CSCI-GA.3033-004

Graphics Processing Units (GPUs): Architecture and Programming

CUDA

Advanced Techniques 2

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Alignment
Memory Alignment

• Memory access on the GPU works much better if the data items are aligned at 64 byte boundaries.
• Hence, allocating 2D (or 3D) arrays so that every row starts at a 64-byte boundary address will improve performance.
• Difficult to do for a programmer!
2D Arrays

• CUDA offers special versions of:
  – Memory allocation of 2D arrays so that every row is padded (if necessary). The function determines the best pitch and returns it to the program. The function name is `cudaMallocPitch()`
  – Memory copy operations that take into account the pitch that was chosen by the memory allocation operation. The function name is `cudaMemcpy2D()`
cudaMallocPitch( void** devPtr,
size_t* pitch,
size_t widthInBytes,
size_t height)

• This allocates at least $width \times height$ array.
• The value returned in pitch is the width in bytes of the allocation.
• The above function determines the best pitch and returns it to the program.
• It is strongly recommends the usage of this function for allocating 2D (and 3D) arrays.
cudaError_t cudaMemcpy2D ( void * dst,
    size_t dpitch,
    const void * src,
    size_t spitch,
    size_t width,
    size_t height,
    enum cudaMemcpyKind kind )

• \textit{dst} - Destination memory address
• \textit{dpitch} - Pitch of destination memory
• \textit{src} - Source memory address
• \textit{spitch} - Pitch of source memory
• \textit{width} - Width of matrix transfer (in bytes)
• \textit{height} - Height of matrix transfer (rows)
• \textit{kind} - Type of transfer

the widths in memory in bytes including any padding added to the end of each row
Example

```c
int main(int argc, char * argv[]) {
    float * A, *dA;
    size_t pitch;

    A = (float *)malloc(sizeof(float)*N*N);
    cudaMallocPitch(&dA, &pitch, sizeof(float)*N, N);

    //copy memory from unpadded array A of 760 by 760 dimensions
    //to more efficient dimensions on the device
    cudaMemcpy2D(dA,pitch,A,sizeof(float)*N,sizeof(float)*N,N,
                cudaMemcpyHostToDevice);
    ...
}
```
So..

Pitch is a good technique to speedup memory access

• There are two drawbacks that you have to live with:
  • Some wasted space
  • A bit more complicated elements access
Multi-GPU System
Nebulae: #10 in Top 500 list (June 2012)

Intel Xeon X5650 and Nvidia GPU Tesla c2050
Tsubame 2.0: #5 in Top 500 list

Intel Xeon X5670 and Nvidia GPU
Flavors

- Multiple GPUs in the same node (e.g. PC)
- Multi-node system (e.g. MPI).

Multi-GPU configuration is here to stay!
Hardware Example: Tesla S870 Server
Hardware Example: Tesla S870 Server

Connected to a single-host
Hardware Example: Tesla S870 Server

Connected to a two host systems
Why Multi-GPU Solutions

• Scaling-up performance
• Another level of parallelism
• Power
• Reliability
// Run independent kernel on each CUDA device
int numDevs = 0;
cudaGetDeviceCount(&numDevs);
...
for (int d = 0; d < numDevs; d++) {
    cudaSetDevice(d);
    kernel<<<blocks, threads>>>(args);
}
CUDA Support

• `cudaGetDeviceCount( int * count)`
  - Returns in *count the number of devices

• `cudaGetDevice( int * device )`
  - Returns in *device the device on which the active host thread executes the device code.
CUDA Support

- `cudaSetDevice(devID)`
  - Device selection within the code by specifying the identifier and making CUDA kernels run on the selected GPU.

```c
size_t size = 1024 * sizeof(float);
cudaSetDevice(0); // Set device 0 as current
float* p0;
cudaMalloc(&p0, size); // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
cudaSetDevice(1); // Set device 1 as current
float* p1;
cudaMalloc(&p1, size); // Allocate memory on device 1
MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1
```
Peer-to-Peer Access
CUDA Support: Peer to peer memory Access

- Peer-to-Peer Memory Access
  - `cudaDeviceEnablePeerAccess()` to check peer access

```c
cudaSetDevice(0); // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size); // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
cudaSetDevice(1); // Set device 1 as current
cudaDeviceEnablePeerAccess(0, 0); // Enable peer-to-peer access with device 0

// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address p0
MyKernel<<<1000, 128>>>(p0);
```
What we want to do ...
Does the device support P2P?

cudaError_t cudaDeviceCanAccessPeer (
  int* canAccessPeer,
  int device,
  int peerDevice)

• Returns 1 in canAccessPeer if device can access peerDevice.
• You need to check both directions.
cudaError_t cudaDeviceEnablePeerAccess (int peerDevice, unsigned int flags)

Access granted by this call is unidirectional (i.e. current device can access peer device)

peerDevice ID
Always set to zero

cudaError_t cudaDeviceDisablePeerAccess (int peerDevice)
CUDA Support

Peer to peer memory Copy

• Using cudaMemcpyPeer()
cudaMemcpyPeer ( void * dst,
    int dstDevice,
    const void * src,
    int srcDevice,
    size_t count )

• This function is asynchronous with respect to the host.
• This function is serialized with respect to all pending and future asynchronous work into the current device.

Important: If GPU supports Unified Virtual Address, then no need to the above function.
(We will see shortly)
The Evolution of CPU-GPU Memory Operations

**Milestones**

- Traditional `cudaMemcpy()` (We already saw this!)
- Zero-copy
- Unified Virtual Address (CUDA 4.0 and up)
- Managed Memory (CUDA 6.0 and up)
The Evolution of CPU-GPU Memory Operations

Milestones

- Traditional cudaMemcpy()
- Zero-copy
- Unified Virtual Address (CUDA 4.0 and up)
- Unified Memory (CUDA 6.0 and up)
Unified Virtual Address Space (UVA)

- From CUDA 4.0
- puts all CUDA execution, CPU and GPU, in the same address space
- Requires Fermi-class GPU and above
- Requires 64-bit application
- Call cudaGetDeviceProperties() for all participating devices and check unifiedAddressing flag
Unified Virtual Addressing
Easier to Program with Single Address Space

No UVA: Multiple Memory Spaces

UVA: Single Address Space
Easier Memory Copy

• Between host and multiple devices:
  cudaMemcpy(gpu0_buf, host_buf, buf_size, cudaMemcpyDefault)
  cudaMemcpy(gpu1_buf, host_buf, buf_size, cudaMemcpyDefault)
  cudaMemcpy(host_buf, gpu0_buf, buf_size, cudaMemcpyDefault)
  cudaMemcpy(host_buf, gpu1_buf, buf_size, cudaMemcpyDefault)

• Between two devices:
  cudaMemcpy(gpu0_buf, gpu1_buf, buf_size, cudaMemcpyDefault)

• cudaMemcpy() knows that our buffers are on different devices
• (UVA), will do a P2P copy
• Note that this will transparently fall back to a normal copy through the host if P2P is not available
Example: Direct N-Body

- Simulation of dynamical system of N-bodies
- $O(N^2)$
- Compute-Bound application
- Assume we have $K$ GPUs
  - So each GPU is responsible for $N/K$ bodies
- For each iteration:
  - Get all $N$ up-to-date positions onto each GPU
  - Compute accelerations ($N/k$ per GPU)
  - Integrate position, velocity ($N/k$ per GPU)
Example: Direct N-Body

• Sharing data among GPUs: options
  – Explicit copies via host
  – Zero-copy shared host array
    (cudaMallocHost())
  – Per-device arrays with peer-to-peer exchange transfers (UVA)
  – Peer-to-peer memory access
for(;;) {
    for (int d = 0; d < devs; d++) {
        cudaSetDevice(d);
        cudaMemcpyAsync(pos[d], in, devs*bytes, H2D, s[d]);
    }
    for (int d = 0; d < devs; d++) {
        cudaSetDevice(d);
        integrate<<<b, t, 0, s[d]>>>(pos[d], otherArgs);
    }
    for (int d = 0; d < devs; d++) {
        cudaSetDevice(d);
        cudaMemcpyAsync(&out[offset[d]], pos[d], bytes, D2H, s[d]);
    }
}
Example: Direct N-Body

• Sharing data among GPUs: options
  – Explicit copies via host
  – **Zero-copy shared host array** (direct device access to host memory, through PCIe, which is slow) ... cudaMallocHost() or cudaHostAlloc() ... so, use it when:
    • You copy data to the device only once and access it there AND/OR
    • You generate data on the device and copy back to host without reuse AND/OR
    • Your kernel(s) that access the memory are compute bound
  – UVA
  – Peer-to-peer memory access
N-Body Zero-copy

// Create input and output arrays
cudaHostAlloc(&in, bytes, cudaHostAllocMapped | cudaHostAllocPortable);
cudaHostAlloc(&out, bytes, cudaHostAllocMapped | cudaHostAllocPortable);

Allocates size bytes of host memory that is page-locked and accessible to the device.

Important: If GPU supports Unified Virtual Address, then no need to the above function. (We will see shortly)
// Create input and output arrays
cudaHostAlloc(&in, bytes, cudaHostAllocMapped | cudaHostAllocPortable);
cudaHostAlloc(&out, bytes, cudaHostAllocMapped | cudaHostAllocPortable);

for (int d = 0; d < devCount; d++) {
    cudaSetDevice(d);
    cudaMemcpy(dout[d], hostPtr, 0);
    cudaMemcpy(din[d], hostPtr, 0);
}

pointer that will be passed to the device to access host memory
Example: Direct N-Body

• Sharing data among GPUs: options
  – Explicit copies via host
  – Zero-copy shared host array
    (cudaMallocHost() )
  – Per-device peer-to-peer exchange transfers
    • UVA as we have seen
    • Non-UVA:
      – cudaMemcpyPeer()
      – Copies memory from one device to memory on another device
  – Peer-to-peer memory access
Example: Direct N-Body

• Sharing data among GPUs: options
  – Explicit copies via host
  – Zero-copy shared host array
    (cudaMallocHost() )
  – Per-device peer-to-peer exchange transfers
  – Peer-to-peer memory access
    • Pass pointer to memory on device A to kernel running on device B
    • Requires UVA
    • Must first enable peer access for every pair:
      • cudaDeviceEnablePeerAccess
The Evolution of CPU-GPU Memory Operations

Milestones

- Traditional cudaMemcpy()
- Zero-copy
- Unified Virtual Address (CUDA 4.0 and up)
- Unified Memory (CUDA 6.0 and up)

Source of the next few slides:
Unified Memory

- From Kepler architecture (CC 3.0 and up)
- Creates a pool of managed memory that is shared between the CPU and GPU.
- Managed memory is accessible to CPU and GPU with single pointers.
- Under the hood: data automatically migrates from CPU to GPU.
Unified Memory

Isn’t it like UVA?

• Unified memory depends on UVA.
• UVA does NOT move data automatically between CPU and GPU.
• Unified memory gives higher performance than UVA.
Advantages of Unified Memory

• Ease of programming
• Data is migrated on demand.
  – offer the performance of local data on the GPU
  – while providing the ease of use of globally shared data
• Very efficient with complex data structures (e.g. linked lists, structures with pointers, ...).

Note: The physical location of data is invisible to the program and may be changed at any time
Disadvantages of Unified Memory

- Carefully tuned CUDA program that uses streams to efficiently overlap execution with data transfers may perform better than a CUDA program that only uses Unified Memory.
How to allocated managed memory?

• **Option 1**: `cudaMallocManaged()` routine, which is semantically similar to `cudaMalloc()`

• **Option 2**: defining a global `__managed__` variable, which is semantically similar to a `__device__` variable
int main() {

    int *ret;

    cudaMallocManaged(&ret, 1000 * sizeof(int));

    AplusB<<<1, 1000>>>(ret, 10, 100);
    cudaDeviceSynchronize();

    for (int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);

    cudaFree(ret);
    return 0;
}
__managed__

__device__ __managed__ int ret[1000];

__global__ void AplusB(int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {

    AplusB<<<1, 1000 >>>>(10, 100);
    cudaDeviceSynchronize();

    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);

    return 0;
}
Final Notes About Unified Memory

- Coherence is ahead of performance in runtime implementation. Data has to be coherent across CPUs and GPUs in the system.

- Page faulting is implemented in systems with compute capability 6.x and up \( \rightarrow \text{cudaMallocManaged} \) will not run out of memory as long as there is enough system memory available for the allocation.

- Before that, all managed data must move to the GPU before kernel launch (automatically of course) \( \rightarrow \text{Devices of compute capability lower than 6.x cannot allocate more managed memory than the physical size of GPU memory} \)
Dynamic Parallelism
The Usual case

• Data travels back and forth between the CPU and GPU many times.
• Reason: because of the inability of the GPU to create more work on itself depending on the data.
With Dynamic Parallelism:

- GPU can generate work on itself **without** involvement of CPU.
- Permits Dynamic Run time decisions.
- Kernels can start new kernels
- Streams can spawn new streams.

CUDA 5.0 and later on devices of Compute Capability 3.5 or higher
A kernel can call another kernel that calls another kernel up to 24 nested ... Subject to the availability of resources.
When do we need that?

- Nested for-loop for example
- The need for adaptive grids

Source: https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/
As in the host, device kernel launch is asynchronous.

Successful execution of a kernel launch means that the kernel is queued;
- it may begin executing immediately,
- or it may execute later when resources become available.

Note that every thread that encounters a kernel launch executes it. So be careful!

Child grids always complete before the parent grids that launch them, even if there is no explicit synchronization.
Important

- The CUDA Device Runtime guarantees that parent and child grids have a fully consistent view of global memory when the child starts and ends.

By default, grids launched within a thread block are executed sequentially.

This happens even if grids are launched by different threads within the block.

To deal with this drawback → streams

Streams created on the host cannot be used on the device.

Streams created in a block can be used by all threads in that block.

cudaStream_t s;
cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
Important

• If the parent kernel needs results computed by the child kernel to do its own work → it must ensure that the child grid has finished execution before continuing
  – by explicitly synchronizing using `cudaDeviceSynchronize(void)`.
  – This function waits for completion of all grids previously launched by the thread block from which it has been called.
Example

```c
void threadBlockDeviceSynchronize(void)
{
    __syncthreads();
    if(threadIdx.x == 0)
        cudaDeviceSynchronize();
    __syncthreads();
}
```

To ensure all launches have been made.
What do we gain?

- Reduction in trips to CPU
- Recursion
- More freedom where data generated by the kernel decides how to partition the data for lower-level of the hierarchy.
How to Compile and Link?

```
nvcc -arch=sm_35 -rdc=true myprog.cu -lcudadevrt
```

generate relocatable device code, required for later linking
Hyper-Q
Till Fermi

- Only one work queue
- Even though Fermi allows 16 concurrent kernels.
- GPU resources not fully utilized
Fermi already supported 16 way concurrency of kernel launches from separate streams. Pending work is bottlenecked on 1 work queue. GPU’s computational resources not being utilized fully.
With Hyper-Q

• Starting with Kepler
• We can have connection from multiple CUDA streams, Message Passing Interface (MPI) processes, or multiple threads of the same process.
  – 32 concurrent work queues, can receive work from 32 process cores at the same time.
  – 3X Performance increase on Fermi
With Hyper-Q

With Hyper-Q

KEPLER
32 Simultaneous MPI Tasks
Stream Queue Management

Grid Management Unit
(1000s of pending grids)

Work Distributor
(16 active grids)

SMX SMX SMX SMX
Conclusions

- There are many performance enhancement techniques in our arsenal:
  - Alignment
  - Streams
  - Pinned pages
  - Asynchronous execution
  - Dynamic Parallelism
  - Multi-GPU
- There are tools to help you!