Talks: 18-min talk + 2-min Q&A
- Zhijing Li, Christina Chang, Juan Albrecht, Leandro Dorta Duque
  *Insertion Sort*
- Anant Desai, Rick Lin, Aasta Gandhi, Vidya Ramesh
  *Mergesort*
- Yirong Cheng, Joyce Huang, Yixiao Zhang
  *Convolution Engine*

Q&A = Questions, Quizzes & Answers

Vote for your favorite talk after the Thursday class
Announcements

- Lab 4 is released

- Project abstract due Monday 10/29
  - Three project examples are posted on the course website
Final Project (32 pts)

- **3-4 students per group** to work on an open-ended project (9 groups expected)
  - **Form groups on CMS by Thursday 10/25**
  - Project meetings will be scheduled on Friday
    - Instructor will post doodle link on Piazza

- Project abstract due Monday 10/29
  - One page: Title & Theme, Team members, Project idea

- Three project themes
  - **App**: Accelerator design for compute-demanding applications
  - **Arch**: Domain-specific architectures
  - **Tool**: Accelerator-centric compilation/synthesis techniques
Theme 1 (App):
Application-Specific Accelerator Design

- Leverage HLS and FPGA to quickly create hardware accelerators for compute-demanding applications
  - Hardware/software co-design with CPU + FPGA
    - Speed up interesting applications from emerging domains such as computer vision, cryptography, machine learning, etc.

- Design languages: C/C++, possibly OpenCL
- Software tools: Xilinx Vivado HLS, SDSoC, and SDAccel
- FPGA platform: ZedBoard, ZC-706, AWS F1
Theme 2 (Arch):
Domain-Specific Architectures

- Devise novel architectural mechanisms for domain-specific accelerators
  - e.g., systolic array for deep learning, accelerator for sparse linear algebra, programmable crypto engines

- **Software infrastructure**
  - Commercial: Vivado HLS, SDAccel
  - Open source: LLVM, gem5, RISC-V
Theme 3 (Tool): Accelerator-Centric Compilation/Synthesis

- Develop new compilation and/or HLS techniques
  - Hardware-oriented compiler analysis & transformations
  - Improving core HLS algorithms: scheduling, pipelining, binding
  - Optimizing new design metrics (e.g., security, reliability)

- Software infrastructure
  - Commercial: Vivado HLS
  - Open source: LLVM, LegUp
Insertion Sort

Zhijing Li, Christina Chang, Juan Albrecht, Leandro Dorta Duque

Parallel Programming for FPGAs - Chapter 10 Sorting Algorithm
Vocabulary Describing Sorting Algo

- **In-place**: A list of N elements can be sorted using N memory elements. Or it only requires a constant amount of additional memory space (see next slide).
- **Stable**: When two items in the input data have the same sort keys, the order in which the two items appear in the sorted list should match the order in which the items were inputted.
  - Example: Sorting by age in ascending order, input list read from left to right
    - input list [Age:47, Name: “Mike”; Age:25, Name: “John”; Age:25, Name:”Jane”]
    - Sorted output list [Age:25, Name: “John”; Age:25, Name:”Jane”; Age:47, Name:”Mike”]
    - The order of “John” and “Jane” is preserved.
- **Online**: Data can be sorted as it is received.
- **Adaptive**: Efficient for data that is somewhat sorted.
Insertion Sort: Animation

Base case: sorted

The implementation in the book skips the base case by starting at $i = 1$.
Insertion Sort: Animation

Iter 1:

Size = 2;
A[j-1] > item, replace A[j]
with A[j-1], shifting the
array content to the right
to make space for item
Insertion Sort: Animation

Size = 2;
A[j-1] > item, replace A[j] with A[j-1], shifting the array content to the right to make space for item
Insertion Sort: Animation

j==0; replace A[j] with item 1

A = [3, 3, 0, 2, 5, 4]
Insertion Sort: Animation

Size = 3;
A[j-1] > item,
replace A[j]
with A[j-1] to
shift to the right

Iter 1:

A

1 3 0 2 5 4

item

0
Insertion Sort: Animation

Iter 2:

\[ j \quad i \]

\[ A[j-1] > \text{item}; \text{replace } A[j] \text{ with } A[j-1] \]

\[ \text{Size} = 3; \]

\[ A \]

\[ 1 \quad 3 \quad 3 \quad 2 \quad 5 \quad 4 \]

\[ \text{item} \]

\[ 0 \]
Insertion Sort: Animation

Iter 2:

A

1 1 3 2 5 4

\[ A[j-1] > \text{item}; \text{replace } A[j] \text{ with } A[j-1] \]

\[ \text{Size } = 3; \]
Insertion Sort: Animation

Iter 1:

From previous iteration, $j == 0$, so simply replace $A[j]$ with the item (was 0)


A

| 0 | 1 | 3 | 2 | 5 | 4 |

i

j

item 2
Insertion Sort: Animation

Size = 4;
A[j-1] <= item,
replace A[j]
with item
Insertion Sort: Animation Final Result

\[\begin{array}{cccccc}
A & 0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}\]
How One Would Implement Insertion Sort in Software

```c
#include "insertion_sort.h"

void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(int i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        int j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && A[j-1] > item) {
            #pragma HLS pipeline II=1
            A[j] = A[j-1];
            j--;
        }
        A[j] = item;
    }
}
```

No data recurrence here since j is moving leftward.

Some Options for Hardware Optimization

1. Attempt to pipeline L2 (while loop) with II = 1
2. Attempt to pipeline L2 (while loop) with II = 1 AND unroll L2
3. Attempt to pipeline L1 (outer for loop)

Challenges:

- Control recurrence previously mentioned
- Conditional execution of L2 makes scheduling and pipelining L1 (outer for loop) hard
- How to partition the array to do parallel insertion
- How to insert one element per cycle (current implementation has cycles that are performing only swaps)
Option 1 - L2: pipeline II=1

#include "insertion_sort.h"

void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(int i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        int j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && A[j-1] > item
             #pragma HLS pipeline II=1
             A[j] = A[j-1];
             j--;
        }
        A[j] = item;
    }
}

Can we achieve II = 1 for inner loop?

- No significant data recurrences in data path

No!

\[
\begin{array}{c|c|c|c|c|c}
\end{array}
\]
Option 1 - L2: pipeline II=1

No solution?
- Explicitly speculate the read of A[i-1] out of the loop
- Enables the loop exit check to be scheduled with II=1
- At the expense of an additional array access on the last iteration.

```
#include "insertion_sort.h"

void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && t > item) {
            #pragma HLS pipeline II=1
            A[j] = t;
            t = A[j-2];
            j--;
        }
        A[j] = item;
    }
```
Option 2 - L2: pipeline = 1, unroll factor = 2

Constraint: array_partition variable=A, cyclic factor = 2

- array_partition directive results in implementing a number of completely separate memories
- Generating two separate memories from array A[]
- Memory banking can map independent access into different Memory

Can we achieve II = 1 using the given array partitioning?

- Each array access cannot be assigned to a different memory bank
- Vivado HLS is not able to achieve II=1 under the memory constraint
Option 3 - L1: pipeline II=1

Can we achieve II = 1 for the outer loop?

- No.
- Inner loop does not have a statically computable loop bound
- But the question is interesting to explore with appropriate code construction

```c
#include "insertion_sort.h"

void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(int i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        int j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && A[j-1] > item)
            #pragma HLS pipeline II=1
            A[j] = A[j-1];
            j--;
        }
        A[j] = item;
    }
```
Parallelising Insertion Sort
(By restructuring the code)

★ Objective: inserting a new element each cycle
★ Less intuitive
★ We attempt to demonstrate several concepts:
  ○ Writing efficient high-level synthesis code requires that the designer must understand hardware concepts like unrolling and partitioning
  ○ Designers must diagnose any throughput problems, which requires substantial knowledge about both the application and the hardware implementation of that design
  ○ To achieve the best results (high performance and low area) it is often required to rewrite the code in a manner that will create an efficient hardware architecture
Parallelising Insertion Sort

```c
#include "insertion_sort.h"

void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(int i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        int j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && A[j-1] > item) {
            #pragma HLS pipeline II=1
            A[j] = A[j-1];
            j--;
        }
        A[j] = item;
    }
}
```
Parallelising Insertion Sort

```c
#include "insertion_sort.h"

void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(int i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        int j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && A[j-1] > item) {
            #pragma HLS pipeline II=1
            A[j] = A[j-1];
            j--;
        }
        A[j] = item;
    }
}
```
Parallelising Insertion Sort

```c
#include "insertion_sort.h"

void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(int i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        int j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && A[j-1] > item) {
            //pragma HLS pipeline II=1
            A[j] = A[j-1];
            j--;
        }
        A[j] = item;
    }
}
```

```c
#include "insertion_sort_parallel.h"
#include "assert.h"

void insertion_sort_parallel(DTYPE A[SIZE], DTYPE B[SIZE]) {
    //pragma HLS array_partition variable=B complete
    L1:
    for(int i = 0; i < SIZE; i++) {
        //pragma HLS pipeline II=1
        DTYPE item = A[i];
        L2:
        for(int j = SIZE-1; j >= 0; j--) {
            DTYPE t;
            if(j > i) {
                t = B[j];
            } else if(j > 0 && B[j-1] > item) {
                t = B[j-1];
            } else {
                t = item;
                if(j > 0)
                    item = B[j-1];
            }
            B[j] = t;
        }
    }
}
```
Parallelising Insertion Sort

```c
#include "insertion_sort.h"

void insertion_sort(DTYPE A[SIZE]) {
  L1:
  for(int i = 1; i < SIZE; i++) {
    DTYPE item = A[i];
    int j = i;
    DTYPE t = A[j-1];
    L2:
    while(j > 0 && A[j-1] > item) {
      #pragma HLS pipeline II=1
      A[j] = A[j-1];
      j--;
    }
    A[j] = item;
  }
}
```

```c
#include "insertion_sort_parallel.h"
#include "assert.h"

void insertion_sort_parallel(DTYPE A[SIZE], DTYPE B[SIZE]) {
  #pragma HLS array_partition variable=B complete
  L1:
  for(int i = 0; i < SIZE; i++) {
    #pragma HLS pipeline II=1
    DTYPE item = A[i];
    L2:
    for(int j = SIZE-1; j >= 0; j--) {
      DTYPE t;
      if(j > i) {
        t = B[j];
      } else if((j > 0) && (B[j-1] > item)) {
        t = B[j-1];
      } else {
        t = item;
        if(j > 0)
          item = B[j-1];
      }
      B[j] = t;
    }
  }
```
Parallelising Insertion Sort

```c
#include "insertion_sort.h"

void insertion_sort(DTYPE A[SIZE]) {
    for(int i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        int j = i;
        DTYPE t = A[j-1];
        while(j > 0 && A[j-1] > item) {
            #pragma HLS pipeline II=1
            A[j] = A[j-1];
            j--;
        }
        A[j] = item;
    }
}
```

```c
#include "insertion_sort_parallel.h"
#include "assert.h"

void insertion_sort_parallel(DTYPE A[SIZE], DTYPE B[SIZE]) {
    #pragma HLS array_partition variable=B complete
    for(int i = 0; i < SIZE; i++) {
        DTYPE item = A[i];
        for(int j = SIZE-1; j >= 0; j--) {
            DTYPE t;
            if(j > i) {
                t = B[j];
            } else if(j > 0 && B[j-1] > item) {
                t = B[j-1];
            } else {
                t = item;
                if(j > 0)
                    item = B[j-1];
            }
            B[j] = t;
        }
    }
}
```
Parallelising Insertion Sort

Unsorted

Sorted
Explicit Systolic Array for Insertion Sort

- Systolic arrays are often described as components communicating by streams of data
- It exploits the advantages of explicit streams and the `dataflow` directive to optimize data processing
- Hardware approach to analyzing “elements” of an array
Insertion Cells Architecture
Insertion Cell Implementation

```c
void cell0(hls::stream<DTYPE> & in, hls::stream<DTYPE> & out)
{
    static DTYPE local = 0;
    DTYPE in_copy = in.read();
    if(in_copy > local) {
        out.write(local);
        local = in_copy;
    }
    else {
        out.write(in_copy);
    }
}
```

- Smaller value is passed to the output: `out = min(in, local);`
- The output of cell $i$ is passed as input to next cell $i+1$
- Code uses a streaming interface `hls::stream`
- Static variable `local` to preserve its content across multiple function calls
- Each cells needs to have a `local` variable storing the sorted element on that position
Insertion Sort for 8 Insertion Cells

```c
void insertion_cell_sort(hls::stream<DTYPE> & in, hls::stream<DTYPE> & out)
{
    #pragma HLS DATAFLOW
    hls::stream<DTYPE> out0("out0_stream");
    hls::stream<DTYPE> out1("out1_stream");
    hls::stream<DTYPE> out2("out2_stream");
    hls::stream<DTYPE> out3("out3_stream");
    hls::stream<DTYPE> out4("out4_stream");
    hls::stream<DTYPE> out5("out5_stream");
    hls::stream<DTYPE> out6("out6_stream");

    // Function calls;
    cell0(in, out0);
    cell1(out0, out1);
    cell2(out1, out2);
    cell3(out2, out3);
    cell4(out3, out4);
    cell5(out4, out5);
    cell6(out5, out6);
    cell7(out6, out);
}
```

★ Expanding this to a larger number of elements simply requires duplicating more cell functions and connecting them appropriately.

★ The `insertion_cell_sort()` function must be called multiple times in order to sort the entire data.

★ To achieve a task-pipelined architecture, the program used the `dataflow` directive.

**KEY POINT:** given how each insertion cell is independent and inputs to the sort function flow to the right, there are no **recurrences** between cells: resource utilization determines MIU.
Mergesort Background

- Invented by John von Neumann in 1945
- Divide and Conquer algorithm
- $N \log N$
- 3 parts:
  - Break into 2 subarrays
  - Sort subarrays
  - Merge subarrays
MERGE SÖRT

1

2

3

4

https://visualgo.net/bn/sorting
Recursive Implementation

mergesort(item_type s[], int low, int high)
{
    int i; /* counter */
    int middle; /* index of middle element */

    if (low < high) {
        middle = (low+high)/2;
        mergesort(s,low,middle);
        mergesort(s,middle+1,high);
        merge(s, low, middle, high);
    }
}

function merge($x[1...k], y[1...l])$

if $k = 0$: return $y[1...l]$

if $l = 0$: return $x[1...k]$

if $x[1] \leq y[1]$:  
    return $x[1] \circ merge(x[2...k], y[1...l])$

else: 
    return $y[1] \circ merge(x[1...k], y[2...l])$
Learn to program

Make recursive function

No exit condition

Learn to program

Make recursive function

No exit condition

Learn to program

Make recursive function

No exit condition
Issues with Recursion

- Most HLS tools don’t support recursion
  - Unknown bounds at compile time
- All recursive functions can be implemented iteratively
- This includes Mergesort
  - Uses for-loops to iteratively partition into larger and larger subarrays
    - Start at subarray size 1
  - Use another for-loop to loop through all partitions and merge adjacent ones
Iterative Mergesort

```c
void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
    // Each time through the loop, we try to merge sorted subarrays of width elements
    // into a sorted subarray of 2*width elements.
    stage:
    for (int width = 1; width < SIZE; width = 2 * width) {
        merge_arrays:
        for (int i1 = 0; i1 < SIZE; i1 = i1 + 2 * width) {
            // Try to merge two sorted subarrays:
            // A[i1..i1+width-1] and A[i1+width..i1+2*width-1] to temp[i1..2*width-1]
            int i2 = i1 + width;
            int i3 = i1 + 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            merge(A, i1, i2, i3, temp);
        }
    }
    // Copy temp[] back to A[] for next iteration
    copy:
    for(int i = 0; i < SIZE; i++) {
        A[i] = temp[i];
    }
}
```
```c
#include "merge_sort.h"
#include "assert.h"

// subarray1 is in[i1..i2−1], subarray2 is in[i2..i3−1], result is in[i1..i3−1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // Foreach element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
        // Select the smallest available element.
        if((f1 < i2 && in[f1] <= in[f2]) || f2 == i3) {
            out[index] = in[f1];
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = in[f2];
            f2++;
        }
    }
}
```
Restructured Mergesort

- Issues with iterative mergesort:
  - Redundant reads
  - Recursion within variables
- Solution: pipeline merge() function
  - Achieve II = 1
  - Reduces latency, not resources
Restructured Mergesort

```c
#include "merge_sort.h"
#include "assert.h"

// subarray1 is in[i1..i2−1]; subarray2 is in[i2..i3−1]
// sorted merge is stored in out[i1..i3−1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // Foreach element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
        #pragma HLS pipeline II=1
        DTYPE t1 = in[f1];
        DTYPE t2 = in[f2];
        // Select the smallest available element.
        if((f1 < i2 && t1 <= t2) || f2 == i3) {
            out[index] = t1;
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = t2;
            f2++;
        }
    }
}
```
Restructured Mergesort Behavior
Pipeline Problems

- Pipeline is not very deep
- Have to flush out repeatedly
- Results in lots of pipeline bubbles after flushing pipeline
- Flushing pipeline every outer loop iteration kills performance

```c
for (i=0; i<N; i++){
    for (j=0; j<M; j++){
        Op1;
        ...
        OpL;
    }
}
```
Flattening the inner loop

- Nested loops flattened into a single loop
- HLS automatically flattens perfect loop nests
- More complex loops: flatten manually
- Resulting loop has a constant number of iterations
After flattening

```c
for (i=0; i<N*M; i++){
    Op1;
    ...
    OpL;
}
```
Transition to Dataflow

```c
merge_arrays:
    for (int i = 0; i < SIZE; i++) {
        #pragma HLS pipeline II=1
        DTYPE t1 = A[f1];
        DTYPE t2 = A[f2];
        if((f1 < i2 && t1 <= t2) || f2 == i3) {
            temp[i] = t1;
            f1++;
        } else {
            assert(f2 < i3);
            temp[i] = t2;
            f2++;
        }
    }
    if(f1 == i2 && f2 == i3) {
        f1 = i3;
        i2 += 2*width;
        i3 += 2*width;
        if(i2 >= SIZE) i2 = SIZE;
        if(i3 >= SIZE) i3 = SIZE;
        f2 = i2;
    }
```
Dataflow

- Task level pipelining
- Unroll the stage loop in merge_sort()
- Use dataflow directive
- merge_arrays loop separated into instances → handled as separate tasks
Dataflow

```c
void merge_arrays(DTYPE in[SIZE], int width, DTYPE out[SIZE]) {
    int f1 = 0;
    int f2 = width;
    int i2 = width;
    int i3 = 2*width;
    if(i2 >= SIZE) i2 = SIZE;
    if(i3 >= SIZE) i3 = SIZE;
    merge_arrays:
    for (int i = 0; i < SIZE; i++) {
        #pragma HLS pipeline II=1
        DTYPE t1 = in[f1];
        DTYPE t2 = in[f2];
        if((f1 < i2 && t1 <= t2) || f2 == i3) {
            out[i] = t1;
            f1++;
        } else {
            assert(f2 < i3);
            out[i] = t2;
            f2++;
        }
        if(f1 == i2 && f2 == i3) {
            f1 = i3;
            i2 += 2*width;
            i3 += 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            f2 = i2;
        }
    }
}
```

```c
void merge_sort_parallel(DTYPE A[SIZE], DTYPE B[SIZE]) {
    #pragma HLS dataflow
    DTYPE temp[STAGES-1][SIZE];
    #pragma HLS array_partition variable=temp complete dim=1
    int width = 1;
    merge_arrays(A, width, temp[0]);
    width *= 2;
    for (int stage = 1; stage < STAGES-1; stage++) {
        #pragma HLS unroll
        merge_arrays(temp[stage-1], width, temp[stage]);
        width *= 2;
    }
    merge_arrays(temp[STAGES-2], width, B);
}
```
Final Thoughts

- Mergesort requires \( \log N \) comparisons every clock cycle over \( N \) cycles
- Compared to insertion sort, this is:
  - fewer comparisons
  - more memory utilization
  - higher latency
- Insertion sort is good when data is already mostly sorted
References:

Images:

http://cs.lmu.edu/~ray/notes/alganalysis/


https://idea-instructions.com/merge-sort/

Computer Science Memes for Travelling Salesman Teens

Books:


Convolution Engine

Yirong Cheng, Joyce Huang, Yixiao Zhang

“Convolution engine: balancing efficiency and flexibility in specialized computing”
Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark Horowitz
Overview

Limitations

Convolution Engine

Performance Evaluation
The balancing point of both flexibility and efficiency

- Domain-specific architecture
  - Customized architecture for a set of similar computation tasks
  - Has both satisfiable flexibility and computation efficiency.
  - Convolution Engine is a domain-specific architecture for convolution.
Convolution

- Widely used in several popular applications:
  - Image processing
  - Video processing
  - Convolutional Neural Network

- What is the problem?
  - Very computational heavy and cause great energy consumption in CPU and GPU.

- What is the solution?
  - Let the processor to hundreds of operations per instruction to amortize the instruction cost.
  - Exploit the dataflow and data locality (temporal/spatial reuse) patterns in the algorithms to reduce the number of memory fetch.
What about SIMD?

- Single instruction multiple data extension:
  - ‘average’ the cost of one instruction over multiple operations

- Can achieve 10x better performance with the programmability remained.

- Does not scale well to larger degree of parallelism due to register structure.

How can better performance be achieved?

Utilize the data-flow and data-locality of convolution!
Convolution

\[
(Img \ast f)[n] \overset{\text{def}}{=} \sum_{k=-\infty}^{\infty} Img[k] \cdot f[n-k]
\]  

(1)

\[
(Img \ast f)[n,m] \overset{\text{def}}{=} \sum_{l=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} Img[k,l] \cdot f[n-k,m-l]
\]  

(2)

Computing Abstraction: Map and Reduce
Kernels

**Integer Motion Estimation (IME)**
IME searches for the matching block in the reference image using the SAD operator.

**Fractional Motion Estimation (FME)**
refines the initial match obtained at the IME step to a quarter-pixel resolution.
Kernels

**Scale Invariant Feature Transform** looks for distinctive features in an image.

**Demosaic**
Reconstruct a full color image from the incomplete color samples output from Bayer filtered samples.
Mapping kernels to convolution abstraction

<table>
<thead>
<tr>
<th></th>
<th>Map</th>
<th>Reduce</th>
<th>Stencil sizes</th>
<th>Data flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>IME SAD</td>
<td>Abs diff</td>
<td>Add</td>
<td>4 × 4</td>
<td>2D convolution</td>
</tr>
<tr>
<td>FME 1/2 pixel up-sampling</td>
<td>Multiply</td>
<td>Add</td>
<td>6</td>
<td>1D horizontal and vertical convolution</td>
</tr>
<tr>
<td>FME 1/4 pixel up-sampling</td>
<td>Average</td>
<td>None</td>
<td>–</td>
<td>2D matrix operation</td>
</tr>
<tr>
<td>SIFT Gaussian blur</td>
<td>Multiply</td>
<td>Add</td>
<td>9, 13, 15</td>
<td>1D horizontal and vertical convolution</td>
</tr>
<tr>
<td>SIFT DoG</td>
<td>Subtract</td>
<td>None</td>
<td>–</td>
<td>2D matrix operation</td>
</tr>
<tr>
<td>SIFT extrema</td>
<td>Compare</td>
<td>Logical AND</td>
<td>9 × 3</td>
<td>2D convolution</td>
</tr>
<tr>
<td>Demosaic interpolation</td>
<td>Multiply</td>
<td>Complex</td>
<td>3</td>
<td>1D horizontal and vertical convolution</td>
</tr>
</tbody>
</table>
Convolution Engine: Shifted Registers

Shift Registers
Coefficient Registers
ALU/Multipliers
Map and reduce logic

Map:
- Interface Unit
- Functional Unit

Reduction Unit

Other Hardware

32-element SIMD unit

Customized unit
Programming the convolution engine

Major instructions added to processor ISA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_CE_OPS</td>
<td>Set arithmetic functions for MAP and REDUCE steps</td>
</tr>
<tr>
<td>SET_CE_OPSIZE</td>
<td>Set convolution size</td>
</tr>
<tr>
<td>LD_COEFF_REG_n</td>
<td>Load $n$ bits to specified row of 2D coeff register</td>
</tr>
<tr>
<td>LD_1D_REG_n</td>
<td>Load $n$ bits to 1D shift register. Optional Shift left</td>
</tr>
<tr>
<td>LD_2D_REG_n</td>
<td>Load $n$ bits to top row of 2D shift register. Optional shift row down</td>
</tr>
<tr>
<td>ST_OUT_REG_n</td>
<td>Store top row of 2D output register to memory</td>
</tr>
<tr>
<td>CONVOLVE_1D_HOR</td>
<td>1D convolution step—input from 1D shift register</td>
</tr>
<tr>
<td>CONVOLVE_1D_VER</td>
<td>1D convolution step—column access to 2D shift register</td>
</tr>
<tr>
<td>CONVOLVE_2D</td>
<td>2D convolution step with 2D access to 2D shift register</td>
</tr>
</tbody>
</table>
Programming the convolution engine

// Set MAP function = MULT, Reduce function = ADD
SET_CE_OPS(CE_MULT, CE_ADD);

// Set convolution size 8
SET_CE_OPSIZE(8);

// Load eight rows of eight 8-bit coefficients
// into Coeff Reg's rows 0 to 7
for (i = 0; i < 8; i++) {
    LD_COEFF_REG_128(coeffPtr, 0);
    coeffPtr += coeffWidth;
}

// Load & shift seven rows of sixteen input pixels
// into 2D shift register
for (i = 0; i < 7; i++) {
    LD_2D_REG_128(inPtr, SHIFT_ENABLED);
    inPtr += width;
}

// Filtering loop
for (y = 0; y < height; y++) {
    // Load & Shift 16 more pixels
    LD_2D_REG_128(inPtr, SHIFT_ENABLED);

    // Filter first 8 locations. Because we have
    // access to 128-ALUS, we can filter two 8x8
    // blocks in parallel
    for (RW_OFFSET = 0; RW_OFFSET < 8; RW_OFFSET+=2) {
        CONVOLVE_2D(RW_OFFSET, RW_OFFSET);
    }

    // Add 2 to row 0 of output register
    SIMD_ADD_CONST(0, 2);

    // Store 8 output pixels
    ST_OUT_REG_64(outPtr);
    inPtr += width;
    outPtr += width;
}
Evaluation: Energy/Area optimization

**Energy**

- **Energy normalized to custom** (lower is better)

**Area**

- **Ops/mm² normalized to custom** (higher is better)
Evaluation

SIMD implementation

the instruction overheads and data access energy are still large relative to the amount of computation done

Fixed Function Custom accelerators

exploit the parallelism and data-reuse in their respective kernels, fully amortizing instruction and data fetch.
Evaluation: Computer Engine Performance

Better than SIMD

CE is closely matched to the **data-flow** of convolution-based algorithms

(Similar to the instruction stream to fixed function units)

Worse than fixed function unit

To provide **flexibility**
Evaluation: energy/area vs programmability

Energy

Area