MAPS: Optimizing Massively Parallel Applications using Device-Level Memory Abstraction

Presented by: Eri Rubin
Eri Rubin (HUJI), Ely Levy (HUJI),
Prof. Amnon Barak (HUJI), Tal Ben-Nun (HUJI)
Talk Outline

- Motivation - What's the problem?
- Maps, written by CUDA programmers for CUDA programmers
- See the syntax!
- Very little theory 😊
- Some technical details
- Index mapper
- Code sample
- Index mapper code sample
- Performance
Motivation

- GPUs can achieve amazing acceleration for heavy compute tasks
- However getting this high performance is hard
- There are already many different tools to make development easier
- But something is missing
- After over 8 years and many projects of developing many applications, advising on many projects and teaching numerous courses in CUDA I got tired of:
  - Writing the same annoying pieces of code over and over again
  - Spending many hours debugging annoying bugs
  - Seeing my colleagues and students fall into the same potholes
- So I developed MAPS with the help of my colleagues at the lab
MAPS - Fun Facts:

- Most software is actually memory bound (if you wrote the compute in a smart way ...)
- Memory optimizations on GPUs are not fun, and lead to the “Indexing Hell”
- These can induce hard to find bugs, and prolong development time significantly
- Most algorithms actually use a very small set of access patterns. So addressing this set is a feasible task.
MAPS – Goals

• Easily let a programmer utilize advanced memory optimizations without even knowing about it!
• Remove the “Indexing Hell” by using iterators, no need to calculate indexes at all!
• Doesn’t break the CUDA programming model, so if it stops working for you, there is no need to rewrite the whole algorithm.
• Familiar STL like Container/Iterator interface
template<int RAD, int BLOCK_W>
__global__ void convMAPS(const float *in, float *out, int size) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    if (x >= size) return;

typedef maps::Window1D<float, BLOCK_W, RAD> window1DType;
__shared__ window1DType wnd;
wnd.init(in, size);
float result = 0.f;

#pragma unroll
    for (window1DType::iterator iter = wnd.begin(); iter != wnd.end(); ++iter)
        result += (*iter) * dev_convKernel[iter.id()];
    out[x] = result;
}
**Access patterns**

- Similar patterns are used in multiple algorithms
- To validate we look at “Berkeley's parallel dwarfs”
- Berkeley's parallel dwarfs – a set of algorithmic building blocks with which any parallel algorithm can supposedly be built.
- We found a set of access patterns that are used by these parallel dwarfs
### Parallel Dwarfs access pattern

<table>
<thead>
<tr>
<th>Parallel Dwarf</th>
<th>Data Structure</th>
<th>Access Patterns</th>
<th>Typical Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dense Linear Algebra</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vector-Vector</td>
<td>Block (1D)</td>
<td>Dot product</td>
</tr>
<tr>
<td></td>
<td>Matrix-Vector</td>
<td>Block (2D, 1D)</td>
<td>Matrix-Vector Mult.</td>
</tr>
<tr>
<td></td>
<td>Matrix-Matrix</td>
<td>Block (2D, Transposed)</td>
<td>Matrix Mult.</td>
</tr>
<tr>
<td><strong>Sparse Linear Algebra</strong></td>
<td>CSR/CSC Matrix</td>
<td>Adjacency</td>
<td>SpMV</td>
</tr>
<tr>
<td></td>
<td>Banded Matrix</td>
<td>Block (1D)</td>
<td>Banded Solver</td>
</tr>
<tr>
<td><strong>Spectral Methods</strong></td>
<td>Vector/Matrix</td>
<td>Permutation</td>
<td>FFT</td>
</tr>
<tr>
<td><strong>N-Body Methods</strong></td>
<td>Array</td>
<td>Block (1D)</td>
<td>All pairs N-Body</td>
</tr>
<tr>
<td></td>
<td>Octree</td>
<td>Traversal</td>
<td>Barnes-Hut N-Body</td>
</tr>
<tr>
<td><strong>Structured Grids</strong></td>
<td>Grid Matrix</td>
<td>Window</td>
<td>Convolution</td>
</tr>
<tr>
<td><strong>Unstructured Grids</strong></td>
<td>Graph</td>
<td>Adjacency</td>
<td>Cloth Simulation</td>
</tr>
<tr>
<td><strong>Graph Traversal</strong></td>
<td>Graph</td>
<td>Varies</td>
<td>BFS</td>
</tr>
<tr>
<td><strong>MapReduce</strong></td>
<td>Varies</td>
<td>Varies</td>
<td>Histogram</td>
</tr>
<tr>
<td><strong>Combinational Logic</strong></td>
<td>Varies</td>
<td>Varies</td>
<td>CRC</td>
</tr>
<tr>
<td><strong>Dynamic Programming</strong></td>
<td>Varies</td>
<td>Varies</td>
<td>Needleman-Wunsch</td>
</tr>
<tr>
<td><strong>Backtrack/Branch-and-Bound</strong></td>
<td>Varies</td>
<td>Varies</td>
<td>A*, DFS</td>
</tr>
<tr>
<td><strong>Graphical Models</strong></td>
<td>Varies</td>
<td>Varies</td>
<td>HMM</td>
</tr>
<tr>
<td><strong>Finite State Machine</strong></td>
<td>Varies</td>
<td>Varies</td>
<td>Any FSM</td>
</tr>
</tbody>
</table>
Proposed Framework - MAPS

- Base on the “Memory Dwarfs”
- Using a familiar STL style Container/Iterator interface
- Hide the “Indexing Hell”
- Does not limit or hinder the developer in any way
- Maintain optimized performance
Naïve flow

Host Memory

Data Structure

Host

Device (GPU)

Data Structure

Global Memory

Block

Threads

Shared Memory

Local Memory

Kernel

Local Memory

Kernel

Code

Store/Load
Optimized flow

Host Memory

Local Memory

Kernel

Device (GPU)

Code

Data Structure

Global Memory

Data Structure

Shared Memory

Block

Threads

Local Memory

Data

Kernel

Host

Store/Load

Optimized flow
MAPS Framework

Host Memory

Device (GPU)

Block

Threads

Container

Shared Memory

Local Memory

Local Memory

Iterator

Kernel

Map

Data Structure

Data Structure

Mapped Indices

Global Memory

Host Memory

Mapper

Global Memory

Mapper
Matrix Multiplication Sample: Naïve

```c
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;

int aBegin = n * BLK_SIZE * by;
int bBegin = BLK_SIZE * bx;

for (int i = 0; i < n; ++i)
{
    Csub += A[aBegin + n * ty + i] * B[bBegin + k * i + tx];
}
```
__shared__ float As[BLK_SIZE*BLK_SIZE];
__shared__ float Bs[(BLK_SIZE+1)*BLK_SIZE];
float Csub = 0;

for (a = wA * BLK_SIZE * blockIdx.y, b = BLK_SIZE * blockIdx.x;
    a <= wA * BLK_SIZE * blockIdx.y + wA - 1;
    a += BLK_SIZE, b += BLK_SIZE * wB)
{
    As[threadIdx.y*BLK_SIZE+threadIdx.x] = A[a + wA * threadIdx.y + threadIdx.x];
    Bs[threadIdx.y*(BLK_SIZE+1)+threadIdx.x] = B[b + wB * threadIdx.y + threadIdx.x];

    __syncthreads();

#pragma unroll
    for (int k = 0; k < BLK_SIZE; ++k){
        Csub += As[threadIdx.y*BLK_SIZE+k] * Bs[k*(BLK_SIZE+1)+threadIdx.x];
    }
    __syncthreads();
}
Matrix Multiplication Sample: MAPS

Block2D <float, BLK_SIZE> matConA;
Block2D<float, BLK_SIZE> matConB;
matConA.init(A, m, n, As);
matConB.init(B, n, k, Bs);
Block2D <float, BLK_SIZE>::iterator matAIt;
Block2D<float, BLK_SIZE>::iterator matBIt;

while (!matConA.isDone())
{
    #pragma unroll
    for (matAIt = matConA.begin(), matBIt = matConB.begin(); matAIt != matConA.end();
         ++matAIt, ++matBIt)
    {
        Csub += (*matAIt) * (*matBIt);
    }
    matConA.nextChunk(); matConB.nextChunk();
}
A graph has a topology which in many cases is unstructured, thus cannot be known in compile time, yet in many cases it is fairly constant for a certain use case.

- Naïve access from a node to its neighbors leads to random memory access
- Yet if index locality is maintained, a group of near by nodes will have allot of overlap with their neighbors.
- Caching these items beforehand can significantly reduce the overhead of random access
Host Index Mapping

For these cases, the MAPS framework includes the **index mapper** component.

This component processes data structures on the host to find an optimal caching strategy for each thread-block.
First we need to build the graph, and create the index maps

```cpp
maps::GraphMapper indexMapper(blockSize, false);
indexMapper.init(rows);

std::vector<matCell>::iterator matIt;
for (matIt = spMat.begin(); matIt != spMat.end(); ++matIt)
    indexMapper.addEdge(matIt->i, matIt->j);

indexMapper.setMaxNodeRankSize(maxNRank);

indexMapper.createIndexMap();

int sharedMemSize_con = sizeof(float)*indexMapper._MaxNumOfConstVecsInBlock;
unsigned int numPartRoundUp = maps::RoundUp(cols, 512)*512;

SPmV_maps_kernel_maps <<<gridDim, blockDim, sharedMemSize_con>> (rows, d_A_val, d_A_j_ind,
    d_A_lineStartInd, d_x, d_b, indexMapper._gpuData, cols, indexMapper._MaxNumOfConstVecsInBlock,
    numPartRoundUp);
```
extern __shared__ float sdata[];

maps::Adjacency<float,false> MyGraph;

MyGraph.init(threadIdx.x, blockDim.x, g_x ,sdata , GraphGPUData, MaxNumOfConstVecsInBlock, global_ind, numPartRoundUp);

if (global_ind < N) {
    int lineStartIndex = g_A_lineStartInd[global_ind];
    int nextLineStartInd = g_A_lineStartInd[global_ind+1];

    float res=0.f;

    maps::Adjacency<float,false>::iterator gIter = MyGraph.begin();
    for (int i=lineStartIndex; i<nextLineStartInd; ++i,++gIter)
        res += g_A_val[i] * (*gIter);

    g_b[global_ind] = res;
}
Dense Matrix Multiplication – Block 2D

![Graph showing relative speedup for different matrix sizes (64x64, 128x128, 256x256, 512x512, 1024x1024, 2048x2048) comparing Naive, Optimized, and MAPS methods. The x-axis represents matrix size, and the y-axis represents relative speedup.]
Cloth Simulation - Adjacency

![Graph showing relative speedup for different resolutions and methods.]

- **Relative Speedup**
- **Cloth Nodes**
- **Naive**
- **Optimized**
- **MAPS**

### Resolutions:
- 128x128
- 256x128
- 256x256
- 512x256
- 512x512
2D Convolution

The graph shows the relative speedup for different kernel sizes (3x3, 5x5, 7x7, 9x9, 11x11, 13x13) compared to the naive approach. The x-axis represents the kernel size, and the y-axis shows the relative speedup. The bars for the optimized approach are consistently higher than those for the naive approach for all kernel sizes, indicating improved performance.
Cross Platform Benchmark – Fused Convolution + Histogram

![Graph showing relative speedup for Kepler (K40c) and Maxwell (750 Ti). The graph compares different operations: NPP, CUB: Global Atomics, CUB: Sorting, CUB: Shared Atomics, and Fused MAPS.]
Conclusion

This work presented a novel framework

Defined a set of “Memory Dwarfs” based on Parallel Dwarfs

The MAPS framework creates an abstraction, exposing a familiar STL-like API

An implementation has been written and is publicly available
Future Work

Writing a port of the library over higher level languages such as C++ AMP, Python.

Enhance the model to allow for even more automatic optimizations (e.g. ILP).

Integration with other libraries.
Thank you

Questions?

Questions can be sent to:

- eri.rubin@gmail.com
- talbn@cs.huji.ac.il

Library to be published at http://www.cs.huji.ac.il/~talbn/maps

This research was partially supported by the Ministry of Science and Technology, Israel.