INTRODUCTION TO GPU COMPUTING WITH CUDA C++

Jiri Kraus, Senior DevTech Compute, 26th November 2019, GPU Seminar, JSC
WHY GPUs ARE FASTER?

**GPU**
28 pJ/flop (SP)
Optimized for Throughput

Pascal
16 nm

**CPU**
126 pJ/flop (SP)
Optimized for Latency

Broadwell E5 v4
14 nm
WHY ARE GPUS MORE ENERGY EFFICIENT?

1. The end of voltage scaling

The Good Old Days
Leakage was not important, and voltage scaled with feature size

\[
\begin{align*}
L' &= \frac{L}{2} \\
C' &= \frac{C}{2} \\
V' &= \frac{V}{2} \\
E' &= C'V'^2 = \frac{E}{8} \\
f' &= 2f \\
D' &= \frac{1}{L'^2} = 4D
\end{align*}
\]

\[
P' = E'f'D' = \left(\frac{E}{8}\right) \cdot 2f \cdot 4D = P
\]

Halve L and get 4x the transistors and 8x the capability for the same power

The New Reality
Leakage has limited threshold voltage, largely ending voltage scaling

\[
\begin{align*}
L' &= \frac{L}{2} \\
C' &= \frac{C}{2} \\
V' &= V \\
E' &= C'V'^2 = \frac{E}{2} \\
f' &= f \\
D' &= \frac{1}{L'^2} = 4D
\end{align*}
\]

\[
P' = E'f'D' = \left(\frac{E}{2}\right) \cdot f \cdot 4D = 2P
\]

Halve L and get 4x the transistors and 4x the capability for 2x the power, or the same capability for half the power in \(\frac{1}{4}\) the area.
WHY ARE GPUs MORE ENERGY EFFICIENT?

2. simpler cores are more energy efficient
ACCELERATED COMPUTING

**GPU**
Optimized for Throughput

**CPU**
Optimized for Latency
ACCELERATED COMPUTING

CPU Strengths

• Small number of threads can run very quickly
• Very fast clock speeds
• Latency optimized via large caches
• Very large main memory

CPU Weaknesses

• Relatively low memory bandwidth
• Cache misses very costly
• Low performance/watt
ACCELERATED COMPUTING

GPU
Optimized for Throughput

GPU Strengths
- High throughput
- High performance/watt
- Significantly more compute resources
- Latency tolerant via parallelism
- High memory bandwidth

GPU Weaknesses
- Relatively low memory capacity
- Low per-thread performance
CPUS AND GPUS
High throughput or Low Latency

CPU architecture must minimize latency within each thread
GPU architecture hides latency with computation from other thread warps
HOW GPU ACCELERATION WORKS

Application Code

Compute-Intensive Functions

Rest of Sequential CPU Code

GPU

CPU
WHY DO I NEED CPUS AND GPUs?
Two Computing Models For Accelerators

Many-Weak-Cores (MWC) Model
Single CPU Core for Both Serial & Parallel Work

Heterogeneous Computing Model
Complementary Processors Work Together

Xeon Phi (And Others)
Many Weak Serial Cores

CPU
Optimized for Latency

GPU Accelerator
Optimized for Throughput
AMDAHL’S LAW ANALYSIS

98% Parallel Work

1 CPU

2 MWC (0.25x CPU)

1 CPU + 1 GPU

10x Speedup assumed for GPU and 1 MWC
AMDAHL’S LAW ANALYSIS
90% Parallel Work

1 CPU
2 MWC (0.25x CPU)
1 CPU + 1 GPU

10x Speedup assumed for GPU and 1 MWC
AMDAHL’S LAW ANALYSIS

70% Parallel Work

1 CPU

2 MWC (0.25x CPU)

1 CPU + 1 GPU

10x Speedup assumed for GPU and 1 MWC
ACCELERATED COMPUTING

Conclusion

- GPUs offer higher compute throughput through many, simple and energy efficient low clocked cores
- GPUs are latency tolerant by hiding latency with independent work
- Latency hiding and many cores require parallelism
- Parallelism requires high memory bandwidth
- Limited capacity of high bandwidth memory makes it important to manage data locality
- Latency optimized CPUs are still required due to Amdahl’s Law
Identify Available Parallelism

Optimize Kernel Performance

Optimize Data Locality

Express Parallelism
EXPRESS PARALLELISM
SAXPY IN CUDA C++

CPU

```cpp
void saxpy( float* y, float a, float* x, int n ) {
    for ( int i = 0; i < n; ++i ) {
        y[i] += a*x[i];
    }
}
saxpy(y,1.0f,x,n);
```

GPU (serial)

```cpp
__global__
void saxpy( float* y, float a, float* x, int n ) {
    for ( int i = 0; i < n; ++i ) {
        y[i] += a*x[i];
    }
}
saxpy<<<1,1>>>(y,1.0f,x,n);
cudaDeviceSynchronize();
```

Inspired by: [https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/](https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/); See Backup for CUDA Fortran
COMPILING, LINK AND RUN
Using NVCC

Compile with NVCC:

$ nvcc -arch=sm_70 -std=c++11 saxpy.cu -o saxpy
$ ls saxpy
saxpy

Run:

$ ./saxpy
BW (GiB/s) = 0.185674
## GV100 SM

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>7.0</td>
</tr>
<tr>
<td>FP32 Cores</td>
<td>64</td>
</tr>
<tr>
<td>INT32 Cores</td>
<td>64</td>
</tr>
<tr>
<td>FP64 Cores</td>
<td>32</td>
</tr>
<tr>
<td>Tensor Cores</td>
<td>8</td>
</tr>
<tr>
<td>Register File</td>
<td>256 KiB</td>
</tr>
<tr>
<td>L1 cache</td>
<td>Up to 128 KiB*</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>Up to 96 KiB*</td>
</tr>
<tr>
<td>Active Threads</td>
<td>2048</td>
</tr>
<tr>
<td>Active Blocks</td>
<td>32</td>
</tr>
</tbody>
</table>

*Configurable
THREADS, (WARPS) AND BLOCKS

A GV100 SM can handle up to 2048 threads, but we are
starting only 1:

\[
\text{saxpy}<<<1,1>>>(y,1.0f,x,n);
\]

Threads are scheduled in groups of 32 threads (warps)

Up to 32 warps (1024 threads) can be grouped into a
block executed on one SM

\[
\text{saxpy}<<<1,1024>>>(y,1.0f,x,n);
\]

\[
\text{threadIdx.x}
\]

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & \ldots & 31
\end{array}
\]

\[
\text{blockIdx.x} = 0
\]
__global__
void saxpy( float* y,  
float a, float* x, int n ) {
    for ( int i = threadIdx.x;  
i < n;  
i += blockDim.x ){
        y[i] += a*x[i];
    }  
}

saxpy<<<1,1024>>>(y,1.0f,x,n);
cudaDeviceSynchronize();

$ nvcc -arch=sm_70 -std=c++11 saxpy.cu -o saxpy$
$ ./saxpy$
BW (GiB/s) = 21.0611 (was 0.185674)
TESLA V100 GPU: GV100

80 SMs
5120 FP32 Cores
7.8 TF Double Precision
15.7 TF Single Precision
125 TF Tensor Core Math
16 GiB HBM2
898 GiB/s Bandwidth
**USING THE FULL GPU**

**Grids of Blocks**

Each GV100 SM can run up to 32 Blocks and there are 80 SMs.

Blocks can be grouped to form a grid executed on one GPU.

\[
\text{gridDim.x} = 8192
\]

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```
__global__

void saxpy( float* y,
            float a, float* x, int n ) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if ( i < n ) {
        y[i] += a*x[i];
    }
}

saxpy<<<((n-1)/128)+1,128>>>(y,1.0f,x,n);
cudaDeviceSynchronize();
SAXPY PERFORMANCE
Bandwidth and Achieved Occupancy vs. size
EXPRESS PARALLELISM

Conclusion

- In CUDA Parallelism is expressed using light weight threads
- Threads are scheduled in groups of 32 (warps), aka SIMT
  => Block size should be a multiple of 32
- Threads can be grouped into blocks of up 1024 threads executing on a single SM
  - Allows fast synchronization and coordination of threads in a block
- A GV100 SM can execute up to 2048 threads and 32 blocks concurrently
  => To allow running at full occupancy at least 64 threads per block are required (other resources might still limit occupancy)
- A kernel can be launched as a grid of multiple thread blocks running on one GPU
OPTIMIZING DATA LOCALITY
HETEROGENEOUS PLATFORM

Memory hierarchy
UNIFIED MEMORY
Starting with Kepler and CUDA 6

Custom Data Management

System Memory

GPU Memory

Developer View With Unified Memory

Unified Memory
UNIFIED MEMORY

**SAXPY**

```c
float* x, *y;
cudaMallocManaged( &x, n*sizeof(float) );
cudaMallocManaged( &y, n*sizeof(float) );
init<<<((n-1)/128)+1,128>>>(x,y,n);
saxpy<<<((n-1)/128)+1,128>>>(y,1.0f,x,n);
cudaDeviceSynchronize();
for (int i = 0; i < n; ++i) {
    if ( std::fabs( y[i] - 1.0f ) > eps ) {
        std::cerr<<"ERROR y["<<i<<"] = ""<<y[i]"") != 1.0"<<std::endl;
        break;
    }
}
cudaFree( y );
cudaFree( x );
```
UNIFIED MEMORY ON PASCAL

On-demand page migrations

cudaMallocManaged
page fault

GPU memory
~0.9 TB/s

interconnect

page fault

System memory
~0.1 TB/s

map VA to
system memory
UNIFIED MEMORY PERFORMANCE TUNING

General guidelines

Minimize page fault overhead:

Fault handling can take 10s of μs, while execution stalls

Keep data local to the accessing processor:

Higher bandwidth, lower latency

Minimize thrashing:

Migration overhead can exceed locality benefits
UNIFIED MEMORY PERFORMANCE TUNING

New hints in CUDA 8

cudaMemPrefetchAsync(ptr, length, destDevice, stream)

Unified Memory alternative to cudaMemcpyAsync
Async operation that follows CUDA stream semantics

cudaMemAdvise(ptr, length, advice, device)

Specifies allocation and usage policy for memory region
User can set and unset advices at any time
UNIFIED MEMORY PREFETCHING

Code example

```c
float* x, *y;
cudaMallocManaged(&x, n*sizeof(float));
cudaMallocManaged(&y, n*sizeof(float));
init_on_cpu(x, y, n);
cudaMemPrefetchAsync(x, n*sizeof(float), dev_id);
cudaMemPrefetchAsync(y, n*sizeof(float), dev_id);
saxpy<<<((n-1)/128)+1, 128>>>(y, 1.0f, x, n);
cudaMemPrefetchAsync(y, n*sizeof(float), cudaCpuDeviceId);
cudaDeviceSynchronize();
for (int i = 0; i < n; ++i) {
    if (std::fabs(y[i] - 1.0f) > eps) {
        std::cerr<<"ERROR y["<<i<<"] = ""<<y[i]"" != 1.0""<<std::endl;
        break;
    }
} cudaFree(y); cudaFree(x);
```

Physical Memory is allocated by first touch

Faults are expensive, prefetch to avoid excess faults
float * x;
cudaMalloc( &x, n*sizeof(float) );
float * y;
cudaMalloc( & y, n*sizeof(float) );
cudaMemcpyAsync( x, x_h, n*sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpyAsync( y, y_h, n*sizeof(float), cudaMemcpyHostToDevice );

saxpy<<<((n-1)/128)+1,128>>>(y,1.0f,x,n);
cudaMemcpyAsync( y_h, y, n*sizeof(float), cudaMemcpyDeviceToHost );
cudaDeviceSynchronize();
cudaFree( b ); cudaFree( a );

Allocate
Copy to GPU
Use on GPU
Copy to CPU
Free
OPTIMIZING DATA LOCALITY

Conclusions

- Maximize data locality by keeping data close to the processor using it
- Minimize impact of faults and page migrations by prefetching data
- Pipeline independent data transfers and computation to hide transfer times and maximize resource utilization (see backup slides)
- Maximize copy throughput by registering memory (see backup slides)
OPTIMIZE KERNEL PERFORMANCE
ACCESSING GLOBAL MEMORY

Best Practices

- Threads issue instructions in groups of 32 threads (warp) operating in SIMT mode
- Each issued memory instructions generates multiple 32byte transactions
- 4 consecutive 32byte segments form a cache line
- The memory subsystem works most efficient if all moved data is consumed and as few memory instructions as possible are used

threadIdx.x

0 1 2 3 ... 31
ACCESSING GLOBAL MEMORY

optimal access pattern (4byte words) - fully coalesced

float x_val = x[threadIdx.x];
ACCESSING GLOBAL MEMORY
worst case access pattern (4-byte words) - fully uncoalesced

```c
float x_val = x[32*threadIdx.x];
```
SAXPY PERFORMANCE
Achieved Bandwidth for different memory access patterns
OPTIMIZE KERNEL PERFORMANCE

Conclusions

- Size of a memory transactions is 32byte
- Size of a cache line is 128byte
- Coalescing memory accesses is at the largest possible granularity is important for the performance of kernels limited by the memory subsystem
- Possible mitigation strategies:
  - Changing data structures
  - Share data in a block (via shared memory)
THANK YOU FOR YOUR ATTENTION

NVIDIA Application Lab at Jülich:
- Andreas Herten, JSC
- Jiri Kraus, NVIDIA

Jülich GPU Courses
- GPU Programming with CUDA - May 2020
- Directive-based GPU programming with OpenACC - October 2020

Helmholtz GPU Hackathon, March 2020, Berlin: https://www.gpuhackathons.org/
SAXPY CUDA FORTRAN

module mymodule contains
  attributes(global) subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, I
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<n) y(i) = a*x(i)+y(i)
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  real, device :: x_d(2**20), y_d(2**20)
  x_d = 1.0, y_d = 2.0
  ! Perform SAXPY on 1M elements
  call saxpy<<4096, 256>>(2**20, 2.0, x_d, y_d)
end program main

From: https://devblogs.nvidia.com/six-ways-saxpy/
PIPELINING

- Tesla GPUs have DMA engines for host to device, device to host and device to device copies (Copy Engines)
- All DMA engines can operate independently of each other and compute work on the GPU and CPU (requires pinned memory)
- Pipelining independent copies, CPU and GPU work allows to maximize resource utilization and hide transfer times
- CUDA streams allow to express independent streams of work
- CUDA events allow to coordinate work in different streams
PIPELINING

Simple Example without pipelining

H2D, GPU Kernel, D2H, CPU Function, H2D, GPU Kernel, D2H, CPU Function
PIPELINING
Simple Example with optimal pipeline
PIPELINING
Double buffered pipeline: Create streams and pipeline prolog

cudaStream_t streams[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate( streams+i );

// Pipeline prolog
cudaMemcpyAsync( in[0], in_h[0], size, cudaMemcpyHostToDevice, streams[0] );
gpu_kernel<<<grid,block,0,streams[0]>>>(out[0],in[0],n);
cudaMemcpyAsync( out_h[0], out[0], size, cudaMemcpyDeviceToHost, streams[0] );
for (int chunk = 1; chunk < num_chunks; ++chunk) {
    // asynchronously launch GPU work on next chunk
    int next = chunk % 2;
    cudaMemcpyAsync(in[next], in_h[chunk], size, cudaMemcpyHostToDevice, streams[next]);
    gpu_kernel<<<grid,block,0,streams[next]>>>(out[next], in[next], n);
    cudaMemcpyAsync(out_h[chunk], out[next], size, cudaMemcpyDeviceToHost, streams[next]);

    // wait for last chunk to finish and do CPU processing
    cudaStreamSynchronize(streams[(chunk - 1) % 2]);
    cpu_function(out_h[chunk], n);
}
PIPELINING
Double buffered pipeline: Pipeline epilog and destroy streams

//CPU processing of last chunk
cudaStreamSynchronize( streams[(num_chunks-1)%2] );
cpu_function(out_h[num_chunks-1], n);

for (int i = 0; i < 2; ++i)
    cudaStreamDestroy( streams[i] );
PIPELINING

Simple Example with double buffered pipeline
REGISTERING HOST MEMORY

```cpp
int* ptr;
cudaMallocHost( &ptr, n*sizeof(int) );
cudaFreeHost( ptr );

int* ptr = new int[n];
cudaHostRegister( &ptr, n*sizeof(int),
    cudaHostRegisterMapped );
int* ptr_d;
cudaHostGetDevicePointer( &ptr_d, ptr, 0 );
cudaHostUnregister( ptr );
ptr_d = 0;
delete[] ptr;
```

- Accelerates Host to Device and Device to Host copies
- Required for truly asynchronous Host to Device and Device to Host copies
- Allows direct access from the GPU without a copy (Zero Copy - equivalent to direct mapped Unified Memory)
REGISTERING HOST MEMORY
Host to Device Bandwidth on x86 + P100 PCI-E 16GB

![Graph showing host to device bandwidth with pageable and pinned memory]