/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2019 - Lecture 10: “GPGPU (3)”

Welcome!

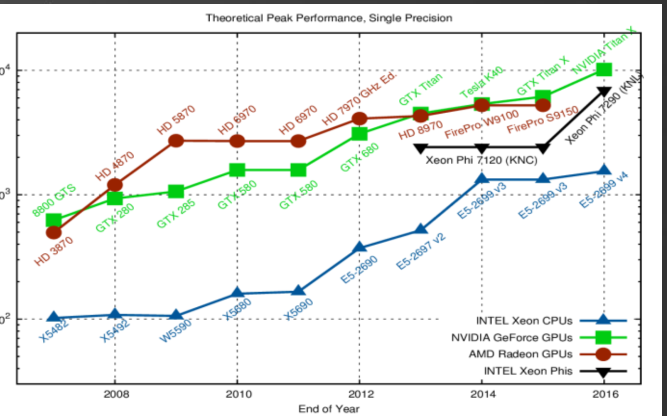
Today's Agenda:

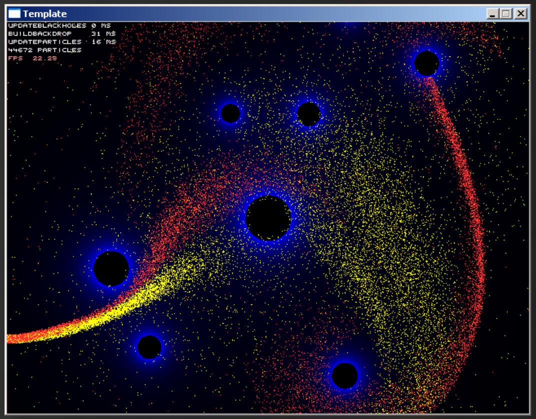
- GPU Execution Model
- GPGPU Flow
- GPGPU Low Level Notes
- P3

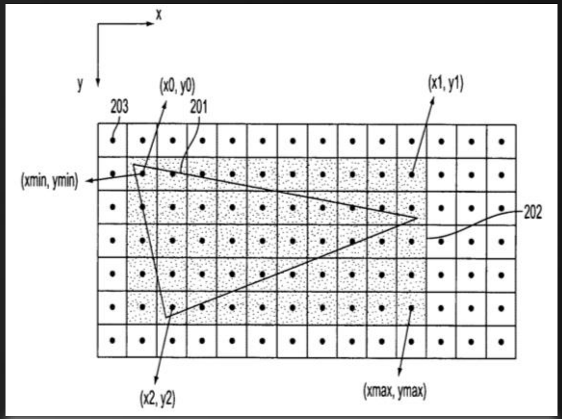
Model

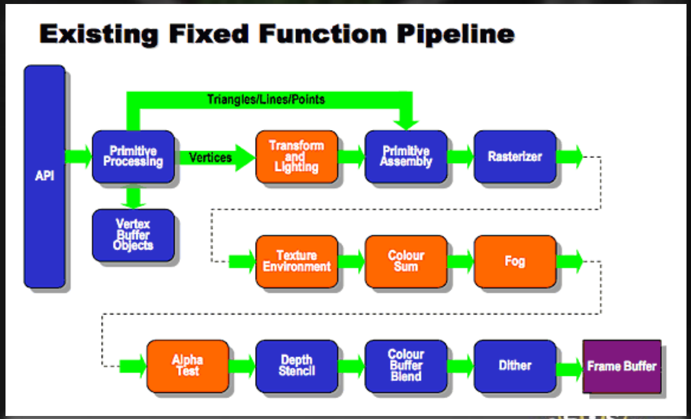
Recap

- The GPU is a co-processor, which needs a host.
- GPUs have a history of fixed-function pipelines.
- Typical GPU work is fundamentally data-parallel.
- GPU programming is similar to SIMD programming.
- For parallel tasks, a GPU is very fast (worth the effort!).









Model

SIMT Recap

S.I.M.T.: *Single Instruction, Multiple Thread.*

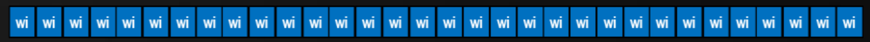
```
for (float i = 0.0; i < 4095.0f; i += 1.0)
{
    dz = (float2)(2.0f * (z.x * dz.x - z.y * dz.y) + 1.0f, 2.0f * (z.x * dz.y + z.y * dz.x));
    z = cmul( z, z ) + c;
    float a = sin( tm * 1.5f + i * 2.0f ) * 0.3f + i * 1.3f;
    float2 t = (float2)(cos( a ) * z.x + sin( a ) * z.y, -sin( a ) * z.x + cos( a ) * z.y);
    if (fabs( t.x ) > 2.0f && fabs( t.y ) > 2.0f) { it = i; break; }
}
float z2 = z.x * z.x + z.y * z.y, t = log( z2 ) * sqrt( z2 ) / length( dz ), r = sqrt( z2 );
float q = zoom * 0.016f * (1.0f / j.x + 1.0f / j.y), d = length( j ), w = q * d / 400.0f;
float s = q * d / 80.0f, f = 0.0f, g = 0.0f;
```

Model

SIMT Recap

S.I.M.T.: *Single Instruction, Multiple Thread.*

```
for (float i = 0.0; i < 4095.0f; i += 1.0)
{
    dz = (float2)(2.0f * (z.x * dz.x - z.y * dz.y) + 1.0f, 2.0f * (z.x * dz.y + z.y * dz.x));
    z = cmul( z, z ) + c;
    float a = sin( tm * 1.5f + i * 2.0f ) * 0.3f + i * 1.3f;
    float2 t = (float2)(cos( a ) * z.x + sin( a ) * z.y, -sin( a ) * z.x + cos( a ) * z.y);
    if (fabs( t.x ) > 2.0f && fabs( t.y ) > 2.0f) { it = i; break; }
}
float z2 = z.x * z.x + z.y * z.y, t = log( z2 ) * sqrt( z2 ) / length( dz ), r = sqrt( z2 );
float q = zoom * 0.016f * (1.0f / j.x + 1.0f / j.y), d = length( j ), w = q * d / 400.0f;
float s = q * d / 80.0f, f = 0.0f, g = 0.0f;
```



Model

SIMT Recap

S.I.M.T.: *Single Instruction, Multiple Thread.*

Adding two arrays, C/C++ way:

```
for( int i = 0; i < N; i++ ) c[i] = a[i] + b[i];
```

Adding two arrays in MatLab: c = a + b

Adding two arrays using SIMD:

```
void add(int* a, int* b, int* c, int N)
{
   for( int i = 0; i < N; i += 4 )
   {
      __m128 a4 = ((__m128*)a)[i];
      __m128 b4 = ((__m128*)b)[i];
      ((__m128*)c)[i] = a4 + b4;
   }
}
```

Adding two arrays using SIMT:

```
void add(int* a, int* b, int* c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
  c[i] += a[b[i]];     // via a lut

  // look ma, no loop!
}
```

Model

SIMD versus SIMT

```
void add(int* a, int* b, int* c, int N)
{
   for( int i = 0; i < N; i += 4 )
   {
      __m128 a4 = ((__m128*)a)[i];
      __m128 b4 = ((__m128*)b)[i];
      ((__m128*)c)[i] = a4 + b4;
   }
}
```

```
void add(int* a, int* b, int* c)
{
  int i = blockIdx.x * blockDim.x +
threadIdx.x;
  c[i] = a[i] + b[i];
  c[i] += a[b[i]];     // via a lut

  // look ma, no loop!
}
```

Benefit of SIMT:

- Easier to read and write; similar to regular scalar flow.

Drawbacks of SIMT:

- Redundant data (here: pointers `a`, `b` and `c`).
- Redundant data (variable `i`).
- A 'warp' is 32-wide, regardless of data size.
- Scattered memory access is not discouraged.
- Control flow.
- **We need *tons* of registers.**

Model

Register Pressure

On a CPU:

AX (‘accumulator register’)	AH, AL (8-bit)	EAX (32-bit)	RAX (64-bit)
BX (‘base register’)	BH, BL	EBX	RBX
CX (‘counter register’)	CH, CL	ECX	RCX
DX (‘data register’)	DH, DL	EDX	RDX
BP (‘base pointer’)		EBP	RBP
SI (‘source index’)		ESI	RSI
DI (‘destination index’)		EDI	RDI
SP (‘stack pointer’)		ESP	RSP
			R8..R15
		st0..st7	
		XMM0..XMM7	XMM0..XMM15
			YMM0..YMM15
			ZMM0..ZMM31

Model

Register Pressure

On a CPU:

RAX (64-bit)
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8..R15
YMM0..YMM15 (256-bit)

Model

Register Pressure

On a GPU:

- Each thread in a warp needs its own registers (32 * N);
- The GPU relies on SMT to combat latencies (32 * N * M).

SMT on the CPU: each core *avoids* latencies.

- Super-scalar execution
- Out-of-order execution
- Branch prediction
- Cache hierarchy
- Speculative prefetching

And, as a ‘last line of defense’, if a latency happens anyway:

- SMT

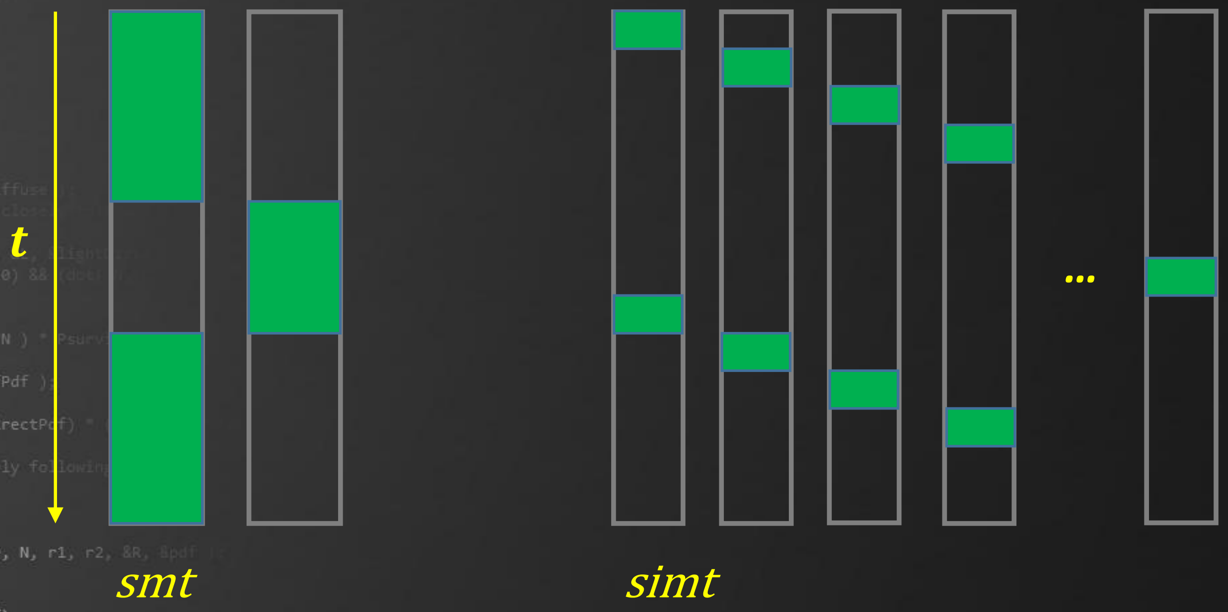
Model

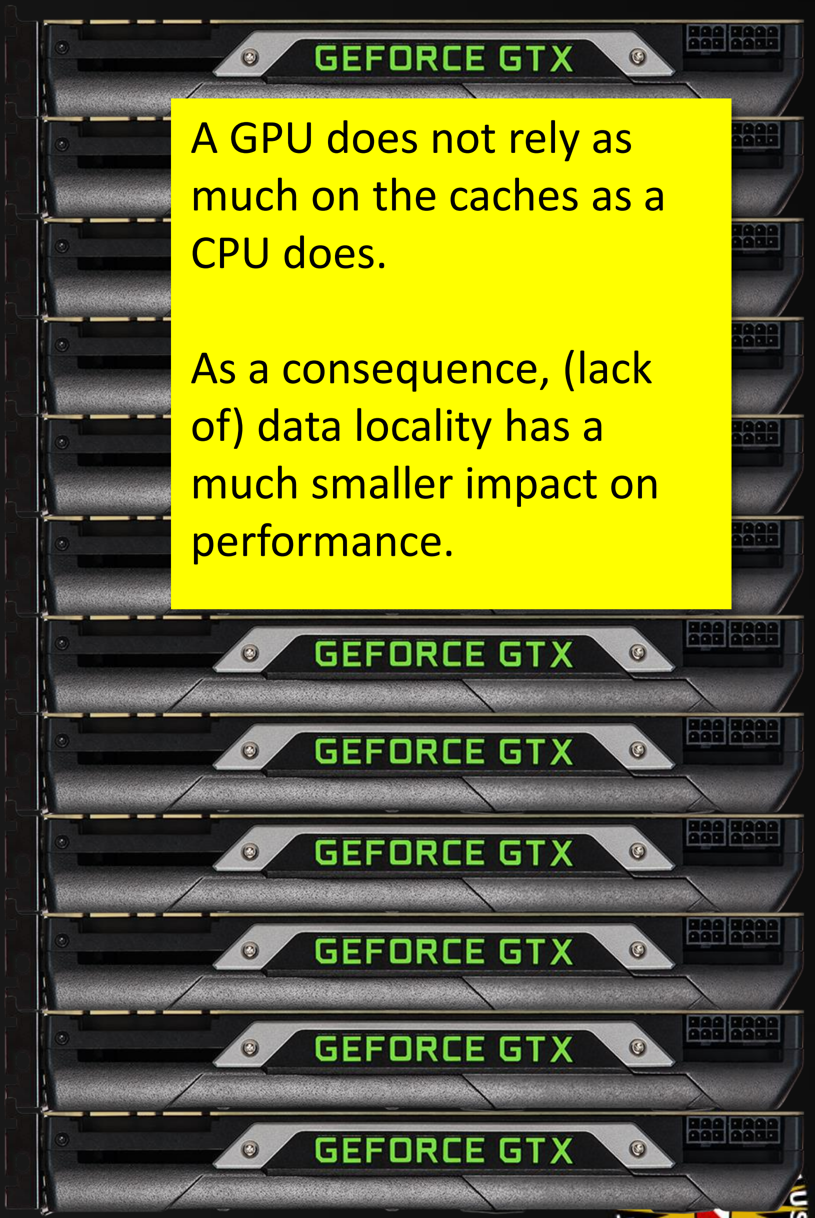
Register Pressure

On a GPU:

- Each thread in a warp needs its own registers (32 * N);
- The GPU relies on SMT to combat latencies (32 * N * M).

SMT on the GPU: *primary weapon* against latencies.





A GPU does not rely as much on the caches as a CPU does.

As a consequence, (lack of) data locality has a much smaller impact on performance.

Model

Register Pressure

On a CPU, hyperthreading typically *hurts* single thread performance
➔ SMT is limited to 2, max 4 threads.

On a GPU, 2 warps per SM is not sufficient: we need 4, 8, 16 or more.

For 16 warps per SM we get:

32 * N * 16, where N is the number of registers one thread wishes to use.

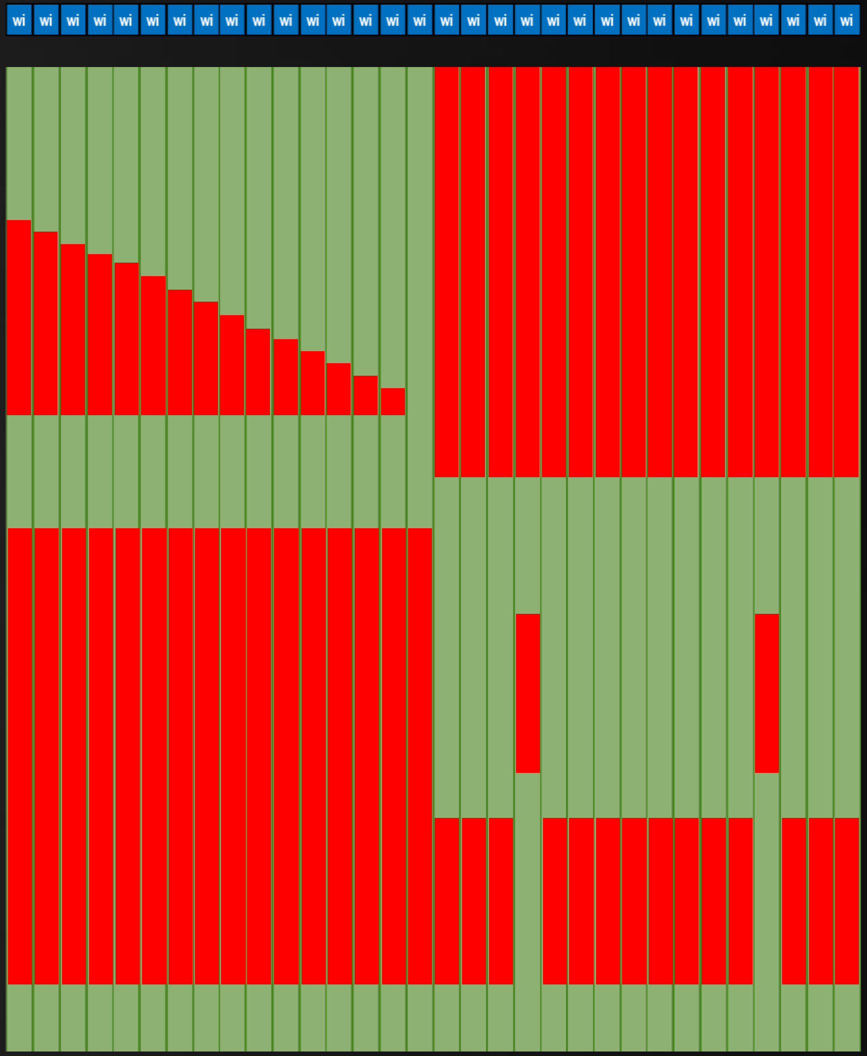
On a typical CPU we have 32 registers ore more available, many of these 256-bit (8-wide AVX registers), others 64-bit.

On a modern GPU, we get 256KB of register space per SM:
32 * 32 * 64 = 65536 32-bit registers per SM.

Model

Control Flow

```
if (threadIdx.x < 16)
{
    for( int i = 0; i < threadIdx.x; i++ )
    {
        // ...
    }
}
else
{
    if (y == 5
    {
        // ...
    }
    else
    {
        // ...
    }
}
```



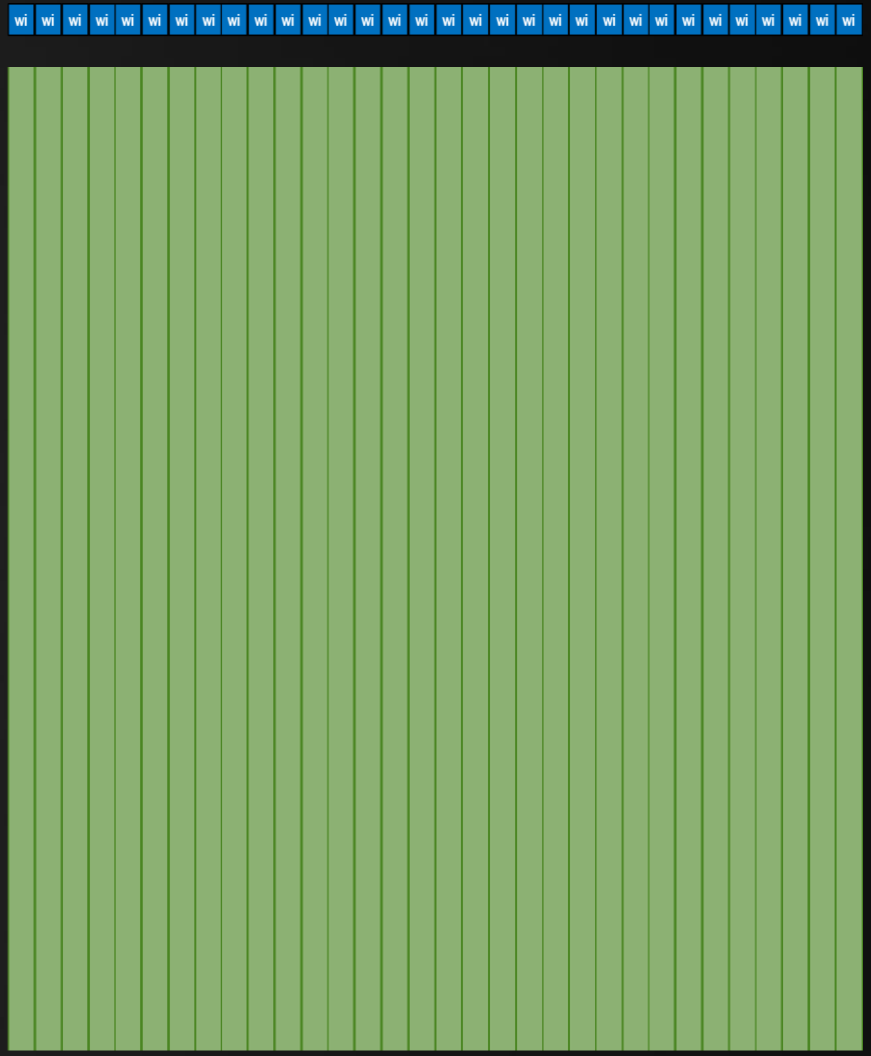
Model

Control Flow

```
while (1)
{
   // ...
   if (Rand() < 0.05f) break;
}


while (1)
{
   if (threadIdx.x == 0)
   {
      if (Rand() < 0.05f) a[0] = 1;
   }
   if (a[0] == 1) break;
}
```

Careful: thread 0 is not necessarily the first one to reach the break.

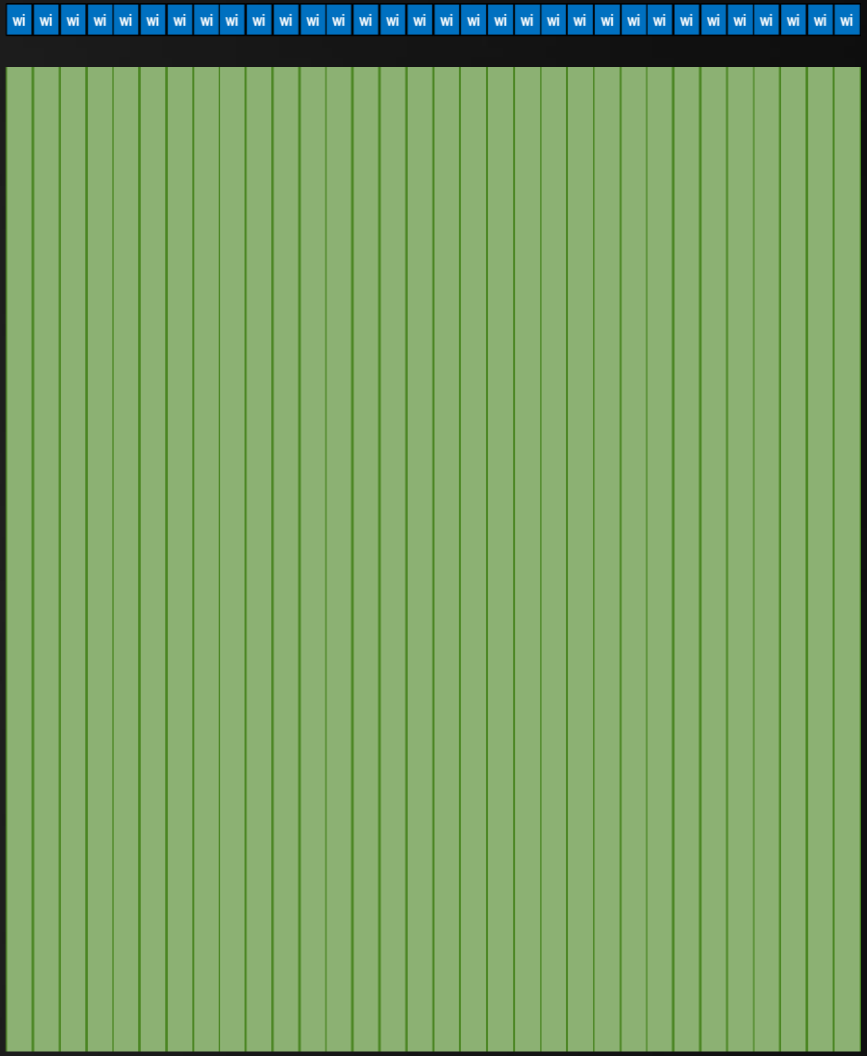


Model

Control Flow

```
while (1)
{
   // ...
   if (Rand() < 0.05f) break;
}


while (1)
{
   if (threadIdx.x == 0)
   {
      if (Rand() < 0.05f) a[0] = 1;
   }
   __syncthreads();
   if (a[0] == 1) break;
}
```



Model

Synchronization

CPU / GPU synchronization: *streams* (CUDA), *queues* (OpenCL).

An OpenCL command is executed *asynchronously*:
it simply gets added to the queue.

Example:

```
void Kernel::Run()
{
   glFinish();                // wait for OpenGL to finish
   clEnqueueNDRangeKernel( queue, kernel, 2, 0, workSize, localSize, 0, 0, 0 );
   clFinish( queue );     // wait for OpenCL to finish
}
```

Model

Synchronization

Fundamental approach to synchronization of GPU threads: don’t do it.

...But, if you must:

```
__syncthreads();
```

For free:

```
__shared__ int firstSlot;
if (threadIdx.x == 0) firstSlot = atomic_inc( &counter, 32 );
int myIndex = threadIdx.x;
array[firstSlot + myIndex] = resultOfComputation;
```

Warps execute in lockstep, and are therefore synchronized*.

*: On Volta and Turing use __syncwarp(), see: https://devblogs.nvidia.com/inside-volta, section “Independent Thread Scheduling”.

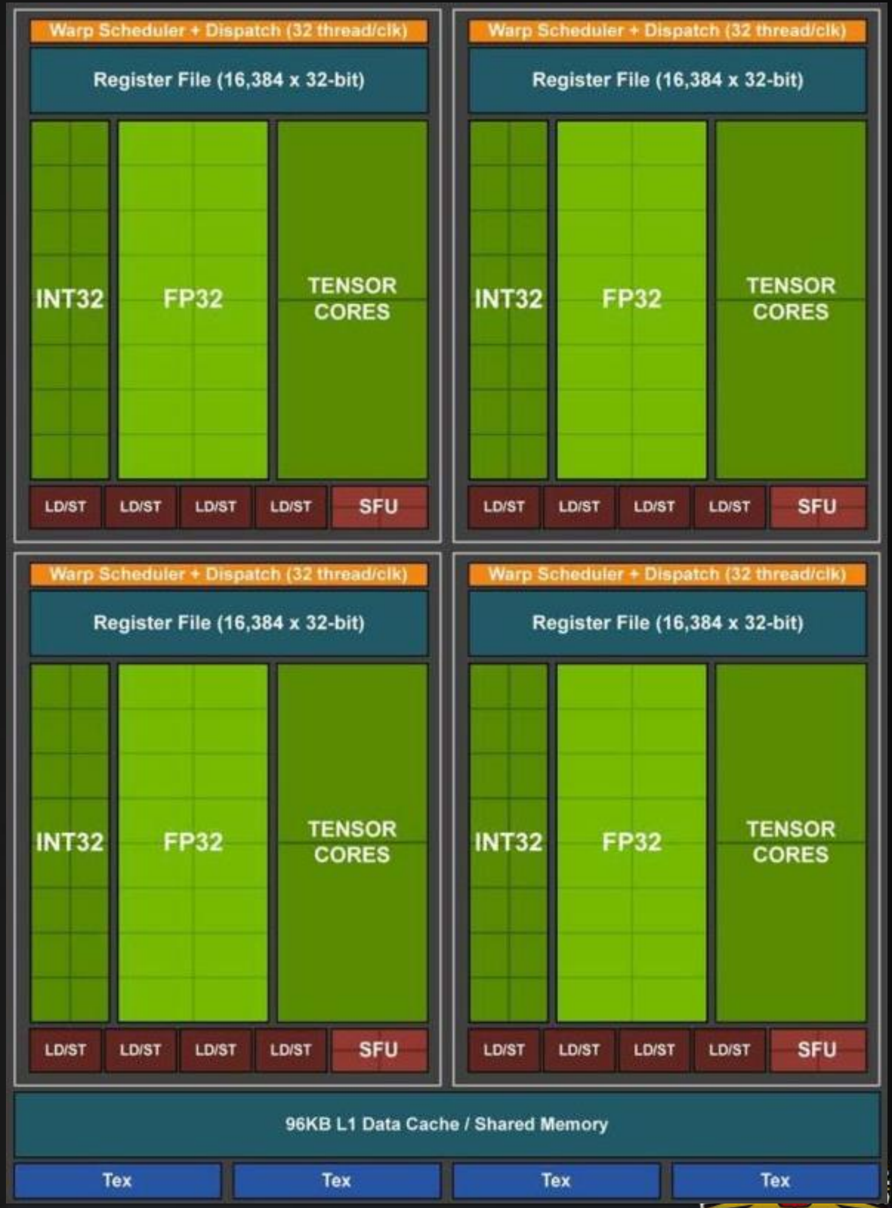
Model

Synchronization

Threads on a single SM can communicate via global memory, or via shared memory.

In CUDA:

```
__global__ void reverse( int* d, int n )
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```



Model

Synchronization

Threads on a single SM can communicate via global memory, or via shared memory.

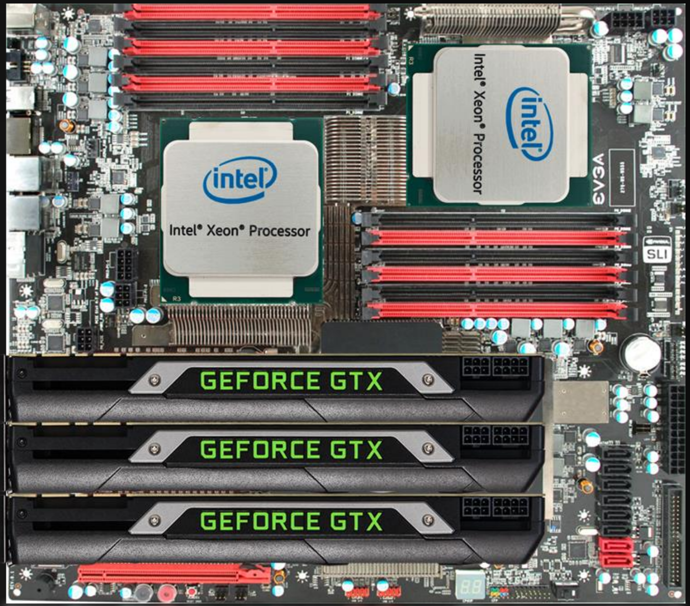
In OpenCL:

```
__kernel void reverse( global int* d, int n )
{
  __local int s[64];
  int t = get_local_id(0);
  int tr = n-t-1;
  s[t] = d[t];
  barrier( CLK_LOCAL_MEM_FENCE);
  d[t] = s[tr];
}
```

Today's Agenda:

- GPU Execution Model
- GPGPU Flow
- GPGPU Low Level Notes
- P3

Flow



A Typical GPGPU Program

Calculating anything using a GPU kernel:

1. Setup input data on the CPU
2. Transfer input data to the GPU
3. Operate on the input data
4. Transfer the result back to the CPU
5. Profit.

Amdahl’s law:

$$S_{peedup} < \frac{1}{1-p},$$

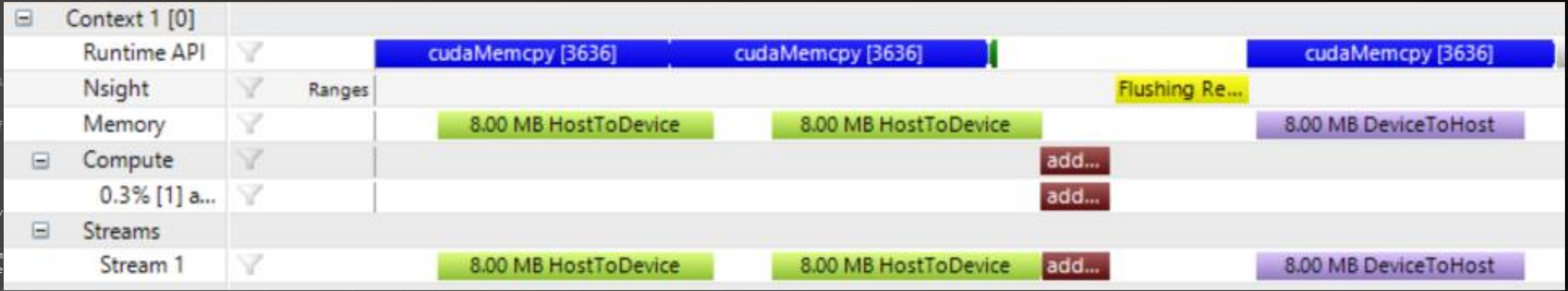
where p is the portion of the code that is parallelizable.

Flow

A Typical GPGPU Program

2. Transfer input data to the GPU.

```
void add(int* a, int* b, int* c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  a[i] = b[i] + c[i];
}
```



Flow

A Typical GPGPU Program

2. Transfer input data to the GPU.

```
void add(int* a, int* b, int* c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  a[i] = b[i] + c[i];
}
```

Optimizing transfers:

- Reduce the *number* of transfers first, then their size.
- Only send changed data.
- Use asynchronous copies.

If possible:

- Produce the input data on the GPU.

For visual results:

- Store visual output directly to a texture.

Flow

Asynchronous Copies

OpenCL supports multiple queues:

queue = clCreateCommandQueue(context, devices[…], 0, &error);

Kernels and copy commands can be added to any queue:

```
clEnqueueNDRangeKernel( queue, kernel, 2, 0, workSize, 0, 0, 0, 0 );
clEnqueueWriteBuffer( Kernel::GetQueue(), ... );
```

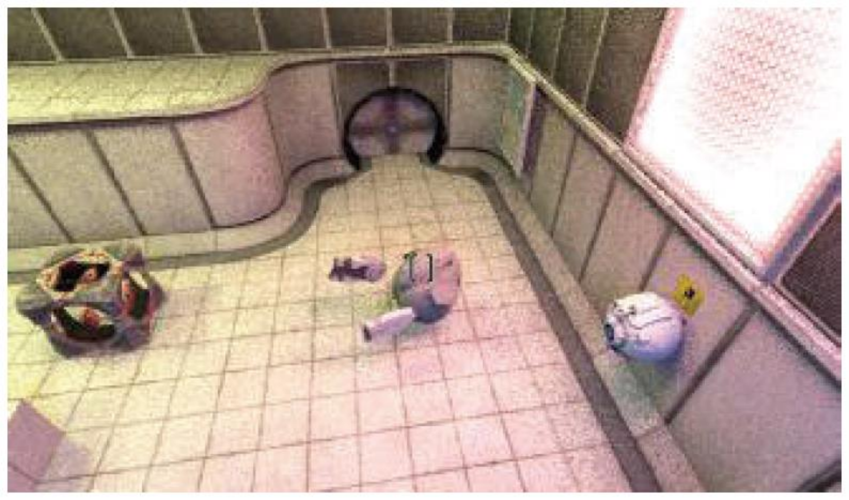
Queues can wait for a signal from another queue:

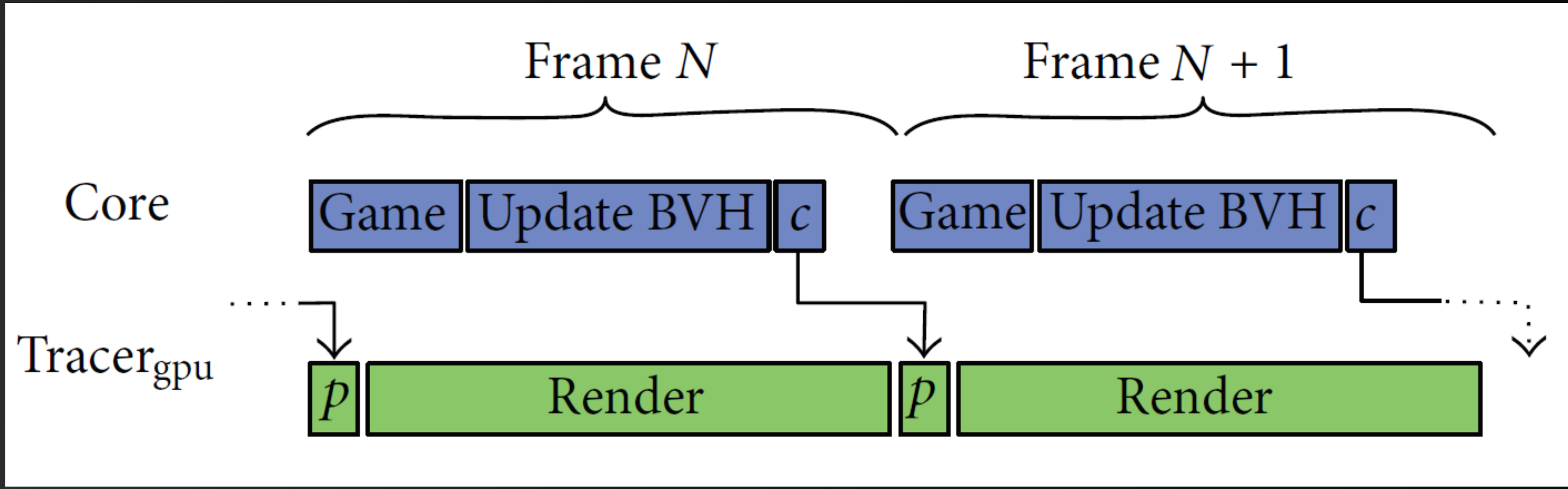
```
clEnqueueBarrierWithWaitList( … );
```

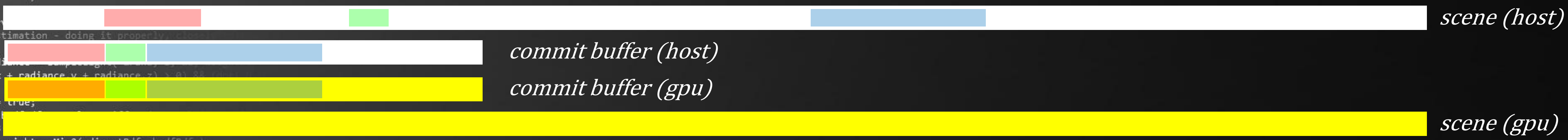
CUDA provides similar functionality.

Flow

Asynchronous Copies







*: The Brigade Renderer: A Path Tracer for Real-Time Games, Bikker & Van Schijndel, 2013.

Today’s Agenda:

- GPU Execution Model
- GPGPU Flow
- GPGPU Low Level Notes
- P3

P3

Your Mission

“Optimize an application using the process and means discussed in INFOMOV.”

“An application”:

1. One of your own. Requirement: functionality must be ‘done’, optimization may not purely be a port to C/C++.
2. One of Roland’s Projects. Additional benefit: goodies if you win. Also: winning. Will be introduced today in The Final Hour.
3. One of my projects. Options: animation module of Lighthouse 2, and a simpler application. Simple application grade will be capped at 7.
4. A single-header library from GitHub. Lists: here and here. You will have to setup your own test case, and you are expected to submit the optimized code (INFOMOV-branded) to the original repo.
5. Any GitHub / open source project, if you think you can handle it. Warning: last option on this list for a reason.

P3

Your Mission

“Optimize an application using the process and means discussed in INFOMOV.”

“The Process”:

1. Establish optimization goal (optional).
2. Profile.
3. Apply high-level optimization (on hotspot).
4. Profile.
5. Multi-thread / vectorize / apply GPGPU, if applicable.
6. Profile.
7. Apply low-level optimizations.
8. Repeat step 6 and 7 until time runs out.
9. Report.

Your report should provide clear proof that you approached the optimization in a structured manner, i.e. it will provide profiling information at every step.

P3

Your Mission

"Optimize an application using the process and means discussed in INFOMOV."

"Means":

1. High-level optimizations (typically those that change algorithmic complexity).
2. Low-level optimizations (see "Rules of Engagement").
3. Data-Oriented Design.
4. Anything else to please the cache.
5. SIMD.
6. GPGPU.
7. Compiler output inspection, compiler choice, compiler settings.

Note that overclocking is not in this list.

P3

Your Mission

“Optimize an application using the process and means discussed in INFOMOV.”

Notes:

1. Do not alter functionality.
2. If you skip optimizations to maintain readability: indicate this in the report.
3. Multiple teams may work on the same base code. Do not share optimized code in these cases; sharing ideas is still allowed however.

Don’t forget to maintain a healthy work/life balance. Or fix that after the deadline.

/INFOMOV/

END of “GPGPU (3)”

next lecture: “fixed point”