Sparse Linear Algebra in CUDA

Oliver Meister
November 27\textsuperscript{th} 2013
Table of Contents

Recap on Coalesced Memory Access

Homework - Worksheet 2

PageRank algorithm

Compressed Sparse Row (CSR) Kernels
  Scalar kernel
  Vectorized kernel
Last Tutorial

Matrix Multiplication: Optimizations

- coalesced memory access
- overlapped memory access and computation

Roofline Model

- Compares hardware and kernel performance limits
- Hints for optimizations through ceilings

Performance tools

- Nvidia Visual Profiler: GUI for easy performance debugging
- Important measures: Occupancy and Warp serialize
The third generation SM introduces several architectural innovations that make it not only the most powerful SM yet built, but also the most programmable and efficient.

- **512 High Performance CUDA cores**
- Each SM features 32 CUDA processors—a fourfold increase over prior SM designs. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).
- Prior GPUs used IEEE 754-1985 floating point arithmetic. The Fermi architecture implements the new IEEE 754-2008 floating-point standard, providing the fused multiply-add (FMA) instruction for both single and double precision arithmetic. FMA improves over a multiply-add (MAD) instruction by doing the multiplication and addition with a single final rounding step, with no loss of precision in the addition. FMA is more accurate than performing the operations separately.
- GT200 implemented double precision FMA.
- In GT200, the integer ALU was limited to 24-bit precision for multiply operations; as a result, multi-instruction emulation sequences were required for integer arithmetic. In Fermi, the newly designed integer ALU supports full 32-bit precision for all instructions, consistent with standard programming language requirements. The integer ALU is also optimized to efficiently support 64-bit and extended precision operations. Various instructions are supported, including Boolean, shift, move, compare, convert, bit-field extract, bit-reverse insert, and population count.

- **16 Load/Store Units**
- Each SM has 16 load/store units, allowing source and destination addresses to be calculated for sixteen threads per clock. Supporting units load and store the data at each address to cache or DRAM.

(source: NVIDIA – Fermi/Kepler Whitepapers)
Recap: Coalescing

Hardware limitations:
- Fermi: 1 dispatcher per warp $\Rightarrow$ 1 concurrent instruction
- Kepler: 2 dispatchers per warp $\Rightarrow$ 2 concurrent instructions

Bottleneck:
- biggest LD/ST instruction can transfer 128 B chunk
- if threads in a warp access multiple 128 B chunks in global memory, multiple LD/ST instructions per warp have to be dispatched.
- if all threads access the same 128 B chunk, a single LD/ST per warp of size 128 B is enough.
Recap: Coalescing

On this slide:
\[
\text{tx} = \text{threadIdx.x}; \text{ty} = \text{threadIdx.y}; \text{tz} = \text{threadIdx.z};
\]

Coalescing:

- Ideally: each thread in a warp accesses the same 128 B chunk
- In order to achieve that, we must know which threads of a block are in a warp.
- CUDA uses row-major order for thread indices, so:
  \[
  \text{threadID} = ((\text{tz} \times \text{blockDim}.y + \text{ty}) \times \text{blockDim}.x + \text{tx});
  \]
  \[
  \text{warpID} = \text{threadID} / 32;
  \]
- Rule of thumb: Memory access to a float array \( A[f(\text{threadID})] \) is coalesced, if \( f(\text{threadID}) / 32 \) is equal for all threads with equal \( \text{warpID} = \text{threadID} / 32 \).

Task:
- Solve assignment 1 on your worksheet.
- Besides coalescing, no further hardware optimizations are assumed to happen.
Assignment 1a, b, c - Roofline Model

Roofline: matrix multiplication on NVS 5200M

computational intensity

Oliver Meister: Sparse Linear Algebra in CUDA
Tutorial Parallel Programming and High Performance Computing, November 27th 2013
Assignment 1d - Hardware Profiler

Results:

- Occupancy:
- Warp serialize:
Assignment 1d - Hardware Profiler

Results:
- Occupancy: good
- Warp serialize:
Assignment 1d - Hardware Profiler

Results:

- Occupancy: good
- Warp serialize: bad - bottleneck due to serialization
Assignment 1d - Hardware Profiler

Results:

- Occupancy: good
- Warp serialize: bad - bottleneck due to serialization

One level deeper:

- Branches:
- Memory access:
Assignment 1d - Hardware Profiler

Results:
- Occupancy: good
- Warp serialize: bad - bottleneck due to serialization

One level deeper:
- Branches: good
- Memory access:
Assignment 1d - Hardware Profiler

Results:

- Occupancy: good
- Warp serialize: bad - bottleneck due to serialization

One level deeper:

- Branches: good
- Memory access: bad - uncoalesced access
Assignment 2a, b - Coalesced memory access

Most of you got it right

- Multiple indices always error-prone
- Test your code properly
Assignment 3a - Prefetching

```c
__global__ void mm_o(float* Ad, float* Bd, float* Cd, int n) {
    /** snip **/

    float Celem = 0;
    float Areg = Ad[ i*n + tx];
    float Breg = Bd[ ty*n + k];

    for(int m=1; m < n/TILE_SIZE; m++) {
        Ads[ty][tx] = Areg;
        Bds[ty][tx] = Breg;
        __syncthreads();

        Areg = Ad[ i*n + m*TILE_SIZE+tx];
        Breg = Bd[ (m*TILE_SIZE+ty)*n + k];

    }/** cont. **/
```
/** cont. **/

    for(int j=0; j<TILE_SIZE; j++)
        Celem += Ads[ty][j]*Bds[j][tx];

    __syncthreads();
};

Ads[ty][tx] = Areg;
Bds[ty][tx] = Breg;
__syncthreads();

    for(int j=0; j<TILE_SIZE; j++)
        Celem += Ads[ty][j]*Bds[j][tx];

    Cd[i*n+k] += Celem;
}
Assignment 3a - Prefetching

Some things worth mentioning:

- Allocating local registers corresponds to declaring a local variable in C, no need for further keywords
- Careful: `register float a[n];` allocates kernel-local memory, but it is actually located in global memory and thus very slow
- After computation of the last tile, no sync is required (thanks, Arno!)
- Overlapping: Memory instructions BEFORE computation
Assignment 2c, 3b, c - Updated roofline model

Roofline: matrix multiplication on NVS 5200M

→ What are your results?
Assignment 2c, 3b, c - Updated roofline model

The Roofline Model for NVIDIA GeForce GT 425M

- Single precision performance: 215.04 GFlop/s
- Memory bandwidth: 25.6 GB/s
- Best performance record: 40.319 GFlop/s (using cublasSgemm())

Oliver Meister: Sparse Linear Algebra in CUDA
Tutorial Parallel Programming and High Performance Computing, November 27th 2013

pengcheng.li@tum.de
Prefetching - a resource trade-off

Prefetching: no massive performance gain on all systems!
Example: NVidia Quadro NVS 290

- before prefetching 10 Registers per thread $\rightarrow$
  $16 \times 16 \times 10 = 2560$ registers per block
- 8192 registers per SM $\rightarrow$ 3 active blocks
Prefetching: no massive performance gain on all systems!
Example: NVidia Quadro NVS 290

- before prefetching 10 Registers per thread $\rightarrow$ $16 \times 16 \times 10 = 2560$ registers per block
- 8192 registers per SM $\rightarrow$ 3 active blocks
- now 16 registers $\rightarrow$ $16 \times 16 \times 16 = 4096$ registers per block
- only 8192 registers per SM $\rightarrow$ only 2 active blocks per SM!
Prefetching - a resource trade-off

Prefetching: no massive performance gain on all systems!
Example: NVidia Quadro NVS 290

- before prefetching 10 Registers per thread \( \rightarrow 16 \times 16 \times 10 = 2560 \) registers per block
- 8,192 registers per SM \( \rightarrow 3 \) active blocks
- now 16 registers \( \rightarrow 16 \times 16 \times 16 = 4096 \) registers per block
- only 8,192 registers per SM \( \rightarrow \) only 2 active blocks per SM!

Number of active threads

- before: 3 active blocks \( \rightarrow 24 \) active warps \( \rightarrow 768 \) active threads \( \rightarrow \) optimal occupancy
- after: 2 active blocks \( \rightarrow 16 \) active warps \( \rightarrow 512 \) active thread \( \rightarrow \) less parallelism, less latency hide
Towards High-Performance Matrix Multiplication

More options for optimization:

- loop unrolling (save loop instructions and address arithmetics)
- thread granularity: compute $1 \times 2$ or $1 \times 4$ blocks per thread (requires to load Ads or Bds only once)
- how do different optimizations interact with resource limitations (available registers, etc.)
Sparse Linear Algebra

Nathan Bell and Michael Garland
Efficient Sparse Matrix-Vector Multiplication on CUDA.
2008.

Goals:

- Large matrices in big applications always sparse
- Efficient treatment of sparsity
- Towards higher numbers of unknowns
PageRank algorithm

Let $B \in \mathbb{R}^{n \times n}$ be a non-negative matrix, $\alpha \in (0, 1)$. If $B$ is not left stochastic, apply step a of the algorithm.

Goal: Find non-zero $x \in [0, 1]^n$ with $x = Bx$.

a if $B$ has a 0-column, no solution exists. Otherwise, set $B \leftarrow B C^{-1}$, where $C$ is a diagonal matrix that contains the column sums of $B$.

b initialize solution vector: $x \leftarrow \frac{1}{n} e$, where $e = (1, 1, 1, \ldots)^T$

c multiply matrix with vector: $y \leftarrow Bx$

d regularize: $x \leftarrow \alpha y + (1 - \alpha) \frac{1}{n} e$

e while error criterion is not fulfilled, back to step c
Compressed Sparse Row (CSR)

CSR matrix-vector multiplication:

```c
const int N;  // number of matrix rows
const int K;  // number of nonzero matrix entries
float a[K];   // array of nonzero matrix entries
float j[K];   // array of column indices
float start[N+1];  // array of row start indices
float x[N];    // input vector x
float y[N];    // result vector y

for(int i = 0; i < N; i++) {
    y[i] = 0;
    for(k = start[i]; k < start[i + 1]; k++) {
        y[i] += a[k] * x[j[k]];
    }
}
```
Compressed Sparse Row (CSR) Kernel 1

```c
__global__ void csr_matvec_s(ptr, indices, data, x, y) {

/** TODO **/
}
```

Task: Implement a simple CSR matrix-vector multiplication:

- Assign one thread to a matrix row
- Compute a row \( \times \) vector product for each thread.
- Add the result to output vector.
Compressed Sparse Row (CSR) Kernel 1

Solution:

```c
__global__ void csr_matvec_s(ptr, indices, data, x, y) {
    int row = blockDim.x * blockIdx.x + threadIdx.x;
    if (row < num_rows) {
        float dot = 0;
        int row_start = ptr[row];
        int row_end = ptr[row + 1];

        for (int jj = row_start; jj < row_end; jj++) {
            dot += data[jj] * x[indices[jj]];
        }

        y[row] += dot;
    }
}
```
Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:
- contiguous, fully compressed storage of column and value vectors
- \( x \) is accessed randomly
- non-coalesced memory access to indices and data, coalesced access to \( y \)
- non-uniform distribution of nonzeros may lead to serialization

Example data:

\[
\text{ptr} = [0 
2 
4 
7 
9]
\]

Access pattern to indices and data by row / thread ID (0-3):

\[
\begin{align*}
\text{jj} &= \text{row}_\text{start} & [0 & 1 & 2 & 3] \\
\text{jj} &= \text{row}_\text{start} + 1 & [0 & 1 & 2 & 3] \\
\text{jj} &= \text{row}_\text{start} + 2 & [ & & 2 & ]
\end{align*}
\]
Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:
- contiguous, fully compressed storage of column and value vectors
- \( x \) is accessed randomly

Example data:

\[
\text{ptr} \quad [0 \ 2 \ 4 \ 7 \ 9]
\]

Access pattern to indices and data by row / thread ID (0-3):

\[
\begin{align*}
\text{jj} &= \text{row}\_\text{start} \quad [0 \ 1 \ 2 \ 3] \\
\text{jj} &= \text{row}\_\text{start} + 1 \quad [0 \ 1 \ 2 \ 3] \\
\text{jj} &= \text{row}\_\text{start} + 2 \quad [\quad 2 \quad]
\end{align*}
\]
Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:
- contiguous, fully compressed storage of column and value vectors
- \( x \) is accessed randomly
- **non-coalesced** memory access to indices and data, **coalesced** access to \( y \)

Example data:

\[
\text{ptr} = [0 \ 2 \ 4 \ 7 \ 9]
\]

**Access pattern to indices and data by row / thread ID (0-3):**

\[
\begin{align*}
\text{jj} &= \text{row\_start} \quad [0 \ 1 \ 2 \ 3] \\
\text{jj} &= \text{row\_start} + 1 \quad [0 \ 1 \ 2 \ 3] \\
\text{jj} &= \text{row\_start} + 2 \quad [\quad 2 \quad]
\end{align*}
\]
Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:
- contiguous, fully compressed storage of column and value vectors
- $x$ is accessed randomly
- **non-coalesced** memory access to indices and data, **coalesced** access to $y$
- non-uniform distribution of nonzeros may lead to serialization

Example data:

```
ptr [0 2 4 7 9]
```

**Access pattern to indices and data by row / thread ID (0-3):**

```
jj = row_start [0 1 2 3 ]
jj = row_start + 1 [ 0 1 2 3]
jj = row_start + 2 [   2   ]
```
Compressed Sparse Row (CSR) Kernel 2

Idea: each warp does a row $\times$ vector multiplication.

Requires the following steps:

- Assign one warp to a matrix row
- Allocate a shared array $vals[]$ for the partial results of a block
- Compute one row $\times$ vector product in a loop. This time, parallelize the loop over all 32 threads in the warp. Take care that access to the arrays $indices$ and $data$ is coalesced.
- Use a reduction of some kind (ideally: binary fan-in) to add up the partial sums in $vals[]$ and add the output to the result vector.

→ Homework!