Pipelining
Announcements

- HW 2 will be released tomorrow
- Lab 3 due Monday
- Complete reading assignment on *modulo scheduling* before the Tuesday lecture
Agenda

- Common parallelization techniques
  - Parallel processing vs. Pipelining

- Introduction to pipelining
  - Common forms in hardware accelerators
  - Throughput restrictions: resources and recurrences

- Systolic arrays: combining parallel processing and pipelining
  - Uniform recurrence equations
  - Case study on matrix multiplication
Recap: Compatibility and Conflict Graphs

- **Compatibility graph**
  - Partition the graph into a **minimum number of cliques**
    - Clique in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge

- **Conflict graph**
  - Color the vertices by a **minimum number of colors** (chromatic number), where adjacent vertices cannot use the same color

![Graph Diagrams](image_url)

A scheduled DFG

**Clique partitioning** on compatibility graph

**Coloring** on conflict graph
# Exercise: Meeting Assignment Problem

<table>
<thead>
<tr>
<th>Meeting</th>
<th>Schedule (am)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9:00~11:00</td>
</tr>
<tr>
<td>B</td>
<td>9:30~10:00</td>
</tr>
<tr>
<td>C</td>
<td>10:00~11:00</td>
</tr>
<tr>
<td>D</td>
<td>11:00~11:30</td>
</tr>
</tbody>
</table>

**Conflict graph**
(chromatic number?)

**Compatibility graph**
(clique cover?)

Gantt chart
Parallelization Techniques

- **Parallel processing**
  - Emphasizes concurrency by *replicating* a hardware structure several times (typically homogeneous)
    - High performance is attained by having all structures execute simultaneously on different parts of the problem to be solved

- **Pipelining**
  - Takes the approach of *decomposing* the function to be performed into smaller stages and allocating separate hardware to each stage (typically heterogeneous)
    - Data/instructions flow through the stage of a hardware pipeline at a rate (often) independent of the length of the pipeline

[source: Peter Kogge, The Architecture of Pipelined Computers]
Case Study: Digit Recognition Lab

- Use a simple machine learning algorithm to recognize handwritten digits
  - 2000 training instances per digit
  - Each training/test instance is a 7x7 bitmap after downsampling

MNIST dataset: http://yann.lecun.com/exdb/mnist/
K-Nearest-Neighbor (KNN) Implementation

```c
bit4 digitrec( digit input )
{
    #include "training_data.h"
    // This array stores K minimum distances per training set
    bit6 knn_set[10][K_CONST];
    // Initialize the knn set
    for ( int i = 0; i < 10; ++i )
        for ( int k = 0; k < K_CONST; ++k )
            // Note that the max distance is 49
            knn_set[i][k] = 50;

L2000: for ( int i = 0; i < TRAINING_SIZE; ++i ) {
    L10:  for ( int j = 0; j < 10; j++ ) {

        // Read a new instance from the training set
digit training_instance = training_data[j * TRAINING_SIZE + i];
        // Update the KNN set
update_knn( input, training_instance, knn_set[j] );
    }
}
```

Assuming 10 cycles per innermost loop (L10)
~200K cycles by default without optimizations
10x Speedup through Parallel Processing

```c
bit4 digitrec( digit input )
{
    #include "training_data.h"
    // This array stores K minimum distances per training set
    bit6 knn_set[10][K_CONST];
    // Initialize the knn set
    for ( int i = 0; i < 10; ++i )
        for ( int k = 0; k < K_CONST; ++k )
            // Note that the max distance is 49
            knn_set[i][k] = 50;

    // Read a new instance from the training set
    digit training_instance = training_data[j * TRAINING_SIZE + i];
    // Update the KNN set
    update_knn( input, training_instance, knn_set[j] );
}
```

Unroll inner loop completely

Partition training set into 10 banks

10 instances of “update_knn” running in parallel

~20K cycles after parallelization
Further Speedup through Pipelining

```c
bit4 digitrec( digit input )
{
    #include "training_data.h"
    // This array stores K minimum distances per training set
    bit6 knn_set[10][K_CONST];
    // Initialize the knn set
    for ( int i = 0; i < 10; ++i )
        for ( int k = 0; k < K_CONST; ++k )
            // Note that the max distance is 49
            knn_set[i][k] = 50;

    for ( int i = 0; i < TRAINING_SIZE; ++i )
    
        // Read a new instance from the training set
        digit training_instance = training_data[j * TRAINING_SIZE + i];
        // Update the KNN set
        update_knn( input, training_instance, knn_set[j] );

}
```

Unroll inner loop completely

Pipeline outer loop

Partition training set into 10 banks

Outer loop (L2000) pipelined to II=1
~2K cycles after pipelining
Common Forms of Pipelining

- **Operator pipelining**
  - Fine-grained pipeline (e.g., functional units, memories)
  - Execute a sequence of operations on a pipelined resource

- **Loop/function pipelining** *(focus of this class)*
  - Statically scheduled
  - Overlap successive loop iterations / function invocations at a fixed rate

- **Task pipelining**
  - Coarse-grained pipeline formed by multiple concurrent processes (often expressed in loops or functions)
  - Dynamically controlled
  - Start a new task before the prior one is completed
Operator Pipelining

- Pipelined multi-cycle operations
  - $v_3$ and $v_4$ can share the same pipelined multiplier (3 stages)
Loop Pipelining

- Pipelining is one of the most important optimizations for HLS
  - Key factor: **Initiation Interval (II)**
  - Allows a new iteration to begin processing, II cycles after the start of the previous iteration (II=1 means the loop is fully pipelined)

```c
for (i = 0; i < N; ++i)
    p[i] = x[i] * y[i];
```

Pipelined schedule

- **ld** – Load (memory read)
- **st** – Store (memory write)

Dataflow of loop body

Time (cycles)
Function Pipelining

- Function pipelining: Entire function is becomes a pipelined datapath

```c
void fir(int *x, int *y)
{
    static int shift_reg[NUM_TAPS];
    const int taps[NUM_TAPS] =
        {1, 9, 14, 19, 26, 19, 14, 9, 1};
    int acc = 0;
    for (int i = 0; i < NUM_TAPS; ++i)
        acc += taps[i] * shift_reg[i];
    for (int i = NUM_TAPS - 1; i > 0; --i)
        shift_reg[i] = shift_reg[i-1];
    shift_reg[0] = *x;
    *y = acc;
}
```

Pipeline the entire function of the FIR filter
(with all loops unrolled and arrays completely partitioned)
Task Pipelining

A coarse-grained pipeline for the optical flow algorithm

```c
void gradientWeightingH(unsigned short width, unsigned short height,
  short gradientOrigin[HEIGHT*WIDTH][3],
  short interGradientWeighting[HEIGHT*WIDTH][3]
  )
{
  static unsigned int inIdx = 0;
  static unsigned int outIdx = 0;
  unsigned int k, m, i, j;
  short gradientWeightingRowWindow[3][WindowSize];
  short tmpOutput[3];
  short tmpInput[3];

  for (i = 0; i < height; ++i) { // loop over rows
    for (j = 0; j < width + WeightRadius; ++j) { // loop over columns
      for (m = 0; m < 3; ++m)
        tmpOutput[m] = 0;

      if (j < width) { // make sure it read height*width times
        for (m = 0; m < 3; ++m)
          tmpInput[m] = gradientOrigin[inIdx][m];
        ++inIdx;
      }

      if (j < width && i > WeightRadius && i < height - WeightRadius) {
        for (m = 0; m < 3; ++m) {
          for (k = 0; k < WeightSize-1; ++k)
            gradientWeightingRowWindow[m][k] = gradientWeightingRowWindow[m][k+1];
        }
      }
    }
  }
}
```
Restrictions of Pipeline Throughput

▷ Resource limitations
  – Limited compute resources
  – **Limited Memory resources (esp. memory port limitations)**
  – Restricted I/O bandwidth
  – Low throughput of subcomponent
...

▷ Recurrences
  – Also known as feedbacks, carried dependences
  – **Fundamental limits of the throughput of a pipeline**
Resource Limitation

- Memory is a common source of resource contention
  - e.g. memory port limitations

Assuming arrays A and B are held in two different SRAMs

Only one read port per SRAM → 1 load / cycle

Port conflict

for (i = 1; i < N; ++i)

Recurrence Restriction

- Recurrences restrict pipeline throughput
  - Computation of a component depends on a previous result from the same component

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>cycle 1</th>
<th>cycle 2</th>
<th>cycle 3</th>
<th>cycle 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\text{ld}_1$</td>
<td>$\text{ld}_2$</td>
<td>+</td>
<td>$\text{st}$</td>
</tr>
<tr>
<td>1</td>
<td>$\text{ld}_1$</td>
<td>$\text{ld}_2$</td>
<td>+</td>
<td>$\text{st}$</td>
</tr>
</tbody>
</table>

\textbf{ld} – Load  \textbf{st} – Store

Assume operation chaining is not allowed here due to cycle time constraint
Type of Recurrences

- **Recurrence** – if one iteration has *dependence* on the same operation in a previous iteration
  - Direct or indirect
  - Data or control dependence

- **Types of dependences**
  - True dependences, anti-dependences, output dependences
  - Inter-iteration, intra-iteration

- **Distance** – *number of iterations* separating the two dependent operations
  (0 = same iteration or intra-iteration)
True Dependences

- True dependence
  - Aka flow or RAW (Read After Write) dependence
  - \( S_1 \rightarrow^t S_2 \)
    - Statement \( S_1 \) precedes statement \( S_2 \) in the program and computes a value that \( S_2 \) uses

**Example 1**

```c
for (i = 0; i < N; i++)
```

Inter-iteration true dependence on \( A \) (distance = 1)

**Example 2**

```c
for (i = 0; i < N; i++)
    sum += A[i];
```

Inter-iteration true dependence on \( \text{sum} \) (distance = 1)
Anti-Dependences

- **Anti-dependence**
  - Aka WAR (Write After Read) dependence
  - $S_1 \rightarrow^a S_2$
    - $S_1$ precedes $S_2$ and may read from a memory location that is later updated by $S_2$
  - Renaming (e.g., SSA) can resolve many of the WAR dependences

Example

```c
for ( ... i++ ) {
    A[i-1] = b - a;
    B[i] = A[i] + 1
}
```

Inter-iteration anti-dependence on $A$ (distance = 1)
Output Dependences

- **Output dependence**
  - Aka WAW (Write After Write) dependence
  - S1 precedes S2 and may write to a memory location that is later (over)written by S2
  - Renaming (e.g., SSA) can resolve many of the WAW dependences

**Example**

```c
for (... i++) {
    B[i] = A[i-1] + 1
    A[i] = B[i+1] + b
    B[i+2] = b - a
}
```

Inter-iteration output dependence on B (distance = 2)
Systolic Arrays

- An array of processing elements (PEs) that process data in a systolic manner using nearest-neighbor communication
  - Systolic means “data flows from memory in a rhythmic fashion, passing through many processing elements before it returns to memory” – H.T. Kung

In Sparse Matrix Proceedings, 1978
Uniform Recurrence Equations (UREs)

- Any systolic algorithm can be described by a set of UREs
  - i.e., an n-dimensional loop nest where the recurrences (inter-iteration dependences) must have constant distances

\[
y = A \ast x
\]
for (int i = 0; i < N; i++)
y[i] = 0;
for (int j = 0; j < N; j++)
y[i] += A[i, j] \ast x[j]

\[
C = A \ast B
\]
for (int i = 0; i < N; i++)
for (int j = 0; j < N; j++)
C[i, j] = 0;
for (int k = 0; k < N; k++)
C[i, j] += A[i, k] \ast B[k, j]

Matrix Vector Multiplication (MV) in UREs

\[
Z[i, j] = 0, \text{ when } j = 0
\]
\[
Z[i, j] = Z[i, j - 1] + A[i, j] \ast x[j], \text{ when } j > 0
\]
\[
y[i] = Z[i, N - 1]
\]

Matrix Matrix Multiplication (MM) in UREs

\[
Z[i, j, k] = 0, \text{ when } k = 0
\]
\[
Z[i, j, k] = Z[i, j, k - 1] + A[i, k] \ast B[k, j], \text{ when } k > 0
\]
\[
C[i, j] = Z[i, j, N - 1]
\]
Mapping MM to a Systolic Array

- Map the n-dimensional iteration space into a physical array of PEs

\[ Z[i, j, k] = 0, \text{when } k = 0 \]
\[ Z[i, j, k] = Z[i, j, k - 1] + A[i, k] \cdot B[k, j], \text{when } k > 0 \]
\[ C[i, j] = Z[i, j, N - 1] \]
MM Running on a Systolic Array

- An array of processing elements that process data in a systolic manner

\[ C = A \times B \]
Next Lecture

- More pipelining
Acknowledgements

- These slides contain/adapt materials developed by
  - Prof. Scott Mahlke (UMich)
  - Prof. Jason Cong (UCLA)