L5: Writing Correct Programs, cont.

Outline

- How to tell if your parallelization is correct?
- Race conditions and data dependences
- Tools that detect race conditions
- Abstractions for writing correct parallel code
  - Mapping these abstractions to CUDA
- Reading (if you can find it):

Administrative

- Next assignment (a homework) given out on Monday

Is this CUDA code correct?

```c
__host__ callkernel() {
    dim3 blocks(10);
    dim3 threads(100);
    float *d_array;
    ...
    cudaMalloc(&d_array1...)
    cudaMalloc(&d_array2...);
    kernelcode<<<blocks,threads,
    D>>>(d_array1, d_array2,
    1000);
}

__global__ kernelcode(float * d_array1,
    d_array2, int N) {
    float result;
    for (int i=0; i<N; i++) {
        d_array1[threadIdx] +=
        d_array2[blockIdx][i];
    }
    for (int i=1; i<N; i++) {
        result += d_array1[threadIdx-1];
    }
}
```

Threads Access Shared Memory!

- Global memory and shared memory within an SM can be freely accessed by multiple threads.
- Requires appropriate sequencing of memory accesses across threads to the same location if at least one access is a write.
  - Recall using `__syncthreads()` within a thread block for synchronization.
  - Not to be used for different blocks within a grid.

Is this CUDA code correct?

```c
__global__ kernelcode(float * d_array, d_array2, int N) {
    for (int i = 0; i < N; i++) {
        d_array[threadIdx] += d_array[blockIdx][i];
    }
    __syncthreads();
    for (int i = 1; i < N; i++) {
        result += d_array[threadIdx-1];
    }
}
```

More Formally:
Race Condition or Data Dependence

- A **race condition** exists when the result of an execution depends on the **timing** of two or more events.
- A **data dependence** is an ordering on a pair of memory operations that must be preserved to maintain correctness.

How about other Shared Memory Architectures?

- Race detection software (e.g., Intel ThreadChecker)
  - Trace memory accesses for each thread
  - Compare addresses accessed by each thread
  - Race condition exists if, between synchronization points,
    - multiple threads access the same memory location
    - and, at least one access is a write.
What can we do to debug parallelization in CUDA?

- deviceemu code (to be emulated on host)
  - Support for pthread debugging?
- Can compare GPU output to CPU output, or compare GPU output to device emulation output
  - Race condition may still be present
- Or can (try to) prevent introduction of race conditions (remainder of lecture)

Data Dependence

- Definition:
  Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write.

  A data dependence can either be between two distinct program statements or two different dynamic executions of the same program statement.

  - Two important uses of data dependence information (among others):
    Parallelization: no data dependence between two computations \(\rightarrow\) parallel execution safe.
    Locality optimization: absence of data dependences & presence of reuse \(\rightarrow\) reorder memory accesses for better data locality (next week)

Data Dependence of Scalar Variables

- True (flow) dependence
  \(i = a\)
- Anti-dependence
  \(a = i\)
- Output dependence
  \(i = a\)
- Input dependence (for locality)
  \(i = a\)

Definition: A data dependence exists from a reference instance \(i\) to \(i'\) iff either \(i\) or \(i'\) is a write operation and \(i\) refers to the same variable \(i\) executes before \(i'\).

Some Definitions (from Allen & Kennedy)

- Definition 2.5:
  Two computations are equivalent if, on the same inputs,
  - they produce identical outputs
  - the outputs are executed in the same order

- Definition 2.6:
  A reordering transformation
  - changes the order of statement execution
  - without adding or deleting any statement executions.

- Definition 2.7:
  A reordering transformation preserves a dependence if
  - it preserves the relative execution order of the dependences' source and sink.
Fundamental Theorem of Dependence

- **Theorem 2.2:** Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

Now we will discuss abstractions and algorithms to determine whether reordering transformations preserve dependences...

Parallelization as a Reordering Transformation in CUDA

```c
__host__ callkernel() {
    dim3 blocks(bx, by);
    dim3 threads(tx, ty, tz);
    ...
    kernelcode<<<blocks,threads>>>(args);
}
__global__ kernelcode(args) {
    /* code refers to threadIdx.x, threadIdx.y, threadIdx.z, blockIdx.x, blockIdx.y */
}
```

**EQUIVALENT?**

For all (or CUDA kernels or Doall) loops:

Loops whose iterations can execute in parallel (a particular reordering transformation)

**Example**

```c
forall (i=1; i<n; i++)
A[i] = B[i] + C[i];
```

Meaning?

Each iteration can execute independently of others
Free to schedule iterations in any order

Why are parallelizable loops an important concept for data-parallel programming models?

CUDA Equivalent to "Forall"

```c
forall (int tidx_x=0; tidx_x<bx; tidx_x++) {
    forall (int tidx_y=0; tidx_y<by; tidx_y++) {
        forall (int tidx_z=0; tidx_z<tz; tidx_z++) {
            * code refers to tidx_x, tidx_y, tidx_z,
            blockIdx_x, blockIdx_y */
        }
    }
}
```
1. Characterize Iteration Space

- Iteration instance: represented as coordinates in iteration space
- n-dimensional discrete cartesian space for n deep loop nests
- Lexicographic order: Sequential execution order of iterations
  \[[1,1], [1,2], ..., [1,n], [2,1], [2,2], ..., [2,n], ...

- Iteration I (a vector) is lexicographically less than I', I < I', iff there exists c (i_1, ..., i_c) = (i'_1, ..., i'_c) and i_c < i'_c.

2. Compare Memory Accesses across Dynamic Instances in Iteration Space

- How to describe relationship between two dynamic instances? e.g., I=[1,1] and I'=[2,2]

Distance Vectors

- Distance vector = \[1,1\]
- A loop has a distance vector D if there exists data dependence from iteration vector I to a later vector I', and D = I' - I
- Since I' > I, D >= 0.
  (D is lexicographically greater than or equal to 0).
Distance and Direction Vectors

- Distance vectors: (infinitely large set)
  \[ \text{\{<,=,>,+=,-,->\}} \]
- Direction vectors: (realizable if 0 or lexicographically positive)
  \[ \text{\{:=,=,>,=,<\}} \]
- Common notation:
  \[ \begin{align*}
  \text{+} & \quad < \\
  \text{-} & \quad > \\
  \text{+/-} & \quad * 
  \end{align*} \]

Parallelization Test: 1-Dimensional Loop

- Examples:
  \[
  \begin{align*}
  \text{for } (i=1; i<N; i++) & \quad \text{for } (j=1; j<N; j++) \\
  \end{align*}
  \]
- Dependence (Distance and Direction) Vectors?
- Test for parallelization:
  \[
  \begin{align*}
  \text{A loop is parallelizable if for all data dependences } D \in D, \\
  D = 0
  \end{align*}
  \]

n-Dimensional Loop Nests

\[
\begin{align*}
\text{for } (i=1; i<N; i++) & \quad \text{for } (j=1; j<N; j++) \\
\end{align*}
\]

- Distance and direction vectors?
- Definition:
  \[ D = (d_1, \ldots, d_n) \text{ is loop-carried at level } i \text{ if } d_i \text{ is the first nonzero element.} \]

A Few Words about n-Dimensional Arrays in C

- Largely conceptual, due to difficulty in expressing this in C for dynamically allocated data structures
- Imagine the following macros,

\[
\begin{align*}
\text{\#define 2dAccess(i,j,\text{dim} \_i) \backslash} \\
& \text{i+j*dim\_i} \\
\text{\#define 3dAccess(i,j,k,\text{dim} \_i,\text{dim} \_j) \backslash} \\
& \text{i+j*dim\_i+k*dim\_i*dim\_j}
\end{align*}
\]
Test for Parallelization

The \( i \)th loop of an \( n \)-dimensional loop is parallelizable if there does not exist any level \( i \) data dependences.

The \( i \)th loop is parallelizable if for all dependences \( D = (d_1, \ldots, d_n) \), either
\[ (d_1, \ldots, d_{i-1}) > 0 \]
or
\[ (d_1, \ldots, d_i) = 0 \]

Safe Parallelization of CUDA Code

- Dependencies must be carried by
  - (a) Loops that execute on the host
  - OR, loops that execute within a kernel function

  *May be able to use synchronization for dependences across threads, but not across blocks (subtle distinction)*

Parallelization Algorithm

- For each pair of dynamic accesses to the same array within a loop nest:
  - determine if there exists a dependence between that pair
- Key points:
  - \( n^2 \) tests for \( n \) accesses in loop!
  - a single access is compared with itself
  - includes accesses in all loops within a nest

Dependence Testing

- Question so far:
  - What is the distance/direction (in the iteration space) between two dynamic accesses to the same memory location?
- Simpler question:
  - Can two data accesses ever refer to the same memory location?

```c
for (i=11;i<=20;i++)
```

```c
for (i=11;i<=20;i++)
```
Restrict to an Affine Domain

for (i=1; i<N; ++i) {
    for (j=1; j<N; ++j) {
        A[i+2*j+3, 4*i+2*j, 3*i] = ...;
        ... = A[1, 2*i+1, j];
    }
}

• Only use loop bounds and array indices which are integer linear functions of loop variables.

• Non-affine example:

for (i=1; i<N; ++i) {
    A[i*j] = A[i*(j-1)];
}

Equivalence to Integer Programming

• Need to determine if \( F(i) \leq G(i) \), where \( i \) and \( i' \) are iteration vectors, with constraints \( i, i' \geq L, U \geq i, i' \).

• Example:

for (i=1; i<=100; ++i) {
    A[i] = A[i-1];
}

• Inequalities:

1 \leq iw \leq 100, \quad ir = iw - 1, \quad ir \leq 100

• Integer Programming is NP-complete

- \( O(\text{size of the coefficients}) \)
- \( O(n^2) \)

Introducing Omega Calculator

• A software tool used in compilers

• Description:
  - Solves “Presburger formulas”, logical formulas built from affine constraints over integer variables, logical connectives and quantifiers.
  - Can formulate these dependence problems, and derive existence of dependences and distances.

• Relevant to locality optimizations as well, next week’s focus

• Can download from:
  - http://www.cs.utah.edu/~chunchen/omega/
  - Also available from CADE Linux machines in:
    - ~cs6963/bin/oc
Using Omega Calculator

- Example:
  for (i=2; i<=100; i++)
  A[i] = A[i-1];
- Define relation iw = i, and iw = ir-1 in the iteration space 2 <= i <= 100.
  R := ([iw] -> [ir]: 2 <= iw, ir <= 100 && iw < ir && iw = ir-1);
  Result: { [iw] -> [ir+1] : 2 <= iw <= 99 }

Using Omega Calculator, cont.

- Example:
  for (i=20; i<=29; i++)
  A[i] = A[i-10];
- Define relation iw = i, and iw = ir-10 in the iteration space 20 <= i <= 29.
  R := ([iw] -> [ir]: 20 <= iw, ir <= 29 && iw < ir && iw = ir-10);
  Result: { [iw] -> [ir] : FALSE }

2-D Omega Example

- Example:
  for (i=0; i<n; i++)
  for (j=i; j<n; j++)
  a[i][j] = a[i-1][j+1];
- Formula (more complex):
  R := ([iw,jw] -> [di,dj]: exists (ir, jr): 1 <= iw, ir <= n && 1 <= jw, jr <= n && iw = ir-1 && jw = jr+1 && di = ir - iw && dj = jr - jw);
  Result: { [iw,jw] -> [ir,jr] : FALSE }

Calculating Distance Vectors

- Example from before:
  for (i=1; i<=100; i++)
  for (j=1; j<=100; j++)
  a[i][j] = a[i-1][j+1];
- Omega formula:
  R := ([iw,jw] -> [di,dj]: exists (ir, jr): 1 <= iw, ir <= n && 1 <= jw, jr <= n && iw = ir-1 && jw = jr+1 && di = ir - iw && dj = jr - jw);
  Result: { [iw,jw] -> [1,-1] : 1 <= iw <= 99 && 2 <= jw <= 100 }
Aside: What about dependences for other data structures?

- Pointer-based
  - Pointer alias analysis
  - Shape analysis
- Objects
  - Escape analysis
- In practice
  - Lots of #pragma and special flags for programmer to assert no dependence

Homework Assigned Monday

- Example questions
  - Given some sequential code, do the following:
    - Show distance vectors and decide which loop can be parallelized
    - Show Omega dependence relations
    - Show correct CUDA code
  - Memory hierarchy optimization
    - Simple tiling example
    - Identify safety of code transformations
    - Given description of goal, show CUDA code to manage memory hierarchy

Summary of Lecture

- Data dependence can be used to determine the safety of reordering transformations such as parallelization
  - Preserving dependences = preserving "meaning"
- Iteration, distance and direction vectors are abstractions for understanding whether reordering transformations preserve dependences
  - Parallelization of CUDA kernel programs can be viewed as a reordering transformation of a sequential implementation
- Dependence testing on array accesses in loops has been shown to be equivalent to integer programming
  - Omega calculator demonstrated

What’s Ahead

- Next week
  - Homework assignment on Monday
  - Managing the memory hierarchy
  - Initial discussion of projects
- February 16: President’s Day holiday
- February 18:
  - Jim Guilkey to present MPM for projects
- February 20 (Friday):
  - Make up class (we’ll discuss control flow)