Scanning Polyhedra: Fourier-Motzkin Elimination

Sanjay Rajopadhye
Colorado State University

Outline

- Parallelizing programs with dependences
- Two levels of parallelism
  - Virtualization, virtualization, virtualization
  - Rules and when/how to break them (Volkov)
- Coarse grain wavefront parallelism
- Fine grain wavefronts
- Locality, Parallelism and Energy
- Breaking the rules – synchronization and communication of coarse grain tasks
Wavefront Parallelization

- How to parallelize computations (e.g., loops in OpenMP) that have dependences:
  - None of the loop iterations are independent

Simple examples

```c
for (i=1; i<N; i++)
  for (j=1; j<M; j++)  
    A[i,j] = foo(A[i,j-1], A[i-1,j])

for (i=1; i<N; i++)
  for (j=1; j<M; j++)
    B[j] = bar(B[j-1], B[j])

for (i=1; i<N; i++)
  for (j=1; j<M; j++)
    C[i] = baz(C[i-1], C[i])
```
Iteration Space & Data Space

- **Iteration Space**: set of values that the loop iterators can take
  - Rectangular region, with “corners” [1,1] and [N-1, M-1]
- **Data Space**: set of values of array indices accessed by the statements in the program
  - Ex 1: 2-D table, (nearly) identical to the iteration space
  - Ex 2: 1D array, bounded by [0, M-1]
  - Ex 3: 1D array, bounded by [0, N-1]

References and Dependences

- **Reference**: a occurrence of an array variable on either
  - left hand side (write reference)
  - right hand side (read)
  - of a statement in the loop body
- **Dependences**: specify which iteration points depend on which others
  - can be refined if/when there are multiple statements in the program
Finding the dependences

- Very hard problem (undecidable in general) but we have simple cases
  - An iteration point \([i, j]\) reads a memory location
  - (Many) iterations (may) have written to that location
  - Find this set (as a function of \([i,j]\))
  - Find the "most recent writer" in this set (again, as a function of \([i,j]\))

Execution order
Solutions to examples

- Ex1 and Ex2 (same solution, even though the data space is very different). Iteration \([i, j]\) depends on:
  - \([i, j-1]\) and \([i-1, j]\) neighbors on west and north
  - Ex2 has an additional (memory based dependence)
    - Iteration \([i-1, j+1]\) reads a memory location that the iteration \([i, j]\) is overwriting, that must also happen before \([i, j]\) so it cannot be executed before its northeast neighbor

- Ex3 is more complicated
  - \([i, j]\) depends on \([i, j-1]\) and \([i-1, M-1]\)
(Finally) the parallelization

- Now that we know the dependences between iterations (the dependence graph)
  - Analyze to determine what can happen at what time (hopefully many things can happen at the same time)
  - Rewrite the program to represent this new order
Ex1 wavefront parallelization

Redraw the graph
Writing the (OpenMP) code

- Node \([i, j]\) is mapped to \([p, t]\)
  - \((i, j \rightarrow p, t) = (i, j \rightarrow i, i+j-1)\)
- Inverse of the transformation:
  - \((p, t \rightarrow i, j) = (p, t \rightarrow p, t-p+1)\)
- Determine the transformed iteration space
- Write loops that traverse this
- Outer loop must be the time
- Inner loop is marked to be executed in parallel with
  
  ```
  #pragma omp parallel for
  ```

- Write the new loop body

Control structure

Source \rightarrow Loop bounds \rightarrow Iteration space (inequalities) \rightarrow Transformed Iteration space (inequalities) 

Target \leftarrow Loop bounds \leftarrow Target
Control structure

\[
\begin{align*}
\text{for } (i &= 1; i < N; i++) \\
\text{for } (j &= 1; j < M; j++) \\
\{i, \ j | 1 \leq i \leq N-1; \ 1 \leq j \leq M-1\} \\
\{p, \ t | 1 \leq p \leq N-1; \ 1 \leq (t-p+1) \leq M-1\} \\
\{p, \ t | 1 \leq p \leq N-1; \ p \leq t \leq M+p-2\} \\
\text{for } (t &= 1; t \leq N + M - 3; t++) \\
\text{for } (p &= \text{max}(1, t - M + 2); p \leq \text{min}(t, N-1); p++) // \text{this is parallel}
\end{align*}
\]

From inequalities to loops

- Work “inside out,” i.e., generate bounds on the innermost dimension (say \(z_n\)) first
- For each inequality, rearrange it into the form:
  \[a_n z_n \geq \exp,\] for some constant coefficient \(a_n\)
  - If \(a_n\) is positive, \(\exp/a_n\) is a lower bound on \(z_n\)
  - Otherwise, \(-\exp/a_n\) is an upper bound
- Let \(l_1 \ldots l_m\) be the lower bound expressions and \(u_1 \ldots u_m\) be the upper bound expressions.
- The innermost loop is (with one caveat):
  \[\text{for } (z_n = \text{max}(l_1 \ldots l_m); z_n < \text{min}(u_1 \ldots u_m); z_n++\]
- Recurse on the outer \(n-1\) dimensions
Recursion: eliminate $z_n$

- For each pair, $u_i, l_j$, introduce an inequality, $u_i \geq l_j$.
- Let the collection of these inequalities define the iteration space $I_{n-1}$:
  - $I_{n-1}$ is an $(n-1)$-dimensional iteration space (doesn't involve $z_n$).
  - The first $n-1$ coordinates of every point in the original iteration space, $I_n$, also satisfy $I_{n-1}$.
  - $I_n$ is the intersection of $I_{n-1}$ and loop bounds.

Example (on doc cam)
New loop body

- At each point \([t, p]\) in the new loop,
  - Determine the original iteration point that was mapped to \([t, p]\) (inverse of the rectangle-to-parallelogram transformation)
    Given \([t, p] = [i+j-1, j]\) solve for \([i, j]\) in terms of \(t\) and \(p\).
  - Add synchronization (optional)
  - Optionally, change memory

```c
int i, j;
for (t=1; t<=N+M-3; t++)
  #pragma omp parallel for private i, j
  for (p=max(1,t-M+2); t<=min(t,N-1); p++) {
    i = p;
    j = t-p+1;
    // insert old loop body (unchanged) here:
    // we chose \(t\) and \(p\) as brand new index names
    A[i,j] = foo(A[i,j-1], A[i-1,j]);
  }
```

Colorado State University
Ex2 wavefront parallelization

Example 2 (contd)

- Mapping is \((i, j \rightarrow p, t) = (i, j \rightarrow i, 2i+j-2)\)
- Inverse of the transformation:
  - \((p, t \rightarrow i, j) = (p, t \rightarrow p, t-2p+2)\)
- Transformed iteration space:
  \[ \{p, t | 1<=p<=N-1; 1<=(t-2p+2)<=M-1\} \]
- Rewrite as:
  \[ \{p, t | 1<=p<=N-1; t-M+3<=2p<=t+1\} \]
- Write the new loop
Example 2 (contd)

```
for (t=3; t<=2N+M-5; t++)
    for (p = max(1, \left\lfloor \frac{t-M+3}{2} \right\rfloor ); p <= \min( \left\lfloor \frac{t+1}{2} \right\rfloor , N-1) \) {

    // NEW LOOP BODY:
    i = p; j = t-2p+2;
    // copy the old body
    B[j]=\text{bar}(B[j], B[j-1]);
    }
```

Better way

- Early preoccupation with memory:
  - Memory allocation of the original program is hurting us
- First parallelize the “full table version”
- Then make it use less memory
Ex1 revisited = Ex 2

```c
int i, j;
for (t=1; t<=N+M-3; t++)
    #pragma omp parallel for private i, j
    for (p=max(1,t-M+1); t<=min(t,N-1); p++) {
        i = p;
        j = t-p+1;
        // A[i,j] = foo(A[i,j-1], A[i-1,j]);
        A[i%2, j] = bar(A[i%2, j-1], A[(i-1)%2, j]);
        if (p==N-1) B[j] = A[i%2, j];
    }
```

So what about CUDA

- Two levels of parallelism
- Focus for now only on fine grain parallelism
- CUDA threads are not the same as OpenMP
- Work done by each thread is explicitly described (parametric function of tid)
- Implicit outer parallel loops, iterating over
  - Block coordinates, and
  - Thread coordinates
- Memory may be
  - private (registers), or
  - shared (shared memory)
- Synchronization may have to be explicit
- Tweak what we did for OpenMP to do CUDA
What’s wrong with outer p?

for (i=1; i<N; i++)
  for (j=1; j<M; j++)
    \{i, j \mid 1\leq i\leq N-1; 1\leq j\leq M-1\}

\{p, t \mid 1\leq p\leq N; 1\leq (t-p+1)\leq M\}

\{p, t \mid 1\leq p\leq N; p\leq t\leq M+p-1\}

#pragma omp parallel for // make the outer loop parallel
for (p=1; p\leq N-1; p++)
  for (t=p; t\leq M+p-1; t++) \{ // body

#pragma omp barrier
  \}

Assume numthreads = N-1

#pragma omp parallel for // make the outer loop parallel
for (p=1; p\leq N-1; p++)
\{
  for (t=0; t<p; t++) \{
    #pragma omp barrier
  \}
  for (t=p; t\leq M+p-1; t++) \{ // body
    #pragma omp barrier
  \}
  for (t=M+p; t\leq tmax; t++) \{
    #pragma omp barrier
  \}
\}
Why is outer p incorrect?

```c
#pragma omp parallel for // make the outer loop parallel
for (p=1; p<=N-1; p++)
    for (t=p; t<=M+p-1; t++) {
        // body
    }
```

- OpenMP semantics: all iterations of the p loop should be independent of each other
  - But iteration p
    - reads data produced by iteration p-1, and
    - produces data to be consumed by p+1
  - Solution – add explicit synchronization
Dummy (nop) nodes

Synchronization in CUDA

- Waruna’s toy kernel (array increment)
- Each thread increments all the elements in an array (in shared memory)
- Avoid races by taking turns
- Thread 0 starts first and updates the first block (of width SUBTILE_WIDTH)
- Next it moves on to the next block while thread 1 updates the one to which thread 0 just wrote
- ...

Colorado State University
What if numthreads < N-1

- This is the more realistic case:
  - N is typically large (limited only by shared memory capacity)
  - numthreads is no more than a machine dependent upper bound (e.g., 512), usually much less than N
- Solution: divide the rows into strips of height \( h = (N-1)/\text{numthreads} \) and have each thread responsible for the whole strip
  - Avoid serialization by splitting columns into strips too
  - Basic unit is a rectangular, \( h \times w \) tile

Dependence Graph (Ex 1)
Details: memory

- Each thread
  - Updates a private array of length h
    - Left/right boundary of tile
    - in registers (private to thread)
  - Communicates to next thread an array of length w
    - Top/bottom boundary of tile
    - must be in shared memory (only mechanism by which threads communicate with each other)
- Double buffering to avoid races
- Synchronize (and sometimes do nops)

Drawbacks and Solutions

- Latency of the pipeline fill
  - Inherent in the problem – graph has a single source
  - Other programs may have better parallelism (e.g., KPDP)
- Only uses a single SM
- Solution:
  - Do the same thing all over again
  - Tile one more level
Smth-Wtrmn code: Waruna

- Real application (biological sequence comparison)
- One twist – a diagonal dependence
- Outer level of (coarse grain parallelism)
  - Sequence of kernel calls from the host
  - Host executes the time loop iterating over the wavefronts
  - Each front has a number of independent tiles
- Inner fine grain parallelism
  - Each tile is executed by a thread block
  - In parallel (using multiple threads)
  - Use the parallelization discussed till now

Knapsack DP parallelization

- Lessons from HW0
  - The code (i.e., function fill-table in HW3) was memory bound
  - omp overhead of thread spawn/sync in each of the outer n iterations
- HW3: Coarse grain parallelism
  - What is the speedup? Is it good?
  - Why?
  - What happens if fine-grain parallelism is added?
Fine Grain Parallel on GPU

- Main challenge – make it compute bound rather than bandwidth bound
- Optimize the loop body
  - No conditionals
  - Especially conditionals using thread-id (caveat)
- Improve the balance

- Use resources on the web

Dependences of the DPKP
"Better" Parallelization

Pros & Cons

- Pipeline fill-flush vs "concurrent start" of all threadblocks
- Load balance
  - During fill/flush
  - In steady state: adjust stile sizes such that # threadblocks is multiple of #SMs – can be parameterized
  - Easy with "concurrent start wavefronts"
  - Complicated with [1,1] wavefront
- So do concurrent start with tiling to improve the balance
Tiled DPKP Dependences

Main Drawback

- Tiling loses the concurrent start property
- Unless
  - Tile height is 1
- Challenge – how to retain/improve balance (arithmetic intensity) with tiles of height 1
- Issue: data must be retrieved from (and restored to) global memory in each kernel call
- Solution: do a sequence of rows in a single kernel call
New issues

- There are dependences between thread-blocks
- Need for synchronization
- Data transfer via global memory read-writes
- Possibility of deadlocks?

Solution: synchronization

- Each threadblock (i.e., one designated thread, sat thread 0 in the threadblock)
  - maintains a counter indicating which row of the table it has successfully finished
  - Array if grid-size (in global memory) call it status[
- A threadblock (numbered p)
  - does not start row number i until status[p-1] has reached i-1.
  - After finishing row i, it
    - Copies the last w[i] values of its section of the row into global memory
    - increments status[p] to i.
Deadlock?

- Previous solution relies on the runtime system's scheduling order
- What if the number of threadblocks is more than number of SMs?
- Block number \( p \) may be running and block \( p-1 \) has not even started
- Deadlock!!
- Currently runtime system schedules in the “correct order”

Deadlock: better solution

- Use atomic operations in CUDA
- Each threadblock does a “test-and-increment” to obtain a unique counter, whose value is, say \( ctr \). Properties
  - There must be \( ctr-1 \) threadblocks that are already active (may even have terminated)
  - Therefore I will start working on the \( ctr \)th chunk of data, regardless of what my block-id is.
- Dynamic (rather than static) allocation of work to threadblocks.