SINTEF

CONSERVATION LAWS ON GPUS: COMPUTING PI WITH CUDA

André R. Brodtkorb
Researcher, Department of Mathematics and Cybernetics, SINTEF Digital
Associate Professor, OsloMet – Oslo Metropolitan University

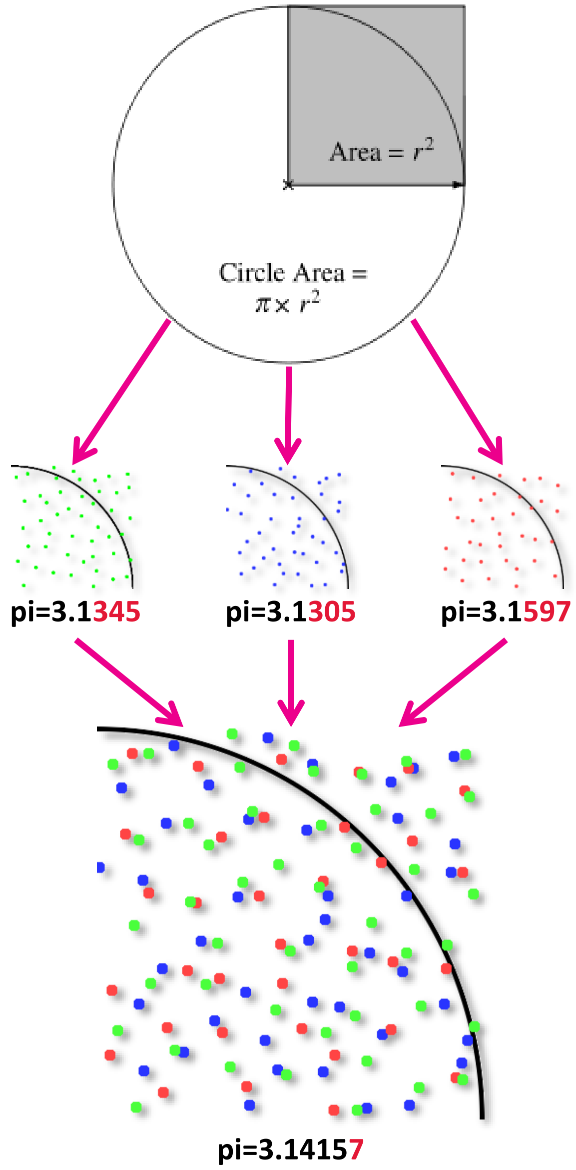
Computing π with CUDA

Computing π with CUDA

- 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

- 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☺

SINTEF

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();
        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;
}
```

1

2 & 3

4

SINTEF

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

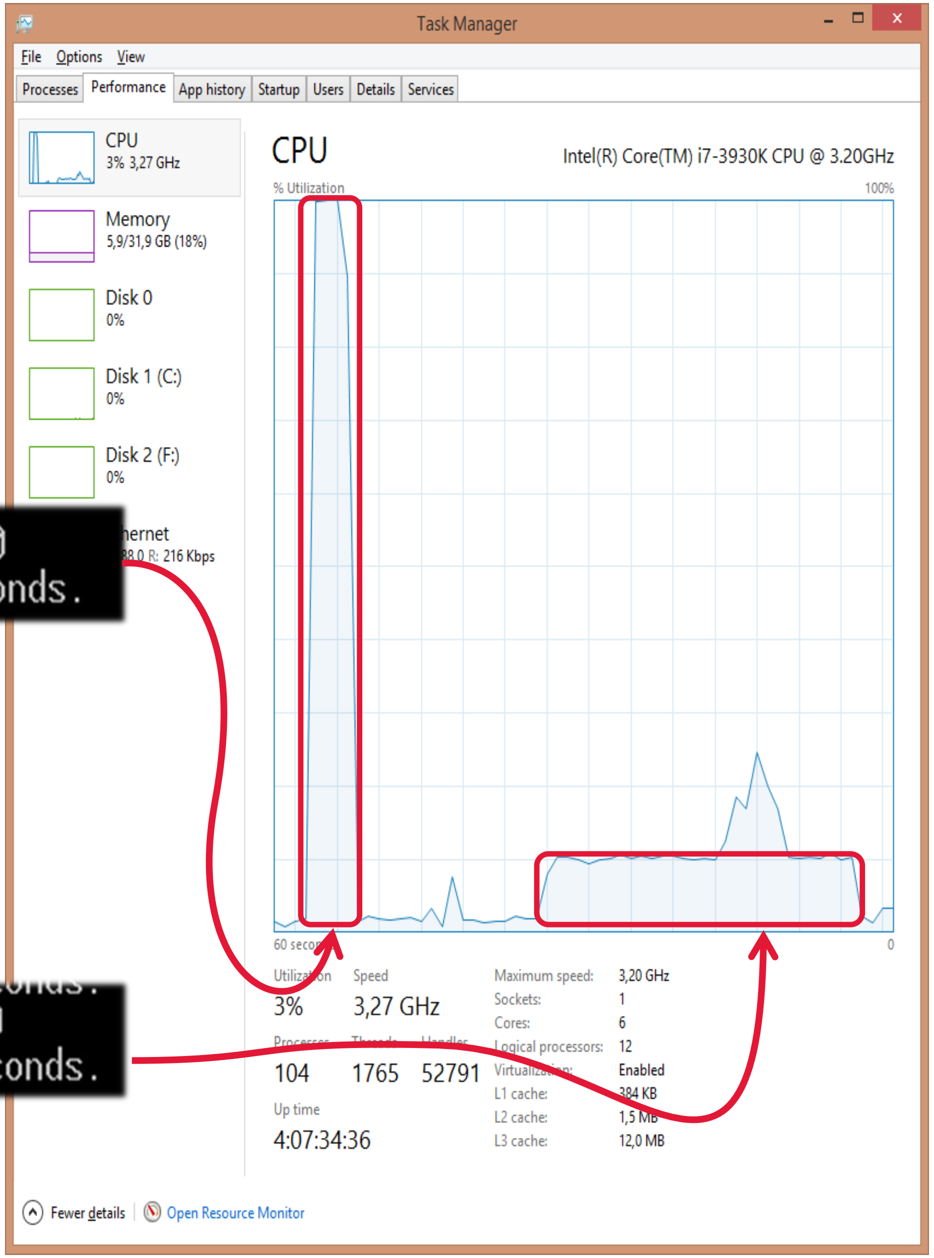
Performance

- Parallel: 3.8 seconds @ 100% CPU

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

- Serial: 30 seconds @ 10% CPU

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



SINTEF

Parallel GPU version 1 (CUDA) 1/3

```
__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;
     }
}
```

GPU function

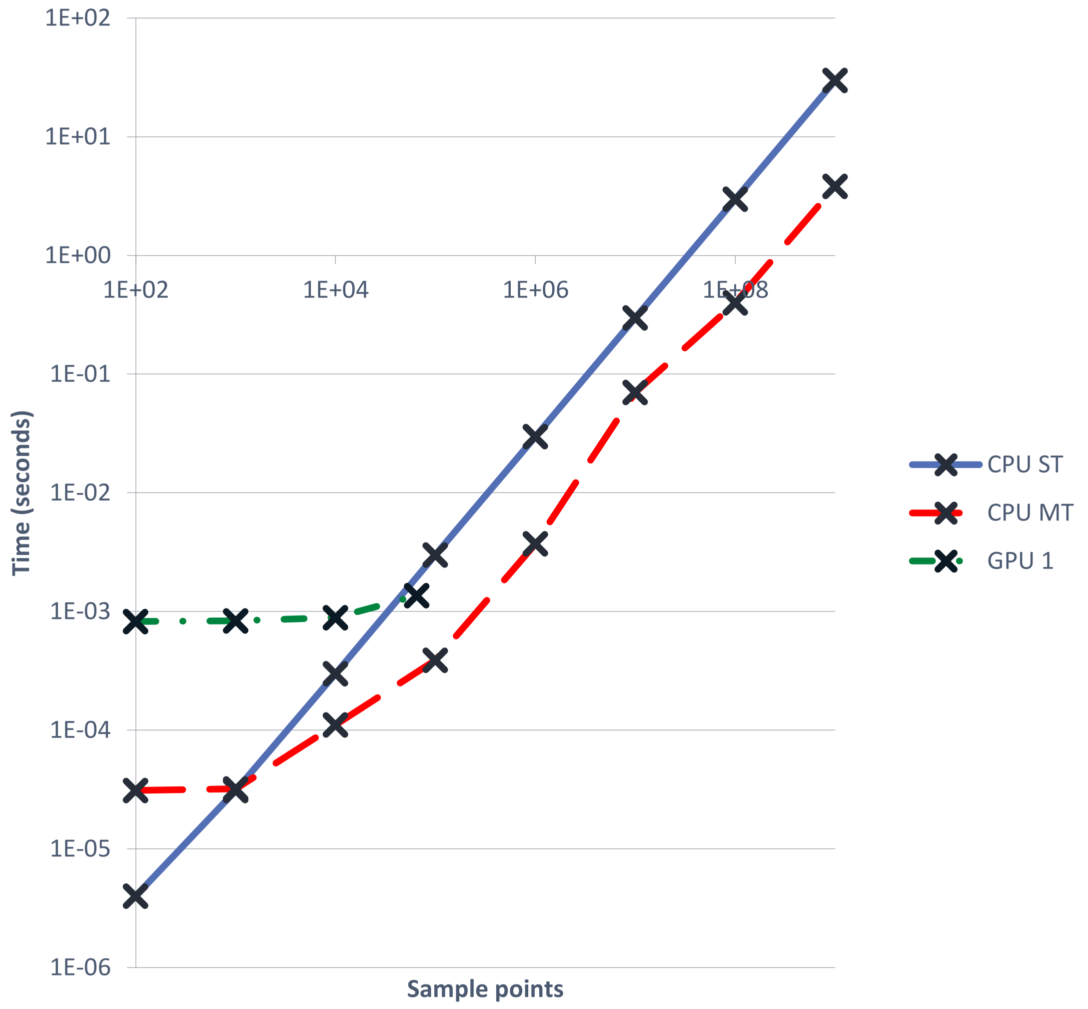
*Random numbers on GPUs can be a slightly tricky, see cuRAND for more information

Parallel GPU version 1 (CUDA) 2/3

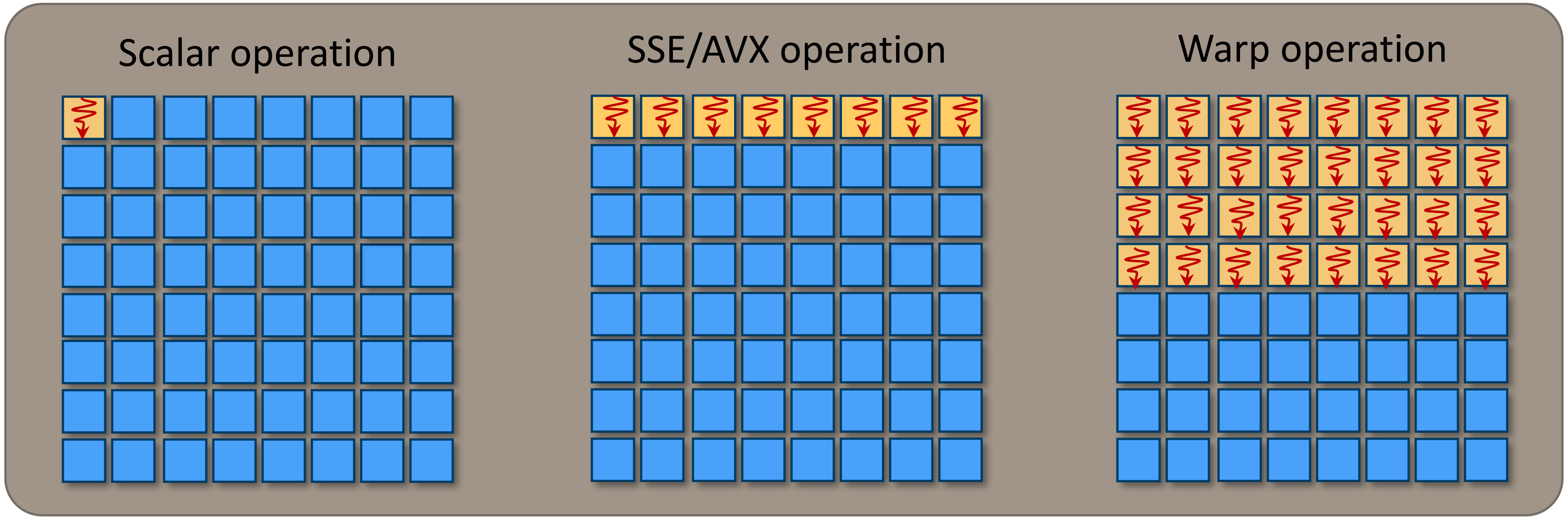
```
float computePi(int n_points) {
    dim3 grid = dim3(n_points, 1, 1);
    dim3 block = dim3(1, 1, 1);

    //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.

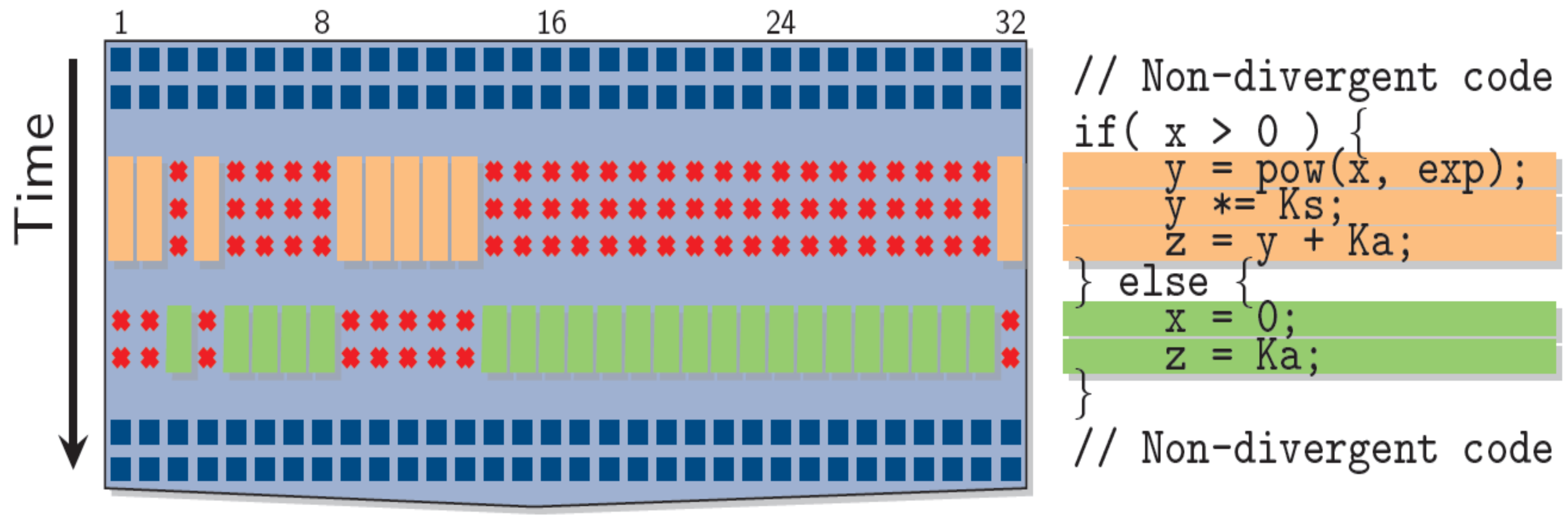


GPU Vector Execution Model



- **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



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;
      }
}
```

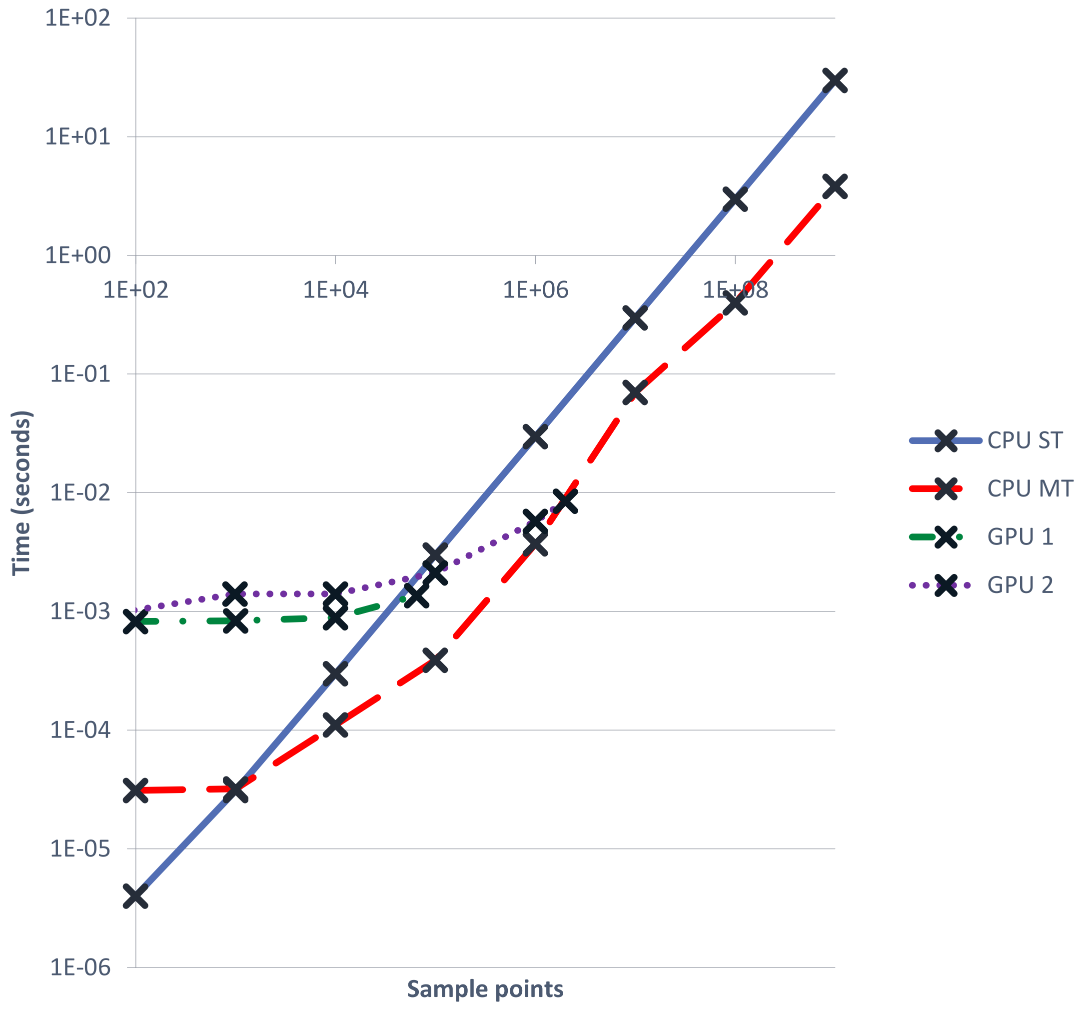
New indexing

```
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

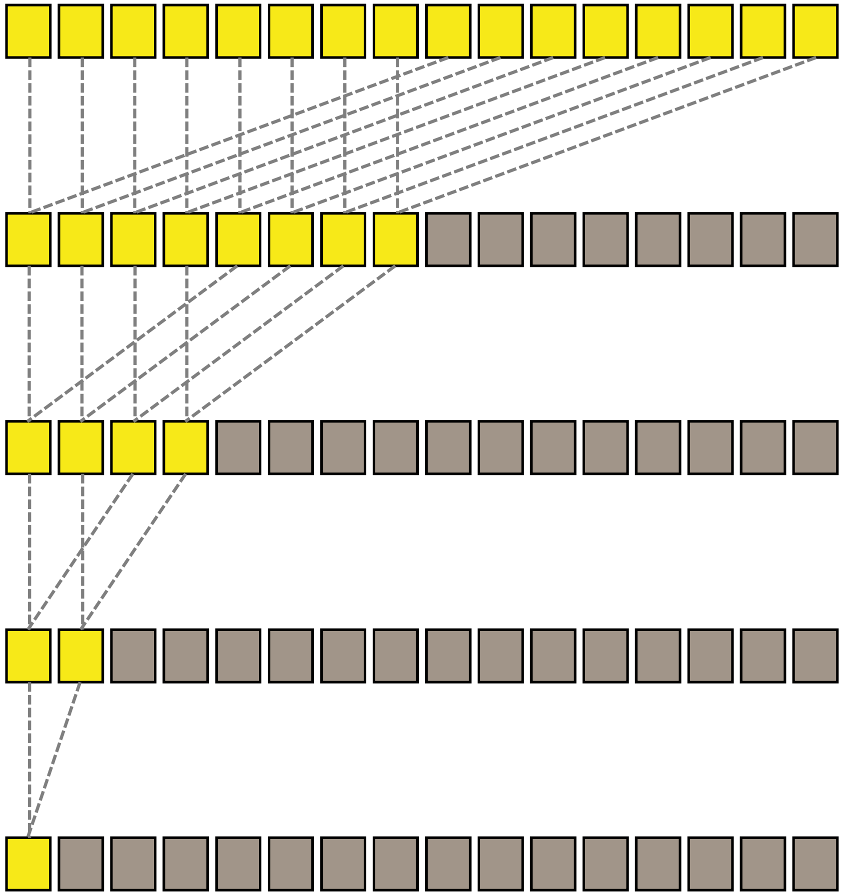
Parallel GPU version 3 (CUDA) 2/4

```
… //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) 3/4

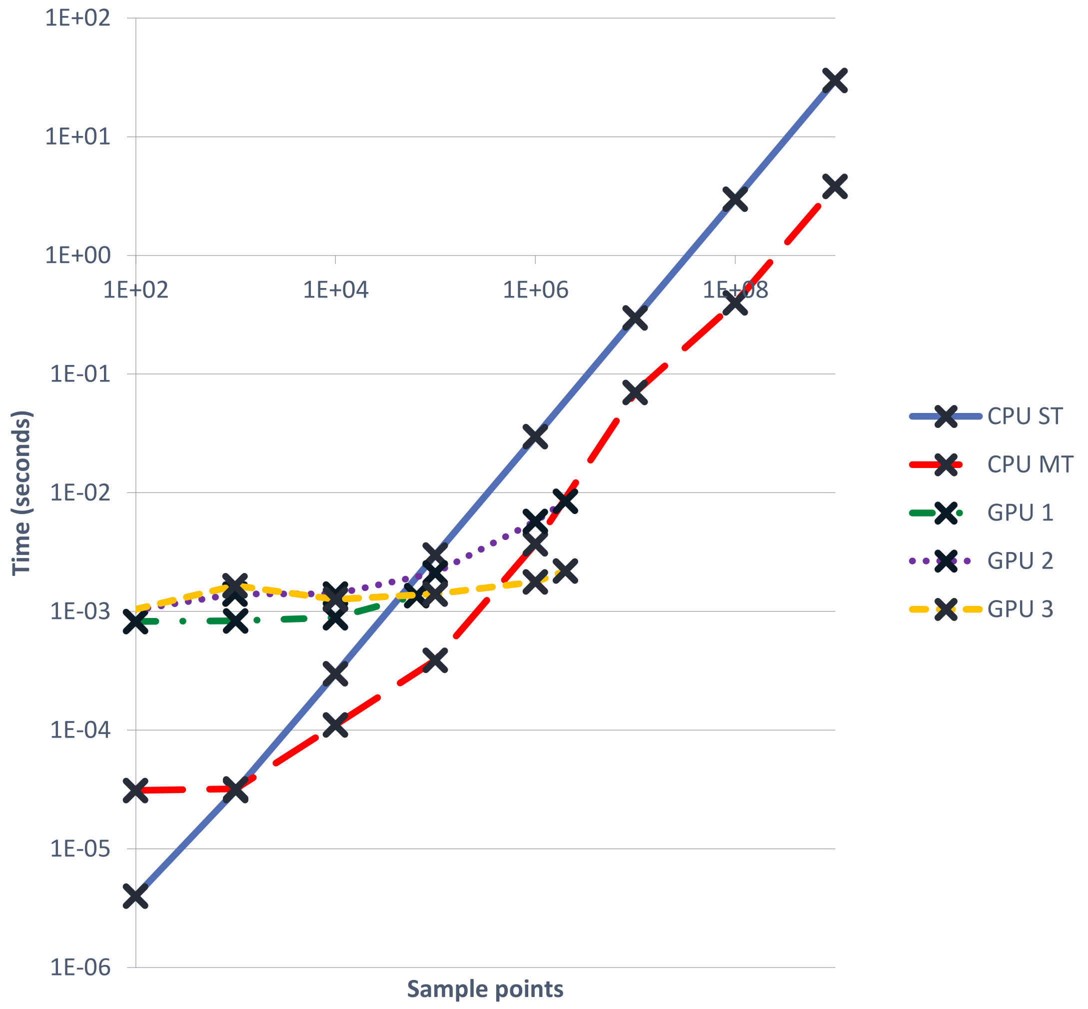
- Shared memory is a kind of programmable cache
  - Fast to access (just slightly slower than registers)
  - Programmers responsibility to move data into shared memory
  - All threads in one block can see the same shared memory
  - Often used for communication between threads

- Sum all elements in shared memory using shared memory reduction



Parallel GPU version 3 (CUDA) 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

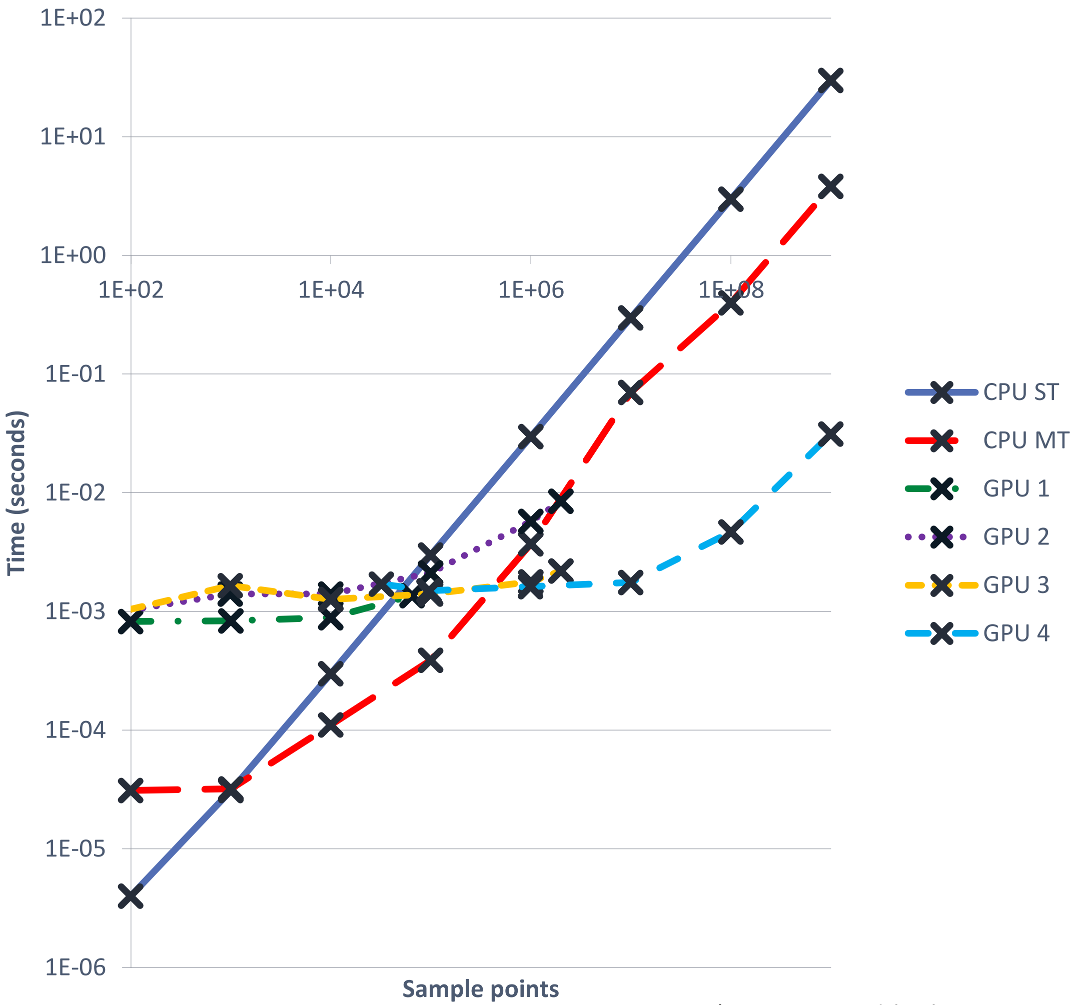
```
__global__ void computePiKernel4(unsigned int* output) {
        int n_inside = 0;

        //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 <u>fair</u>:
  - Same generation CPU (Core i7 3930K) and GPU (GTX 780)
  - Code available on Github: you can test it yourself!

- Reasons why the comparison is <u>unfair</u>:
  - 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).