A Tutorial on CUDA Performance Optimizations

Amit Kalele
Prasad Pawar
Parallelization & Optimization CoE
TCS Pune
Outline

Overview of GPU architecture

Optimization Part I
• Block and Grid size
• Shared memory
• Coalesced memory access
• Bank Conflict
• Constant memory

Optimizations Part II
• Dynamic Parallelism
• Hyper-Q Streams
• Unified Memory Access
• GPUDirect with MultiGPUs
• Default streaming
• Warp Shuffle
• Read Only Memory
• GPUs – High latency, high but more simpler ALUs
• CPUs – low latency, less but more powerful ALUs
• CPUs – Host or Master, offloads work to GPU via PCI-x16
• GPUs – Device or Co processor, worker
• C2075 Tesla has 14 SMs with 32 cores each
What is CUDA?

- CUDA Architecture
  - Expose general-purpose GPU computing as first-class capability
  - Retain traditional DirectX/OpenGL graphics performance
- CUDA C
  - Based on industry-standard C
  - A handful of language extensions to allow heterogeneous programs
  - Straightforward APIs to manage devices, memory, etc
CUDA C basics

Device Code

```c
__global__ void kernel( void ) {}`
```

- **Keyword `__global__`** Indicates that a function
  - Runs on the device
  - Called from host code

- **`nvcc`** splits source file into host and device components
  - NVIDIA’s compiler handles device functions like `kernel()`
  - Standard host compiler handles host functions like `main()`
    - `gcc`
    - Microsoft Visual C++
    - `icc`
In its simplest form it looks like:

```c
kernel_routine<<<gridDim, blockDim>>>(args);
```

*gridDim* is the number of instances/blocks of the kernel (the “grid” size)

*blockDim* is the number of threads within each Instance (the “block” size)

*args* is a limited number of arguments, usually mainly pointers to arrays in graphics memory, and some constants which get copied by value
Optimizations Part I

• Block and Grid size
• Shared memory
• Coalesced memory access
• Bank Conflict
• Constant memory
# of blocks > # of multi-processors
So all multiprocessors have at least one block to execute

# of blocks / # of multiprocessors > 2
Multiple blocks can run concurrently in a multiprocessor, subject to resource availability like registers, shared memory

- Thread block size is a multiple of warp size (32).
- Even if you request fewer threads, HW rounds up.
- Thread blocks should not be too small or too big.
Thread Block Sizing

Too few threads per block

Number of warps allowed by SM resources

Too many threads per block
**Problem** – 3x3 matrix multiplication and 60k pairs

60K pairs: Each thread block supports max 1K threads, so we have
NUM_BLK = 60, THRDS_PER_BLK = 1024
Kernel<<<60, 1024>>>

All issues points towards reducing number of threads per block
• Use 64 threads per block as a guideline
• Launch configuration: NUM_BLK = 60K/64, THRDS_PER_BLK = 64
  Kernel<<<60K/64, 64>>>

<table>
<thead>
<tr>
<th>Optimization Parameter</th>
<th>Kernel time</th>
<th>GFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch configuration (60, 1024)</td>
<td>1.03ms</td>
<td>3.22 GF</td>
</tr>
<tr>
<td>Launch configuration (960, 64)</td>
<td>413µsec</td>
<td>8.01GF</td>
</tr>
</tbody>
</table>
Shared Memory

- Read/write per-block
- Speed equivalent to local cache
- 100x faster than Global memory
- Limit up to 48KB
Shared Memory

• **When to use shared memory**
  – Shared memory enables cooperation between threads in a block.
  – When multiple threads in a block use the same data from global memory, shared memory can be used to access the data from global memory only once.
  – Shared memory can also be used to avoid un-coalesced memory accesses by loading and storing data in a coalesced pattern from global memory and then reordering it in shared memory.

• **Issues**
  – Bank conflict
Shared Memory - Example

Matrix Multiplication without shared memory

```c
__global__ void simpleMultiply(float *a, float* b, float *c, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
        sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

Matrix Multiplication with shared memory

```c
__global__ void sharedABMultiply(float *a, float* b, float *c, int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM], bTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
    __syncthreads();
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
    }
    c[row*N+col] = sum;
}
```
Shared Memory Bank Conflicts

Address Mapping in 8-byte Mode

Byte-address: 0 4 8 12 16 20 24 28 32 36 40

Data: 

0 1 2 3 4 5 6 7 8 9
(or 4B-word index)

Bank-0 Bank-1 Bank-2 Bank-3 Bank-31

Source -
A bank conflict occurs when:

2 or more threads in a warp access different 8-B words in the same bank

- Think: 2 or more threads access different “rows” in the same bank

N-way bank conflict: N threads in a warp conflict

Note there is no bank conflict if:

- Several threads access the same word
- Several threads access different bytes of the same word
Shared Memory Bank Conflicts

Addresses from a warp: no bank conflicts
One address access per bank

Addresses from a warp: no bank conflicts
Multiple addresses per bank, but within the same word

Shared Memory Bank Conflicts

**Addresses from a warp: 2-way bank conflict**
2 accesses per bank, fall in two different words

```
Bank-0  Bank-1  Bank-2  Bank-3  ...
```

**Addresses from a warp: 3-way bank conflict**
4 accesses per bank, fall in 3 different words

```
Bank-0  Bank-1  Bank-2  Bank-3  ...
```
Coalesced Memory Access

• Recall that all threads in a warp execute the same instruction.

• When all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive memory locations.

• The most favorable global memory access is achieved when the same instruction for all threads in a warp accesses global memory locations.

• If thread 0 accesses location n, thread 1 accesses location n + 1, ... thread 31 accesses location n + 31, then all these accesses are coalesced, that is: combined into one single access.
typedef struct Node {
    int Type;
    real_dt Vel[2];
    real_dt Density;
    real_dt F[9];
    real_dt Ftmp[9];
} Node;
Case Study - Lattice Boltzmann Method (LBM)

**T - 0  T - 1**

(All Threads simultaneously accessing **Density**)

**Stride**

**T - 0  T - 1**

(All Threads simultaneously accessing **Density**)

**Coalesced Access pattern**
## Case Study LBM - Result

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Experiment</th>
<th>Time in Sec</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Strided Memory Access pattern for Node Structure</td>
<td>11.213</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Coalesced Memory Access pattern for Node Structure</td>
<td>4.008</td>
<td>2.79x</td>
</tr>
</tbody>
</table>
Constant memory

Where is constant memory?
- Data is stored in the device global memory
- Read data through multiprocessor constant cache
- 64KB constant memory and 8KB cache for each SM

```c
__constant__ float cst_ptr[size];
```

copy data from host to constant memory
```
cudaMemcpyToSymbol(cst_ptr,host_ptr,data_size)
```

When to use constant memory?
- The same data is required by all the threads
- Data is constant through out the kernel execution
- The data size is less than or equal to 64kB
Performance with ease of programming

- Dynamic Parallelism
- Hyper-Q Streams
- Unified Memory Access
- GPUDirect with MultiGPUs
- Warp Shuffle
- Default streaming
- Read Only Memory
Without Dynamic Parallelism

Data travels back and forth between the CPU and GPU many times.

This is because of the inability of the GPU to create more work on itself depending on the data.

With Dynamic Parallelism:

Allows a kernel to call another kernel from within it without returning to the host.

Permits Dynamic Run Time decisions to call kernel with different grid and block size.

Leaves the CPU free to do other work.
Notice the kernel call is a standard syntax and allows each kernel call to have different grid/block structures
Dynamic Parallelism

Programming Model Basics

- CUDA Runtime syntax & semantics
- Launch is per-thread
- Sync includes all launches by any thread in the block
- `cudaDeviceSynchronize()` does not imply `syncthreads`
- Asynchronous launches only

```c
__device__ float buf[1024];
__global__ void dynamic(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];
    __syncthreads();
    if(tid == 0) {
        launch<<< 128, 256 >>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();
    cudaMemcpyAsync(data, buf, 1024);
    cudaDeviceSynchronize();
}
```

Case Study : Time of Use Tariff Model

Fixed tariff model had revenue: $R_{std}$
Variable charge model was chosen with following parameters:
• Daily Fix charges (DFC)
• Off peak charges (Or)
• Shoulder peak charges (Sr)
• Peak Charges(Pr)
• Critical peak charges(Cr)
  • 12 max consumption days from last year was chosen and these are charged additionally
• Revenue under this model is $R_{mld}$:
  where Oc, Sc, Pc, Cc are respective consumption for these zones

$$R_{mdl} = CHG_{fix} + CHG_{var}$$

$$CHG_{fix} = Days \times Total_{users} \times DFC$$

$$CHG_{var} = O_r \times \sum_{user} O_c + S_r \times \sum_{user} S_c + P_r \times \sum_{user} P_c + C_r \times (\sum_{i=1}^{12} C_c[i])$$
Case Study : Time of Use Tariff Model

Rate parameters DFC, Or, Sr, Pr, Cr required to be chosen with care
• If the chosen value results in $R_{mld} \ll R_{std}$ then it’s a financial loss to company
• If $R_{mld} \gg R_{std}$ then it is not acceptable to regulators

Problem Statement: Find the values of rate parameters DFC, Or, Sr, Pr, Cr such that the $|R_{mld} - R_{std}|$ is minimized.

Since it’s an absolute difference, the problem becomes quadratic:
Find the values of rate parameters DFC, Or, Sr, Pr, Cr such that the $(R_{mld}^2 + R_{std}^2 - 2*R_{mld}*R_{std})$ is minimized.
Grid Search Problem

\[ f(A, b) \quad \text{where say } A : 1 \text{ to } 5, \ b : 0.1 \text{ to } 1 \]

**Problem** Find parameters \((A, b)\) such that \(|f(A, b)| - Cr \leq t\) for some reference value \(Cr\) and a user-defined threshold \(t\).

Both parameters \((A, b)\) are continuous, so to perform grid search, one selects a finite set of “reasonable” values for each, say \(A\={1,3,5}\) and \(b\={0.1, 0.3, 0.5, 0.7, 0.9}\).
Grid Search Problem

Grid Search Space (A x Beta)

- The grid search space is represented by a 2D plane with axes A and Beta.
- The x-axis represents the values of A ranging from 0 to 6.
- The y-axis represents the values of Beta ranging from 0 to 1.
- The grid points are marked with colored dots, indicating different combinations of A and Beta.
- Each dot represents a potential point for exploration in the grid search.
Grid Search Problem

Grid Search Space \((A \times \text{Beta})\) stage 1

Finer Grid space around Point \(P\) stage 2

Finer Grid space around Point \(Q\) stage 3
Hyper-Q

**Fermi architecture** support to launch 16 kernels from separate streams.

But ultimately the streams were all multiplexed into the **same hardware work queue**.

**Kepler architecture** allows 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.

By enabling more MPI processes on the GPU, Hyper-Q maximizes GPU utilization, increasing overall performance.
Hyper-Q

Removes the problem of false intra-stream dependencies.

**Fermi Model**

- **STREAM 1**: A, B, C
- **STREAM 2**: P, Q, R
- **STREAM 3**: X, Y, Z

Single hardware work queue

**Kepler Hyper-Q Model**

- **STREAM 1**: A, B, C
- **STREAM 2**: P, Q, R
- **STREAM 3**: X, Y, Z

Each stream receives its own work queue
Hyper-Q

Hyper-Q : Example

```c
for (int i = 0 ; i < nstreams ; ++i) // nstreams=32
{
    kernel_A<<<1,1,0,streams[i]>>>(A_data);
    kernel_B<<<1,1,0,streams[i]>>>(B_data);
}
```

NVVP result Without Hyper-Q ( on Fermi )

<p>| Stream 6 | kernel_A(long*, lo...) | kernel_B(long*, lo...) |
| Stream 7 | kernel_A(long*, lo...) | kernel_B(long*, lo...) |
| Stream 8 | kernel_A(long*, lo...) | kernel_B(long*, lo...) |
| Stream 9 | kernel_A(long*, lo...) | kernel_B(long*, lo...) |
| Stream 10 | kernel_A(long*, lo...) | kernel_B(long*, lo...) |
| Stream 11 | kernel_A(long*, lo...) | kernel_B(long*, lo...) |
| Stream 12 |
| Stream 13 |
| Stream 14 |
| Stream 15 |</p>
<table>
<thead>
<tr>
<th>Streams</th>
<th>kernel_A(long*, long)</th>
<th>kernel_B(long*, long)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Unified Memory Access

Unified Memory -

Simplifies programming by enabling applications to access CPU and GPU memory without the need to manually copy data from one to the other.

Unified Memory Access (UMA) is a shared memory architecture used in parallel computers.

All the processors in the UMA model share the physical memory uniformly.

In a UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data.
Unified memory Access

**Unified Memory**
Dramatically Lower Developer Effort

**Developer View Today**

**Developer View With Unified Memory**

int main(int argc, char *argv[]){
    float *A,*B,*C;
    float *d_A,*d_B,*d_C;
    A = (float *)calloc(1,sizeof(float)*N*SIZE*SIZE);
    B = (float *)calloc(1,sizeof(float)*N*SIZE*SIZE);
    C = (float *)calloc(1,sizeof(float)*N*SIZE*SIZE);

    cudaMemcpy(d_A, A, sizeof(float)*N*SIZE*SIZE, cudaMemcpyHostToDevice) ;
    cudaMemcpy(d_B, B, sizeof(float)*N*SIZE*SIZE, cudaMemcpyHostToDevice) ;
    cudaMemcpy(d_C, 0, sizeof(float)*N*SIZE*SIZE) ;

    Matrix_Mul<<<N/blockSize+1,blockSize>>>(d_A, d_B, d_C);
    cudaMemcpy(C, d_C, sizeof(float)*N*SIZE*SIZE, cudaMemcpyDeviceToHost) ;
}

int main(int argc, char *argv[]){
    float *A, *B, *C;

    checkCudaErrors(cudaMallocManaged(&A, sizeof(float)*N*SIZE*SIZE));
    checkCudaErrors(cudaMallocManaged(&B, sizeof(float)*N*SIZE*SIZE));
    checkCudaErrors(cudaMallocManaged(&C, sizeof(float)*N*SIZE*SIZE));

    Matrix_Mul<<<N/blockSize+1, blockSize>>>(A, B, C);
GPUDirect

Before

```
cudaMemcpy(gpu0_buf, gpu1_buf, buf_size, cudaMemcpyDefault)
```

cudaMemcpy() knows that buffers are on different devices

After
With UVA and CUDA-aware MPI

```c
//MPI rank 0
MPI_Send(s_buf_d,size,...);

//MPI rank n-1
MPI_Recv(r_buf_d,size,...);
```

No UVA and regular MPI

```c
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,...);
MPI_Send(s_buf_h,size,...);

//MPI rank n-1
MPI_Recv(r_buf_h,size,...);
cudaMemcpy(r_buf_d,r_buf_h,size,...);
```

Streams Simplify Concurrency

- CUDA Streams allows concurrency on Device.
- Different Streams may execute their commands concurrently.

```c
kernel<<< blocks, threads, bytes >>>(); // default stream

kernel<<< blocks, threads, bytes, 0 >>>(); // stream 0
kernel<<< blocks, threads, bytes, 1 >>>(); // stream 1
```

**Default Stream before CUDA 7:**

- Each Device has a single default stream used for all host threads.
- This causes implicit synchronization.
- Means two commands from different Streams cannot run concurrently if host thread issues any CUDA command to the default stream between them.
Default Stream

Default Stream with CUDA 7:
Introduces new option, the per-thread default Stream

First Effect:
It gives each host thread its own default Stream.
Command issued to the default Stream by different host threads can run concurrently.

Second Effect:
These default Streams are regular Streams.
Command in the default Stream may run concurrently with commands in non-default Streams.

To enable per-thread default streams in CUDA 7, you can either compile with the nvcc command-line option:

--default-stream per-thread
#define CUDA_API_PER_THREAD_DEFAULT_STREAM

preprocessor macro before you include CUDA headers: (cuda.h or cuda_runtime.h)
void *launch_kernel(void *dummy)  
{  
    float *data;  
    cudaMalloc(&data, N * sizeof(float));  
    kernel<<<1, 64>>>(data, N);  
    cudaStreamSynchronize(0);  
    return NULL;  
}  

int main()  
{  
    const int num_threads = 8;  

    #pragma omp parallel for  
    for (int i = 0; i < num_threads; i++) {  
        launch_kernel(0);  
    }  
    cudaDeviceReset();  

    return 0;  
}
Multi-Threading Example

nvcc omp_test.cu -o omp_legacy
Multi-Threading Example

```
nvcc -default-stream per-thread omp_test.cu -o omp_per_thread
```
Warp shuffle

- Instruction to exchange data in a warp
- Threads can “read” other threads’ registers
- No shared memory is needed
- It is available starting from SM 3.0
- `shfl` requires all threads being accessed to be active.

**Example:**

```c
for (i=0; i<n; i++)
{
    A[i] += B[i+2];
}
```

```c
for (i=0; i<n; i++)
{
    A[i] += B[i-1];
    B[i] = N;
}
```
Warp shuffle

```c
int __shfl(int var, int srcLane);
__shfl(var,1)
```

```c
int __shfl_up(int var, unsigned int delta);
__shfl_up(var,1)
```

```c
int __shfl_down(int var, unsigned int delta);
__shfl_down(var,2)
```
Example of Warp shuffle

Ex. Reduction

```c
for (int i=0; i<8; i++)
    sum += a[i];
for (int i=4; i>0; i=i/2)
    v += __shfl_down(value, i);
```

![Diagram of LaneID and data shuffle](image)

- `v+=__shfl_down(v,4)`
- `v+=__shfl_down(v,2)`
- `v+=__shfl_down(v,1)`
Read-Only Cache Memory

- The read-only data cache was introduced with Compute Capability 3.5 architectures (e.g. Tesla K20c/K20X and GeForce GTX Titan/780 GPUs).
- Each SMX has a 48KB read-only cache.
- The CUDA compiler automatically accesses data via the read-only cache when it can determine that data is read-only for the lifetime of kernel.
- In practice, you need to qualify pointers as const and __restrict__ before the compiler can satisfy this condition.
- Also specify a read-only data cache access with the __ldg() intrinsic function.

```c
__global__ void ReadOnlyFilter(…, float* FilterCoeffs)

__global__ void ReadOnlyFilter(…, float const* __restrict__ FilterCoeffs)
```
Case Study - Result

1. **Time of use Tariff Optimization**: The constrained quadratic optimization problem was solved using recursive 5-dim grid search. The grid search was implemented in CUDA using dynamic parallelism.
   
   - The revenue difference was reduced < $10 from $200,000.

2. **LBM** is a class of CFD methods for fluid simulation
   
   - Converted data from array of structures to structure of arrays to access the global memory in coalesced way.
THANK YOU