Algorithms to code: Case studies

Two case studies, each structured as follows.

- Sketch and analyze algorithm
- Estimate best-case performance, using models and/or benchmarks
- Implement and tune
Case study 1: Sparse direct solve

Streaming small, independent kernels for better resource utilization.
In this section, we consider the design of a *sparse direct solver," specifically, computing the so-called “Cholesky factorization” of a given sparse matrix, A. This kind of computation arises in various engineering applications, such as structural analysis in civil or automotive engineering.

Unlike the matrix multiplication example that we used in Part 1, here the matrix is “sparse," meaning that most of its entries are zero. Thus, for a sparse matrix, we can save both storage and execution time by storing and operating on only the non-zero entries. For an n x n matrix, in a typical problem the number of non-zeros scales like O(n), rather than O(n^2) for a dense matrix, so the savings will in general be substantial. The challenge is that we will have to use a graph-like pointer-oriented data structure to store the matrix, which introduces metadata (integer index) storage overhead per stored non-zero, as well as potentially irregular and indirect memory references.

The goal of a Cholesky factorization is to compute a new sparse matrix L such that the input matrix A = L * transpose(L). In general, we want to keep L sparse as well. The basic structure of the computation might be to take a physical input problem, discretized appropriately (left), form its sparse matrix representation (middle), and perform the parallel computation by operating on this sparse matrix. As it happens, the (parallel) computation can be captured by a DAG, like the ones we analyzed in part 1. In a routine civil engineering analysis problem such as the one shown above, the average intensity will be something like 4 flops per byte in single-precision, which on a CPU is compute-bound but on a GPU is memory bound, as we will see momentarily.
The structure of the parallel computation is captured by something called an “elimination tree,” shown here for a sample matrix. Independent subtrees are themselves independent. This view of the computation is the traditional way that a sparse direct solver is described.
However, the elimination tree actually hides a finer-grained dependency structure, shown here. The “real” computation structure may be captured by a DAG, with each node of the elimination tree expanding to a subgraph of this DAG. Nodes in this finer-grained representation, as it turns out, consist of just four basic primitive *dense* linear algebra operations. When we reported the “average” intensity in the introduction, we were roughly speaking averaging over the nodes in this DAG. That is, each DAG node implies a certain number of flops and a minimum number of memory operations, which we aggregate over the whole DAG to estimate the overall intensity.

The standard library for implementing dense linear algebra is called the “basic linear algebra subroutines,” or “BLAS” library. All major hardware vendors provide a customized and performance-tuned implementation of the BLAS, so, in principle, if we build our application on top of the BLAS, we should get good performance.
Before building the code, we estimated the intensity for a particular problem and did a zeroth-order roofline analysis, which predicts 300 to 500 Gflop/s on a GPU (C1060 or Fermi). We then benchmarked the NVIDIA’s BLAS library, called CUBLAS, on the expected workload to get an overall estimate of best-case performance in the absence of overheads. By “expected workload,” we mean we benchmarked the BLAS on the operand sizes we expected to see in a real solve; note that all of this work is being done *before* actually implementing a real solver. Our performance estimate came to less than 10% of what the roofline predicted!
This chart shows why we achieve such a low-level of performance: The intensity of all of the BLAS operations vary significantly, with as many having very low intensity as having very high intensity. For the vast majority of these operations, we only achieve less than 10% of peak.
The reason is probably obvious: the small problems are too small to utilize the massive thread capacity available in a GPU. Moreover, at the time, calling CUBLAS was restricted to invoking one BLAS operation at a time.
Given a sequence of “small,” independent kernel invocations, we can use the CUDA Streams interface to fill capacity.
Using the Streams interface allows us to pack more independent work onto the GPU, to better use the available thread capacity. However, if optimal performance is the goal, you can’t simply use the interface naively. These results, generated by Kent Czechowski, show effective performance as a fraction of matrix-multiply peak for executing a batch of independent (but mostly small) matrix multiplies, using (a) one-at-a-time execution (red bar); (b) round-robin assignment to a fixed number of work queues (streams) (green bar); and (c) an “optimal” schedule (blue bar).
Case study: FMM for Blood Flow

This section applies our analysis and tuning techniques to a blood flow simulator called "MoBo" (moving boundaries).
MoBo makes two major contributions over codes that came before.

First, MoBo simulates deformable red blood cells (RBCs), which is important if one wishes to accurately model how blood is able to flow in narrow channels, such as tiny capillaries. The prior work based on similarly deformable RBCs were never scaled beyond problem sizes of 14,000 cells, in part because those methods required a very fine-scale representation each cell of roughly tens of thousands of unknowns per cell. MoBo, by contrast, uses a “light” representation of just tens to a few hundreds of unknowns per cell, enabling it to scale to much larger numbers of total cells.

The second major contribution is the scaling itself, which is enabled by use of an “optimal” algorithm called the fast multipole method. For our case study, we’ll focus on part of this component.
Problem formulation and algorithms.
Red blood cells suspended in plasma flow

Cell = boundary points
Plasma = background flow
Intra-cell forces:
- Spherical harmonics (Legendre & Fourier transforms)
- Singular quadratures
- Derivatives
- Surface remeshing
- Semi-implicit solver (BiCGStab)

Dominated by many small but compute-intensive ops, *a la*

Red blood cells suspended in plasma flow
Red blood cells suspended in plasma flow

*Inter-cell forces: Fast multipole method*
The graph above illustrates the performance of different processors with varying Spherical Harmonics' Order. The term "Streams" indicates a performance improvement of approximately 1.5 times compared to the baseline. The processors compared are Nehalem, Tesla, Fermi, and Fermi (Streams). The x-axis represents the Spherical Harmonics' Order, while the y-axis shows GFLOP/s.
Anatomy of a fast multipole method

- Need to evaluate all-pairwise interactions among green points: $O(n^2)$

- FMM instead computes in $O(n)$ time, with an approximation guarantee, using a tree ($n \log n$ to build)
The tree subdivides space into boxes, constructed so that each box contains no more than $q$ points. This figure shows the leaf boxes of such a tree for some 2-D space.

The computation consists of a series of tree traversals that compute interactions among pairs of boxes.
For each target box B, the rest of space is assigned to “regions,” denoted here by U, V, X, W.

The computations between B and boxes in a given region will differ in parallelism and intensity.
The **U region**, which consists of the leaf boxes adjacent to $B$, is the most compute intensive and typically the first bottleneck.

In fact, the interaction $B \otimes U_i$ is the "direct" $O(q^2)$ force/potential interaction.
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source boxes

Target boxes

The direct "U-List" computation
\[ U(B) = \{1, 4, 6\} \]
The direct "U-List" computation

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\[ B \quad \rightarrow \quad * \quad * \quad * \quad \]

\textbf{Source boxes}

\textbf{Target boxes}

\textbf{for} B \textbf{in} TargetBoxes:

\textbf{for} S \textbf{in} U(B):

\[
B \leftarrow B \odot S
\]
Parallelism:
1. All $B$ independent
2. Within $B \otimes S \rightarrow O(q^2)$ flops : $O(q)$ boundary

The direct "U-List" computation
The direct "U-List" computation

<table>
<thead>
<tr>
<th>0</th>
<th>⋯</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>⋯</td>
<td>⋯</td>
<td>*</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
</tr>
</tbody>
</table>
For each green point, sum contribution from all blue points.

The direct “U-List” computation.
The direct “U-List” computation
The direct “U-List” computation

Repeat for each neighbor
Natural mapping:
Target boxes $\rightarrow$ thread blocks
Targets $\rightarrow$ threads

The direct “U-List” computation
Contiguous storage —
points in boxes; boxes;
admits coalescing

The direct “U-List” computation
Inherent reuse of sources and targets

The direct “U-List” computation
Index arrays track box starts and U lists

The direct “U-List” computation
int di = threadIdx.x;  // Local thread ID
int tid_min = TBptr[tboxid];  // Target box start
int tid_max = tid_min + TBn[tboxid];  // Target box end
for (int tid = tid_min + di; tid < tid_max; tid += NB) {  // Targets
    float4 tgt = make_float4(Tx[tid], ..., Tz[tid], Tw[tid]);
    for (int kbox = kbox_min; kbox < kbox_max; kbox++) {  // Source boxes
        int sboxid = Ulist[kbox];  // Source box ID
        int sid_min = SBptr[sboxid];
        int sid_max = sid_min + SBn[sboxid];
        for (int sid = sid_min; sid < sid_max; sid++) {  // Sources
            float4 src = make_float4(Sx[sid], ..., Sw[sid]);  // Global (coalesced)
            float3 dr = make_float3(tgt.x - src.x, tgt.y - src.y, tgt.z - src.z);
            float rsq = dr.x*dr.x + dr.y*dr.y + dr.z*dr.z;
            float r = sqrtf(rsq);
            tgt.w += src.w / r;
        }  // sid loop
    }  // kbox loop
    T[tid].w = tgt.w + (OOFP_R + tw);  // Store result
}  // tid loop

The direct “U-List” computation
Step 1: Set an optimization goal.
What is our optimization goal?

- **Baseline U-list computation:**
  - Nehalem 2x4-core CPU + OpenMP + SSE
    - 60 to 90 Gflop/s in single-precision
    - 2-3x cost of other phases

- Theory: High intensity ➔ **Compute-bound**
  - Verified by examining the distribution of points per box to estimate flop:byte ratio
  - Wrote microbenchmark with no memory operations to estimate instruction mix-specific peak
float4 \textbf{tgt} = \ldots \\
float4 \textbf{src} = \ldots \\
float3 \textbf{dr} = \text{make\_float3} \ (\text{tgt}.x = \text{src}.x, \text{tgt}.y = \text{src}.y, \text{tgt}.z = \text{src}.z); \\
float \text{rsq} = \text{dr.x}\*\text{dr.x} + \text{dr.y}\*\text{dr.y} + \text{dr.z}\*\text{dr.z}; \\
float \text{r} = \text{sqrtf} \ (\text{rsq}); \\
\text{tgt.w} +\gets \frac{\text{src.w}}{\text{r}};

\text{Observations:}

1. \textbf{Chain of 7 dependent flops}

2. “1 / sqrt(x)” $\rightarrow$ Fast “rsqrt(x)” op

\text{SUB} \mid \text{SUB} \mid \text{SUB} \\
$\rightarrow$ \text{MUL} $\rightarrow$ \text{FMA} $\rightarrow$ \text{FMA} \\
$\rightarrow$ \text{SQRT} $\rightarrow$ \text{DIV} $\rightarrow$ \text{ADD}
float4 \textbf{tgt} = \ldots \\
float4 \textbf{src} = \ldots \\
float3 \textbf{dr} = \text{make\_float3}(\text{tgt}.x \text{- src}.x, \text{tgt}.y \text{- src}.y, \text{tgt}.z \text{- src}.z); \\
float \text{rsq} = \text{dr}.x^2 + \text{dr}.y^2 + \text{dr}.z^2; \\
float \textbf{r} = \text{sqrtf}(\text{rsq}); \\
lgl.w += \text{src}.w / \textbf{r}; \\

\begin{equation}
\Rightarrow \text{tgt}.w += \text{src}.w \times \text{rsqrt}(\text{rsq})
\end{equation}

\textbf{Observations:}
1. Chain of 7 dependent flops
2. “1 / sqrt(x)” $\rightarrow$ Fast “rsqrt(x)” op

\textit{SUB | SUB | SUB} \\
$\rightarrow$ \textit{MUL} $\rightarrow$ \textit{FMA} $\rightarrow$ \textit{FMA} \\
$\rightarrow$ \textit{RSQRT} $\rightarrow$ \textit{FMA}
float4 src = ...; // Global memory load: 16 bytes

// Now perform 11 flops...
float3 dr = make float3 (tgt.x - src.x, tgt.y - src.y, tgt.z - src.z);
// ...

Baseline has a minimum intensity of **11 flops : 16 bytes**, for which the roofline predicts ~ **70 Gflop/s**.

Actual measured baseline: **79 Gflop/s**
To summarize, our GPU baseline code achieves $79 \text{ Gflop/s}$ and our optimization goal is $340 \text{ Gflop/s}$.

Step 2: Identify candidate optimizations.
Candidate optimizations

- U-List should be compute-bound, so focus on memory hierarchy optimizations first
  - Blocking/tiling, using shared or texture memory
  - “Vector packing:” Structure-of-arrays vs. AOS
  - Prefetching
  - Unroll-and-jam, on targets (multiple targets / thread)

- Once we are in the compute-bound regime, try low-level optimizations, e.g., reciprocal square-root

- Try all sensible combinations + tune the number of threads per block: \{45 codes\} x
Recall: Inherent reuse of sources and targets

Optimization: Shared memory
Threads cooperatively load and compute…

Optimization: Shared memory
Optimization: Shared memory
Optimization: Shared memory
Optimization: Shared memory
Optimization: Shared memory
Comment: Structure-of-arrays (SOA) vs. AOS layout

- Original SSE-based code uses SOA layout, e.g., one array X to store all x-coordinates, ...

- CUDA folk-wisdom suggests AOS via float2/3/4 primitive types, which may aid aggregation of memory transactions
  - AOS possible with our code
  - But, to maintain compatibility with distributed memory code, would prefer not to abandon SOA
Comment: Unroll-and-jam on targets

- Baseline: Assigns one target (output) per thread and rely on many threads to hide latency

- Alternative: Use unroll-and-jam on targets to increase parallelism and locality (register reuse)
  - Recall notions of “CWP” and “MWP”
  - See also analysis by V. Volkov at NVIDIA GTC’10:
    - Increasing ILP per thread is a viable alternative to increasing thread count
    - Benefits of fewer threads are more registers per thread, whose latency relative to shared memory has
int di = threadIdx.x; // Local thread ID
...
for (int tid = tid_min + di; tid < tid_max; tid += NB) { // Targets
    float4 tgt = T[tid];
    ...

Optimization: Unroll-and-jam on targets
int di = threadIdx.x; // Local thread ID

... for (int tid = tid_min + di; tid < tid_max; tid += NB) // Targets
    float4 tgt = T[tid];
... 

\[ \text{Unroll-and-jam by 2} \]

int di = threadIdx.x; // Local thread ID

... for (int tid = tid_min + di; tid < tid_max; tid += 2*NB) // Targets
    float4 tgt0 = T[tid];
    float4 tgt1 = T[tid+NB];

// Can now reuse sources in registers for two targets

Optimization: Unroll-and-jam on targets
int di = threadIdx.x; // Local thread ID
...
for (int tid = tid_min + di; tid < tid_max; tid += NB) { // Targets
    float4 tgt = T[tid];
    ...
}

Unroll-and-jam by 2

int di = threadIdx.x; // Local thread ID
...
for (int tid = tid_min + di; tid < tid_max; tid += 2*NB) {
    float4 tgt0 = T[tid];
    float4 tgt1 = T[tid+NB];
    // Can now reuse sources in registers for two targets
    // Also doubles ILP, but increases register pressure
    ...

Optimization: Unroll-and-jam on targets
Performance results summary

- Applied transformations in sequence
  - Let $S = \{\text{optimizations}\}$
  - $\text{Base} + (s \text{ in } S) \rightarrow \text{Base} + s_{opt}$
  - $\text{Base} + s_{opt} + (t \text{ in } S-\{s_{opt}\}) \rightarrow \text{Base} + s_{opt} + t_{opt}$
  - $\text{Base} + s_{opt} + t_{opt} + (u \text{ in } S-\{s_{opt}, t_{opt}\})$  
    $\rightarrow \text{Base} + s_{opt} + t_{opt} + u_{opt}$
  - ...

- Top performers use different combinations but achieve the same level of performance
Target ~ 340 Gflop/s

~ 3x from memory optimizations

Compute optimization, but still memory-bound
“Less” memory bound, so compute optimization says off.
FMM Optimization Case Study

- Applied basic principles and microbenchmarks to set performance goal

- Pre-plan “space” of optimizations
  - Parallelism → Locality → low-level optimizations
  - Streams of small problems to improve utilization
  - Non-obvious combinations may be optimal

- Opportunity: Automatable bounds analysis?
Observation:
Occupancy can mislead
Observation:
Roofline is a better guide, but still “coarse.”
FMM Optimization Case Study

- Applied basic principles and microbenchmarks to set performance goal

- Pre-plan “space” of optimizations
  - Parallelism → Locality → low-level optimizations
  - Streams of small problems to improve utilization
  - Non-obvious combinations may be optimal

- Opportunity: Automatable bounds analysis?
Keeneland (ORNL)
150M pts in 0.4s