18-643 Lecture 13: Memory Bound Designs

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today: see examples of customizing memory paths to algorithms

• Notices
  – Handout #7: lab 3, due noon, Monday, 10/26
  – Midterm in class, Wed 10/28
  – Proposal due Friday 10/30, worth 30% of project

• Readings:
“Homework” from Last Time
Blocked CNN Kernel on Local Memory

local cnndata_t BufI[[_Tn][_Tr* _S_wts+_K_wts-1][_Tc* _S_wts+_K_wts-1];
local cnndata_t BufO[[_Tm][_Tr][_Tc];
local cnndata_t BufW[[_Tm][_Tn][_K_wts][_K_wts];

for(row_b=0;row_b<_Tr;row_b++){
    for(col_b=0;col_b<_Tc;col_b++){
        for(to_b=0;to_b<_Tm;to_b++){
            for(ti_b=0;ti_b<_Tn;ti_b++){
                for(i=0;i<_K_wts;i++){
                    for(j=0;j<_K_wts;j++){
                        BufO[to_b][row_b][col_b]++
                            BufW[to_b][ti_b][i][j]*
                            BufI[ti_b][_S_wts*row_b+i][_S_wts*col_b+j];
                    }
                }
            }
        }
    }
}

• Assuming sequential execution, what is the access sequence on **BufW** and **BufI**?

ignore **BufO** for now
BufW is pretty clear

for(row_b=0; row_b<Tr; row_b++) {
  for(col_b=0; col_b<Tc; col_b++) {
    for(to_b=0; to_b<Tm; to_b++) {
      for(ti_b=0; ti_b<Tn; ti_b++) {
        for(i=0; i<K_wts; i++) {
          for(j=0; j<K_wts; j++) {
            Buf0[to_b][row_b][col_b] +=
            BufW[to_b][ti_b][i][j] *
            BufI[ti_b][_S_wts*row_b+i] [
            _S_wts*col_b+j];
        }
      }
    }
  }
}

18-643-F20-L13-S5, James C. Hoe, CMU/ECE/CALCM, ©2020
BufI slightly more interesting

```c
for(row_b=0; row_b<_Tr; row_b++){
    for(col_b=0; col_b<_Tc; col_b++){
        for(to_b=0; to_b<_Tm; to_b++){
            for(ti_b=0; ti_b<_Tn; ti_b++){
                for(i=0; i<_K_wts; i++){
                    for(j=0; j<_K_wts; j++){
                        BufO[to_b][row_b][col_b] +=
                        BufW[to_b][ti_b][i][j]*
                        BufI[ti_b][-_S_wts*row_b+i]
                        [-_S_wts*col_b+j];
                    }
                }
            }
        }
    }
}
```

[Zhang, et al., 2015]
What should happen?

```c
for (row_b=0; row_b<_Tr; row_b++){
    for (col_b=0; col_b<_Tc; col_b++){
        for (to_b=0; to_b<_Tm; to_b++){
            for (ti_b=0; ti_b<_Tn; ti_b++){
                #pragma unroll
                for (i=0; i<_K_wts; i++){
                    #pragma unroll
                    for (j=0; j<_K_wts; j++){
                        BufO[to_b][row_b][col_b]+=BufW[to_b][ti_b][i][j]*BufI[ti_b][_S_wts*row_b+i][_S_wts*col_b+j];
                    }
                }
            }
        }
    }
}
```

initiated sequentially into pipelined body

i and j loops fully unrolled; and pipelined (AOC default)

How many potentially concurrent ‘*’?
How to Layout BufW and BufI

- # of bank, width of bank  
  *(height is derived)*
- bank number, row index, word offset

BTW, K=3 in the labs
What AOC might do (K=3)

 BufW

Bank 0

LD
LD
LD
ST
LD
LD
LD
LD
LD
LD
LD
LD
LD
LD
LD

Requested size: 36 kilobytes
Implemented size: 192
kilobytes = 3 replicates x
2^ceil(log2(Requested size))

Number of banks: 1
Bank width (word size): 32
bits
Bank depth: 16384 words
RAM Mode: True dual-port

Additional information:
Running memory at 2x clock
to support more concurrent
ports
What should happen?

```c
for(row_b=0; row_b<_Tr; row_b++){
    for(col_b=0; col_b<_Tc; col_b++){
        for(to_b=0; to_b<_Tm; to_b++){
            #pragma unroll
            for(ti_b=0; ti_b<_Tn; ti_b++){
                #pragma unroll
                for(i=0; i<_K_wts; i++){
                    #pragma unroll
                    for(j=0; j<_K_wts; j++){
                        BufO[to_b][row_b][col_b] +=
                        BufW[to_b][ti_b][i][j]*
                        BufI[ti_b][_S_wts*row_b+i]
                        [ _S_wts*col_b+j ];
                    }
                }
            }
        }
    }
}
```

How many potentially concurrent `*`?
What should happen?

```c
for(row_b=0;row_b<_Tr;row_b++){
    for(col_b=0;col_b<_Tc;col_b++){
        for(to_b=0;to_b<_Tm;to_b++){
            #pragma unroll
            for(ti_b=0;ti_b<_Tn;ti_b++){
                #pragma loop_coalesce
                for(i=0;i<_K_wts;i++){
                    for(j=0;j<_K_wts;j++){
                        BufO[to_b][row_b][col_b]+=
                        BufW[to_b][ti_b][i][j]*
                        BufI[ti_b][(_S_wts*row_b+i]
                            [(_S_wts*col_b+j];
                    }
                }
            }
        }
    }
}
```

initiated sequentially into pipelines
unroll into Tn parallel copies of pipelined i-j loop

i and j loops “flattened” into single loop level; and body is pipelined

How many potentially concurrent ‘*’?
How many times BufO elements visited?

```c
for(row_b=0;row_b<_Tr;row_b++){
    for(col_b=0;col_b<_Tc;col_b++){
        for(to_b=0;to_b<_Tm;to_b++){
            for(ti_b=0;ti_b<_Tn;ti_b++){
                for(i=0;i<_K_wts;i++){
                    for(j=0;j<_K_wts;j++){
                        BufO[to_b][row_b][col_b] +=
                        BufW[to_b][ti_b][i][j]*
                        BufI[ti_b][._S_wts*row_b+i]
                        [._S_wts*col_b+j];
                }
        }
    }
}
```
Some Hints on Lab 4

• Accumulation in BufO (with ti, i, j as inner loop) can be done by a “register”; do you still need a local memory buffer?

• AOC can optimize local memory mapping but sometime needs help (pragma, reshaping, etc.)

• Pay attention to DRAM access pattern; data copying speed matters too

• Know your dependencies (memory and data)

• Best tuned kernel to Lab 2 vs 4 different; best tuned kernel to Layer 1 vs 5 (in lab4) different

Read compiler report and Best Practices Guide
With program loop nests and pragma’s

Basic 6-level innermost loop nests (like L13-S10)
Popping Up a Level:

Topic 1: AI
The Performance Balancing Act

1. Kernels’ op/sec requires some byte/sec — a function of kernel size
2. On-chip SRAM “filters” kernel byte/sec down to DRAM byte/sec — a function of SRAM capacity
3. DRAM system offers some aggregate byte/sec — a function of access pattern
Arithmetic Intensity

• An algorithm has a cost in terms of operation count
  – runtime_{compute-bound} = \# operations / FLOPS

• An algorithm also has a cost in terms of number of bytes communicated (ld/st or send/receive)
  – runtime_{BW-bound} = \# bytes / BW

• Which one dominates depends on
  – ratio of FLOPS and BW of platform
  – ratio of ops and bytes of algorithm

• Average Arithmetic Intensity (AI)
  – how many ops performed per byte accessed
  – \# operations / \# bytes
Roofline Performance Model
[Williams & Patterson, 2006]

Attained Performance of a system (op/sec)

\[
\text{runtime} > \max\left(\frac{\# \text{ op}}{\text{FLOPS}}, \frac{\# \text{ byte}}{\text{BW}}\right)
> \# \text{op} \cdot \max\left(\frac{1}{\text{FLOPS}}, \frac{1}{(\text{AI} \cdot \text{BW})}\right)
\]

\[
\text{perf} = \min(\text{FLOPS}, \text{AI} \cdot \text{BW})
\]
AI and Algorithms

harder to speed up & harder to scale up

easier

[Figure from P&H CO&D, COPYRIGHT 2009 Elsevier. ALL RIGHTS RESERVED.]
Simple AI Example: MMM

```plaintext
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    for(k=0; k<N; k++)
      C[i][j]+=A[i][k]*B[k][j];
```

- \(N^2\) data-parallel dot-product’s
  - operation count: \(N^3\) float-mult and \(N^3\) float-add
- External memory access (assume 4-byte floats)
  - assume \(N\) is large s.t. 1 row/col too large for on-chip
  - \(2N^3\) 4-byte reads (of \(A\) and \(B\)) from DRAM
  - \(\ldots\) \(N^2\) 4-byte writes (of \(C\)) to DRAM \(\ldots\)
- Arithmetic Intensity \(\approx \frac{2N^3}{4 \cdot 2N^3}=1/4\)
  
  GTX1080: 8 TFLOPS vs 320GByte/sec
Less Simple AI Example: MMM

for(i0=0; i0<N; i0+=N\_b) 
  for(j0=0; j0<N; j0+=N\_b) 
    for(k0=0; k0<N; k0+=N\_b) 
      for(i=i0; i<i0+N\_b; i++) 
        for(j=j0; j<j0+N\_b; j++) 
          for(k=k0; k<k0+N\_b; k++) 
            C[i][j]+=A[i][k]*B[k][j];

• Imagine a ‘N/N\_b’x‘N/N\_b’ MATRIX of N\_b\times N\_b matrices
  – inner-triple is straightforward matrix-matrix mult
  – outer-triple is MATRIX-MATRIX mult

• To improve AI, hold N\_b\times N\_b sub-matrices on-chip for data-reuse
AI of blocked MMM Kernel ($N_b \times N_b$)

```plaintext
for (i=i0; i<i0+N_b; i++)
    for (j=j0; j<j0+N_b; j++) {
        t = C[i][j];
        for (k=k0; k<k0+N_b; k++)
            t += A[i][k] * B[k][j];
        C[i][j] = t;
    }
```

- Operation count: $N_b^3$ float-mul and $N_b^3$ float-add
- When $A$, $B$ fit in scratchpad ($2 \times N_b^2 \times 4$ bytes)
  - $2N_b^3$ 4-byte on-chip reads ($A$, $B$) (fast)
  - $3N_b^2$ 4-byte off-chip DRAM read $A$, $B$, $C$ (slow)
- $N_b^2$ 4-byte off-chip DRAM writeback $C$ (slow)
- Arithmetic Intensity = $2N_b^3 / (4 \cdot 4N_b^2) = N_b/8$
Topic 2: Data Layout and Access Pattern
Data Layout and Access Pattern: 2D-FFT

- Row-column algorithm:

\[ 2D-\text{DFT}_{n \times n} = (\text{DFT}_n \otimes I_n)(I_n \otimes \text{DFT}_n) \]

Column Stage  Row Stage

Dataset:
(Logical abstraction of the 2D dataset)
Inefficient DRAM Access Patterns

- Row-wise traversal -> Sequential accesses
- Column-wise traversal -> Large strided accesses

\[
\text{row-major 2D array}
\]

```
\begin{array}{c}
\text{n} \\
\text{..}
\end{array}
```

\[
\text{linear mem space}
\]

```
\begin{array}{c}
\text{n}
\end{array}
```

\[
\text{n}^2
\]

```
\begin{array}{c}
\text{n} \\
\text{..}
\end{array}
```

```
\begin{array}{c}
\text{0}
\end{array}
```

```
\begin{array}{c}
\text{Row buffer size}
\end{array}
```

```
\begin{array}{c}
\text{DDR2-800 Bandwidth on DE4 (per channel)}
\end{array}
```

```
\begin{array}{c}
\text{Bandwidth [GB/s]}
\end{array}
```

```
\begin{array}{c}
0.02 \\
0.06 \\
0.13 \\
0.25 \\
0.5 \\
1 \\
2 \\
4 \\
8 \\
16 \\
32
\end{array}
```

```
\begin{array}{c}
\text{Read}
\end{array}
```

```
\begin{array}{c}
\text{Write}
\end{array}
```

```
\begin{array}{c}
\text{Packet Size [KB]}
\end{array}
```

```
\begin{array}{c}
0 \\
1
\end{array}
```

```
\begin{array}{c}
\text{Gather-Scatter}
\end{array}
```
Tiled Layout and Access Patterns

- **row-major “blocked”**
- **in row-buffer sized chunks**
- **linear mem space**
- **k^2**
- **n^2**
- **DDR2-800 Bandwidth on DE4 (per channel)**
  - Bandwidth [GB/s]
  - Packet Size [KB]
  - Read
  - Write
  - Row buffer size
Design Generator w/ Tensor Formalism

\[
2D-\text{DFT}_{n \times n} = \left( \text{DFT}_n \otimes I_n \right) \left( I_n \otimes \text{DFT}_n \right)
\]

\[
= \prod_{i=0}^{1} \left( L_n^{n^2} \left( I_n \otimes \text{DFT}_n \right) I_n^2 \right)
\]

row-column algorithm

symmetric algorithm

row stage

column stage

write tiles column-wise

transpose and re-tile on-chip

FFT processing

read tiles row-wise

linearize on-chip

[\text{symmetric algorithm with tiling}]

[Akin, et al., FCCM 2012]
Topic 3: Irregular
Irregular: Breadth First Search

Large graph has more than millions of nodes with may be handful edges per node
Breadth-First Search

foreach (node n in graph) n.dist=∞;

worklist = {root}; root.dist=0;

foreach (node n in worklist) {
    foreach (neighbor of n) {
        if (n.dist + 1 < neighbor.dist) {
            neighbor.dist = n.dist + 1;
            add neighbor to worklist;
        }
    }
}

(see http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/bread_first_search)
Compressed Sparse Row (CSR)

Adjacency Matrix

- Dense array of all non-0 elements in row-order (holds col/dest index)
- Sparse array indexed by row/src idx (holds offset into element array)

Large graph has millions or more nodes each with may be handful edges per node.
while(wl.mHowmany) { // worklist not empty
    // repeat for each node on frontier
    int curr=wl.mList[wl.mDeq]; // S0
    int myDist=graph->mPerNode[curr].dist; // S1
    int numEdges=graph->mPerNode[curr].fanout; // S1
    int scan=graph->mPerNode[curr].edges; // S1
    { ... dequeue from worklist ...}
    while (numEdges--) {
        // repeat for each neighbor
        int dest=graph->mPerEdge[scan].dest; // S2
        int destDist=graph->mPerNode[dest].dist; // S3
        if ((myDist+1)<destDist) {
            // S4
            graph->mPerNode[dest].dist=myDist+1;
            { ...enqueue dest to worklist...} // S5
        }
        scan++;
    }
}
Elastic HW Processing Pipeline

- **S0**: fetch next node’s index
- **S1**: fetch per-node struct
- **S2**: fetch neighbor per-edge struct
- **S3**: fetch neighbor distance
- **S4**: conditionally update neighbor with new distance
- **S5**: add updated neighbor to Worklist

- **Worklist**: 
  - per-node array
  - per-edge array

- **write-ack**: 100+ns roundtrip
BFS Irregular Access Pattern

• Irregular and graph dependent
  – S0 read worklist: spatial locality, non-temporal
  – S1 read node array (self): no locality
  – S2 read edge array: some spatial locality, non-temporal
  – S3 read node array (neighbor): no locality
  – S4 write node array (neighbor): temporal with S3
  – S5 write worklist: spatial locality, non-temporal

• S3 most problematic of all
  – S1 and S3 lack locality but S3 repeated per neighbor
  – same number of S2 and S3 but S2 has spatial locality
  – BTW, S3 and S4 could have RAW hazard
  – BTW, all read/write granularity is multi-word
“Cache” to the Rescue

- Problem:
  - RAW hazard (different nodes with same neighbor)
  - Updates to neighbors on same multi-word block

  Stall S3 read until conflict free?

- Cache/writebuffer: alloc on S3/dealloc after S4
  - S3 read either hit or go to DRAM then cache
  - S4 write hit in cache then writeback to deallocate

[Wang, FCCM19]
Parting Thoughts

• When scaling data size and performance, memory design quickly become the PROBLEM
  – capacity, bandwidth, latency

• FPGAs specialization is an asset
  – balance memory throughput and compute throughput
  – have data to the right place at the right time
  – alter algorithm to memory constraints

• Designing “memorypath” as important as designing “datapath”