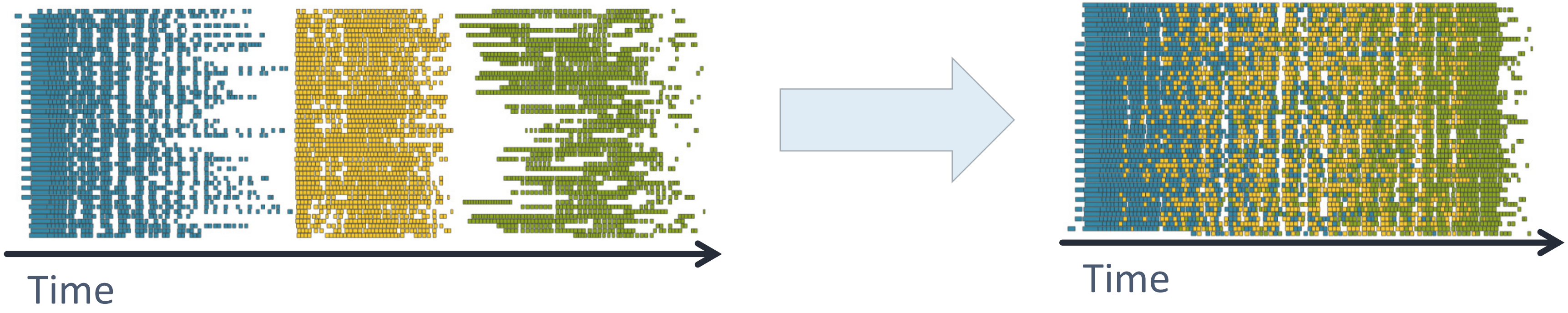
- Note that the gray boxes represent idling: wasted clock cycles!

Task parallel workloads 3/3

- A third way of using task parallelism is
  to represent tasks in a directed acyclic graph (DAG)

- Scales well for millions of tasks, as long as the overhead of executing each task is low

- Example: Cholesky inversion



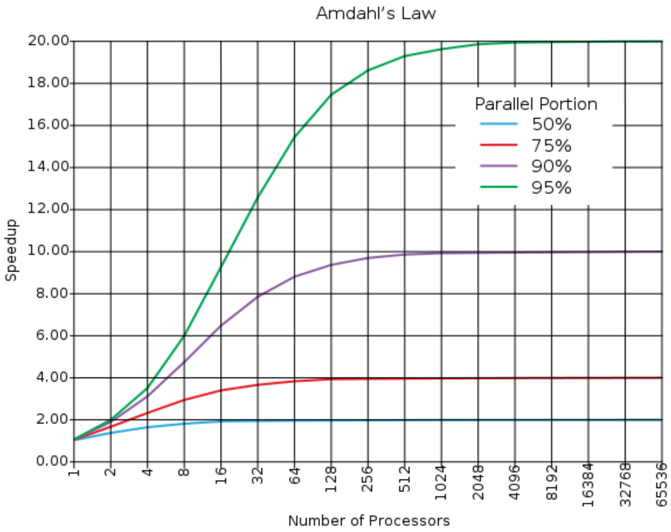
- “Gray boxes” are minimized

Example from Dongarra, On the Future of High Performance Computing: How to Think for Peta and Exascale Computing, 2012

Limits on performance 1/4

- Most algorithms contains a mixture of work-loads:
  - Some serial parts
  - Some task and / or data parallel parts

- <u>Amdahl's law:</u>
  - There is a limit to speedup offered by parallelism
  - Serial parts become the bottleneck for a massively parallel architecture!
  - Example: 5% of code is serial: maximum speedup is 20 times!



$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

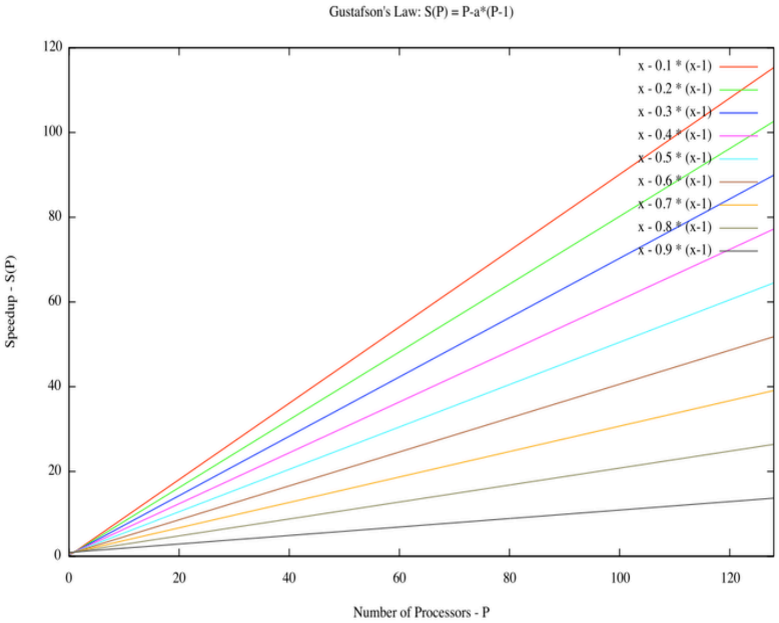
S: Speedup
P: Parallel portion of code
N: Number of processors

Graph from Wikipedia, user Daniels220, CC-BY-SA 3.0

Graph from Wikipedia: Amdahls law

Limits on performance 2/4

- Gustafson's law:
  - If you cannot reduce serial parts of algorithm, make the parallel portion dominate the execution time
  - Essentially: solve a bigger problem!



$$S(P) = P - \alpha \cdot (P - 1).$$

S: Speedup
P: Number of processors
α: Serial portion of code

Graph from Wikipedia, user Peahihawaii, CC-BY-SA 3.0

Limits on performance 3/4

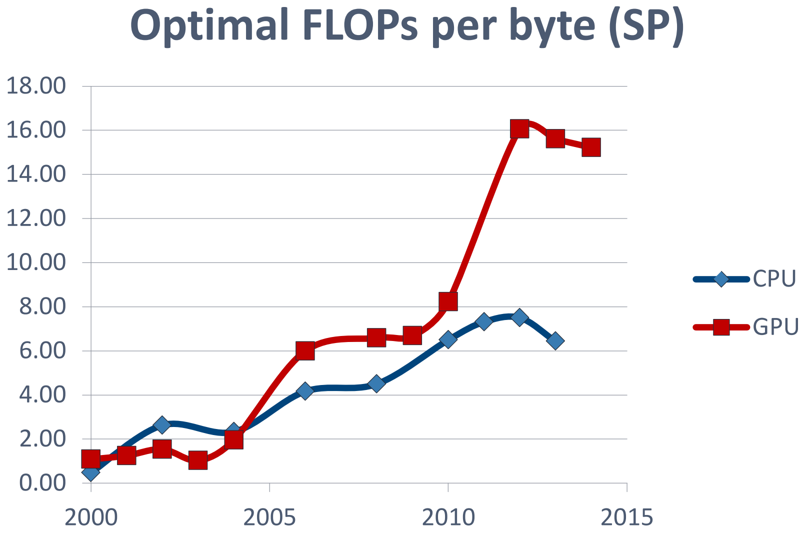
- Moving data has become the major bottleneck in computing.
- Downloading 1GB from Japan to Switzerland consumes roughly the energy of 1 charcoal briquette[1].
- A FLOP costs less than moving one byte[2].
- Key insight: flops are free, moving data is expensive

[1] Energy content charcoal: 10 MJ / kg, kWh per GB: 0.2 (Coroama et al., 2013), Weight charcoal briquette: ~25 grams

[2] Simon Horst, Why we need Exascale, and why we won't get there by 2020, 2014

Limits on performance 4/4

- A single precision number is four bytes
  - You must perform over 60 operations for each float read on a GPU!
  - Over 25 operations on a CPU!
- This groups algorithms into two classes:
  - Memory bound
    Example: Matrix multiplication
  - Compute bound
    Example: Computing π
- The third limiting factor is latencies
  - Waiting for data
  - Waiting for floating point units
  - Waiting for ...

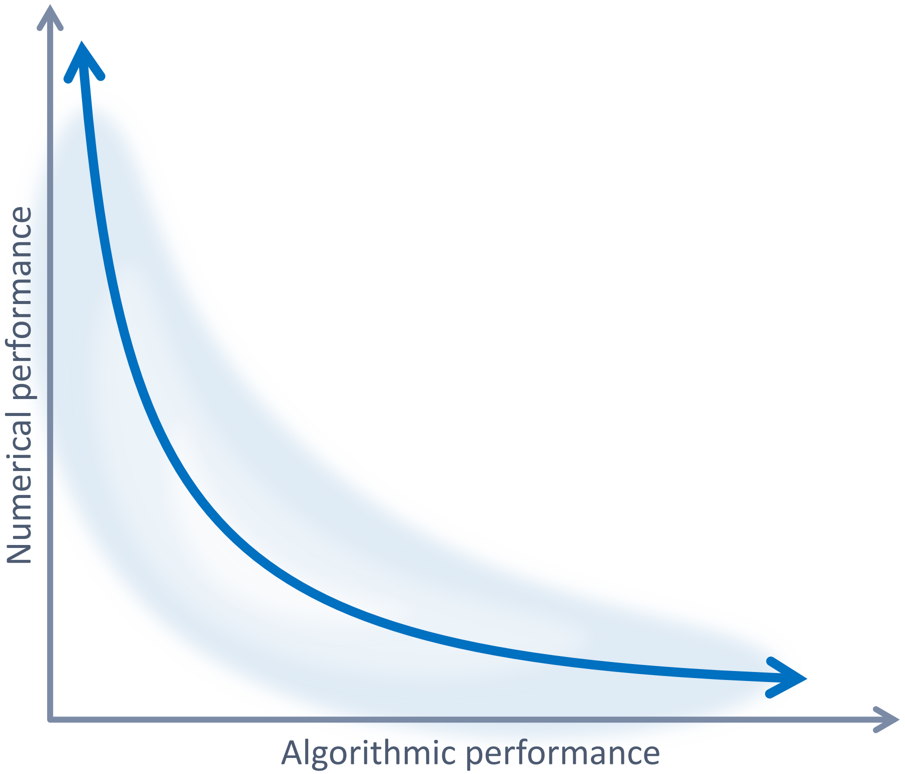


Algorithmic and numerical performance

- Total performance is the product of algorithmic **and** numerical performance
  - Your mileage may vary: algorithmic performance is highly problem dependent

- Many algorithms have low numerical performance
  - Only able to utilize a fraction of the capabilities of processors, and often **worse in parallel**

- Need to consider both the algorithm and the architecture for maximum performance

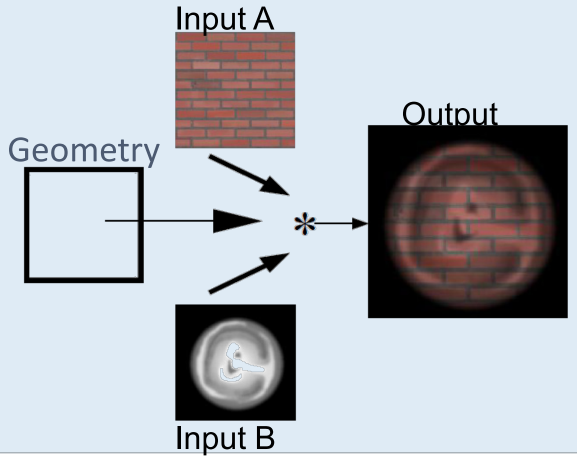


Programming GPUs

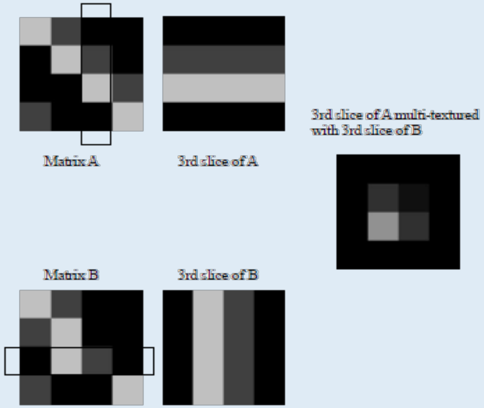
Early Programming of GPUs

- GPUs were first programmed using OpenGL and other graphics languages
  - Mathematics were written as operations on graphical primitives
  - Extremely cumbersome and error prone
  - Showed that the GPU was capable of outperforming the CPU

Element-wise matrix multiplication

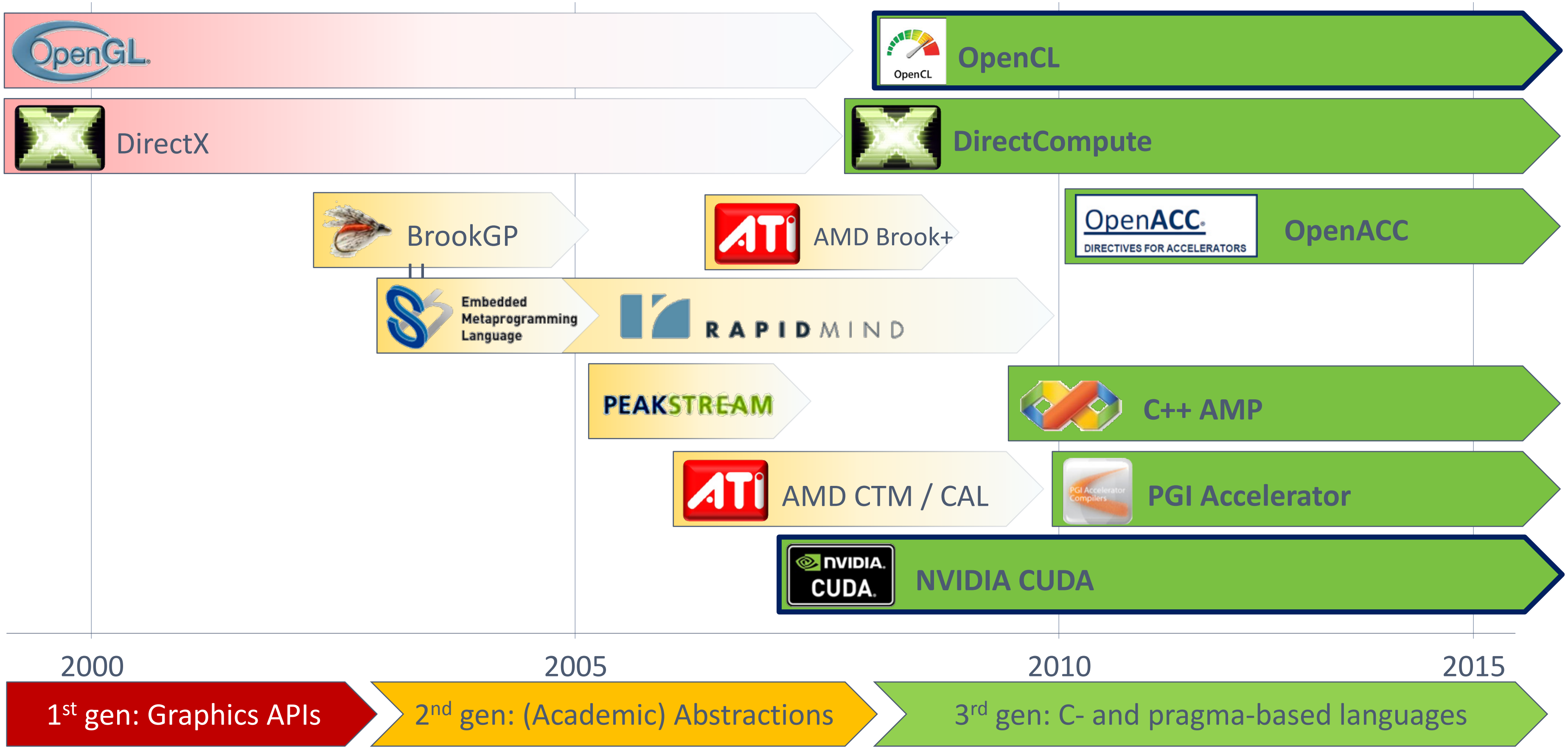


Matrix multiplication



[1] Fast matrix multiplies using graphics hardware, Larsen and McAllister, 2001

GPU Programming Languages



OpenGL

DirectX

OpenCL

DirectCompute

BrookGP

AMD Brook+

OpenACC

Embedded Metaprogramming Language

RAPIDMIND

PEAKSTREAM

C++ AMP

AMD CTM / CAL

PGI Accelerator

NVIDIA CUDA

2000 2005 2010 2015

1st gen: Graphics APIs

2nd gen: (Academic) Abstractions

3rd gen: C- and pragma-based languages

Computing with CUDA

- We will focus on CUDA, as it has the most mature development ecosystem
  - Released by NVIDIA in 2007
  - Enables programming GPUs using a C-like language
  - Essentially C / C++ with some additional syntax for executing a function in parallel on the GPU

- OpenCL is a very good alternative that also runs on non-NVIDIA hardware (Intel Xeon Phi, AMD GPUs, CPUs)
  - Equivalent to CUDA, but slightly more cumbersome.

- For high-level development, languages like OpenACC (pragma based) or C++ AMP (extension to C++) exist
  - Typicall works well for toy problems, and not so well for complex algorithms

Example: Adding two matrices in CUDA 1/2

- We want to add two matrices,
  a and b, and store the result in c.

$$\begin{bmatrix}1 & 3\\ 1 & 0\\ 1 & 2\end{bmatrix} + \begin{bmatrix}0 & 0\\ 7 & 5\\ 2 & 1\end{bmatrix} = \begin{bmatrix}1+0 & 3+0\\ 1+7 & 0+5\\ 1+2 & 2+1\end{bmatrix} = \begin{bmatrix}1 & 3\\ 8 & 5\\ 3 & 3\end{bmatrix}$$

- For best performance, loop through one row at a time
  (sequential memory access pattern)

```
void addFunctionCPU(float* c, float* a, float* b,
                    unsigned int cols, unsigned int rows) {
     for (unsigned int j=0; j<rows; ++j) {
          for (unsigned int i=0; i<cols; ++i) {
               unsigned int k = j*cols + i;
               c[k] = a[k] + b[k];
          }
     }
}
```

C++ on CPU

Matrix from Wikipedia: Matrix addition

Example: Adding two matrices in CUDA 2/2

GPU function

```
__global__ void addMatricesKernel(float* c, float* a, float* b,
                                unsigned int cols, unsigned int rows) {
     //Indexing calculations
     unsigned int global_x = blockIdx.x*blockDim.x + threadIdx.x;
     unsigned int global_y = blockIdx.y*blockDim.y + threadIdx.y;
     unsigned int k = global_y*cols + global_x;

     //Actual addition
     c[k] = a[k] + b[k];
}
```

Indices

Implicit double for loop

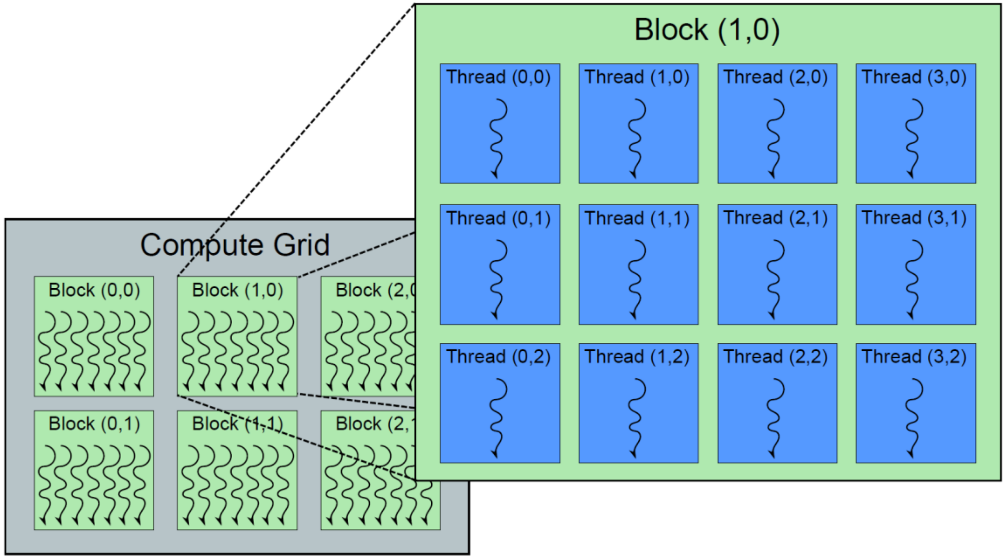
for (int blockIdx.x = 0;
         blockIdx.x < grid.x;
         blockIdx.x) { …

```
void addFunctionGPU(float* c, float* a, float* b,
             unsigned int cols, unsigned int rows) {
     dim3 block(8, 8);
     dim3 grid(cols/8, rows/8);
     ... //More code here: Allocate data on GPU, copy CPU data to GPU
     addMatricesKernel<<<grid, block>>>(gpu_c, gpu_a, gpu_b, cols, rows);
     ...  //More code here: Download result from GPU to CPU
}
```

Run on GPU

Grids and blocks in CUDA

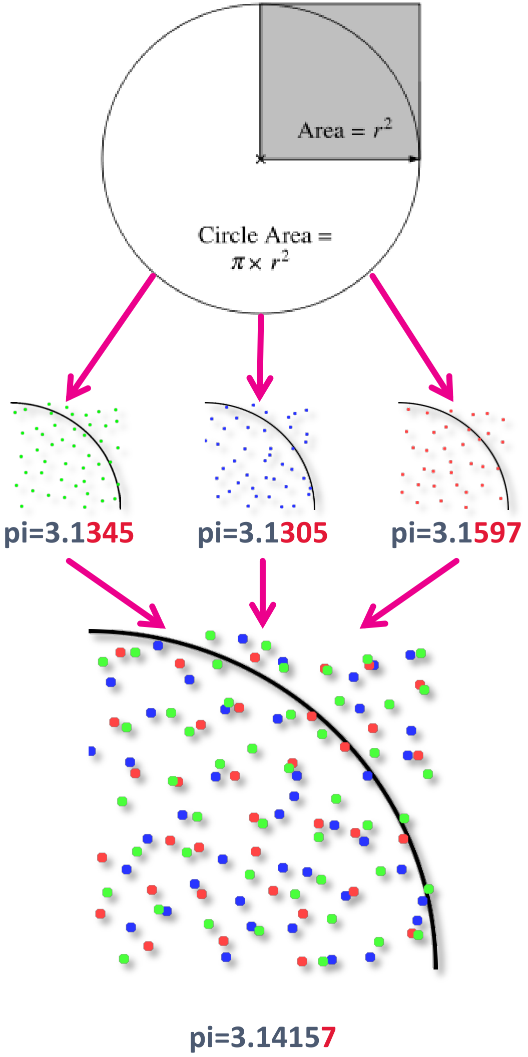
- Two-layered parallelism
  - A block consists of threads: Threads within the same block can cooperate and communicate
  - A grid consists of blocks: All blocks run independently.
  - Blocks and grid can be 1D, 2D, and 3D
- Global synchronization and communication is only possible between kernel launches
  - Really expensive, and should be avoided if possible



Example: Computing π with CUDA

- Algorithm:
  1. Sample random points within a quadrant
  2. Compute distance from point to origin
  3. If distance less than r, point is inside circle
  4. Estimate π as 4 #points inside / #points outside

- Remember: The algorithms serves as an example: it's far more efficient to estimate π as 22/7, or 355/113☺



Serial CPU code (C/C++)

```
float computePi(int n_points) {
        int n_inside = 0;
        for (int i=0; i<n_points; ++i) {
                //Generate coordinate
                float x = generateRandomNumber();                         1
                float y = generateRandomNumber();
                //Compute distance
                float r = sqrt(x*x + y*y);
                //Check if within circle                                  2 & 3
                if (r < 1.0f) { ++n_inside; }
        }
        //Estimate Pi
        float pi = 4.0f * n_inside / static_cast<float>(n_points);       4
        return pi;
}
```

Parallel CPU code (C/C++ with OpenMP)

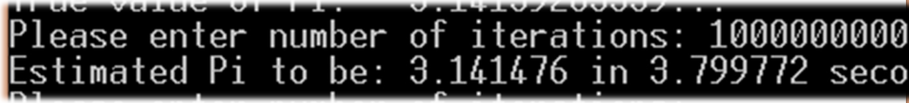
```
float computePi(int n_points) {
        int n_inside = 0;
        #pragma omp parallel for reduction(+:n_inside)
        for (int i=0; i<n_points; ++i) {
              //Generate coordinate
              float x = generateRandomNumber();
              float y = generateRandomNumber();
              //Compute distance
              float r = sqrt(x*x + y*y);
              //Check if within circle
              if (r <= 1.0f) { ++n_inside; }
        }
        //Estimate Pi
        float pi = 4.0f * n_inside / static_cast<float>(n_points);
        return pi;
}
```

Run for loop in parallel using multiple threads

Make sure that every expression involving **n_inside** modifies the global variable using the + operator

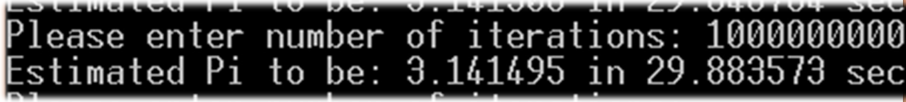
Performance

- Parallel: 3.8 seconds @ 1/1 performance

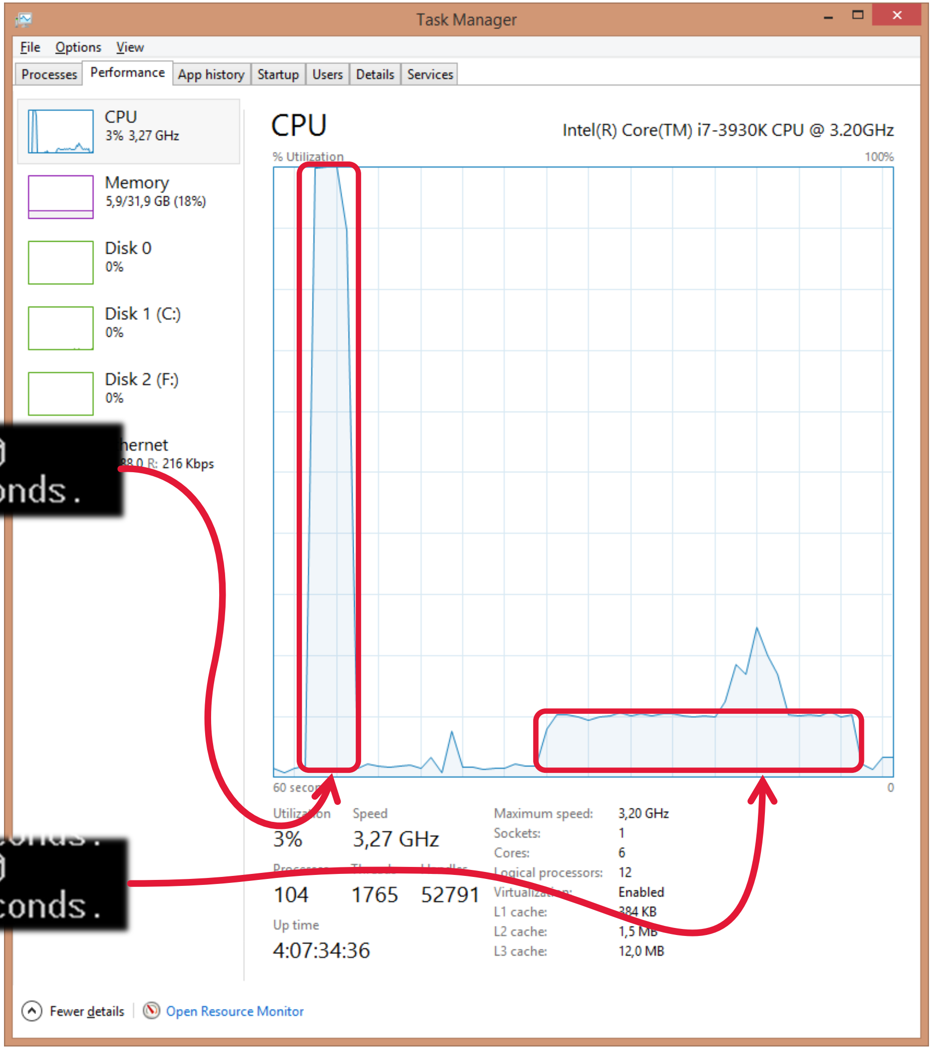


```
Please enter number of iterations: 1000000000
Estimated Pi to be: 3.141476 in 3.799772 seconds.
```

- Serial: 30 seconds @ 1/12 performance



```
Please enter number of iterations: 1000000000
Estimated Pi to be: 3.141495 in 29.883573 seconds.
```



Task Manager

File Options View

Processes | Performance | App history | Startup | Users | Details | Services

CPU 3% 3,27 GHz

Memory 5,9/31,9 GB (18%)

Disk 0 0%

Disk 1 (C:) 0%

Disk 2 (F:) 0%

Ethernet S: 88.0 R: 216 Kbps

CPU Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz

Utilization	3%
Speed	3,27 GHz
Processes	104
Threads	1765
Handles	52791
Up time	4:07:34:36

Maximum speed:	3,20 GHz
Sockets:	1
Cores:	6
Logical processors:	12
Virtualization:	Enabled
L1 cache:	384 KB
L2 cache:	1,5 MB
L3 cache:	12,0 MB

Fewer details | Open Resource Monitor

Parallel GPU version 1 (CUDA) 1/3

GPU function

```
__global__ void computePiKernel1(unsigned int* output) {
        //Generate coordinate
        float x = generateRandomNumber();
        float y = generateRandomNumber();

        //Compute radius
        float r = sqrt(x*x + y*y);

        //Check if within circle
        if (r <= 1.0f) {
                output[blockIdx.x] = 1;
        } else {
                output[blockIdx.x] = 0;
        }
}
```

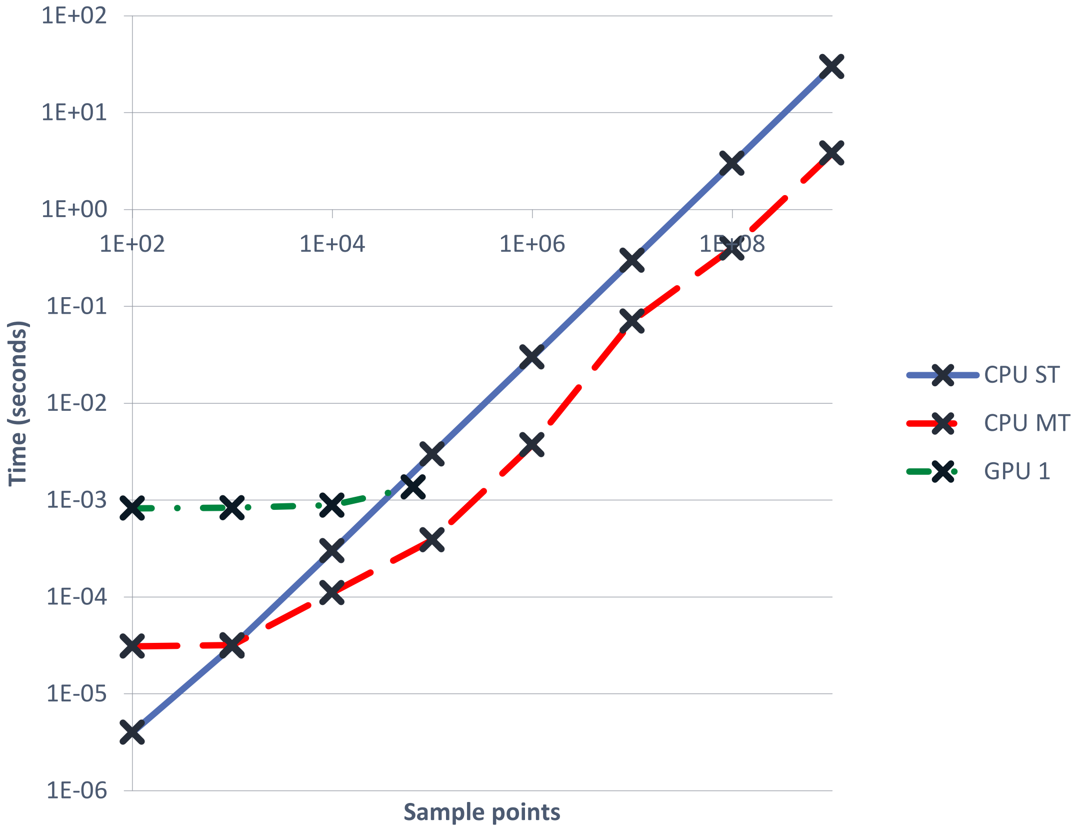
*Random numbers on GPUs can be a slightly tricky, see cuRAND for more information

Parallel GPU version 1 (CUDA) 2/3

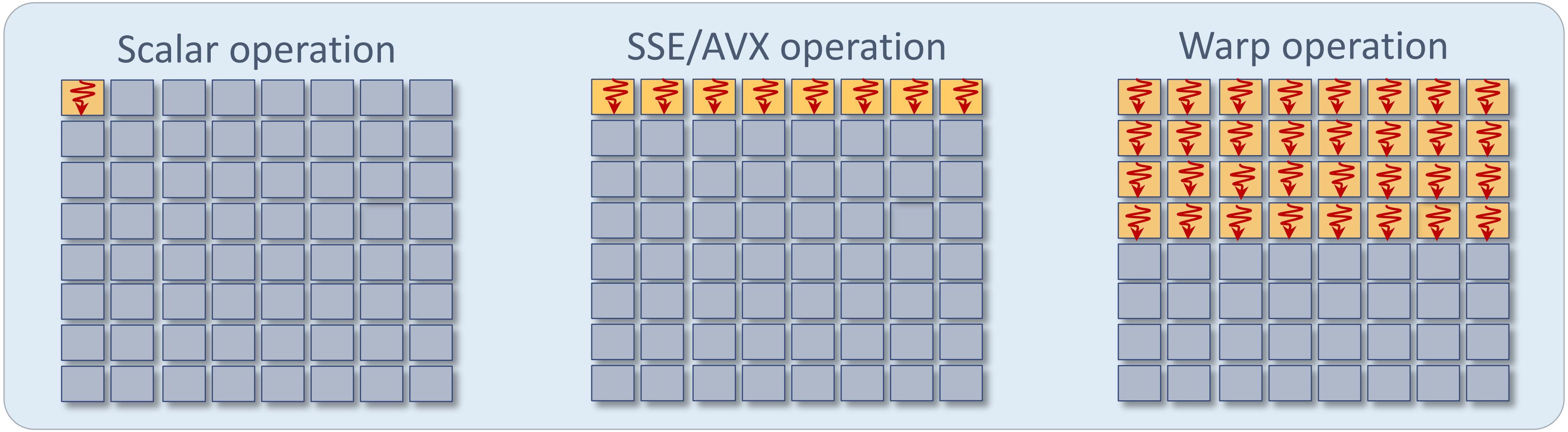
```
float computePi(int n_points) {
      dim3 grid = dim3(n_points, 1, 1);
      dim3 block = dim3(1, 1, 1);

      //Allocate data on graphics card for output
      cudaMalloc((void**)&gpu_data, gpu_data_size);

      //Execute function on GPU (“lauch the kernel”)
      computePiKernel1<<<grid, block>>>(gpu_data);

      //Copy results from GPU to CPU
      cudaMemcpy(&cpu_data[0], gpu_data, gpu_data_size, cudaMemcpyDeviceToHost);

      //Estimate Pi
      for (int i=0; i<cpu_data.size(); ++i) {
            n_inside += cpu_data[i];
      }
      return pi = 4.0f * n_inside / n_points;
}
```

Parallel GPU version 1 (CUDA) 3/3

- Unable to run more than 65535 sample points
- Barely faster than single threaded CPU version for largest size!
- Kernel launch overhead appears to dominate runtime
- The fit between algorithm and architecture is poor:
  - 1 thread per block: Utilizes at most 1/32 of computational power.

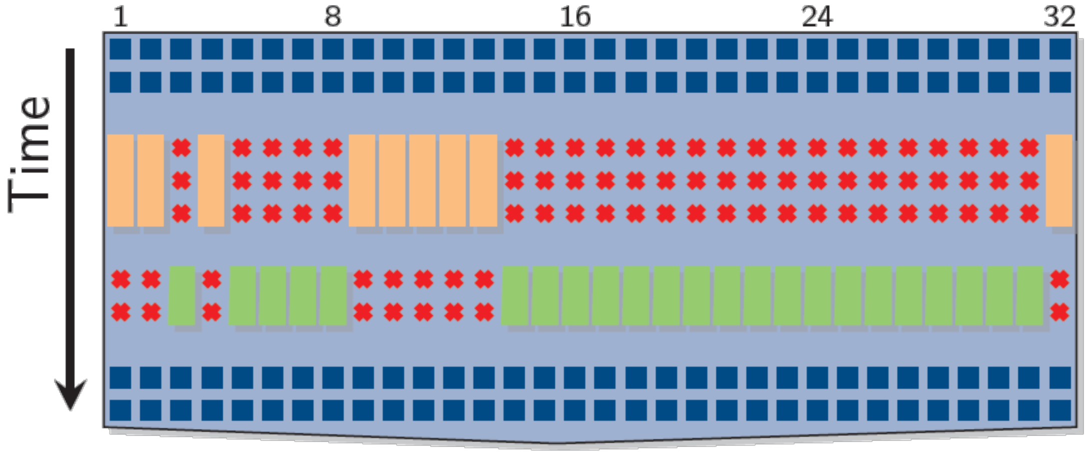


GPU Vector Execution Model



- **CPU scalar:** 1 thread, 1 operand on 1 data element
- **CPU SSE/AVX:** 1 thread, 1 operand on 2-8 data elements
- **GPU Warp:** 32 threads, 32 operands on 32 data elements
  - Exposed as **individual threads**
  - Actually runs the **same instruction**
  - Divergence implies **serialization and masking**

Serialization and masking



```
// Non-divergent code
if( x > 0 ) {
      y = pow(x, exp);
      y *= Ks;
      z = y + Ka;
} else {
      x = 0;
      z = Ka;
}
// Non-divergent code
```

Hardware automatically serializes and masks divergent code flow:

- Execution time is the sum of all branches taken
- Programmer is relieved of fiddling with element masks (which is necessary for SSE/AVX)
- Worst case 1/32 performance
- Important to **minimize divergent code flow <u>within warps</u>**!
  - Move conditionals into data, use min, max, conditional moves.

Parallel GPU version 2 (CUDA) 1/2

```
__global__ void computePiKernel2(unsigned int* output) {
         //Generate coordinate
         float x = generateRandomNumber();
         float y = generateRandomNumber();

         //Compute radius
         float r = sqrt(x*x + y*y);

         //Check if within circle
         if (r <= 1.0f) {
                  output[blockIdx.x*blockDim.x + threadIdx.x] = 1;
         } else {
                  output[blockIdx.x*blockDim.x + threadIdx.x] = 0;
         }
}
```

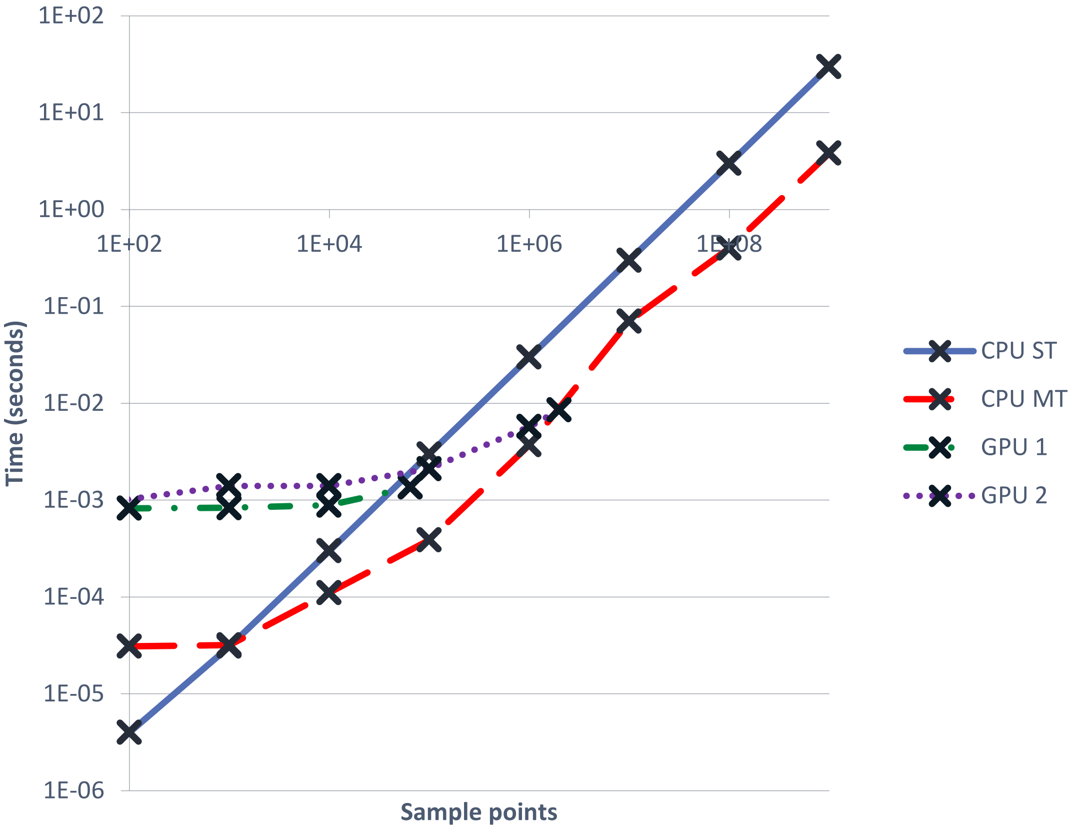
New indexing

```
float computePi(int n_points) {
      dim3 grid = dim3(n_points/32, 1, 1);
      dim3 block = dim3(32, 1, 1);
      …
      //Execute function on GPU (“lauch the kernel”)
      computePiKernel1<<<grid, block>>>(gpu_data);
      …
}
```

32 threads per block

Parallel GPU version 2 (CUDA) 2/2

- Unable to run more than 32*65535 sample points
- Works well with 32-wide SIMD
- Able to keep up with multi-threaded version at maximum size!
- We perform roughly 16 operations per 4 bytes written (1 int): memory bound kernel! Optimal is 60 operations!



Parallel GPU version 3 (CUDA) 1/4

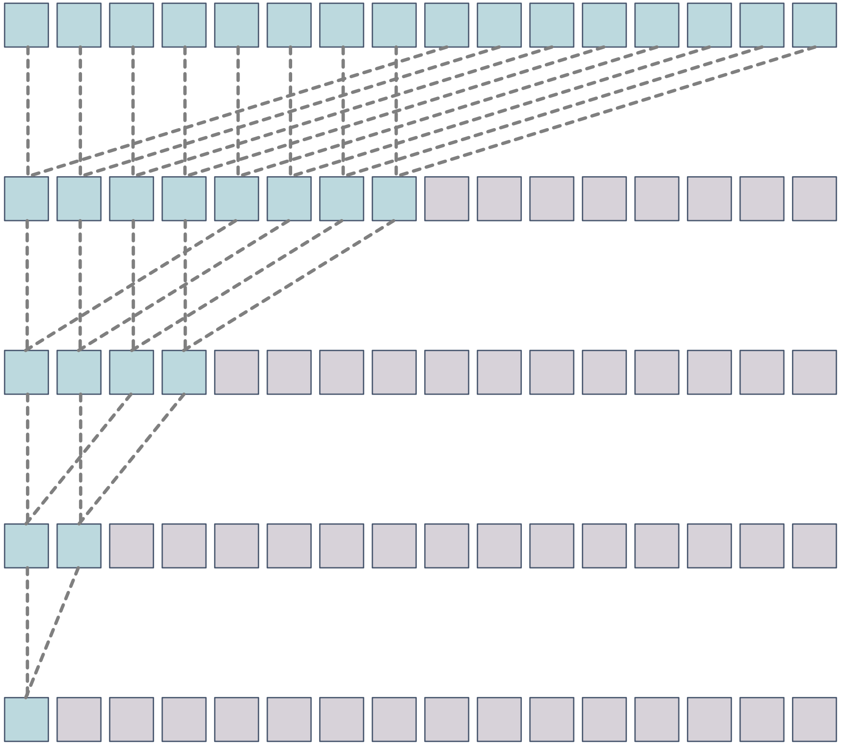
```
__global__ void computePiKernel3(unsigned int* output, unsigned int seed) {
        __shared__ int inside[32];

        //Generate coordinate
        //Compute radius
        …

        //Check if within circle
        if (r <= 1.0f) {
                inside[threadIdx.x] = 1;
        } else {
                inside[threadIdx.x] = 0;
        }

        … //Use shared memory reduction to find number of inside per block
```

Shared memory: a kind of "programmable cache"
We have 32 threads: One entry per thread

Parallel GPU version 3 (CUDA) 2/4

- Shared memory is a kind of programmable cache
  - Fast to access (just slightly slower than registers)
  - Programmers responsibility to move data into shared memory
  - All threads in one block can see the same shared memory
  - Often used for communication between threads

- Sum all elements in shared memory using shared memory reduction

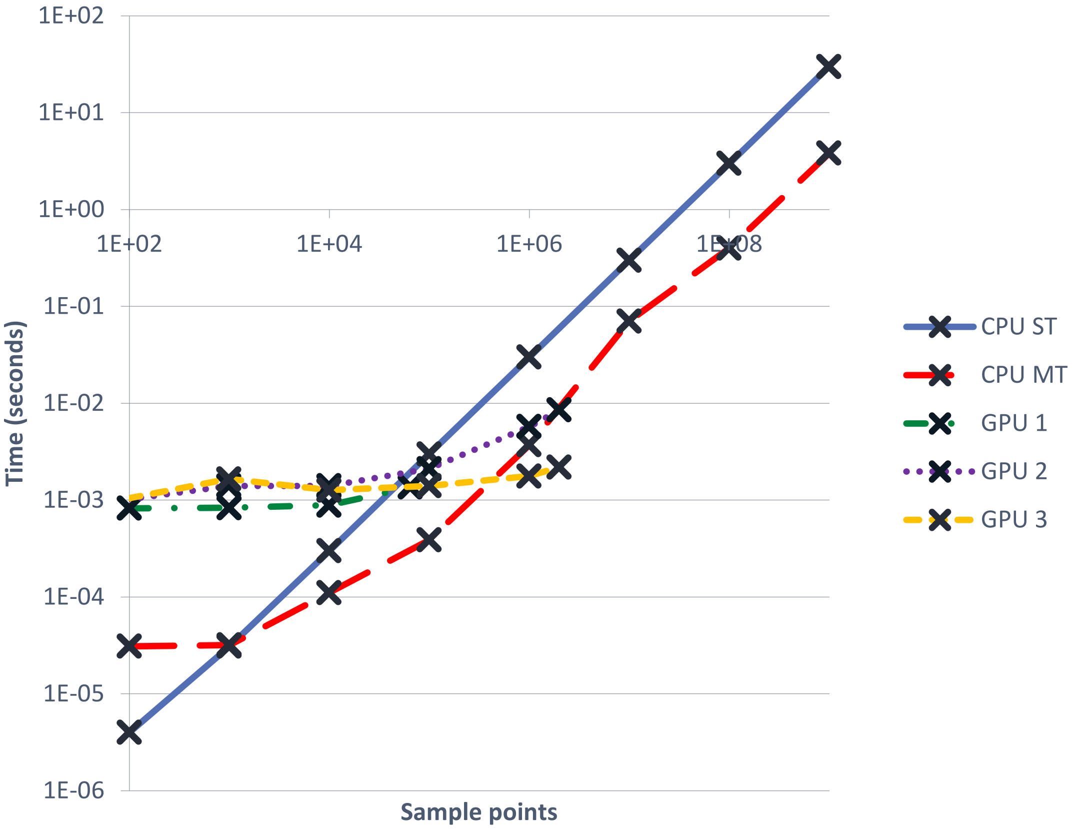


Parallel GPU version 3 (CUDA) 3/4

```
    … //Continued from previous slide

    //Use shared memory reduction to find number of inside per block
    //Remember: 32 threads is one warp, which execute synchronously
    if (threadIdx.x < 16) {
            p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+16];
            p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+8];
            p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+4];
            p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+2];
            p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+1];
    }

    if (threadIdx.x == 0) {
            output[blockIdx.x] = inside[threadIdx.x];
    }
}
```

Parallel GPU version 3 (CUDA) 4/4

- Memory bandwidth use reduced by factor 32!
- Good speed-up over multithreaded CPU!
- Maximum size is still limited to 65535*32.
- Two ways of increasing size:
  - Increase number of threads
  - Make each thread do more work



Parallel GPU version 4 (CUDA) 1/2

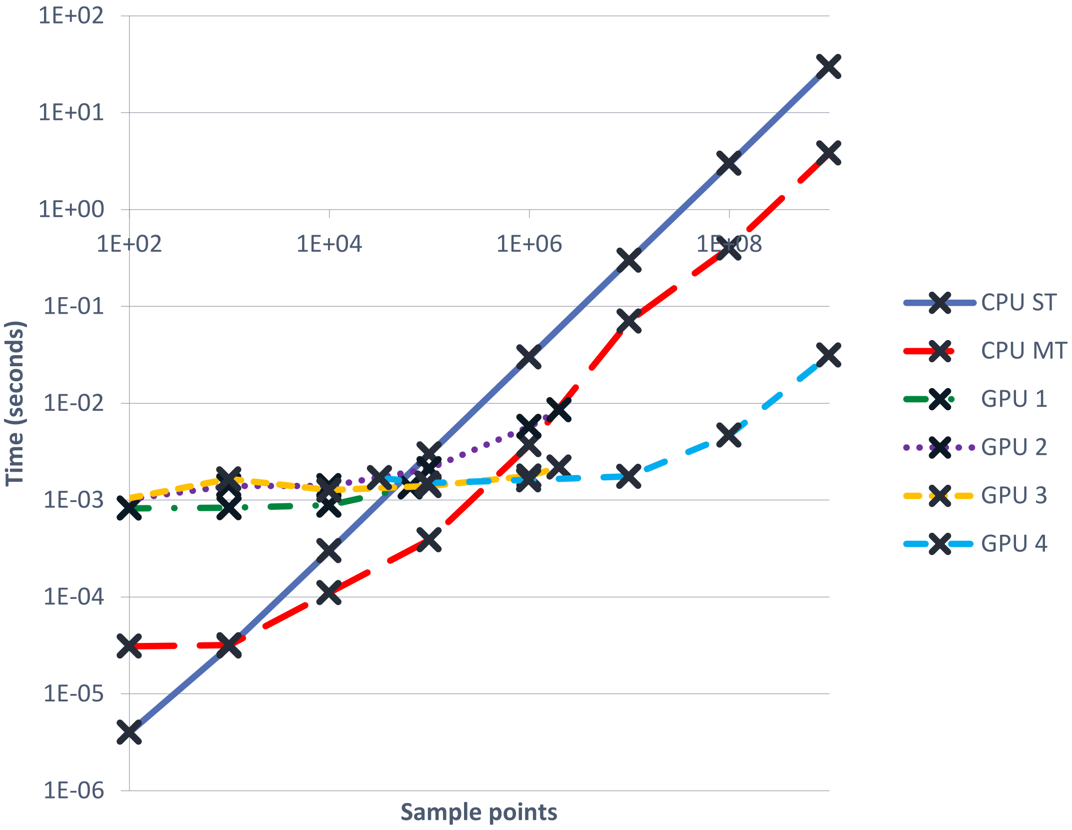
```
__global__ void computePiKernel4(unsigned int* output) {
        int n_inside = 0;

         //Shared memory: All threads can access this
        __shared__ int inside[32];
        inside[threadIdx.x] = 0;

        for (unsigned int i=0; i<iters_per_thread; ++i) {
                //Generate coordinate
                //Compute radius
                //Check if within circle
                if (r <= 1.0f) { ++inside[threadIdx.x]; }
        }

        //Communicate with other threads to find sum per block
        //Write out to main GPU memory
 }
```

Parallel GPU version 4 (CUDA) 2/2

- Overheads appears to dominate runtime up-to 10.000.000 points:
  - Memory allocation
  - Kernel launch
  - Memory copy

- Estimated GFLOPS: ~450
  Thoretical peak: ~4000

- Things to investigate further:
  - Profile-driven development*!
  - Check number of threads, memory access patterns, instruction stalls, bank conflicts, ...



*See e.g., Brodtkorb, Sætra, Hagen, **GPU Programming Strategies and Trends in GPU Computing**, JPDC, 2013

Comparing performance

- Previous slide indicates speedup of
  - 100x versus OpenMP version
  - 1000x versus single threaded version
  - Theoretical performance gap is 10x: why so fast?

- Reasons why the comparison is fair:
  - Same generation CPU (Core i7 3930K) and GPU (GTX 780)
  - Code available on Github: you can test it yourself!

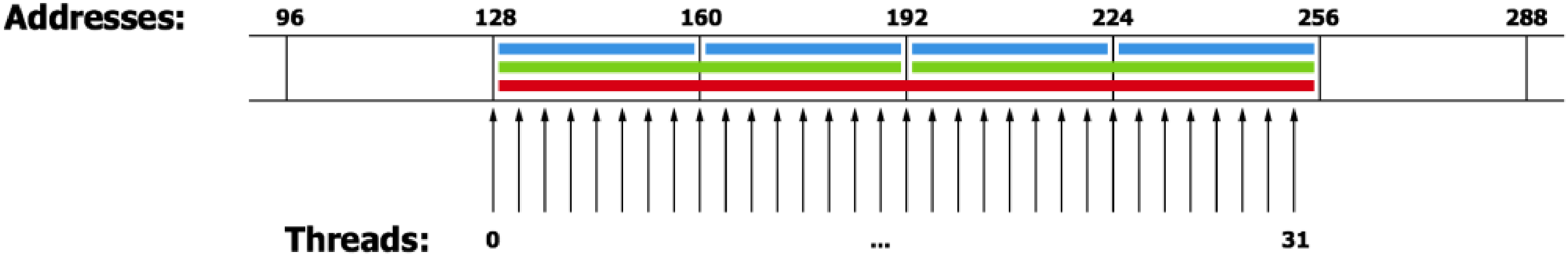
- Reasons why the comparison is unfair:
  - Optimized GPU code, unoptimized CPU code.
  - I do not show how much of CPU/GPU resources I actually use (profiling)
  - I cheat with the random function (I use a simple linear congruential generator).

Optimizing Memory Access

Memory access 1/2

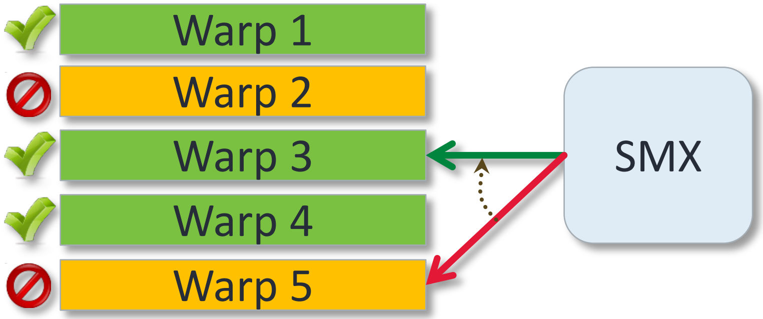
- Accessing a single memory address triggers transfer of a full *cache line* (128 bytes)
  - The smallest unit transferrable over the memory bus
  - Identical to how CPUs transfer data

- For peak performance, 32 threads should use 32 consecutive integers/floats
  - This is referred to as coalesced reads



- On modern GPUs: Possible to transfer 32 byte segments: Better fit for random access!
- Slightly more complex in reality: see CUDA Programming Guide for full set of rules

Memory access 2/2



- GPUs have high bandwidth, and high latency
  - Latencies are on the order of hundreds to thousands of clock cycles
- Massive multithreading hides latencies
  - When one warp stalls on memory request, another warp steps in and uses execution units
- Effect: Latencies are completely hidden as long as you have enough memory parallelism:
  - More than 100 simultaneous requests for full cache lines per SM (Kepler).
  - Far more for random access!

Example: Parallel reduction

- Reduction is the operation of finding a single number from a series of numbers
    - Frequently used parallel building block in parallel computing
    - We've already used it to compute π

- Examples:
    - Find minimum, maximum, average, sum
    - In general: Perform a binary operation on a set data

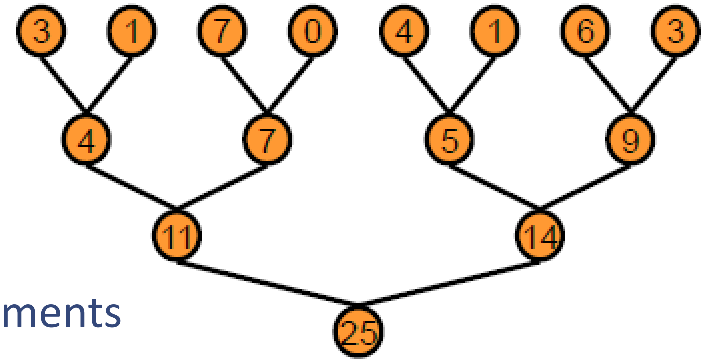
- CPU example:

```
//Initialize to first element
T result = data[0];

//Loop through the rest of the elements
for (int i=1; i<data.size(); ++i) {
        //Perform binary operator (e.g., op(a, b) = max(a, b))
        result = op(result, data[i]);
}
```

Parallel considerations

- This is a completely memory bound application
  - O(1) operation per element read and written.
  - Need to optimize for memory access!

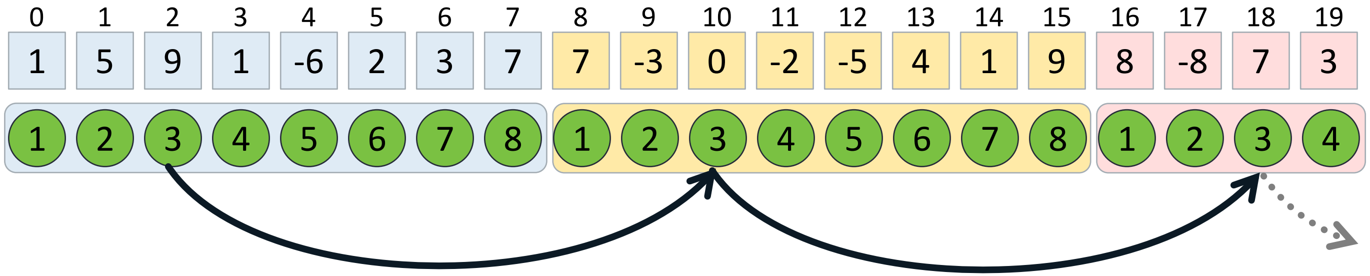
- Classical approach: represent as a binary tree
  - log2(n) passes required to reduce n elements
  - Example: 10 passes to find maximum of 1024 elements



- General idea:
  - Use few blocks with maximum number of threads (i.e., 512 in this example)
  - *Stride* through memory until all items are read
  - Perform shared memory reduction to find single largest

Example based on Mark Harris, Optimizing parallel reduction in CUDA

Striding through data



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	5	9	1	-6	2	3	7	7	-3	0	-2	-5	4	1	9	8	-8	7	3
1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4

```
for (int i=threadIdx.x; i<size; i += blockDim.x) {
        //Perform binary operator (e.g., op(a, b) = max(a, b))
        result = op(result, data[i]);
}
```

- Striding ensures perfect coalesced memory reads
- Thread 2 operates on elements 2, 10, 18, etc. for a block size of 8
- We have block size of 512: Thread 2 operates on elements 2, 514, 1026, …
- Perform "two-in-one" or "three-in-one" strides for more parallel memory requests

Shared memory reduction 1/2

- By striding through data, we efficiently reduce N/num_blocks elements to 512.

- Now the problem becomes reducing 512 elements to 1:
  lets continue the striding, but now in shared memory

- Start by reducing from 512 to 64 (notice use of __syncthreads()):

```
__syncthreads(); // Ensure all threads have reached this point

// Reduce from 512 to 256
if(tid < 256) { sdata[tid] = sdata[tid] + sdata[tid + 256]; }
__syncthreads();

// Reduce from 256 to 128
if(tid < 128) { sdata[tid] = sdata[tid] + sdata[tid + 128]; }
__syncthreads();

// Reduce from 128 to 64
if(tid < 64) { sdata[tid] = sdata[tid] + sdata[tid +  64]; }
__syncthreads();
```

Shared memory reduction 2/2

- When we have 64 elements, we can use 32 threads to perform the final reductions
- Remember that 32 threads is one warp, and execute instructions in SIMD fashion
- This means we do not need the syncthreads:

```
if (tid < 32) {
		volatile T *smem = sdata;
		smem[tid] = smem[tid] + smem[tid + 32];
		smem[tid] = smem[tid] + smem[tid + 16];
		smem[tid] = smem[tid] + smem[tid + 8];
		smem[tid] = smem[tid] + smem[tid + 4];
		smem[tid] = smem[tid] + smem[tid + 2];
		smem[tid] = smem[tid] + smem[tid + 1];
}

if (tid == 0) {
		global_data[blockIdx.x] = sdata[0];
}
```

- Volatile basically tells the optimizer "off-limits!"
- Enables us to safely skip __synchthreads()

Summary so far

- Part 1:
  - Motivation for going parallel
  - Multi- and many-core architectures
  - Parallel algorithm design
- Part 2
  - Example: Computing π on the GPU
  - Optimizing memory access

VeRoLog

EURO Working Group on Vehicle Routing and Logistics Optimization

VeRoLog 2014

Tutorial: GPU and Heterogeneous Computing in Discrete Optimization

Session 2: GPU Computing in Discrete Optimization

Outline

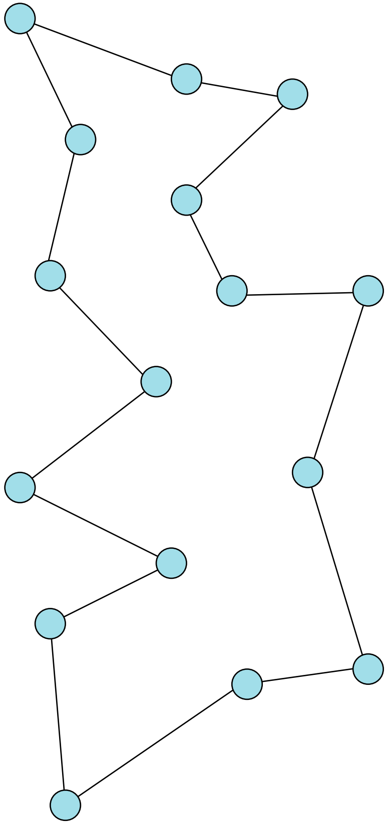
- Local Search
  - Sequential version
  - OpenMP version
  - GPU version
- Profiling the GPU version
- Filtering the GPU version
- Literature Overview

Programming Example – Local Search

(Simple) Local Search for TSP

- Travelling salesman problem:
  - n cities given, want to find shortest tour through all cities
- Best improving local search with swap moves:
  - Given initial/current solution
  - Find swap move the improves tour the most (if improving exists)
  - Apply best move to tour
  - Repeat
- Swap move: Exchange position of 2 cities in tour
  - Change (delta) in tour cost in O(1)

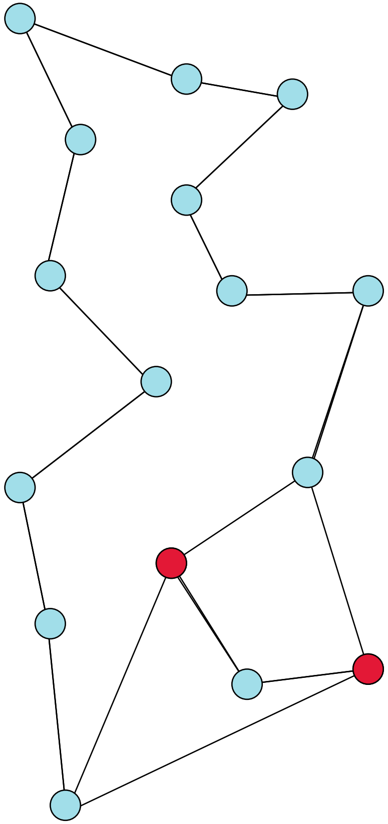
- Why local search as example:
  - Easy, well known
  - Offers clear parallelism
  - Often part of more advanced metaheuristics



(Simple) Local Search for TSP

- Travelling salesman problem:

  n cities given, want to find shortest tour through all cities
- Best improving local search with swap moves:
  - Given initial/current solution
  - Find swap move the improves tour the most (if improving exists)
  - Apply best move to tour
  - Repeat
- Swap move: Exchange position of 2 cities in tour
  - Change (delta) in tour cost in O(1)
- Why local search as example:
  - Easy, well known
  - Offers clear parallelism
  - Often part of more advanced metaheuristics



Sequential CPU Version

- Going through all swap moves by going through all city combinations
- City solution[0] invariant to avoid rotating

(1,2)	(1,3)	(1,4)	…	(1,n-1)
(2,3)	(2,4)	…	(2,n-1)	
(3,4)	…	(3,n-1)		
⋮				
(n-2,n-1)				

- => (x,y), x = 1, …, n-2 and y = x+1, …, n-1

Sequential CPU Version

Make sure best is improving

Iterate through city combinations

```
for (;;) { // Iterations
    Move best_move; float min_delta = 0.0;

    //Loop through all possible moves and find best (steepest descent)
    for (unsigned int x=1; x+2 <= num_nodes;++x) {
         for (unsigned int y=x+1; y+1 <= num_nodes;++y) {

              Move move(x,y,&solution[0]); // Generate move

              float delta = get_delta(move, city_coordinates);

              if (delta < min_delta) { // move improving and best so far?
                  best_move = move;
                  min_delta = delta;
    }    }    }

    // If no move improves the solution, we are finished
    if (min_delta > -1e-7)
         break;

    // Applies best move to current solution
    apply(best_move);
}
```

Generate swap move

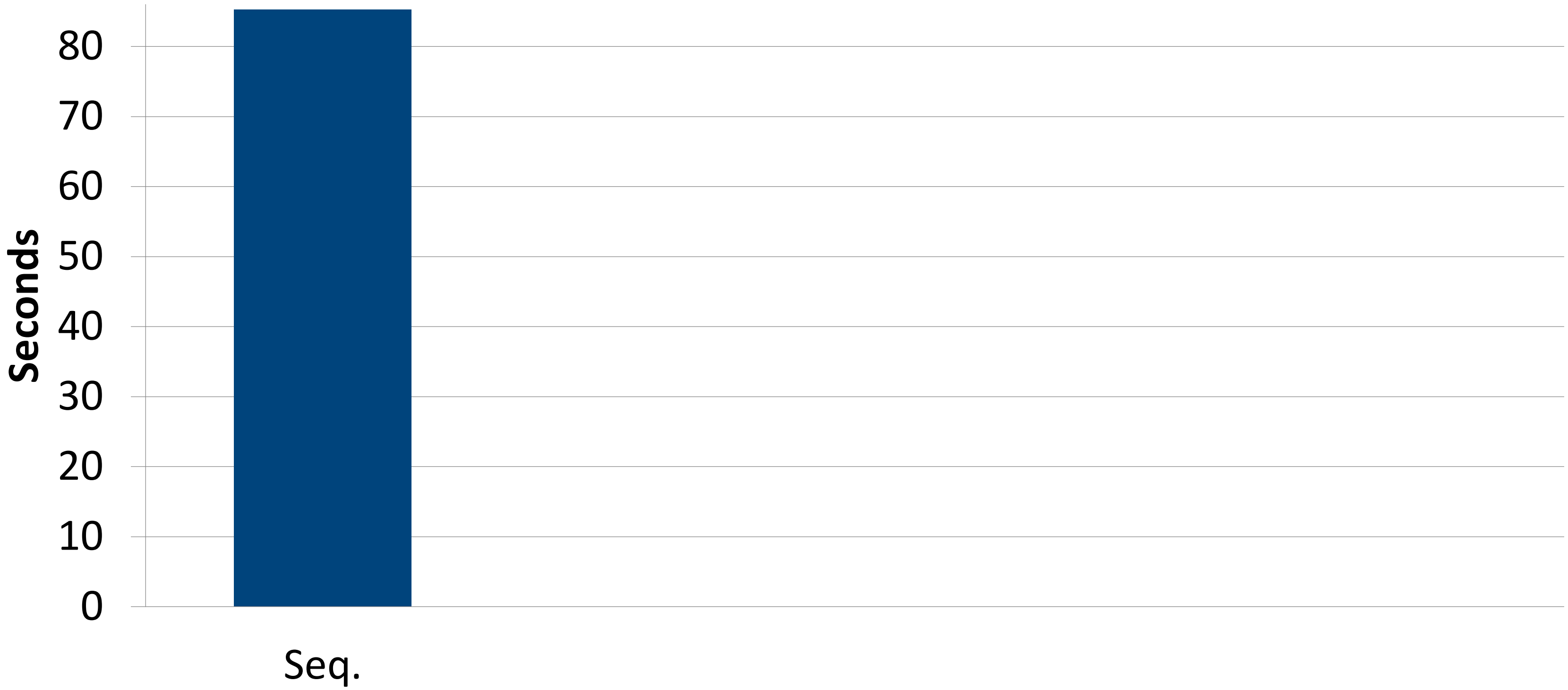
How good is the move?

Keep best

Find best improving

Apply Best

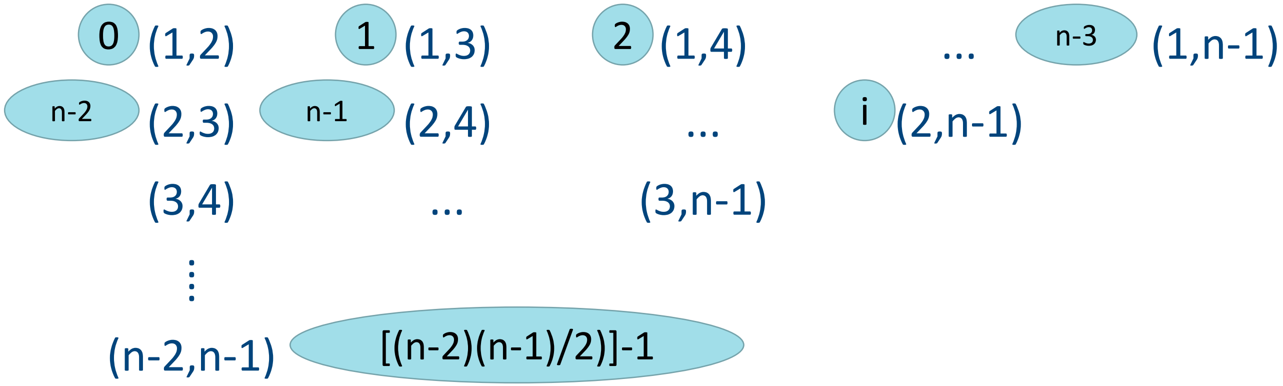
Timing – 1000 cities, 2524 iterations to minimal tour



CPU: Intel® Core™ i7-3740QM CPU 2.7GHz

Parallel OpenMP CPU Version

- Want to parallelize move - loop such that work is distributed between N threads
- Problem: double for-loop x,y
- Solution: Enumerate moves lexicographically



0 (1,2) 1 (1,3) 2 (1,4) ... n-3 (1,n-1)

n-2 (2,3) n-1 (2,4) ... i (2,n-1)

(3,4) ... (3,n-1)

⋮

(n-2,n-1) [(n-2)(n-1)/2)]-1

```
dx = n-2.0f-floor((sqrtf(4.0f*(n-1.0f)*(n-2.0f) - 8.0f*i - 7.0f)-1.0f)/2.0f);
dy = 2.0f+i-(dx-1)*(n-2.0f)+(dx-1.0f)*dx/2.0f;
x = (unsigned int)dx;   y = (unsigned int)dy;
```

Parallel OpenMP CPU Version

```
for (;;) { // Iterations
    Move best_move; float min_delta = 0.0;

    //Loop through all possible moves and find best (steepest descent)
    for (unsigned int x=1; x+2 <= num_nodes;++x) {
        for (unsigned int y=x+1; y+1 <= num_nodes;++y) {

            Move move(x,y,&solution[0]); // Generate move

            float delta = get_delta(move, city_coordinates);

            if (delta < min_delta) { // move improving and best so far?
                best_move = move;
                min_delta = delta;
    }     }     }

    // If no move improves the solution, we are finished
    if (min_delta > -1e-7)
        break;

    // Applies best move to current solution
    apply(best_move);
}
```

Find best improving

Parallel OpenMP CPU Version

```
for (;;) { // Iterations
    Move best_move; float min_delta = 0.0;

    //Loop through all possible moves and find best (steepest descent)
    for (unsigned int i=0; i <= num_moves; ++i) {
        Move move = generate_move(i, num_nodes, &solution[0]);

        float delta = get_delta(move, city_coordinates); // improvement in tour

        if (delta < min_delta) { // move improving and best so far?
            best_move = move;
            min_delta = delta;
    }   }

    // If no move improves the solution, we are finished
    if (min_delta > -1e-7)
        break;

    // Applies best move to current solution
    apply(best_move);
}
```

Find best improving

Parallel OpenMP CPU Version

Would like to do this,
but how to find min_delta?

```
for (;;) { // Iterations
    Move best_move; float min_delta = 0.0;

    //Loop through all possible moves and find best (steepest descent)
    #pragma omp parallel for
    for (unsigned int i=0; i <= num_moves; ++i) {
        Move move = generate_move(i, num_nodes, &solution[0]);
        float delta = get_delta(move, city_coordinates); // improvement in tour
        if (delta < min_delta) { // move improving and best so far?
            best_move = move;
            min_delta = delta;
    }   }

    // If no move improves the solution, we are finished
    if (min_delta > -1e-7)
        break;

    // Applies best move to current solution
    apply(best_move);
}
```

Find best improving

Parallel OpenMP CPU Version

```
for (;;) { // Iterations
    //Loop through all possible moves and find best per thread
    #pragma omp parallel
    {
            int best_move; float min_delta = 0.0;
            #pragma omp for
            for (int i=0; i <= static_cast<int>(num_moves); ++i) {
                    Move move = generate_move(i, num_nodes, &solution[0]);
                    float delta = get_delta(move, city_coordinates);
                    if (delta < min_delta) { // move improving and best so far?
                        best_move = i;
                        min_delta = delta;
            }       }
            best_moves[omp_get_thread_num()] = best_move; // store thread-best-move
            min_deltas[omp_get_thread_num()] = min_delta; // store thread-best-delta
    }
   ⋮ // Choose and apply best improving move
}
```

Find thread best

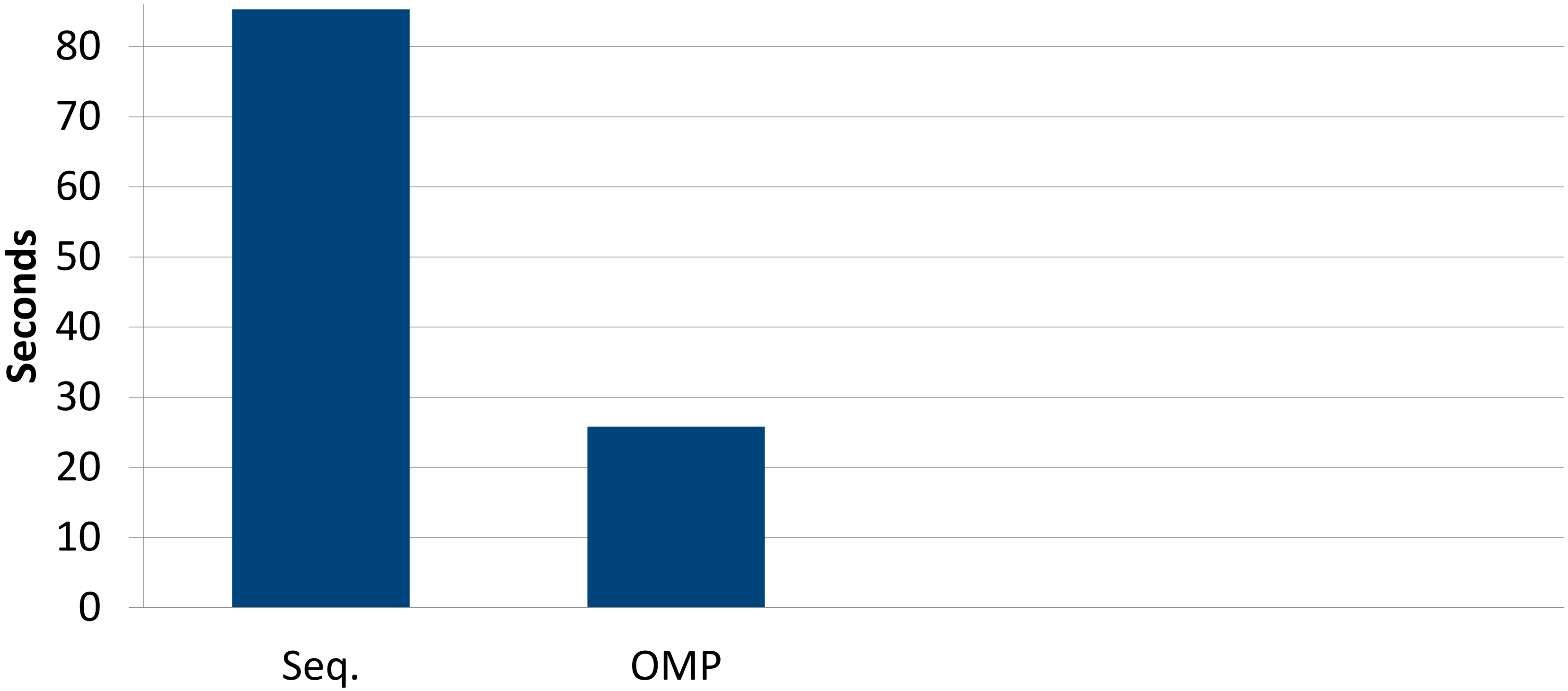
Parallel OpenMP CPU Version

```
for (;;) { // Iterations
    // Loop through all possible moves and find best per thread
    ⋮

    // find best move of threads
    int best_move_id = best_moves[0];
    float min_delta = min_deltas[0];
    for (int i = 1; i < omp_num_threads; ++i)
    {
          if (min_deltas[i] < min_delta) {
                  best_move_id = best_moves[i];
                  min_delta = min_deltas[i];
    }     }

    // If no move improves the solution, we are finished
    if (min_delta > -1e-7)
         break;

    // Applies best move to current solution
    Move best_move = generate_move(best_move_id, num_nodes, &solution[0]);
    apply(best_move);
}
```

Find best of thread-best

Apply Best

Timing – 1000 cities, 2524 iterations to minimal tour



CPU: Intel® Core™ i7-3740QM CPU 2.7GHz

GPU Version

- In OpenMP:
  - Few threads (up to 8, maybe 16)
  - Loop split automatically by OpenMP
  - Choosing best of thread-best-moves simple due to small number of threads
  - Data readily available

- On GPU:
  - Many threads (more than 1000)
  - Need to split work “manually” – need to split loop
  - How to choose best of thread-best-moves?
  - Need to copy data to GPU

Copy Data to GPU

- Allocate space for solution and copy initial solution to GPU

```
cudaError err; unsigned int* solution_gpu; //Pointer to memory on the GPU
```

Allocate

```
//Allocate GPU memory for solution
err = cudaMalloc(&solution_gpu, solution.size()*sizeof(unsigned int));
if (err != cudaSuccess) {
    std::cout << "Could not allocate GPU memory for solution" << std::endl;
    return;
}
```

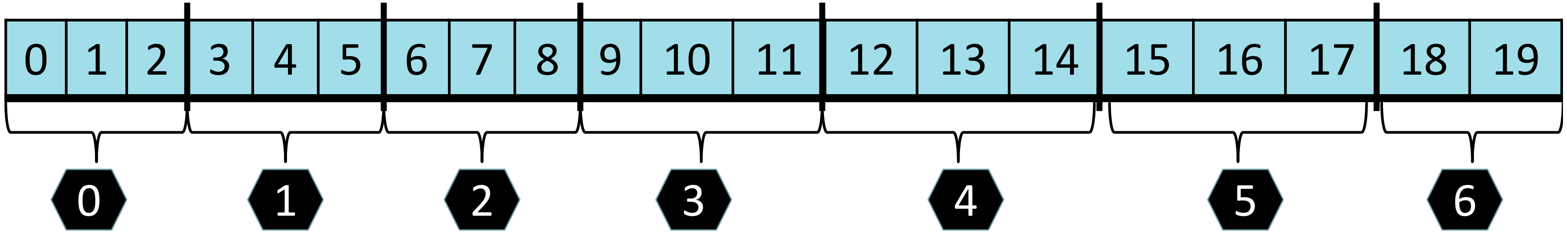
Copy

```
//Copy solution to GPU
err = cudaMemcpy(solution_gpu, &solution[0],
                 solution.size()*sizeof(unsigned int), cudaMemcpyHostToDevice);
if (err != cudaSuccess) {
    std::cout << "Could not copy solution to GPU memory" << std::endl;
    return;
}
```

- Similarly allocate space and copy coordinates
- Allocate space for thread-best-moves and thread-best-deltas

Evaluate Moves on GPU

- Split loop through moves into equal parts
- Each thread goes through its part



- M = Number of moves per thread = ceiling(num_moves / num_threads)
- Thread 0 takes first M moves, thread 1 next M moves, ...
- Thread i takes moves

*M*i, ..., M*i + M-1;* *i = 0, ..., num_threads*

Evaluate Moves on GPU

- Splitting loop over moves “manually”
- Thread i has index `tid`

```
float min_delta = 0.0;
const unsigned int first_move = tid*num_moves_per_thread_;
unsigned int best_move = first_move;
// Find best move in thread
for (int i=first_move; i<first_move+num_moves_per_thread_; ++i) {
    if (i < num_moves) {
        Move move = generate_move(i, num_nodes_, solution_);
        float delta = get_delta(move, city_coordinates_);
        if (delta < min_delta) {
            min_delta = delta;
            best_move = i;
}   }   }
```

Find thread best

Evaluate Moves on GPU

- Need extra evaluation kernel on GPU:

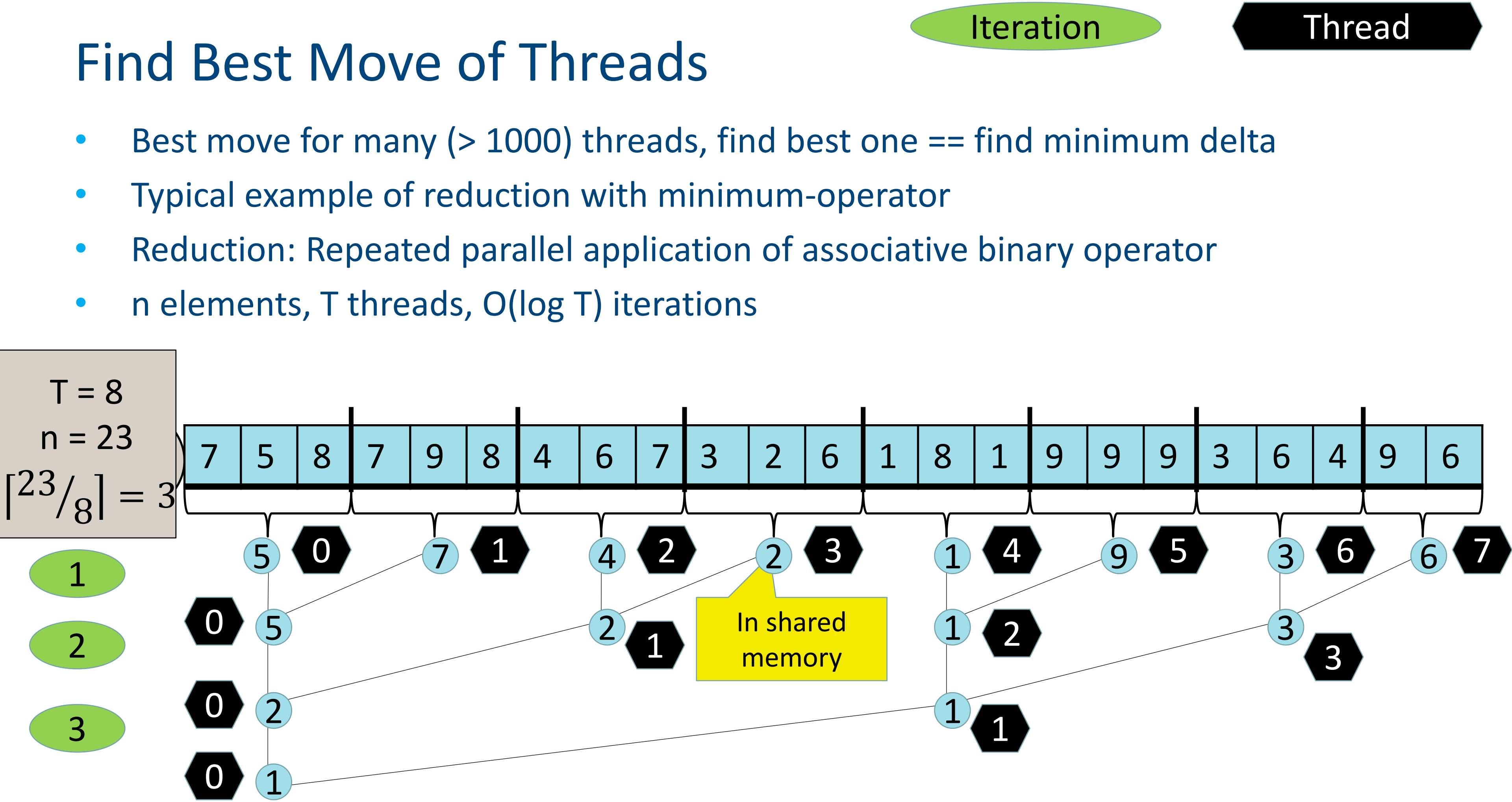
```
__global__ void evaluate_moves_kernel(unsigned int* solution_,
          const float* city_coordinates_, float* deltas_, unsigned int* moves_,
          unsigned int num_nodes_, unsigned int num_moves_per_thread_) {
    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
    const unsigned int num_moves = (static_cast<int>(num_nodes_)- 2)*
                                           (static_cast<int>(num_nodes_)-1)/2;

    float min_delta = 0.0;
    const unsigned int first_move = tid*num_moves_per_thread_;
    unsigned int best_move = first_move;

    // Find best move in thread
    ...
    deltas_[tid] = min_delta; // Store thread-best-delta
    moves_[tid] = best_move; // Store thread-best-move
}
```

Setup

Store thread-best

Find Best Move of Threads

- Best move for many (> 1000) threads, find best one == find minimum delta
- Typical example of reduction with minimum-operator
- Reduction: Repeated parallel application of associative binary operator
- n elements, T threads, O(log T) iterations



Apply best move

```
template <unsigned int threads>
__global__ void apply_best_move_kernel(...) {
   unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x; // Thread id
   __shared__ float deltas_shmem[threads];//Shared memory available to all threads
   __shared__ unsigned int moves_shmem[threads]; // Shared memory

   // Find best move using described algorithm
   ...
   // Now: thread-best-move-id in moves_shmem[0]
   //      thread-best-delta in deltas_shmem[0]

   // Apply move and cleanup
   if (tid == 0) {
      if (deltas_shmem[0] < -1e-7f) {
         Move move = generate_move(moves_shmem[0], num_nodes_, solution_);
         apply_move(move);
      }
      deltas_[0] = deltas_shmem[0]; // store minimum delta in deltas_[0]
}  }
```

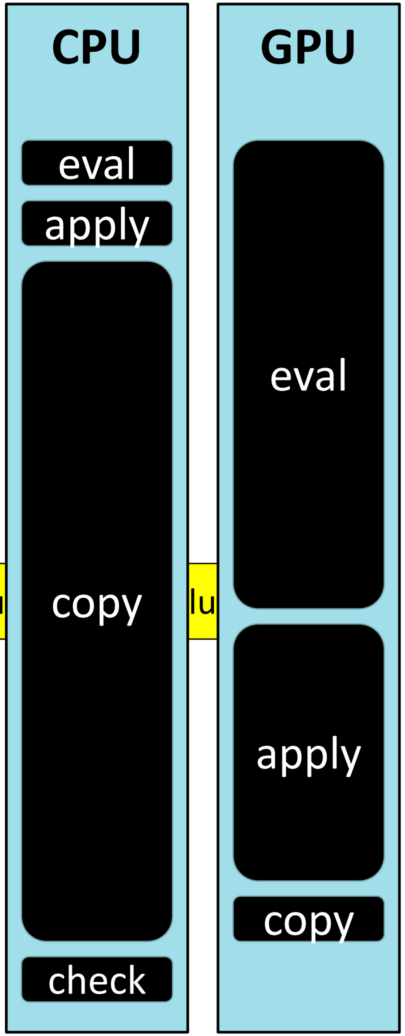
Apply Best

Main Loop

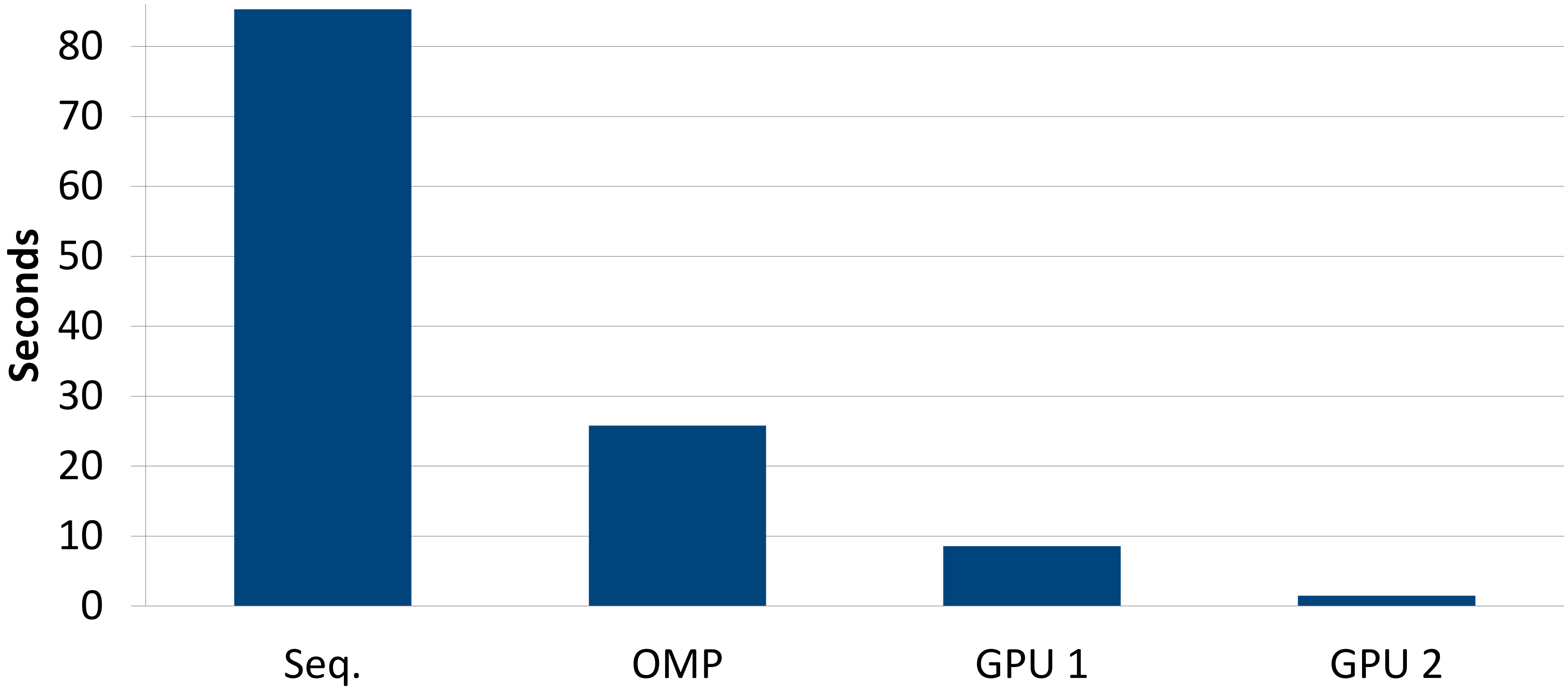
- Have data on GPU
- Have evaluation kernel
- Have best move finding & applying kernel
- Need main loop

Main Loop

```
for (;;) {

    //Loop through all possible moves and find best (steepest d
    evaluate_moves_kernel<<<evaluate_grid, evaluate_block>>>(so
                  coords_gpu, deltas_gpu, moves_gpu, num_nodes, nu
    apply_best_move_kernel<num_apply_threads><<<apply_grid, app
              (solution_gpu, deltas_gpu, moves_gpu, num_nodes, nu

    //Copy the smallest delta and best move to the CPU.
    float min_delta = 0.0;
    err = cudaMemcpy(&min_delta, &deltas_gpu[0],
                          sizeof(float), cudaMemcpyDeviceToHost);
    if (err != cudaSuccess) {
        std::cout << "Could not copy minimum delta to CPU" << st
        return 0;
    }

    // If no moves improve the solution, we are finished
    if (min_delta > -1e-7)
        break;
}
```



Timing – 1000 cities, 2524 iterations to minimal tour



CPU: Intel® Core™ i7-3740QM CPU 2.7GHz

GPU 1: NVS 5200M

GPU 2: GeForce GTX 480

How to use CUDA in Visual Studio

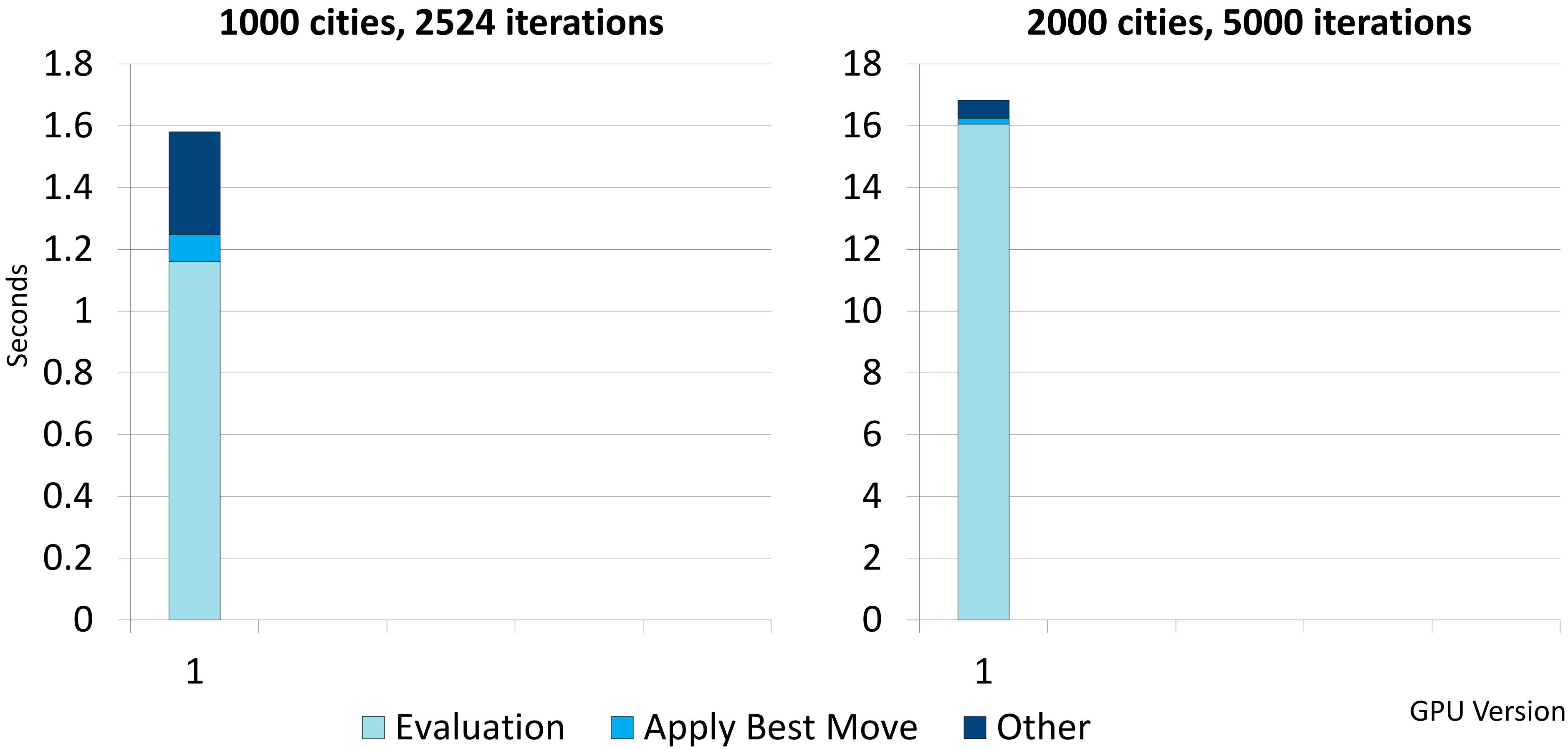
- CUDA is downloadable from NVIDIA’s web-pages
- Comes with
  - CUDA Samples
  - Nsight Visual Studio Edition (for Windows) and Nsight Eclipse Edition (for Linux / Mac OS X)
  - Developer drivers
- Now we will show live how to
  - Create a project in Visual Studio that can use CUDA
  - Using our local search example, how to debug in CUDA
  - How to analyse / profile in CUDA

Get started with CUDA in Visual Studio 2010

- Download latest version and install it
  - https://developer.nvidia.com/cuda-zone
  - Make sure you also installed Drivers and NSight (should come along CUDA)
- Create a CUDA project:
  - Go to File->New->Project
  - Choose NVIDIA->CUDA X.Y (latest version)
  - Fill in project- and solution-name, directory, etc
  - This creates a new CUDA project, already containing an example CUDA code
  - Just compile it and run it
- To Debug
  - Go to NSight->Start CUDA Debugging
  - You can set breakpoints in kernel code just as usual, breakpoints on host code are ignored
  - Activate NSight->Enable CUDA Memory Checker to have memory access checked while debugging
- To Profile
  - Go to NSight->Start Performance Analysis

Profiling the Programming Example

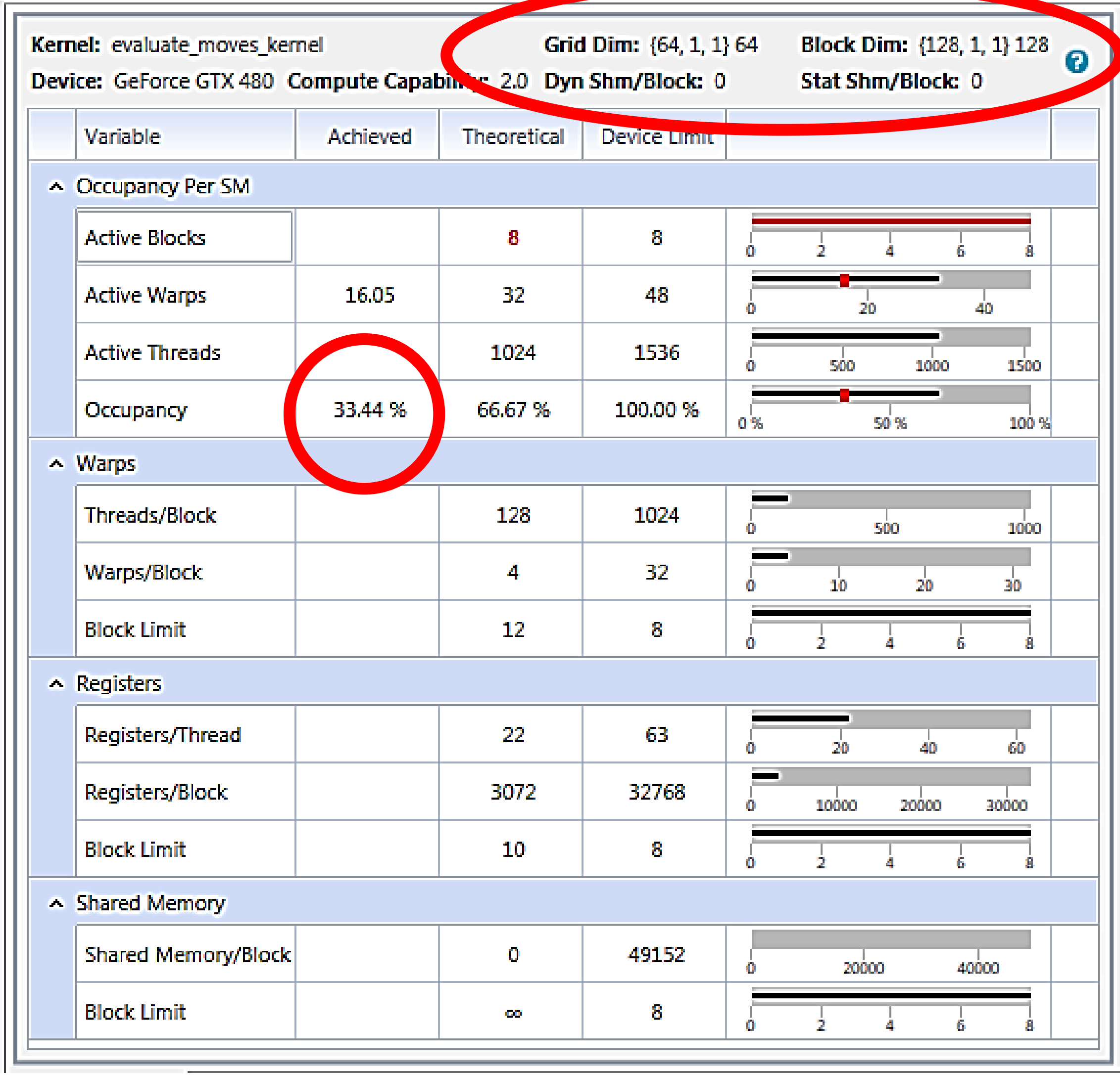
Timing the GPU Version



Profiling the Evaluation Kernel

Occupancy Experiment

- Good
  - 8 / 8 blocks per SM
- Medium/OK
  - 1024 / 1536 threads per SM
  - 22 registers per thread (max 1489 threads per SM)
- Bad
  - 33.44 % Achieved Occupancy
  - Only 128*64 = 8192 threads
  - GTX 480 can support 23040 threads



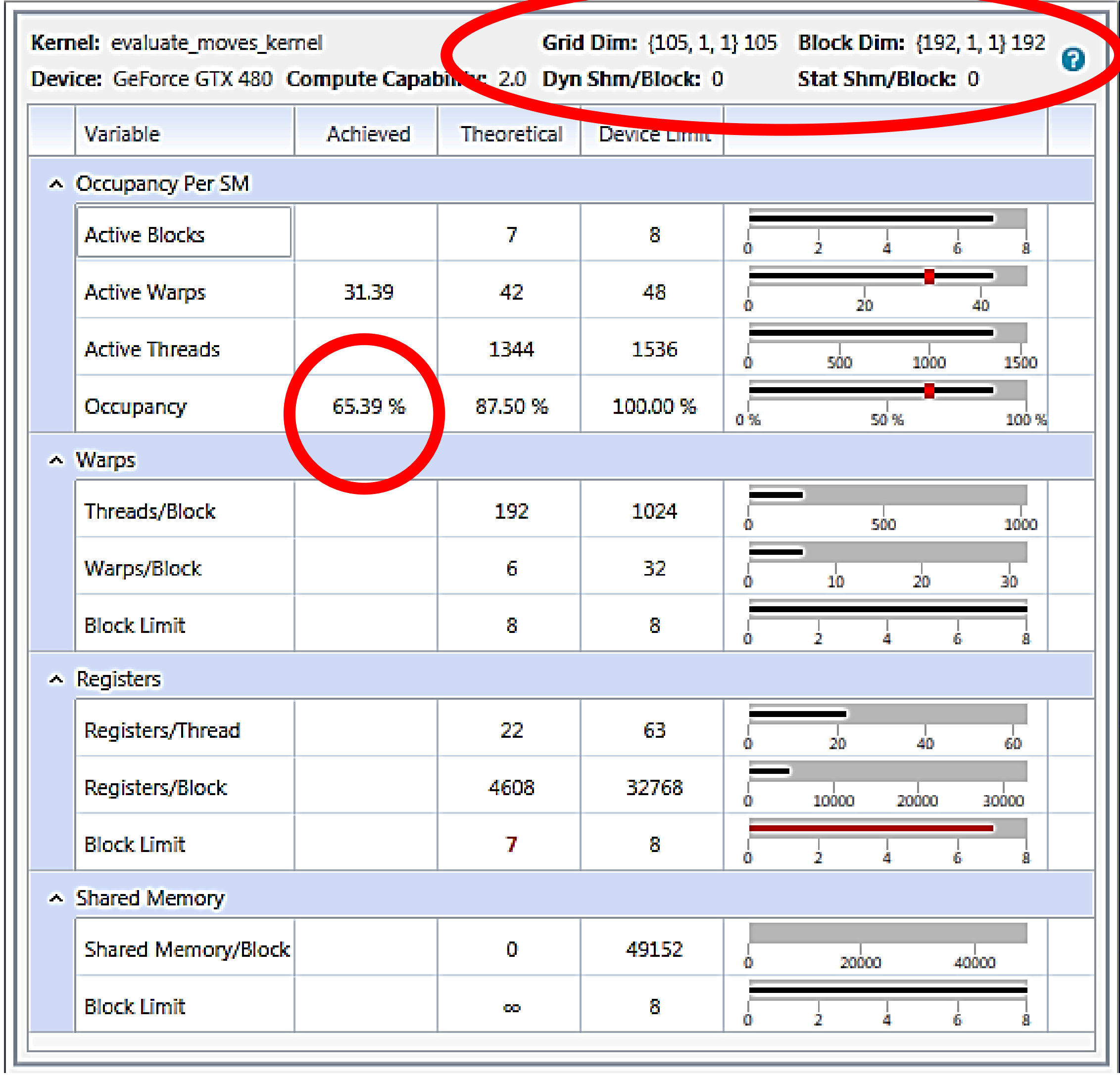
Kernel: evaluate_moves_kernel   Grid Dim: {64, 1, 1} 64   Block Dim: {128, 1, 1} 128

Device: GeForce GTX 480   Compute Capability: 2.0   Dyn Shm/Block: 0   Stat Shm/Block: 0

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		8	8
Active Warps	16.05	32	48
Active Threads		1024	1536
Occupancy	33.44 %	66.67 %	100.00 %
Warps			
Threads/Block		128	1024
Warps/Block		4	32
Block Limit		12	8
Registers			
Registers/Thread		22	63
Registers/Block		3072	32768
Block Limit		10	8
Shared Memory			
Shared Memory/Block		0	49152
Block Limit		∞	8

Profiling the Evaluation Kernel

- Occupancy good

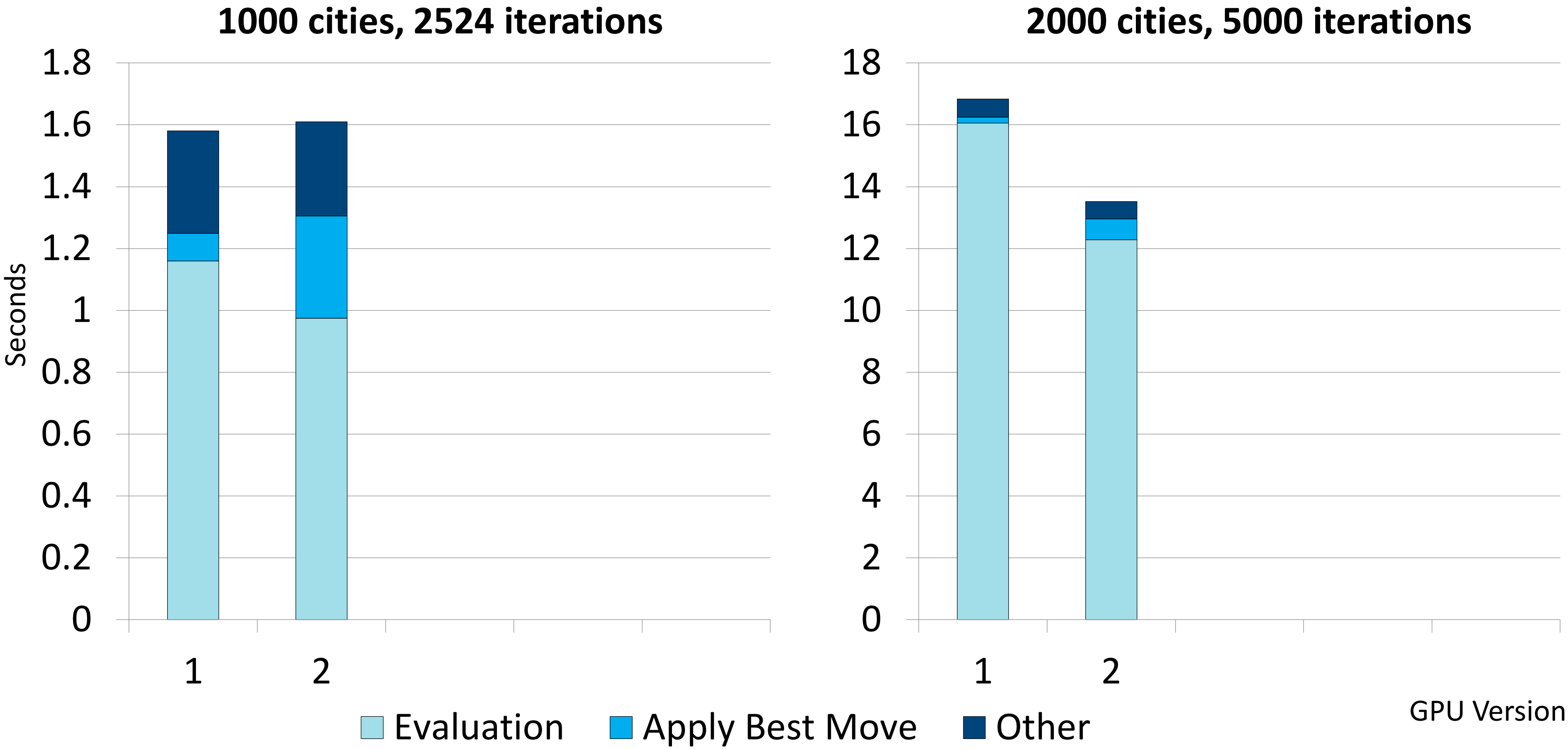


Kernel: evaluate_moves_kernel **Grid Dim:** {105, 1, 1} 105 **Block Dim:** {192, 1, 1} 192

Device: GeForce GTX 480 **Compute Capability:** 2.0 **Dyn Shm/Block:** 0 **Stat Shm/Block:** 0

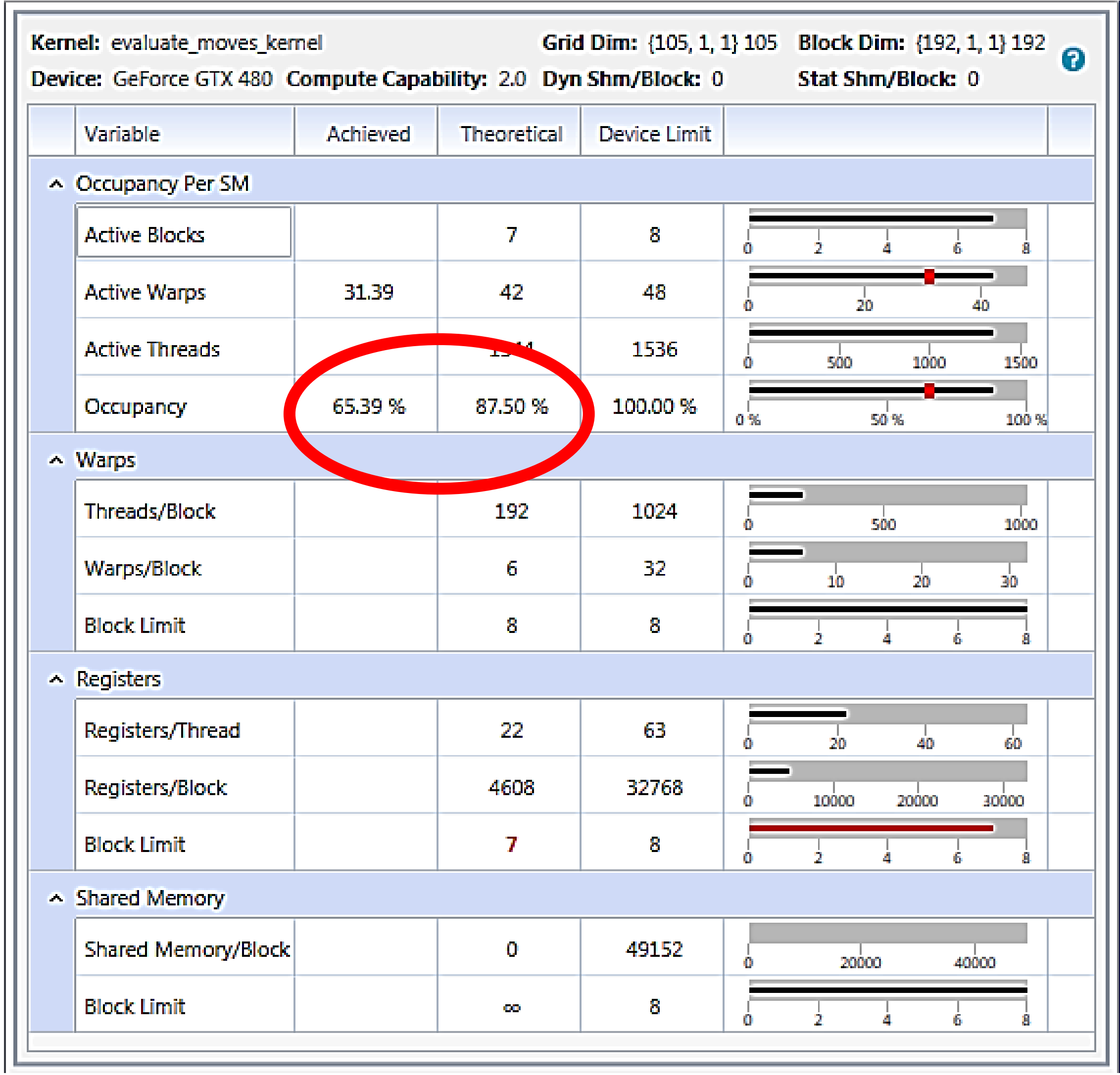
Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		7	8
Active Warps	31.39	42	48
Active Threads		1344	1536
Occupancy	65.39 %	87.50 %	100.00 %
Warps			
Threads/Block		192	1024
Warps/Block		6	32
Block Limit		8	8
Registers			
Registers/Thread		22	63
Registers/Block		4608	32768
Block Limit		7	8
Shared Memory			
Shared Memory/Block		0	49152
Block Limit		∞	8

Timing the GPU Version



Profiling the Evaluation Kernel

- Occupancy good

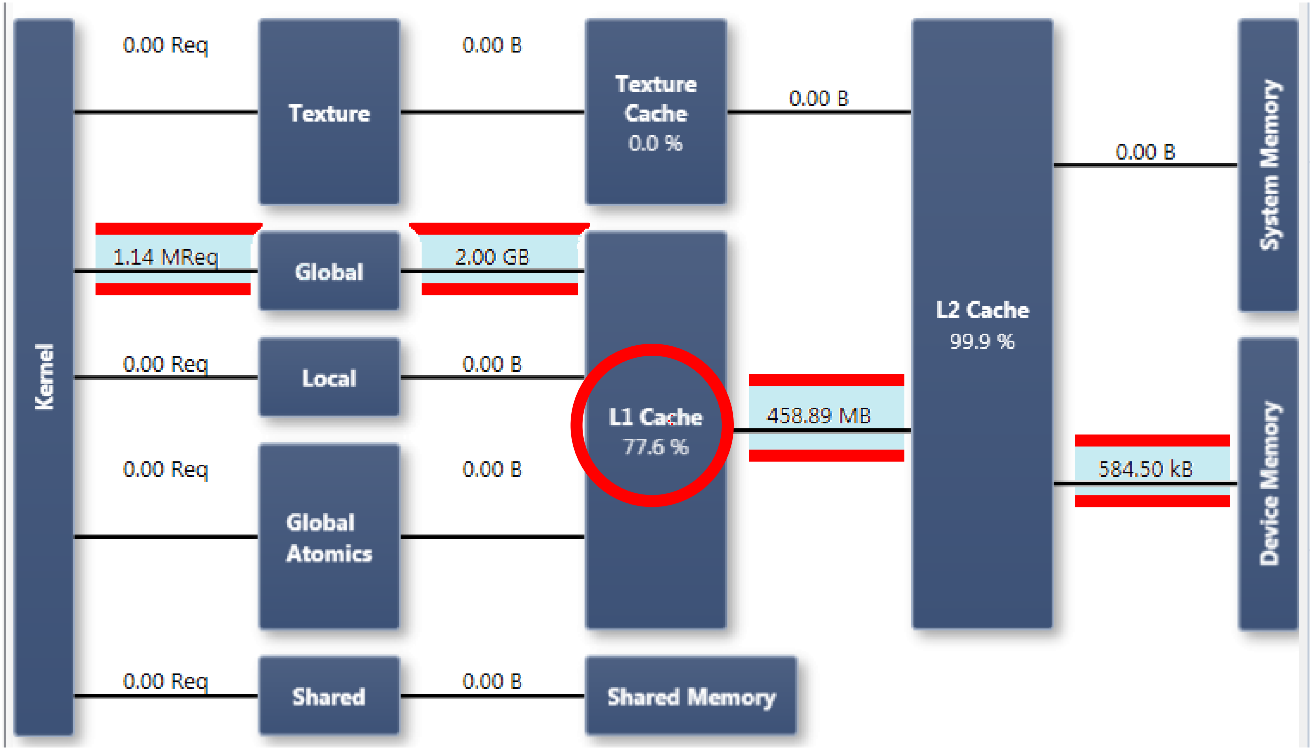


Kernel: evaluate_moves_kernel **Grid Dim:** {105, 1, 1} 105 **Block Dim:** {192, 1, 1} 192
Device: GeForce GTX 480 **Compute Capability:** 2.0 **Dyn Shm/Block:** 0 **Stat Shm/Block:** 0

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		7	8
Active Warps	31.39	42	48
Active Threads		1344	1536
Occupancy	65.39 %	87.50 %	100.00 %
Warps			
Threads/Block		192	1024
Warps/Block		6	32
Block Limit		8	8
Registers			
Registers/Thread		22	63
Registers/Block		4608	32768
Block Limit		7	8
Shared Memory			
Shared Memory/Block		0	49152
Block Limit		∞	8

Profiling the Evaluation Kernel

Memory Statistics Experiment



- Occupancy good
- L1 Cache Hit Rate good, but could be better
- L1 Cache: 16 kB (now) or 48 kB

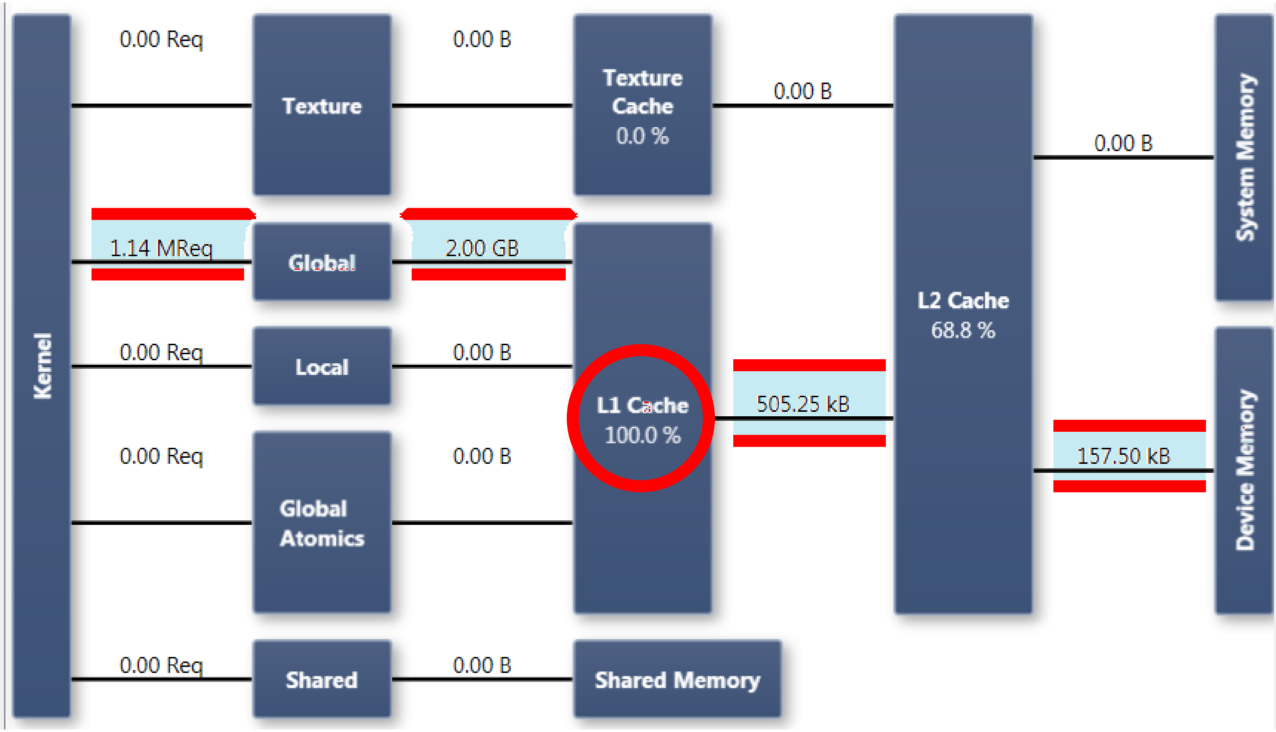
  => Set to 48 kB

```
// Use big L1 cache for move evaluation
err = cudaFuncSetCacheConfig(evaluate_moves_kernel, cudaFuncCachePreferL1);
```

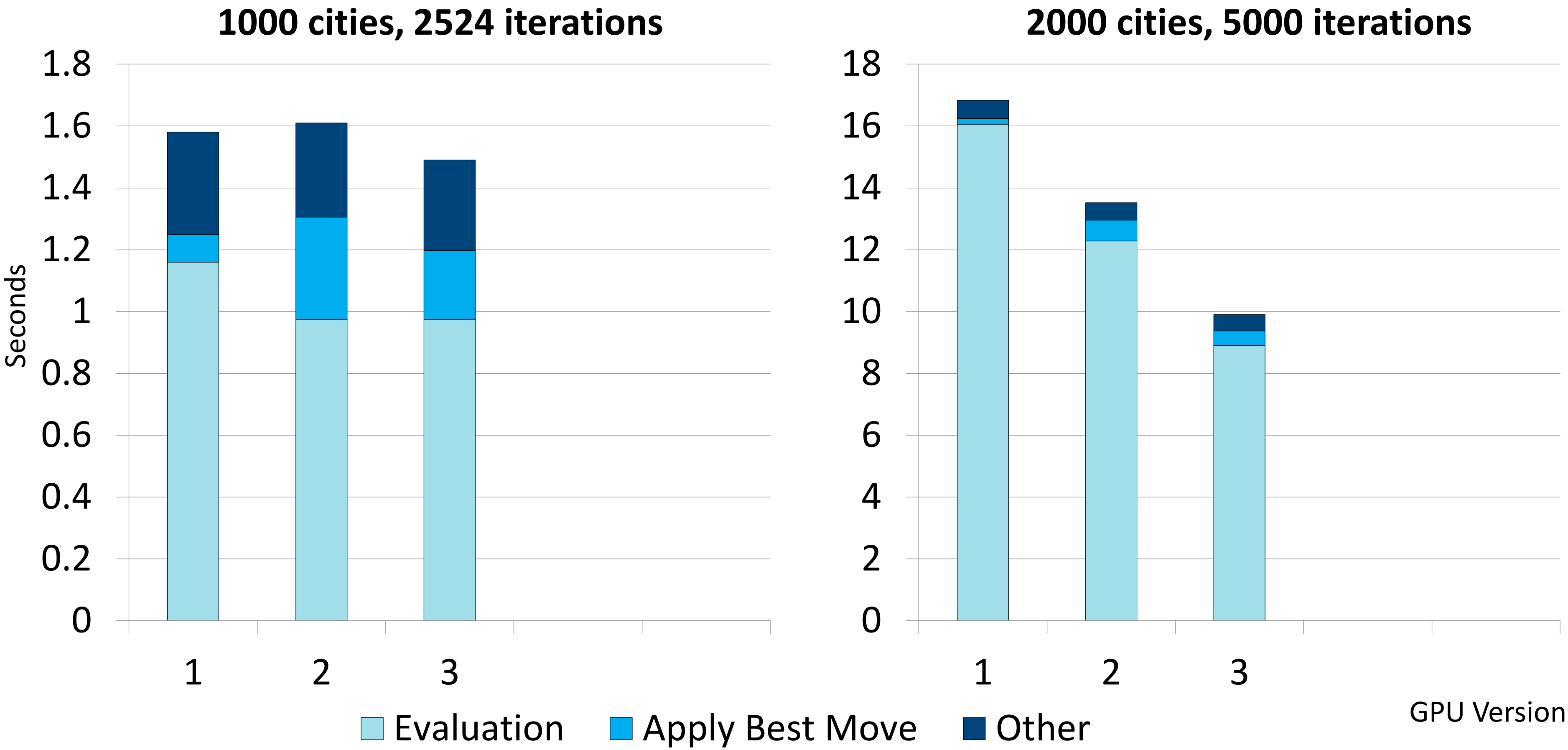
Profiling the Evaluation Kernel

Memory Statistics Experiment - Overview

- L1 Hit Rate perfect (99.98%)



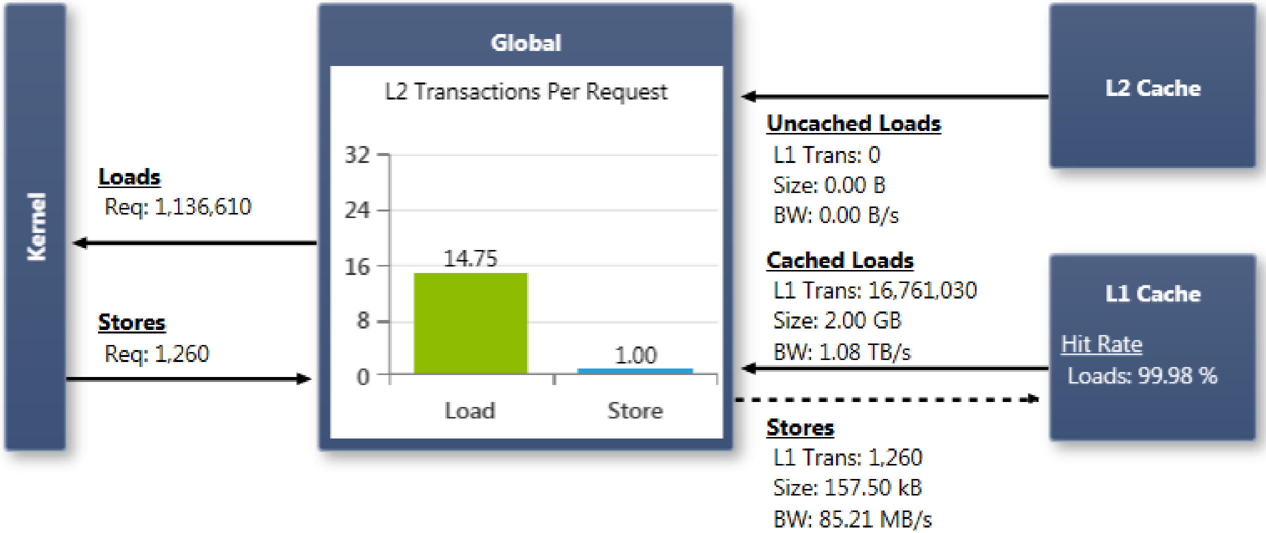
Timing the GPU Version



Profiling the Evaluation Kernel

- L1 Hit Rate perfect (99.98%)
- More details on global memory access yield
- Many transactions per Request
- Remember: Data is read a cache-line at a time

Memory Statistics Experiment - Global



Loads
Req: 1,136,610

Stores
Req: 1,260

L2 Transactions Per Request

	Load	Store
L2 Transactions Per Request	14.75	1.00

Uncached Loads
L1 Trans: 0
Size: 0.00 B
BW: 0.00 B/s

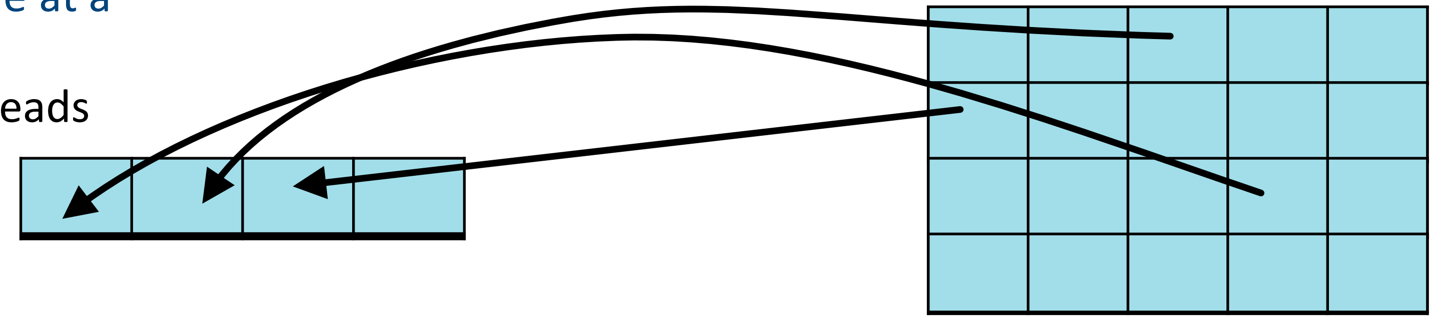
Cached Loads
L1 Trans: 16,761,030
Size: 2.00 GB
BW: 1.08 TB/s

Stores
L1 Trans: 1,260
Size: 157.50 kB
BW: 85.21 MB/s

Hit Rate
Loads: 99.98 %

Cache – 4 lines a 5 elements

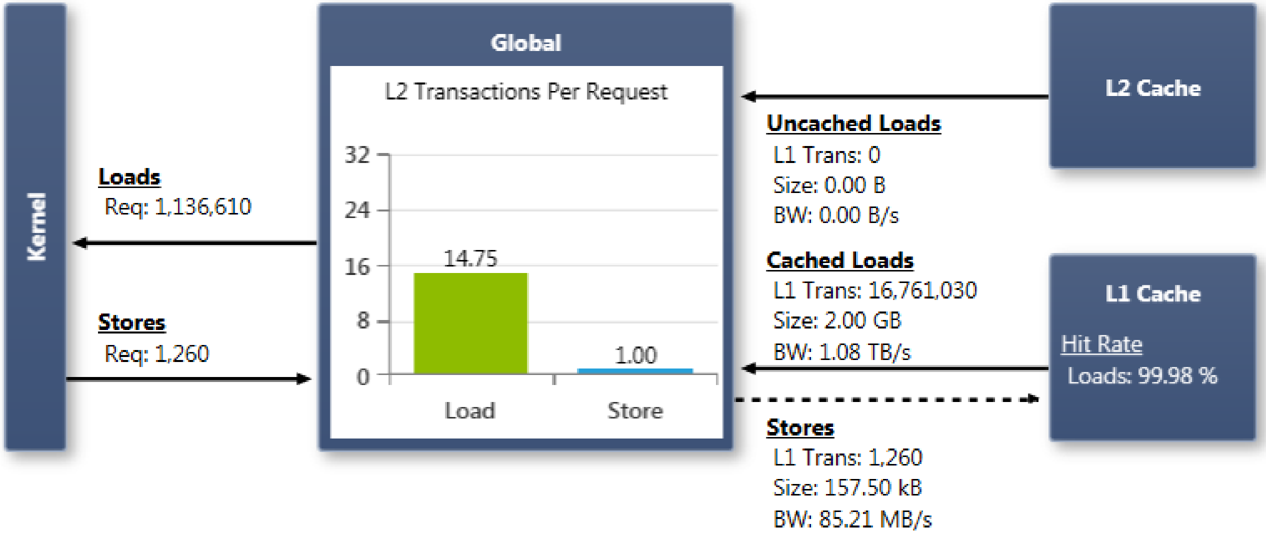
Threads



Profiling the Evaluation Kernel

- L1 Hit Rate perfect (99.98%)
- More details on global memory access yield
- Many transactions per Request
- Remember: Data is read a cache-line at a time

Memory Statistics Experiment - Global



Kernel

Loads
Req: 1,136,610

Stores
Req: 1,260

Global

L2 Transactions Per Request

	Load	Store
	14.75	1.00

L2 Cache

Uncached Loads
L1 Trans: 0
Size: 0.00 B
BW: 0.00 B/s

Cached Loads
L1 Trans: 16,761,030
Size: 2.00 GB
BW: 1.08 TB/s

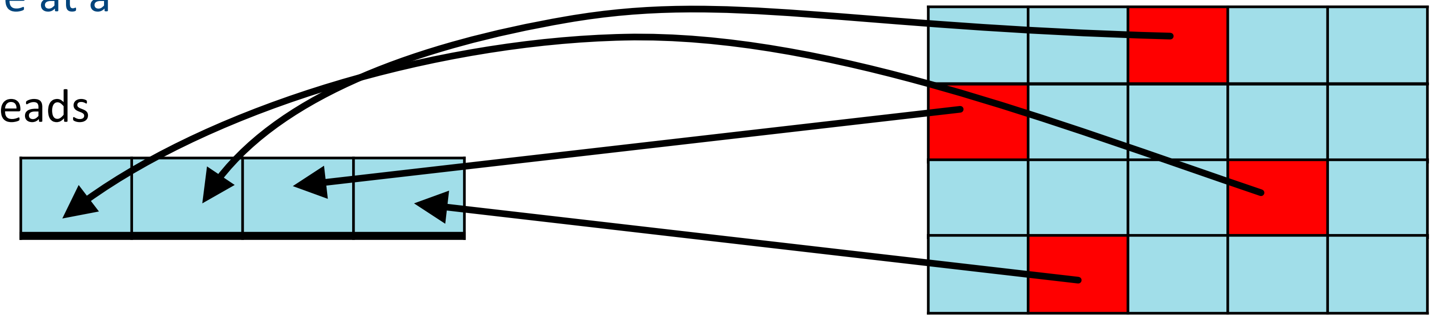
Stores
L1 Trans: 1,260
Size: 157.50 kB
BW: 85.21 MB/s

L1 Cache

Hit Rate
Loads: 99.98 %

Cache – 4 lines a 5 elements

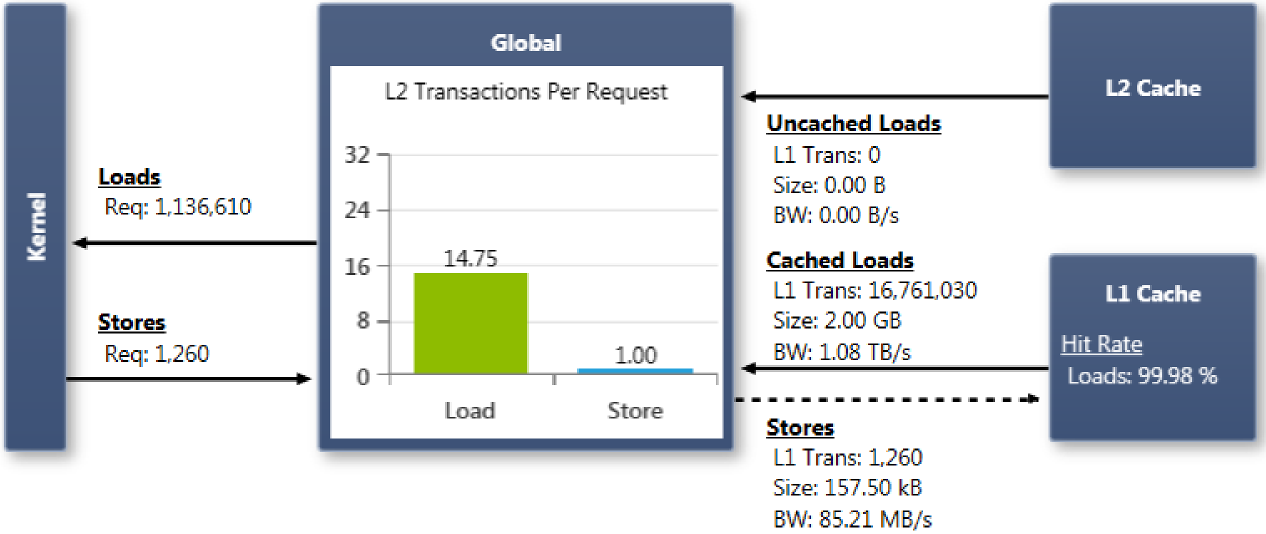
Threads



Profiling the Evaluation Kernel

- L1 Hit Rate perfect (99.98%)
- More details on global memory access yield
- Many transactions per Request
- Remember: Data is read a cache-line at a time

Memory Statistics Experiment - Global



Kernel

Loads
Req: 1,136,610

Stores
Req: 1,260

Global

L2 Transactions Per Request

L2 Cache

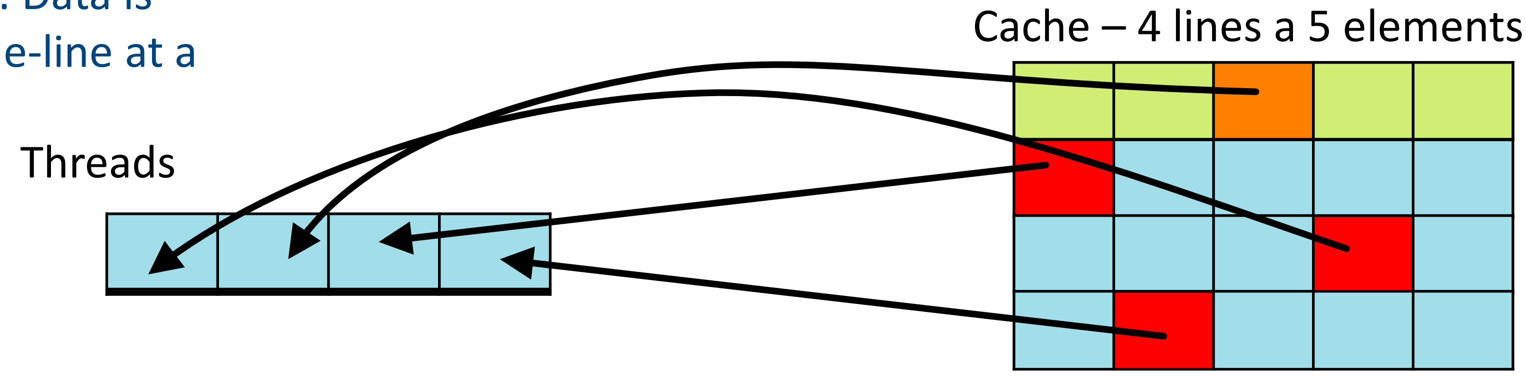
Uncached Loads
L1 Trans: 0
Size: 0.00 B
BW: 0.00 B/s

Cached Loads
L1 Trans: 16,761,030
Size: 2.00 GB
BW: 1.08 TB/s

L1 Cache

Hit Rate
Loads: 99.98 %

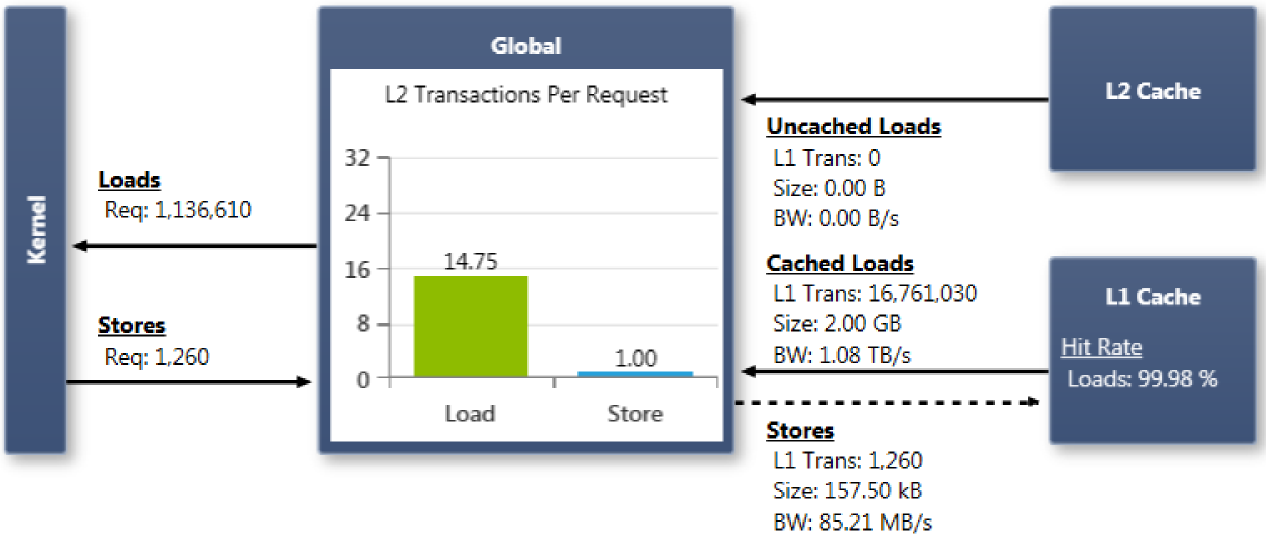
Stores
L1 Trans: 1,260
Size: 157.50 kB
BW: 85.21 MB/s



Profiling the Evaluation Kernel

- L1 Hit Rate perfect (99.98%)
- More details on global memory access yield
- Many transactions per Request
- Remember: Data is read a cache-line at a time

Memory Statistics Experiment - Global



Kernel

Loads
Req: 1,136,610

Stores
Req: 1,260

Global

L2 Transactions Per Request

L2 Cache

Uncached Loads
L1 Trans: 0
Size: 0.00 B
BW: 0.00 B/s

Cached Loads
L1 Trans: 16,761,030
Size: 2.00 GB
BW: 1.08 TB/s

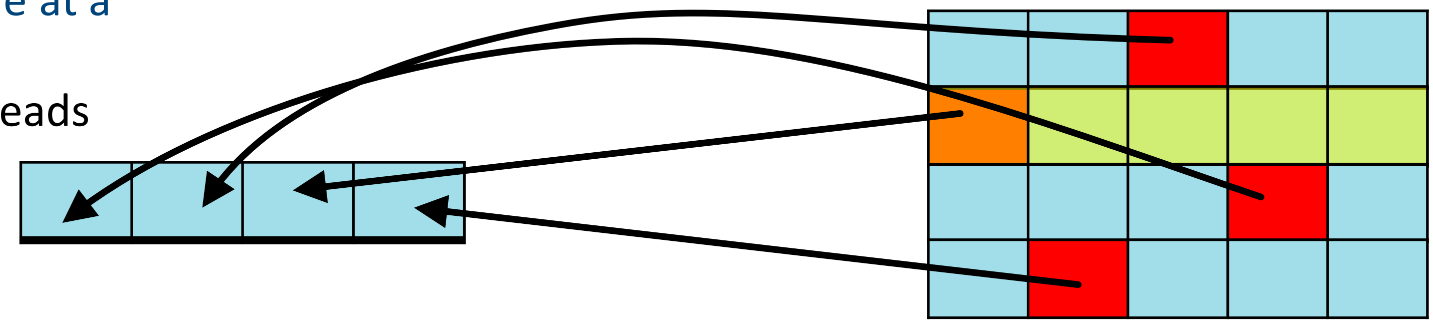
Stores
L1 Trans: 1,260
Size: 157.50 kB
BW: 85.21 MB/s

L1 Cache

Hit Rate
Loads: 99.98 %

Cache – 4 lines a 5 elements

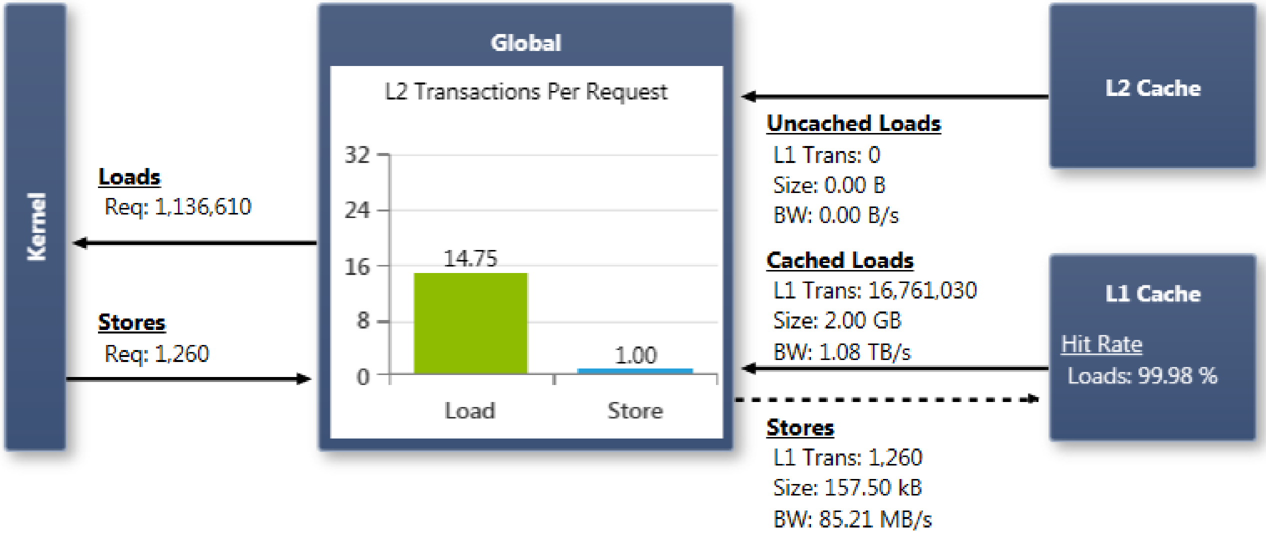
Threads



Profiling the Evaluation Kernel

- L1 Hit Rate perfect (99.98%)
- More details on global memory access yield
- Many transactions per Request
- Remember: Data is read a cache-line at a time

Memory Statistics Experiment - Global



Kernel

Loads
Req: 1,136,610

Stores
Req: 1,260

Global

L2 Transactions Per Request

L2 Cache

Uncached Loads
L1 Trans: 0
Size: 0.00 B
BW: 0.00 B/s

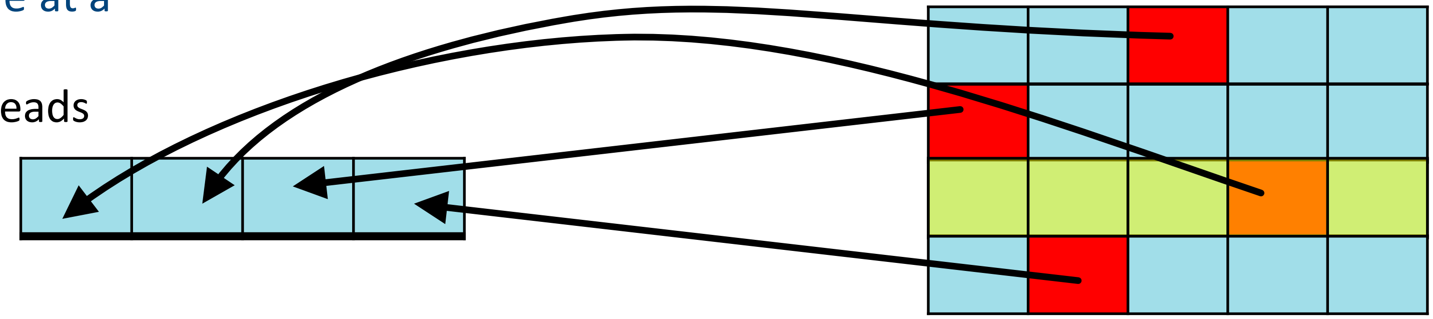
Cached Loads
L1 Trans: 16,761,030
Size: 2.00 GB
BW: 1.08 TB/s

L1 Cache
Hit Rate
Loads: 99.98 %

Stores
L1 Trans: 1,260
Size: 157.50 kB
BW: 85.21 MB/s

Cache – 4 lines a 5 elements

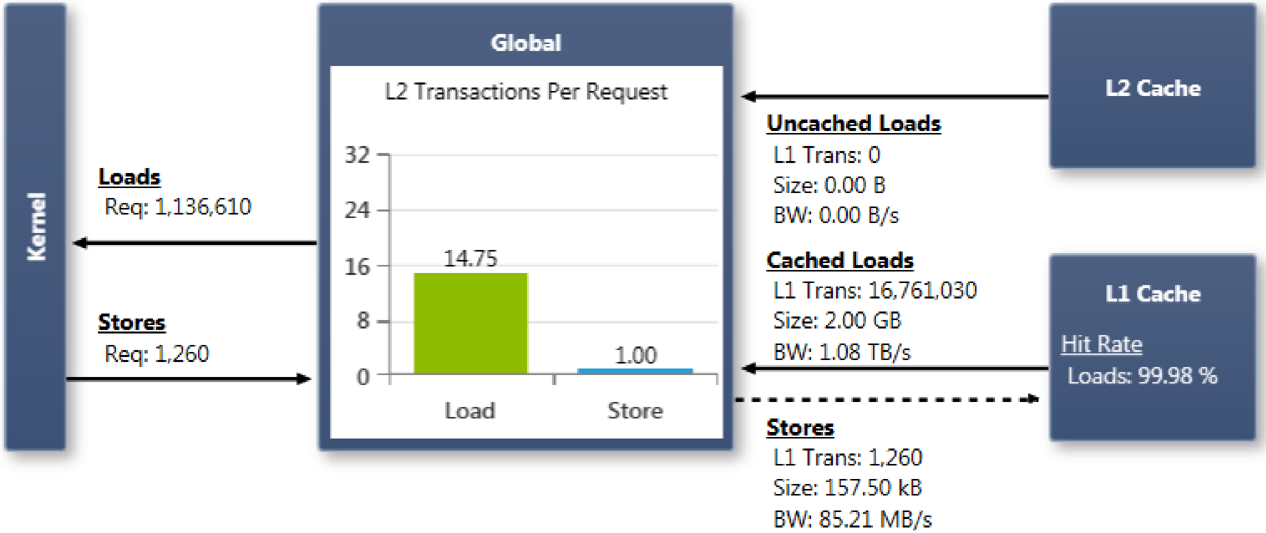
Threads



Profiling the Evaluation Kernel

- L1 Hit Rate perfect (99.98%)
- More details on global memory access yield
- Many transactions per Request
- Remember: Data is read a cache-line at a time

Memory Statistics Experiment - Global



Kernel

Loads
Req: 1,136,610

Stores
Req: 1,260

Global

L2 Transactions Per Request

	Load	Store
	14.75	1.00

L2 Cache

Uncached Loads
L1 Trans: 0
Size: 0.00 B
BW: 0.00 B/s

Cached Loads
L1 Trans: 16,761,030
Size: 2.00 GB
BW: 1.08 TB/s

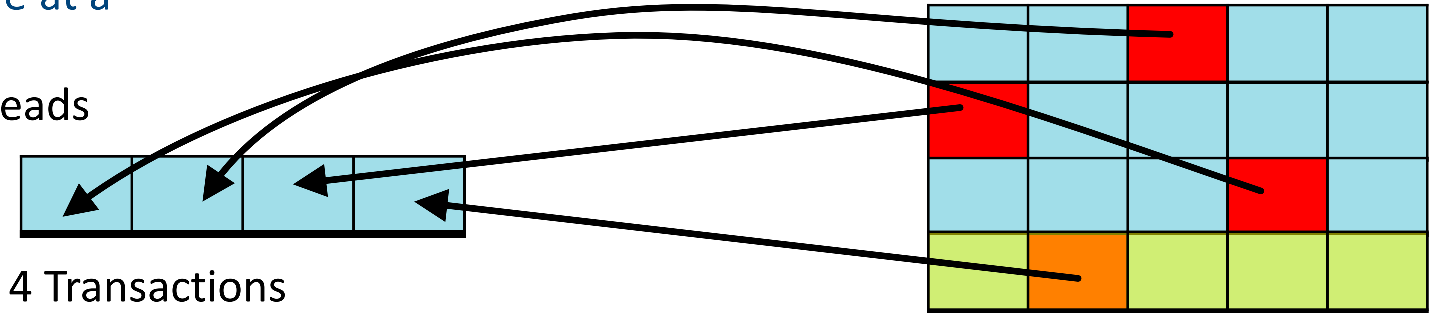
Stores
L1 Trans: 1,260
Size: 157.50 kB
BW: 85.21 MB/s

L1 Cache

Hit Rate
Loads: 99.98 %

Cache – 4 lines a 5 elements

Threads

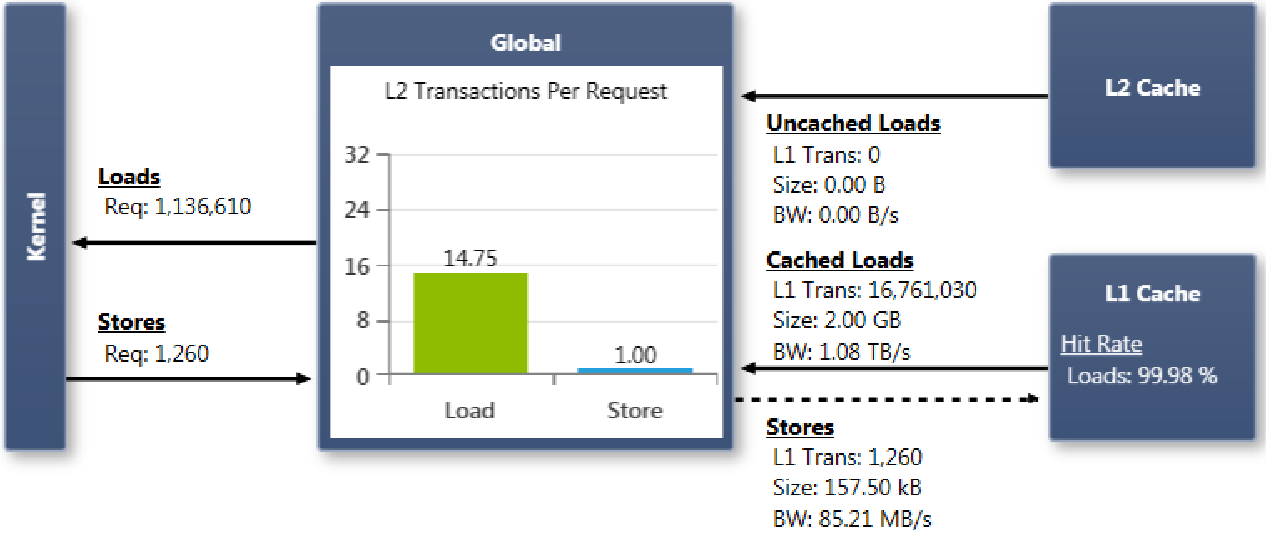


=> 4 Transactions

Profiling the Evaluation Kernel

- L1 Hit Rate perfect (99.98%)
- More details on global memory access yield
- Many transactions per Request
- Remember: Data is read a cache-line at a time

Memory Statistics Experiment - Global


Kernel

Loads
Req: 1,136,610

Stores
Req: 1,260

Global

L2 Transactions Per Request

L2 Cache

Uncached Loads
L1 Trans: 0
Size: 0.00 B
BW: 0.00 B/s

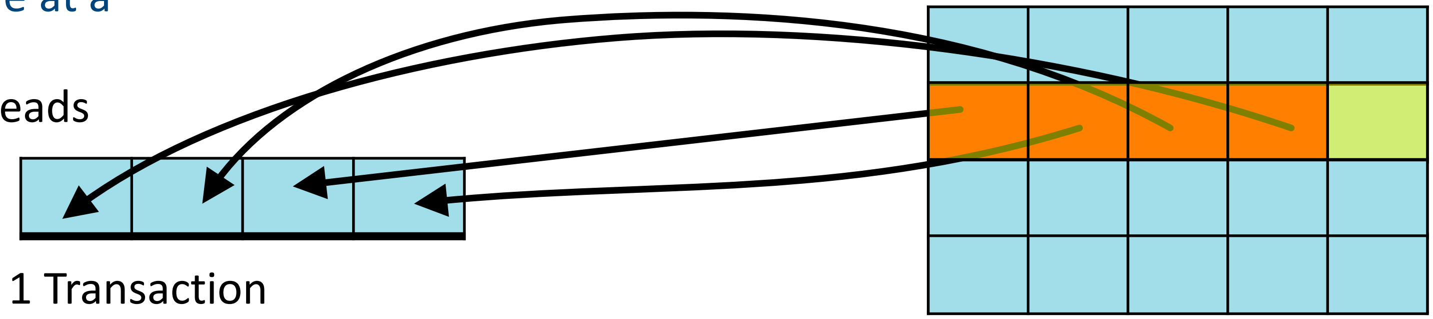
Cached Loads
L1 Trans: 16,761,030
Size: 2.00 GB
BW: 1.08 TB/s

L1 Cache

Hit Rate
Loads: 99.98 %

Stores
L1 Trans: 1,260
Size: 157.50 kB
BW: 85.21 MB/s

Cache – 4 lines a 5 elements

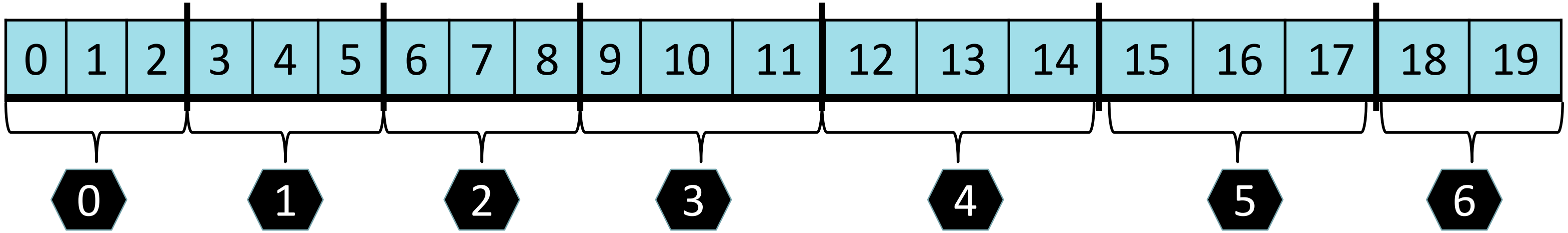

Threads

=> 1 Transaction

Profiling the Evaluation Kernel

- Remember how we split loop through moves:

```
for (int i=first_move; i<first_move+num_moves_per_thread_; ++i)
{... }
```



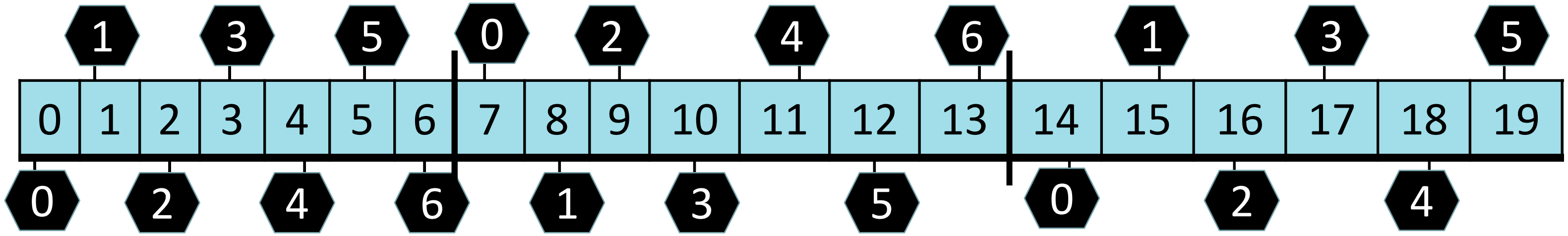
- Moves i and i+1 probably executed by same thread
- Moves i and i+1 are likely to have same first city and neighbouring second city

  => accessing same / neighbouring entries in solution array
- At each iteration thread 0 and thread 1 access i and i+num_moves_per_thread simultaniously

=> Change splitting of loop such that thread 0 and 1 access move i and i+1

Profiling the Evaluation Kernel

- Splitting of loop such that thread 0 and 1 access move i and i+1

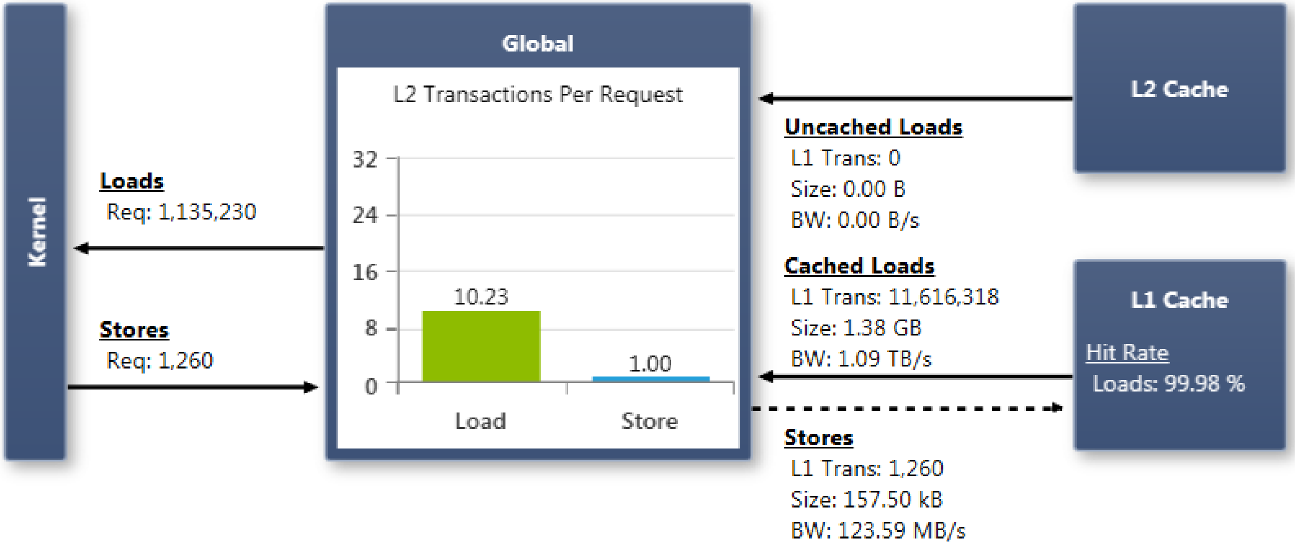


```
unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
const unsigned int grid_size = gridDim.x * blockDim.x;
for (int i = tid; i < num_moves; i += grid_size) {
   ...
}
```

Removes also need for extra check `i < num_moves`

Profiling the Evaluation Kernel

- Number of Transactions reduced
- Still relatively high
- Problem: “Random” access of coordinates (access through a permutation)
- Limited possibilities to improve memory access



Loads
Req: 1,135,230

Stores
Req: 1,260

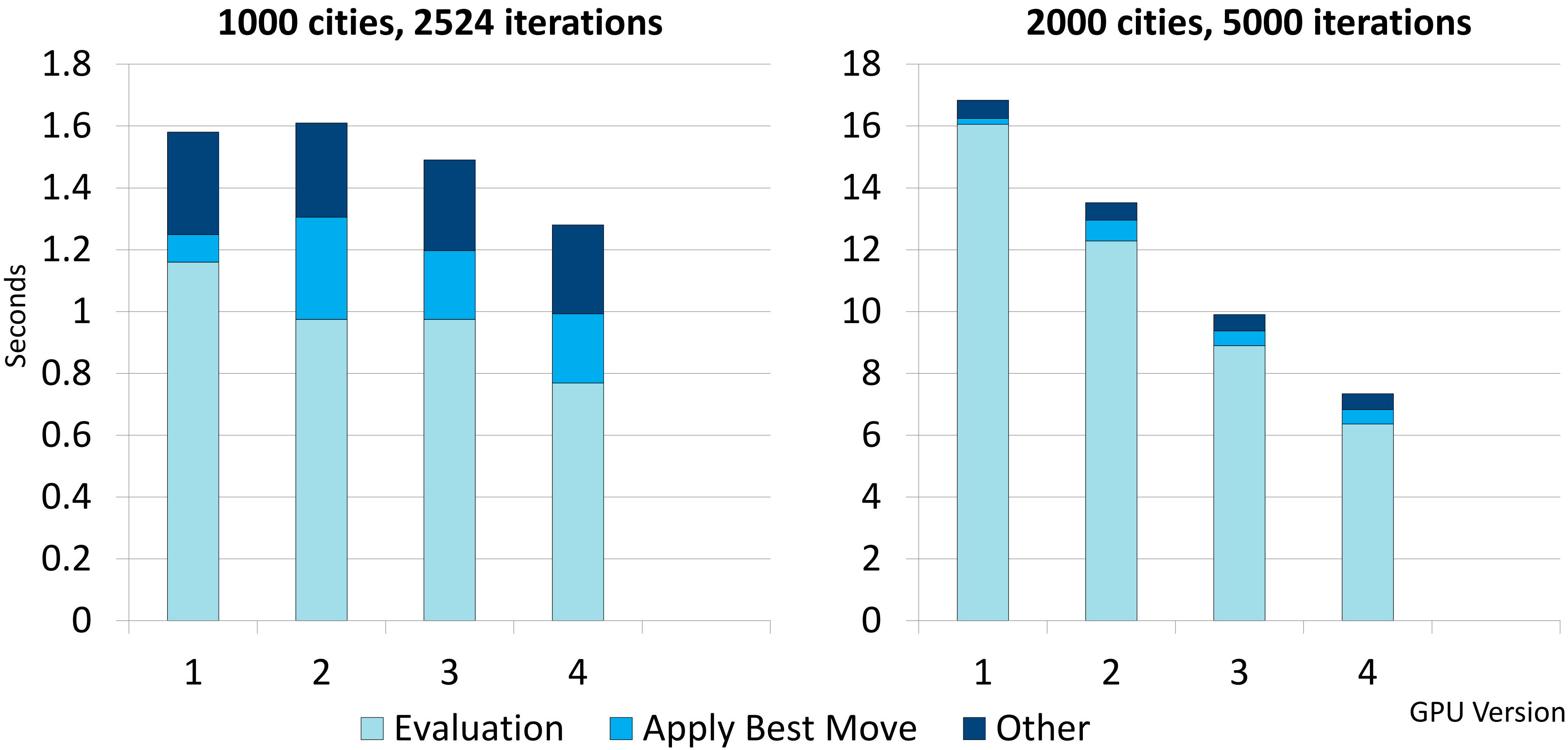
Uncached Loads
L1 Trans: 0
Size: 0.00 B
BW: 0.00 B/s

Cached Loads
L1 Trans: 11,616,318
Size: 1.38 GB
BW: 1.09 TB/s

Stores
L1 Trans: 1,260
Size: 157.50 kB
BW: 123.59 MB/s

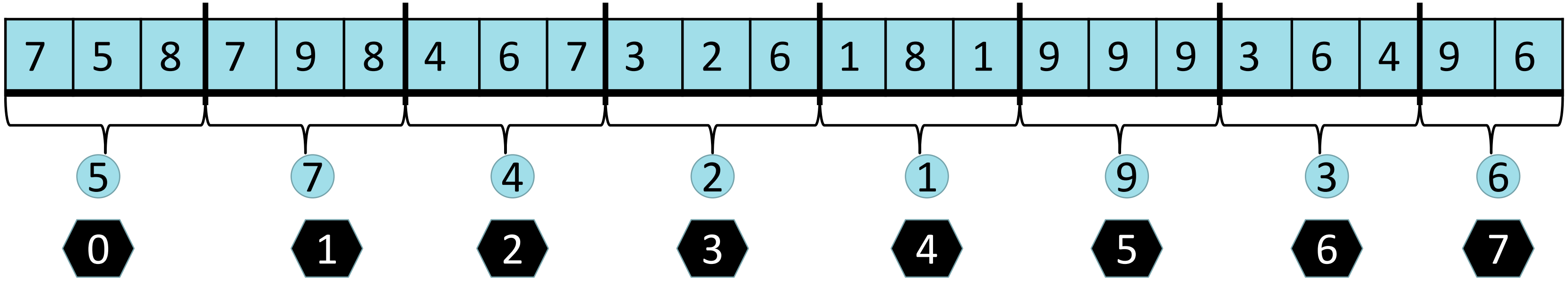
Hit Rate
Loads: 99.98 %

Timing the GPU Version



Improved Reduction

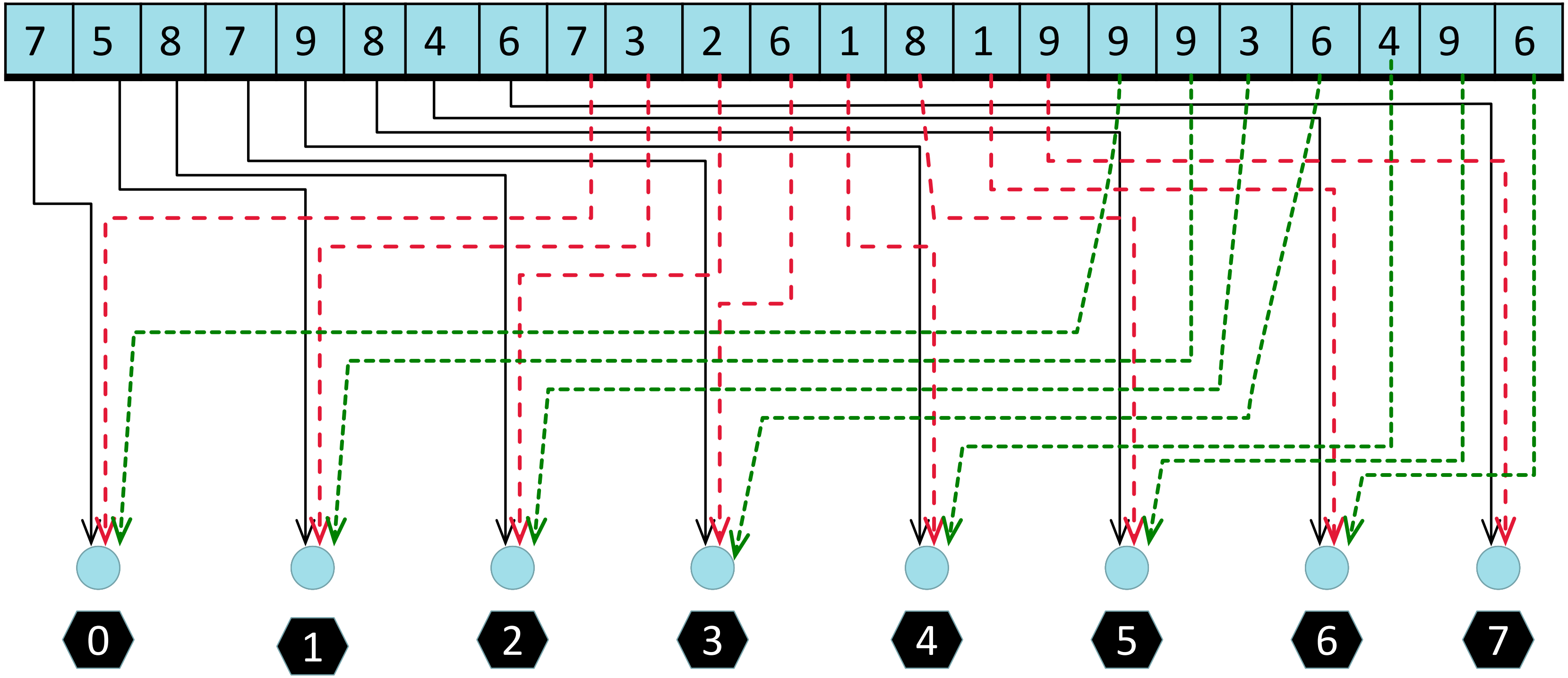
- Remember, we had equally bad access pattern in first step of reduction:



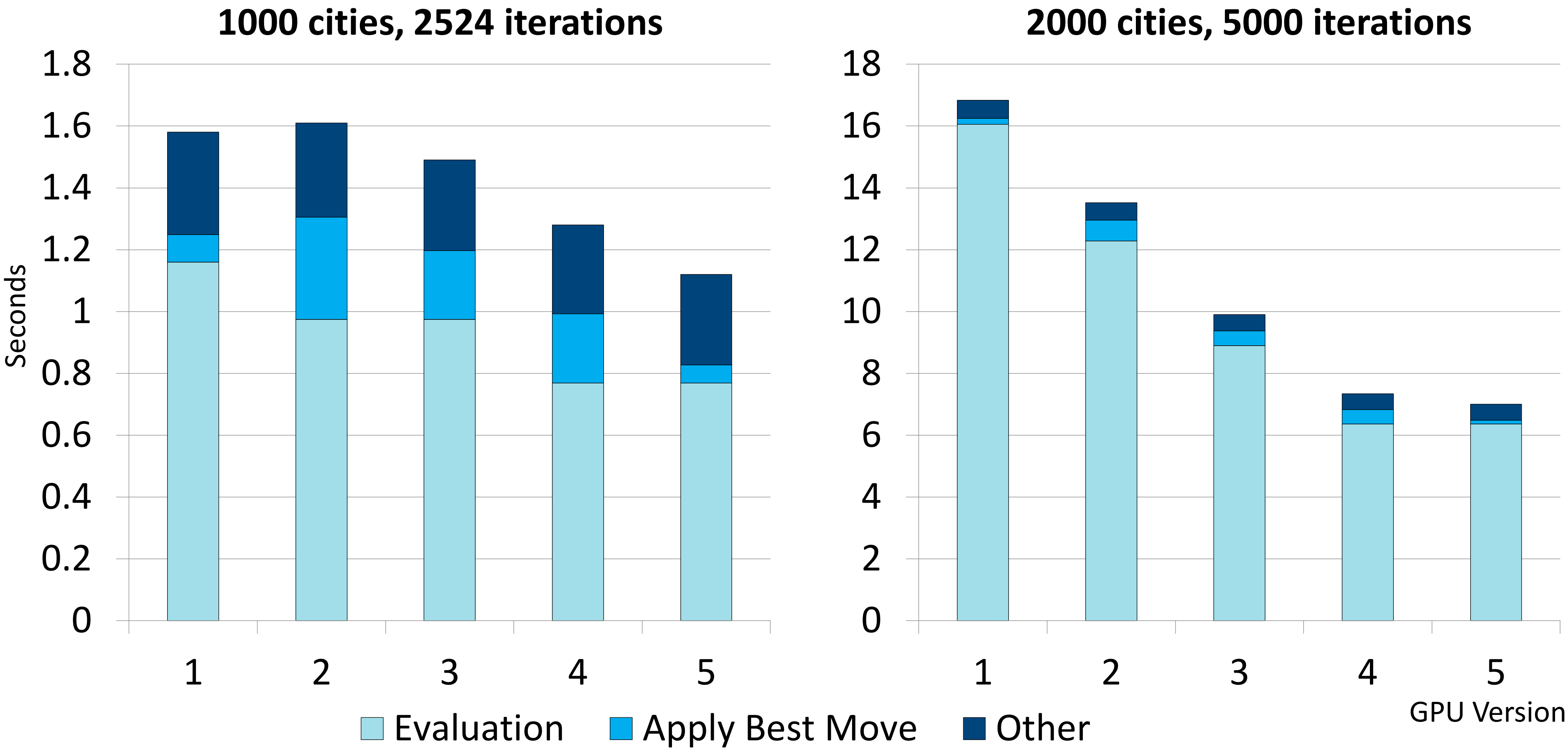
- Minimum is commutative => can change access order

Improved Reduction

- Remember, we had equally bad access pattern in first step of reduction:
- Minimum is commutative => can change access order



Timing the GPU Version



Filtering the Neighborhood

Filtering Moves

- Filtering often used on sequential code to reduce number of moves to evaluate
- How does it perform on GPU?
- Simulate filtering by random filter array

```
__global__ void evaluate_moves_kernel(...) {
    ... // setup
    // Find best move in thread
    for (int i = tid; i < num_moves; i += grid_size) {
        Move move = generate_move(i, num_nodes_, solution_);
        float delta = get_delta(move, city_coordinates_);
        if (delta < min_delta) {
                min_delta = delta;
                best_move = i;
    }   }
    ... // store thread-best-delta and thread-best-move-id
}
```

Find thread best

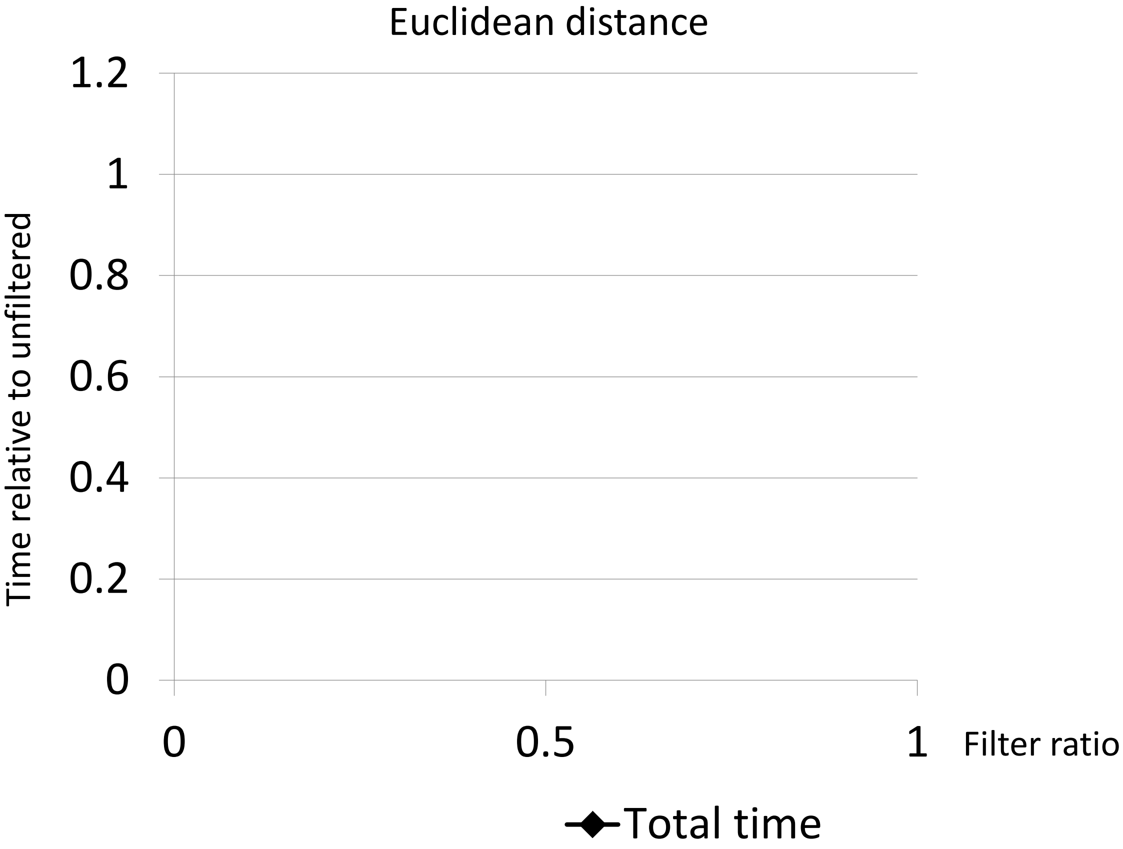
Filtering Moves

- Filtering often used on sequential code to reduce number of moves to evaluate
- How does it perform on GPU?
- Simulate filtering by random filter array

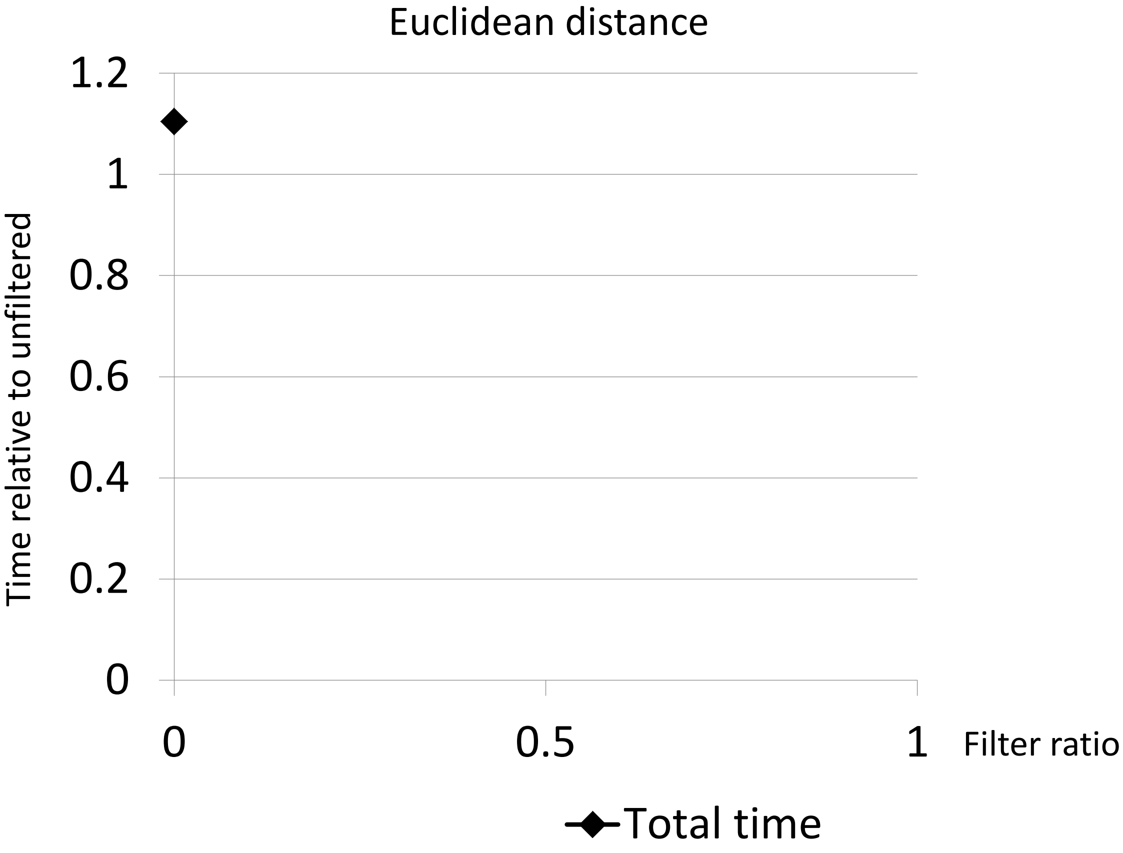
```
__global__ void evaluate_moves_kernel(...) {
    ... // setup
    // Find best move in thread
    for (int i = tid; i < num_moves; i += grid_size) {
        if (filter_[i])
            continue;
        Move move = generate_move(i, num_nodes_, solution_);
        float delta = get_delta(move, city_coordinates_);
        if (delta < min_delta) {
                min_delta = delta;
                best_move = i;
    }   }
    ... // store thread-best-delta and thread-best-move-id
}
```

Find thread best

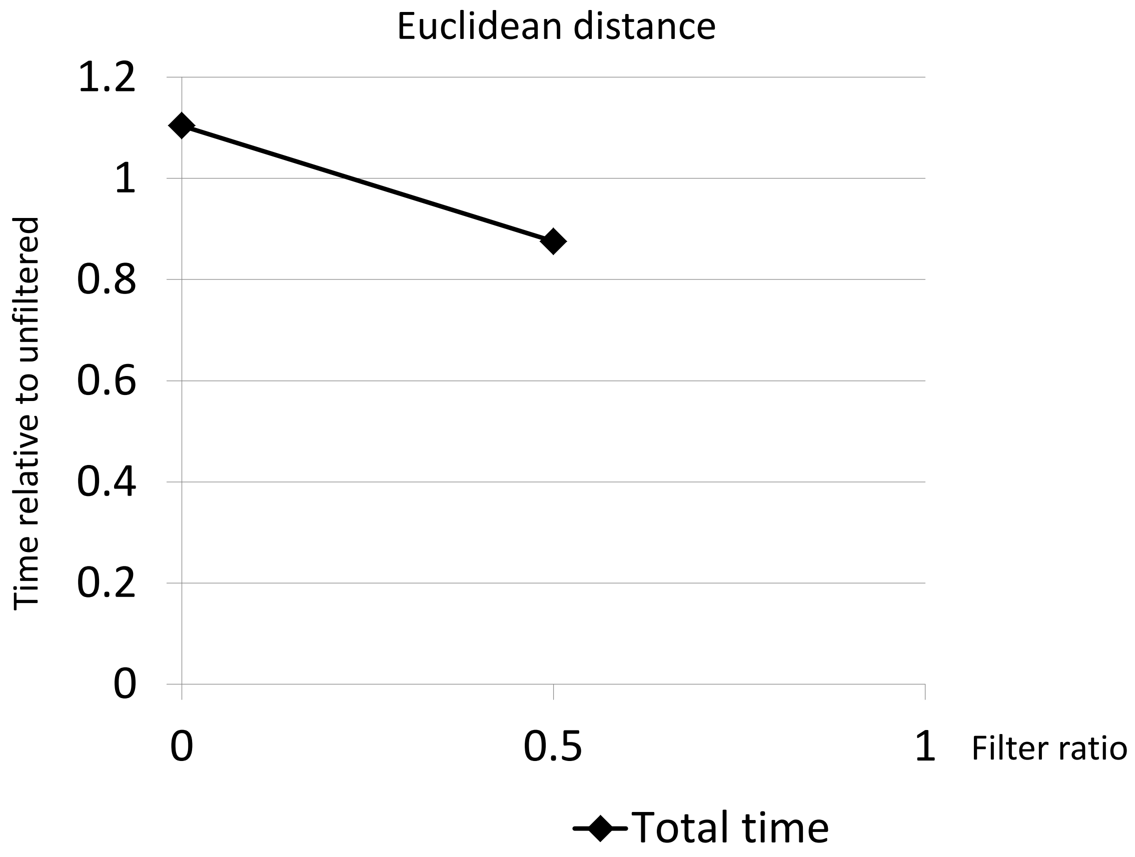
Filtering Moves Experiment for 2000 cities



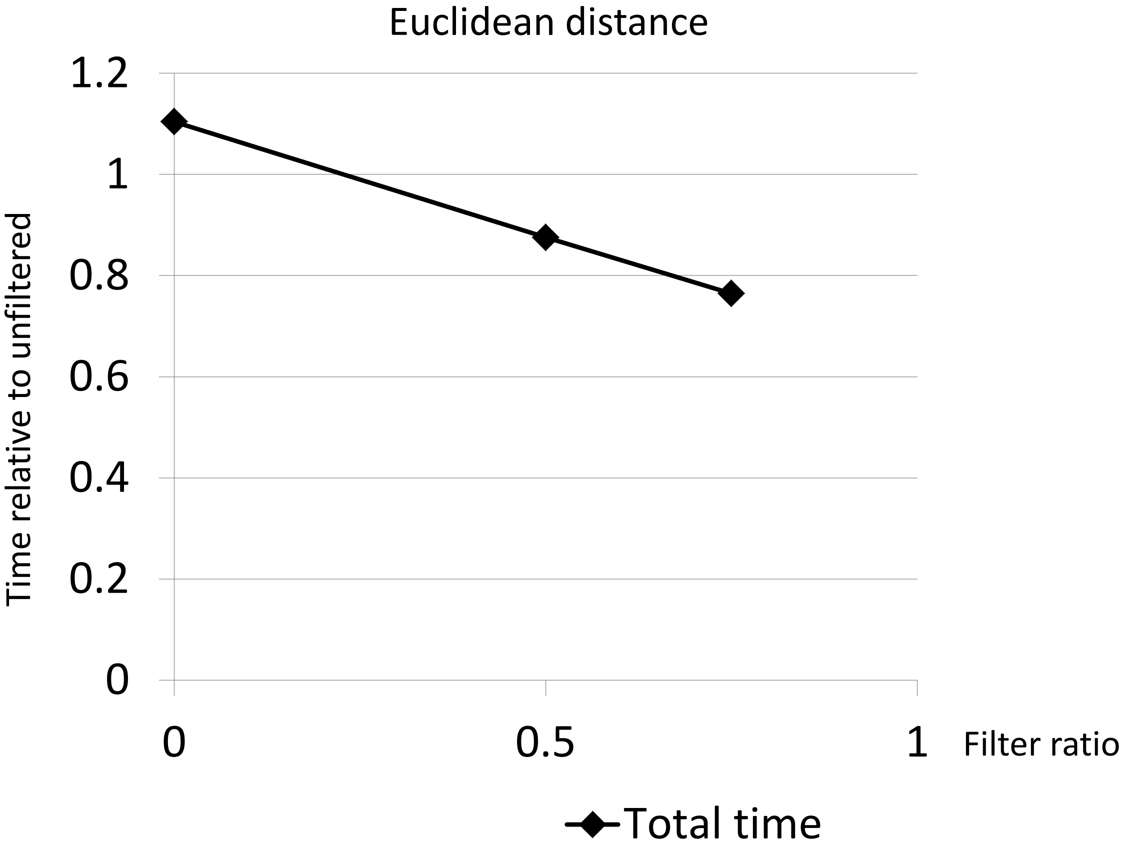
Filtering Moves Experiment for 2000 cities



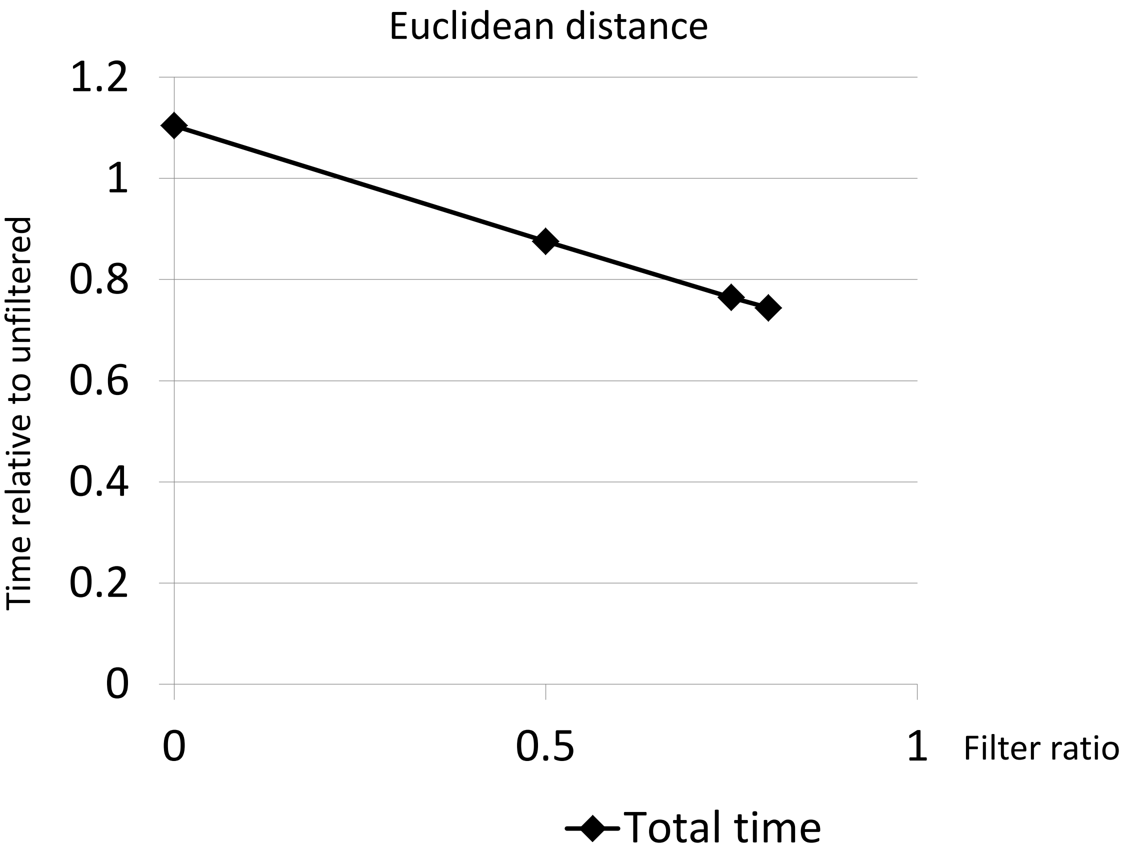
Filtering Moves Experiment for 2000 cities



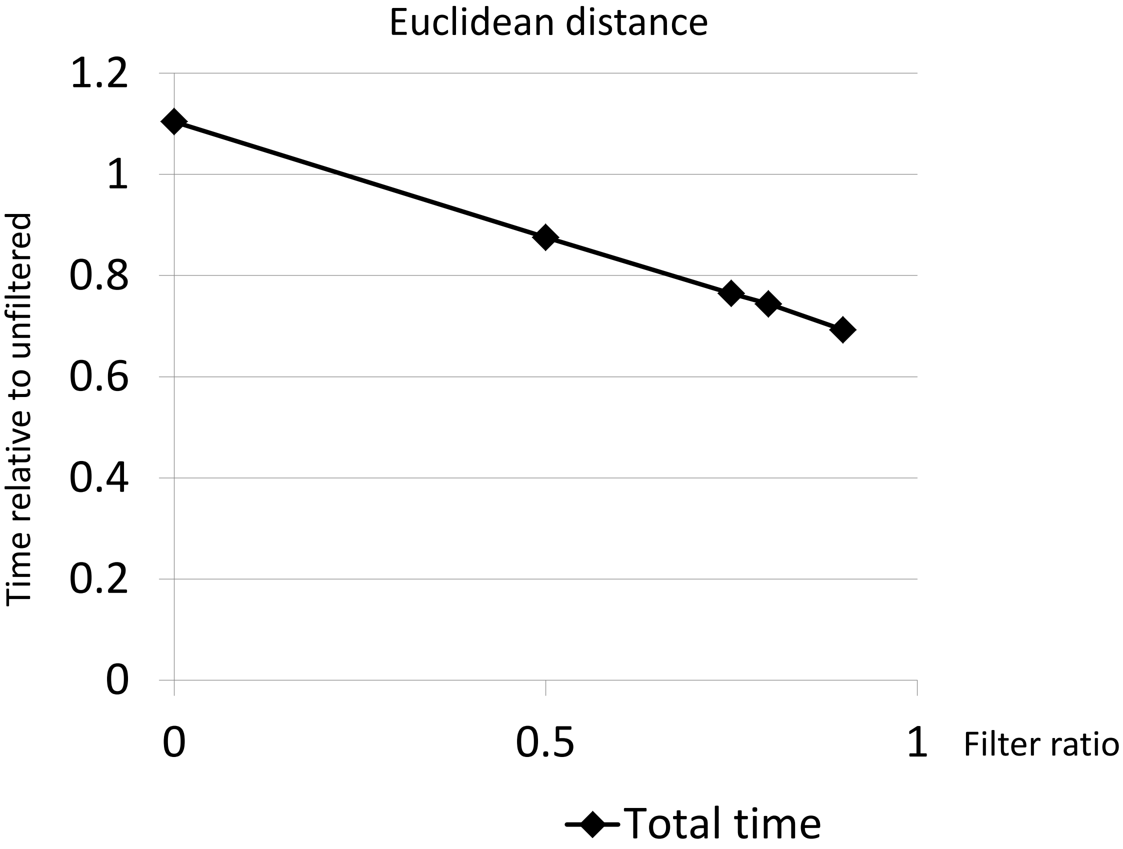
Filtering Moves Experiment for 2000 cities



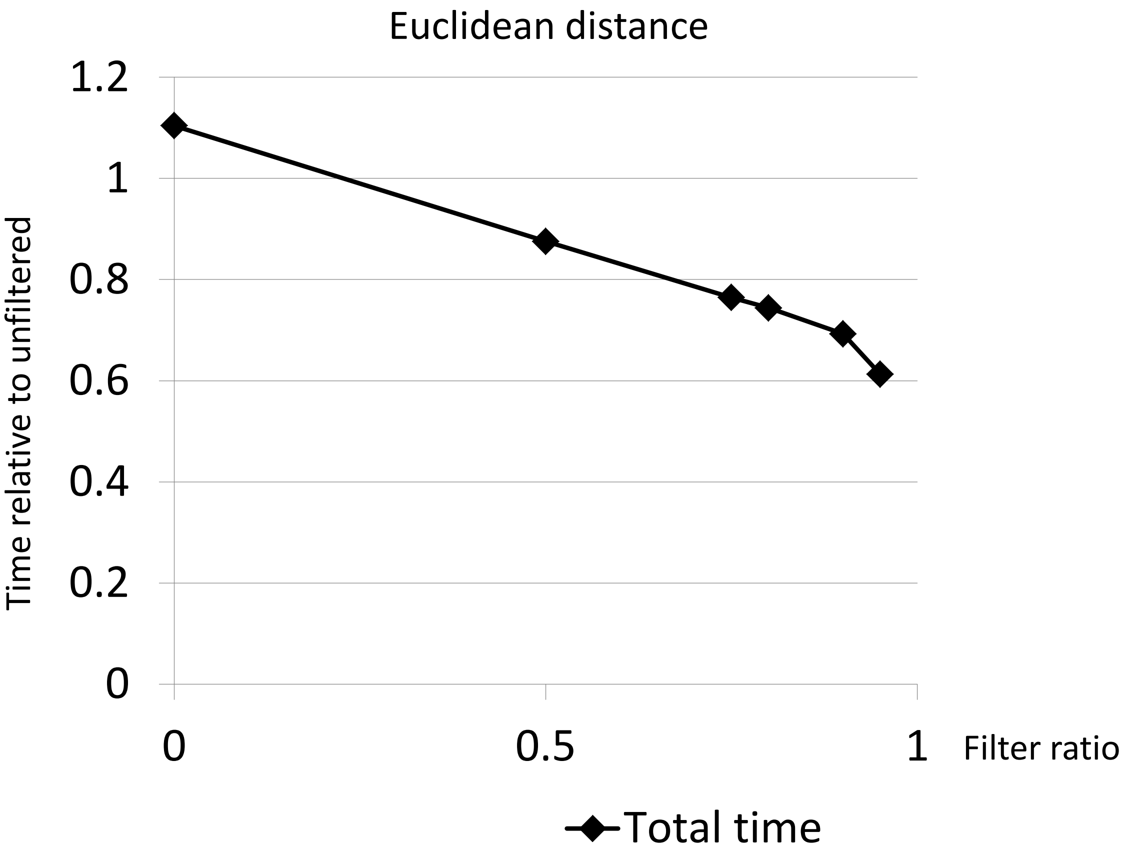
Filtering Moves Experiment for 2000 cities



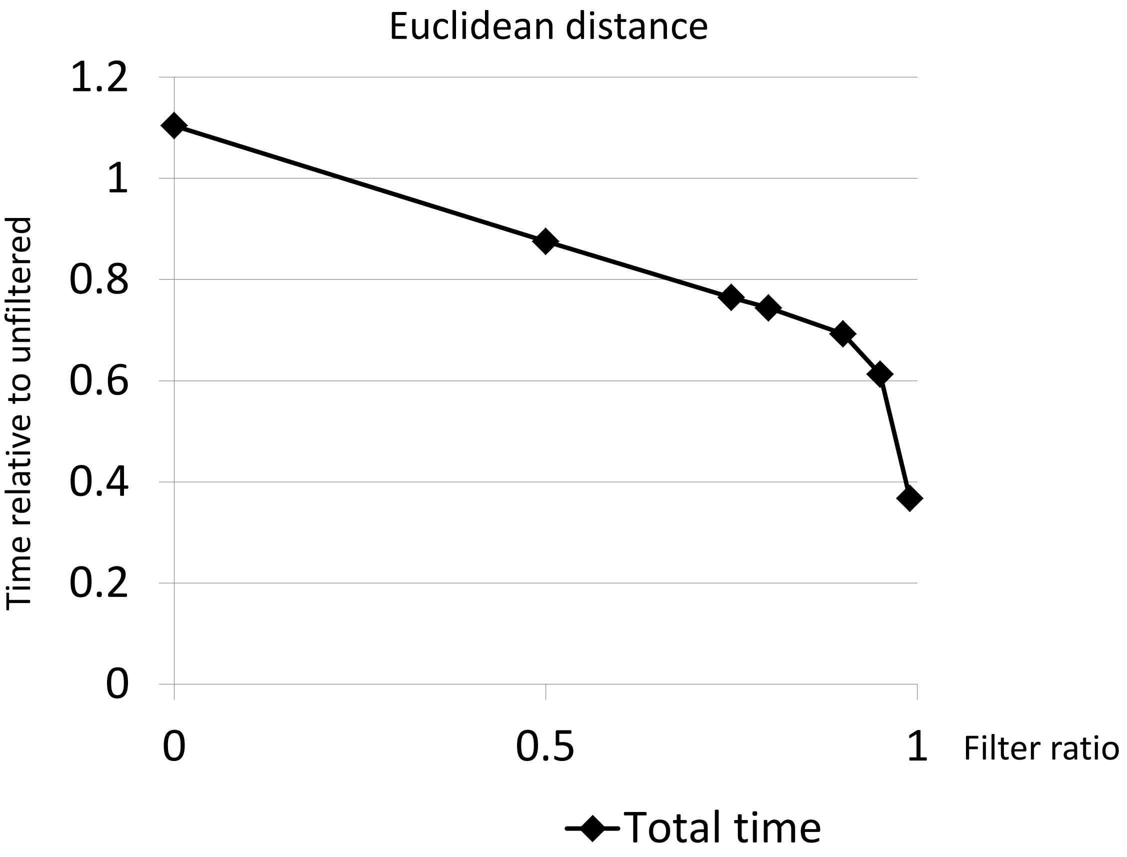
Filtering Moves Experiment for 2000 cities



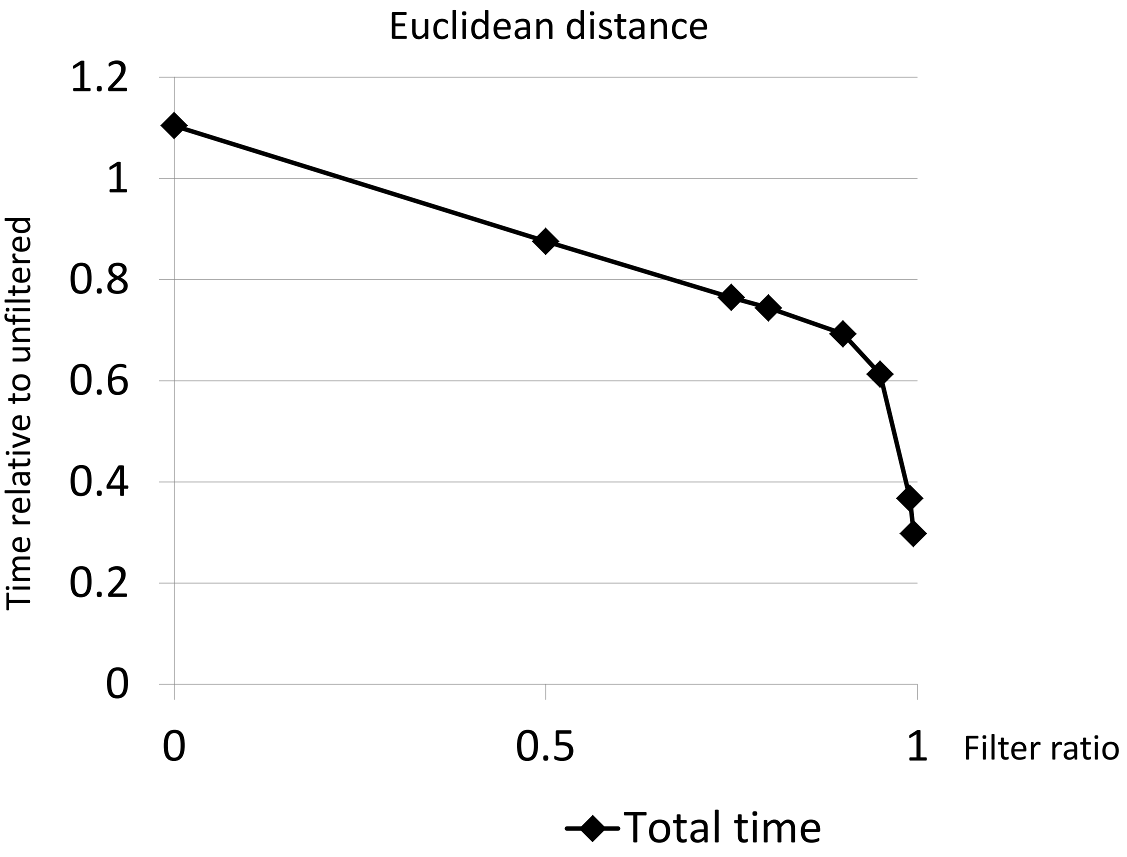
Filtering Moves Experiment for 2000 cities



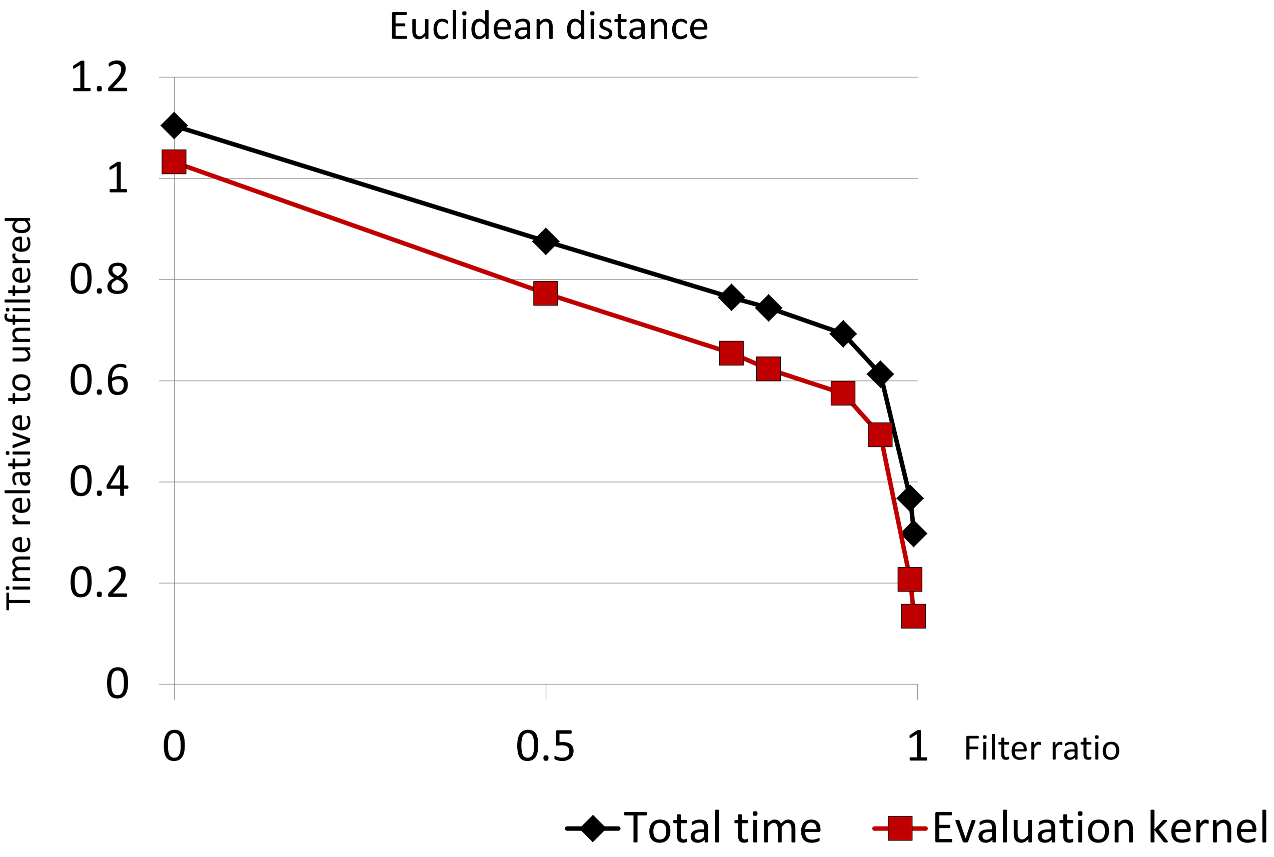
Filtering Moves Experiment for 2000 cities



Filtering Moves Experiment for 2000 cities

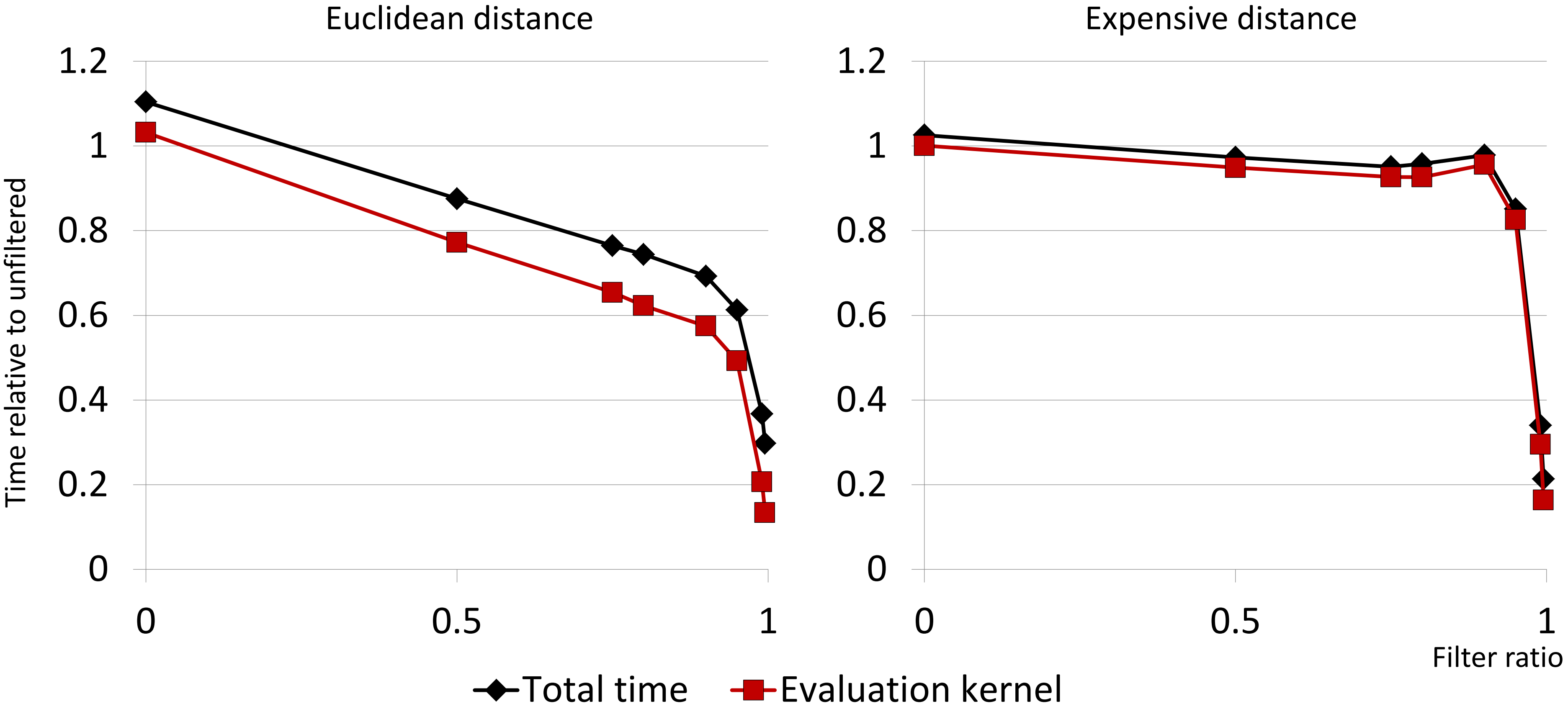


Filtering Moves Experiment for 2000 cities



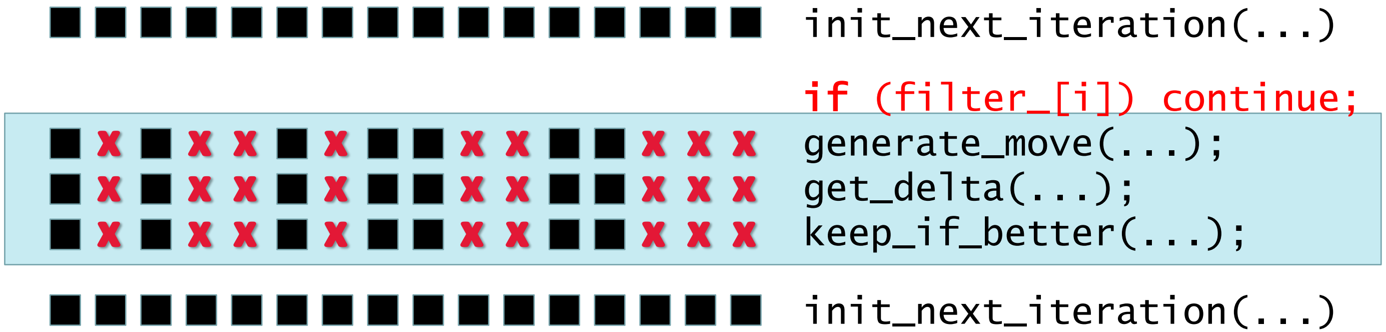
Filtering Moves Experiment for 2000 cities

Simulate more expensive moves



What happened?

- Inside a warp during the iteration: 32 threads, 32 moves



```
init_next_iteration(...)

if (filter_[i]) continue;
generate_move(...);
get_delta(...);
keep_if_better(...);

init_next_iteration(...)
```

- Masking leads to whole warp evaluating moves despite filtering
- 1 move per warp enough for whole warp to evaluate moves
- Less than 1/32 = 0.03125 remaining => whole warps jump iteration

Solution: Compaction

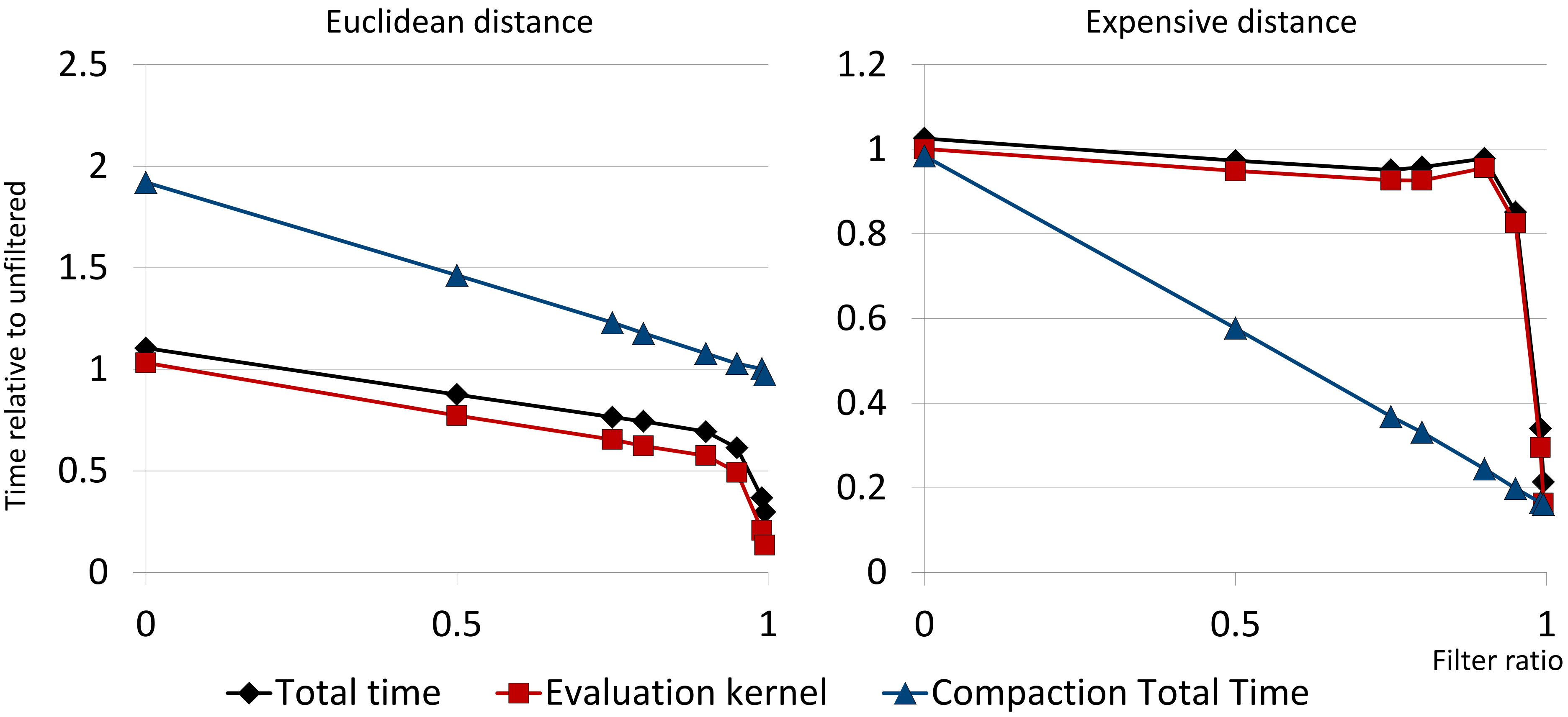
- Given: Array of moves and filter array
- Wanted: Array containing only moves to evaluate (not filtered)

Moves	0	1	2	3	4	5	6	7	8
Filter	1	0	0	1	1	0	1	1	0
Moves to evaluate	1	2	5	8					

- This type of operation is known as ***compaction***
- There exist efficient GPU algorithms for compaction
- Using compacted array:

  Whole warp either performs or skips move evaluation (except for 1 warp)

Filtering Moves Experiment continued



Lesson

- Techniques to reduce work load on CPU might not work on GPU
- Need to ensure enough parallelism available in problem and chosen algorithm
- Parallelism needs to be exploitable by GPU

GPU Computing in Routing Related Discrete Optimization Literature

Methods Implemented on GPU

- Evolutionary algorithms, Genetic algorithms
  - > 40 publications, ≥ 4 routing related
- Ant Colony Optimization
  - > 20 publications, ≥ 9 routing related
- Local Search
  - > 10 publications, ≥ 7 routing related
- Simulated annealing
  - ≥ 3 publications,
- Linear programming
  - ≥ 5 publications

Local Search on GPU in Literature (based on 6 publications)

- Neighbourhoods
  - Swap, relocate, 2-opt
  - 3-opt

Similar to what done in this tutorial
(move depends on x,y)

- Tasks performed on GPU

Move evaluation	6 papers
Best move selection	3 papers
Move application	3 papers (2 of those from move selection)

- Often missing / questionable reason for choice of tasks performed on GPU
- Measure used for justifying GPU usage: Speedup vs. CPU implementation
  - Often limited knowledge about CPU implementation
  - Only one paper specifies usage of more than 1 core on CPU
  - Solution quality not considered
  - Comparison with well known, efficient solvers missing, e.g. LKH2 for TSP

Usefulness of Local Search on GPU

- Usefulness of algorithmic approach not considered in most literature
- Example filtering:
  - Often used in sequential algorithms to reduce amount of work

    => fast & efficient algorithms
  - Filtering on GPU may not yield faster algorithms
- Example best improving:
  - On CPU often first improving is employed due to faster sequential performance
  - Is usage of best improving sensible?
    - Less of number of iterations?
    - Better solution quality?
    - A GPU best improving iteration faster than a CPU first improving iteration?
- Good example of usage of best improving knowledge:

  Burke and Riise, On Parallel Local Search for Permutations
  - perform all independent, improving 2-opts found in iteration => Less number of iterations

Ant Colony Optimization for TSP on GPU (based on 7 publications)

- Ant Colony Optimization for TSP
  - N ants, each ant builds tour guided by edge cost and edge attraction (pheromones)
  - Edge attraction (pheromones) updated by ants after tour construction
  - Repeat
- Tour construction most time consuming task
  - Implemented on GPU by all 7 papers
- Ants are independent => Each thread computes tour construction for 1 ant
  - Need many ants to fill GPU
  - Such many ants beneficial to method?
  - Above issues mentioned in 1 paper
  - 3 papers study GPU implementation details:
    - HW dependent: Shared memory, texture memory
    - Algorithmic: Local search, modified city selection
- Results reported as speedup vs. a CPU version

Ant Colony Optimization for TSP on GPU (based on 7 publications)

- Approach (A): Each thread computes tour construction for 1 ant
- Approach (B):
  - Not enough ants to fill threads of GPU
  - Next city selection per ant offers parallelism

  => One ant per block, threads perform city selection together

- 5 papers implement approach (A), 4 papers implement (B)
- Only 2 papers compare (A) and (B), both favour (B)
- Pheromone update performed on GPU by 6 papers

- Good paper to start reading for ACO (for TSP) on GPU:

  Cecilia et al, Parallelization strategies for ant colony optimization on GPUS, IPDPSW 2011

  - Compares (A) and (B)
  - Examines HW details (e.g. shared memory) and algorithmic changes

Summary

Recap of tutorial

- All current processors are parallel:
  - You cannot ignore parallelization and expect high performance
  - Serial programs utilize 1% of potential!

- Getting started coding for GPUs has never been easier:
  - Nvidia CUDA tightly integrated into Visual Studio
  - Excellent profiling tools available with toolkit

- Low hanging fruit has been picked:
  - The challenge now is to devise new intelligent algorithms that take the architecture into consideration

Some references

- Code examples available online: https://github.com/babrodtk/ParallelPi, https://github.com/sintefmath/DiscreteOptimizationGPUExamples
- NVIDIA CUDA website: https://developer.nvidia.com/cuda-zone
- Brodtkorb, Hagen, Schulz and Hasle, **GPU Computing in Discrete Optimization Part I: Introduction to the GPU,** EURO Journal on Transportation and Logistics, 2013.
- Schulz, Hasle, Brodtkorb, and Hagen, **GPU Computing in Discrete Optimization Part II: Survey Focused on Routing Problems,** EURO Journal on Transportation and Logistics, 2013.
- A. R. Brodtkorb, M. L. Sætra and T. R. Hagen, **GPU Programming Strategies and Trends in GPU Computing**, Journal of Parallel and Distributed Computing, 2013.
- Burke and Riise, **On Parallel Local Search for Permutations**, Journal of the Operational Research Society, 2014.
- Cecilia, Garcia, Ujaldon, Nisbet, and Amos, **Parallelization strategies for ant colony optimization on GPUs**, *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum,* 2011.

Thank you for your attention!

If you found the tutorial interesting,
feel free to contact us!

André R. Brodtkorb, PhD

Christian Schulz, PhD

Email: Andre.Brodtkorb@sintef.no, Christian.Schulz@sintef.no
SINTEF homepage: http://www.sintef.no/math