Technische Universität Berlin

Multicore Architectures – GPU Lab

Jan Lucas | AES

AES Embedded Systems Architecture

CUDA Example

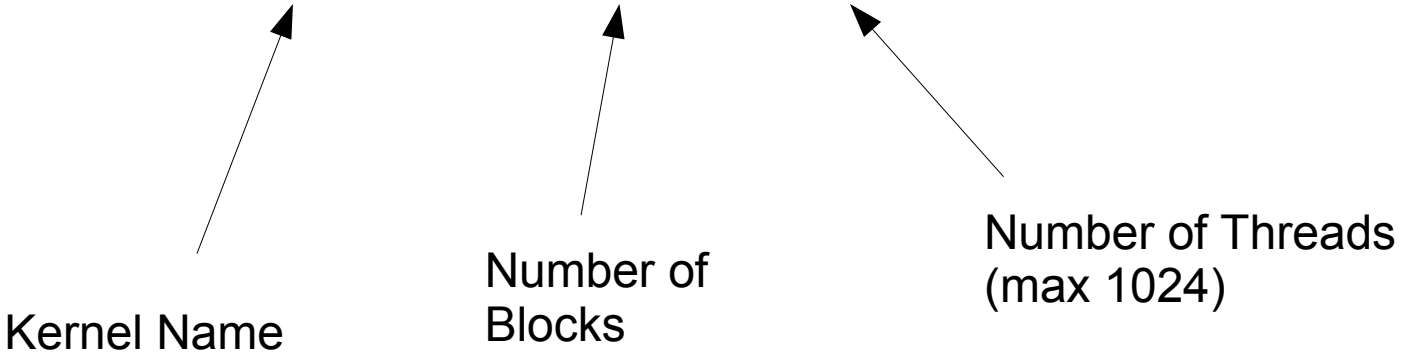
```
#include <stdio.h>

__global__ void mult2(float* input, float* output)
{
    output[0] = input[0] * 2;
}

int main(int argc, char** argv) {
   float in_local=3.14f;
   float out_local;
   float *in,*out;
   cudaMalloc((void**)&in,sizeof(float));
   cudaMalloc((void**)&out,sizeof(float));
   cudaMemcpy(in,&in_local,sizeof(float),cudaMemcpyHostToDevice);
   mult2<<<1,1>>>(in,out);
   cudaMemcpy(&out_local,out,sizeof(float),cudaMemcpyDeviceToHost);
   printf("Output: %f",out_local);
}
```

Launching Kernels

```
kernel<<<blocks,threads>>>(arguments)
```



Built-in Variables

Variable	Description
dim3 gridDim	Size of current grid
dim3 blockDim	Size of current block
dim3 blockIdx	Block index within current grid
dim3 threadIdx	Thread index within current block
int warpSize	Number of threads per warp

```
struct dim3
{
    unsigned int x,y,z;
}
```

Function Qualifiers

Qualifier	Description
__global__	Callable from CPU, executed on GPU
__device__	Callable from GPU, execute on GPU
__host__	Default: call and execution on CPU
__noinline__	Do not inline.

Variable Qualifiers

Qualifier	Description
__shared__	Variable will be placed in shared memory (local SRAM)
__global__	Variable will be placed in global memory (GPU DRAM)
__local__	Variable will be placed in thread local memory (GPU DRAM with special interleaving)
__constant__	Variable will be placed in read-only memory(Initialize using cudaMemcpyToSymbol(...))

Synchronization

```
__syncthreads();
```

- Blocks until all threads in this block have reached `__syncthreads()`.
- All threads in the block must reach the same **__syncthreads()**

Memory fence

– Writes to DRAM or shared memory by one thread are not necessary directly visible to different threads on the GPU or the CPU.

Function	Description
`__threadfence_block()`	Makes all previous writes visible to other threads in the same block
`__threadfence()`	Makes all previous writes visible to all threads on the same device
`__threadfence_system()`	Makes all previous writes visible to all threads on GPU and CPU

Memory Allocation

```
cudaMalloc(void **devptr, size_t size);
```

Example:

```
float *buffer;
cudaMalloc( (void**)&buffer,sizeof(float)*2048);
```

- Allocates a memory buffer for 2048 on the GPU.
- Pointer returned is a GPU pointer!
  - Trying to read or write on the CPU using this pointer will fail!
    e.g.: buffer[123] = 1234.0f on the CPU generates a segfault

Memory Transfer

```
cudaMemcpy(void *dst, void *src, size_t size,
   cudaMemcpyHostToDevice);

cudaMemcpy(void *dst, void *src, size_t size,
   cudaMemcpyDeviceToHost);

cudaMemcpy(void *dst, void *src, size_t size,
   cudaMemcpyDeviceToDevice);
```

Compiling CUDA

nvcc -o OUTPUT file.cu

Compiles “file.cu” into OUTPUT executable

nvcc -ptx file.cu

Compiles GPU kernels into PTX assembler(file.ptx)

Running CUDA

If NVidia GPU is available:

- 1. Install CUDA Toolkit and NVidia Binary drivers
- 2. Run compiled application

- You can enable the CUDA_PROFILER with "export CUDA_PROFILE=1"
- Profiling information will be written to cuda_profile_log_*.log

GPGPU-Sim

- GPGPU-Sim is a GPU-Simulator
- It runs CUDA Applications on simulated GPU.
- No GPU is required
- GPU Architectural parameters can be configured
- Generates detailed statistics.
- http://www.gpgpu-sim.org/

- BUT: extremely slow, applications that run for 0.1 seconds on a real GPU can take days or weeks to simulate
- BUT: Bugs

GPGPU-Sim

- GPGPU-Sim can be used on the Ubuntu 12 IRB Maschines, e.g.: furor.cs.tu-berlin.de
- Some environmental variables need to be set:

```
source /afs/tu-berlin.de/units/Fak_IV/aes/tools/cuda/settings64.sh
```

- We will use GPGPU-Sim 3.2.2 for the lab exercises.

Reading GPGPU-Sim Output

Parameter	Meaning
gpu_sim_cycle	Number of (simulated) clock cycles needed to execute kernel
gpu_sim_insn	Number of instructions execute in kernel
gpu_ipc	Average number of instructions executed per cycle

Reading GPGPU-Sim Output

Look for the line next to “Warp Occupancy Distribution:”

Statistic	Description
Stall:	The number of cycles when the shader core pipeline is stalled and cannot issue any instructions.
W0_Idle	The number of cycles when all available warps are issued to the pipeline and are not ready to execute the next instruction.
W0_Scoreboard	The number of cycles when all available warps are waiting for data from memory.
WX(where X = 1 to 32)	The number of cycles when a warp with X active threads is scheduled into the pipeline.

SIMD-Efficiency & Branch Divergence

- The GPU uses 32-wide SIMD Units, but not always all threads of a warp follow the same control flow.
- Threads that follow a different control flow path will be masked out.

SIMD-Efficiency:

$$SIMD\ Efficiency = \frac{1}{32} \frac{\sum_{n=1}^{32} n W_n}{\sum_{n=1}^{32} W_n}$$

Number of Operations

Number of Instructions

Width of SIMD Unit