T2S: Programming Spatial Architectures for Productive Performance

Hongbo Rong, Xiaochen Hao, Mingzhe Zhang, Yun Liang, Wenguang Chen

Intel Labs, Peking Univ., Tsinghua Univ., Univ. of Science & Technology of China

FCCM, May 12th, 2021
Thanks to the contributions

- Prof. Zhiru Zhang (Cornell)
  ✔ Yi-Hsiang Lai
  ✔ Nitish Srivastava
  ✔ Shaojie Xiang
  ✔ Brendan Sullivan

- Prof. Yun Liang (PKU)
  ✔ Xiaochen Hao
  ✔ Lianwei Cui
  ✔ Size Zheng
  ✔ Yunshan Jia
  ✔ Xiuping Cui

- Prof. Wenguang Chen (TSU)
  ✔ Mingzhe Zhang
  ✔ Guanyu Feng
  ✔ Huanqi Cao

- Prof. Youhui Zhang (TSU)
  ✔ Weihao Zhang

- Prof. Vivek Sarkar (GaTech)
  ✔ Prithayan Barua

- Prof. Jason Cong (UCLA)
  ✔ Jie Wang

And thanks to the support and help of many people at Intel PCL, SSG, PSG, VTT, DevCloud, Legal, et al.
Disclaimer

The software, tutorial and any accompanying documentation ("Materials") are provided “as is” with no warranties of any kind, whether written, oral, implied or statutory, including warranties of merchantability or fitness for a particular purpose, non-infringement or arising from course of dealing or usage in trade.

These Materials contain the general insights and opinions of Intel Corporation ("Intel"). The information in these Materials are provided for information only and are not to be relied upon for any other purpose than educational. Intel makes no representations or warranties regarding the accuracy or completeness of the information in this Material. Intel accepts no duty to update this Material based on more current information. Intel is not liable for any damages, direct or indirect, consequential or otherwise, that may arise, directly or indirectly, from the use or misuse of the information in this Material.
Agenda

• Concept of T2S
• Access to the tool and tutorials
• A deep dive with matrix multiply as an example
• Summary
Spatial architectures: All about power efficiency & performance

- Massive compute resources, plus memory, distributed over a 2-D plane
- Application defines custom pipelines
  - Exploit massive parallelism, and minimize data movement
  - Potential for big boost to power efficiency and thus performance

HPC spatial programming is hard

1. Very deep pipeline parallelism: key for high-performance only on spatial archs
2. Dramatically lower bandwidth/compute ratio than CPU
3. Prohibitively expensive design space exploration
4. Poor debuggability

What to do:
- Reduce coding and verification efforts dramatically

Source of data: Daya Khudia (Intel, SSG), Gorge Powley (Intel, DCG), Yufei Ma (ASU), Jeremy Fowers (Microsoft), Davor Capalija and Tomasz Czajkowski (Intel, PSG)

Source of data: Christopher J. Hughes (Intel, PCL)
T2S: spatial programming for productive perf

Isolate compute
Unrolling Forward
reusable data
Buffering
Distribute data for better scalability
Serialization
Another data path built similarly
Output datapath built similarly

Id A
Id B
Id B
Id A
C = A * B
A_{0k}
A_{1k}
A_{0k}
A_{0k}
C = A * B
A_{0k}
A_{1k}
A_{1k}

CPU
FPG

datapute
Hypotheses and Validation

- Hypothesis: as long as the same set of optimizations are applied, compiler-generated perf should match ninja perf, but with 1-2 orders of magnitudes of higher productivity.

- Validation: the hypothesis holds at least for dense tensor kernels. 82-92% ninja perf with 3% engineering time across FPGA and CGRA.

<table>
<thead>
<tr>
<th>FPGA</th>
<th>LOC</th>
<th>T2S</th>
<th>Ninja</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systolic Array Size</td>
<td>10×8</td>
<td>10×8</td>
<td></td>
</tr>
<tr>
<td>Vector Length</td>
<td>16×float</td>
<td>16×float</td>
<td></td>
</tr>
<tr>
<td># Logic Elements</td>
<td>214K (50%)</td>
<td>230K (54%)</td>
<td></td>
</tr>
<tr>
<td># DSPs</td>
<td>1,282 (84%)</td>
<td>1,280 (84%)</td>
<td></td>
</tr>
<tr>
<td># RAMs</td>
<td>1,384 (51%)</td>
<td>1,069 (39%)</td>
<td></td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>315</td>
<td>245</td>
<td></td>
</tr>
<tr>
<td>Throughput (GFLOPs)</td>
<td>549</td>
<td>626</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGRA</th>
<th>LOC</th>
<th>Throughput w.r.t Ninja</th>
<th>FMA usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEMM</td>
<td>40</td>
<td>92%</td>
<td>100%</td>
</tr>
<tr>
<td>MTKRP</td>
<td>32</td>
<td>99%</td>
<td>100%</td>
</tr>
<tr>
<td>TTM</td>
<td>47</td>
<td>104%</td>
<td>100%</td>
</tr>
<tr>
<td>TTMc</td>
<td>38</td>
<td>103%</td>
<td>95%</td>
</tr>
</tbody>
</table>

New early results on FPGAs

- Capsule S10 338 GFLOPS (71% peak)
- PairHMM S10 39 GCUPS for fixed-sized inputs
- LU A10 24 GFLOPS for 8x8 matrices
Hypotheses and validation (cont.)

• Hypothesis: A wide range of compute patterns and systolic arrays can be expressed based on UREs (Uniform Recurrence Equations) and space-time transforms.

• Validation: the hypothesis holds at least for dense tensors, dynamic programming, and stencils, and for 1-D, 2-D rectangular or triangular systolic arrays.
Access to the tool

- A tool binary, and a set of tutorials, are freely available at Intel DevCloud
  https://github.com/intel/FPGA-Devcloud/tree/master/main/QuickStartGuides/T

Using T2S on FPGA DevCloud

T2S enables software programmers to build systolic arrays on Intel FPGAs for both productivity and performance.

Expressing matrix multiply on Intel FPGAs for productive performance

Capsule Tutorial

LU Decomposition Tutorial

PairHMM Tutorial

Convolution Tutorial
Features

• Dataflow representation in UREs
• Loop transforms
  • Space-time transform, vectorization, unrolling, flattening, infinitization
• Isolation
• Double buffering
• Data scattering and gathering
DevCloud environment
A deep dive with matrix multiply as an example
A dataflow of matrix multiply

Legend: $ijk$ Iteration indexed by $i, j, k$

- $a_{ik}$ is not related with $j$. So reuse it along $j$ dimension
- $b_{kj}$ is not related with $i$. So reuse it along $i$ dimension
- Reduce $c_{ij}$ (initially 0) along $k$ dimension

A dataflow of matrix multiply
UREs of matrix multiply

\[
A_{ijk} = a_{ik} \text{ if } j=0, \ A_{i(j-1)k} \text{ otherwise}
\]

\[
B_{ijk} = b_{kj} \text{ if } i=0, \ B_{(i-1)jk} \text{ otherwise}
\]

\[
C_{ijk} = 0 \text{ if } k=0, \ C_{ij(k-1)} + A_{ijk} B_{ijk} \text{ otherwise}
\]

Final result: \( C_{ijk} \) if \( k \) reaches its max
T2S specification

\[
A_{ijk} = a_{ik} \text{ if } j=0, \quad A_{i(j-1)k} \text{ otherwise}
\]

\[
B_{ijk} = b_{kj} \text{ if } i=0, \quad B_{(i-1)jk} \text{ otherwise}
\]

\[
C_{ijk} = 0 \text{ if } k=0, \quad C_{ij(k-1)} + A_{ijk} B_{ijk} \text{ otherwise}
\]

Final result: \( C_{ijk} \) if \( k \) reaches its max

- Following Halide convention
- Writing arguments starting from the innermost loop
- Matrices are column-major
- select(condition, x, y)
- An expression (condition? x : y).

\[
\begin{align*}
A(k, j, i) &= \text{select}(j == 0, a(k, i), A(k, j - 1, i)) \\
B(k, j, i) &= \text{select}(i == 0, b(j, k), B(k, j, i - 1)) \\
C(k, j, i) &= \text{select}(k == 0, 0, C(k - 1, j, i)) + A(k, j, i) * B(k, j, i) \\
c(j, i) &= \text{select}(k == K - 1, C(k, j, i))
\end{align*}
\]
T2S specification

1 for (i = 0; i < I; i++)
2 for (j = 0; j < J; j++)
3 for (k = 0; k < K; k++)
4   A(k, j, i) = select(j == 0, a(k, i), A(k, j - 1, i));
5   B(k, j, i) = select(i == 0, b(j, k), B(k, j, i - 1));
6   C(k, j, i) = select(k == 0, 0, C(k - 1, j, i)) + A(k, j, i) * B(k, j, i);
7   c( j, i) = select(k == K - 1, C(k, j, i));

A.merge_ures(B, C, c)
.set_bounds(k, 0, K, j, 0, J, i, 0, I);

• A.merge_ures(B, C, c)
  • Merge all functions into a single loop nest
  • A will then represent this loop nest

• Put UREs into the same loop nest
T2S specification

```cpp
#define I 1024
#define J 1024
#define K 256
#define TYPE Float(32)

ImageParam a("a", TYPE, 2), b("b", TYPE, 2);
Var k("k"), j("j"), i("i");
Func A("A", TYPE, {k, j, i}, Place::Device),
    B("B", TYPE, {k, j, i}, Place::Device),
    C("C", TYPE, {k, j, i}, Place::Device),
c("c", Place::Device);

A(k, j, i) = select(j == 0, a(k, i), A(k, j - 1, i));
B(k, j, i) = select(i == 0, b(j, k), B(k, j, i - 1));
C(k, j, i) = select(k == 0, 0, C(k - 1, j, i)) + A(k, j, i) * B(k, j, i);
c(j, i) = select(k == K - 1, C(k, j, i));

A.merge_ures(B, C, c)
    .set_bounds(k, 0, K, j, 0, J, i, 0, I);
```

Parameter

Declare inputs, loop variables, UREs

UREs

Output

Put UREs into the same loop nest.
Buffer<float> ina(K, I), inb(J, K);
Initialize ina, inb (details skipped)
a.set(ina);
b.set(inb);

Target target = get_host_target();
target.set_feature(Target::IntelFPGA);

Buffer<float> result(J, I);
c.realize(result, target);
result.copy_to_host();
Run it

```
# mkdir tutorials
# cd tutorials
# source /data/t2s/setenv.sh a10
# export QUARTUS_HOME=/glob-development-tools/versions/fpgasupportstack/a10/1.2.1/inteldevstack/init_env.sh
# export OPAE_PLATFORM_ROOT=/glob-development-tools/versions/fpgasupportstack/a10/1.2.1/inteldevstack/a10_gx_pac_ias_1_2_1.pv
# export ACI_BOARD_PACKAGE_ROOT=/glob-development-tools/versions/fpgasupportstack/a10/1.2.1/inteldevstack/a10_gx_pac_ias_1_2_1.pv/opencl/opencl_ops
# export OPAE_PLATFORM_ROOT/bin is in PATH already
# export INTELFGPOCLUD_ROOT=/glob-development-tools/versions/fpgasupportstack/a10/1.2.1/intelFPGA_pro/hld
# export ALTERACLSDKROOT=/glob-development-tools/versions/fpgasupportstack/a10/1.2.1/intelFPGA_pro/hld
# export QUARTUS_HOME/bin is in PATH already
# export source /glob-development-tools/versions/fpgasupportstack/a10/1.2.1/intelFPGA_pro/hld/init_opencl.sh
# export INTELFGPOCLUDROOT is set to /glob-development-tools/versions/fpgasupportstack/a10/1.2.1/intelFPGA_pro/hld. Using that.
#
# Will use $QUARTUS_ROOTDIR_OVERRIDE=/glob-development-tools/versions/fpgasupportstack/a10/1.2.1/intelFPGA_pro/quartus to find
# Quartus

# mcc -I /data/t2s/include -I /data/t2s/lib/a10/libHalide.a -lpthread -lz
# u60752@ss01-n137:~/tutorials/fpga/matrix-multiply/run.sh basic emulator
# Adding
# Adding+ cd /home/u60752/tutorials
# Adding+ rm -rf /home/u60752/tutorials/**
# Adding+ g++ /data/t2s/tutorials/fpga/matrix-multiply/basic/main.cpp -I /data/t2s/include /data/t2s/lib/a10/libHalide.a -lpthread -lz
# u60752@ss01-n137:~/tutorials
# export 'INTEL FPGA OCL_PLATFORM_NAME=Intel(R) FPGA Emulation Platform for OpenCL(TM)' CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA="1 CL _CONFIG_CHANNEL_DEPTH_EMULATION_MODE=strict BITSTREAM="/home/u60752/tutorials/a.aocx PRAGMAUNROLL=1 'AOC_OPTION=-march=emulator
# Putting .aoc: OpenCL kernel compilation completed successfully.
# aoc: Linking Object files....
# aoc: Compiling for Emulation ....
# Success!
```
Tiling

- Matrices’ sizes can be flexible
- Partition the output matrix into tiles
- Compute tile by tile use a systolic array
T2S specification

```c
#define II   4
#define JJ   4
#define KK   256
#define III  2
#define JJJ  4
#define KKK  4
#define TYPE Float(32)
#define I    (a.dim(1).extent() / (III * II))
#define J    (b.dim(0).extent() / (JJJ * JJ))
#define K    (a.dim(0).extent() / (KKK * KK))
#define P             kkk,           jjj,     iii,     kk,          jj, ii, k,     j, i
#define P_iii_minus_1 kkk,           jjj,     iii - 1, kk,          jj, ii, k,     j, i
#define P_jjj_minus_1 kkk,           jjj - 1, iii,     kk,          jj, ii, k,     j, i
#define P_kkk_minus_1 kkk - 1,       jjj,     iii,     kk,          jj, ii, k,     j, i
#define P_kk_minus_1  kkk + KKK - 1, jjj,     iii,     kk + KK - 1, jj, ii, k - 1, j, I
#define P_c                          jjj,     iii,                  jj, ii,        j, i
#define total_i       (iii + III * ii + III * II * i)
#define total_j       (jjj + JJJ * jj + JJJ * JJ * j)
#define total_k       (kkk + KKK * kk + KKK * KK * k)
```

**Parameters**

Outermost loops’ extents now determined by the inputs’ sizes!

**Iterations**

Linearized addresses for reading inputs
ImageParam a("a", TYPE, 2), b("b", TYPE, 2);
Var  kkk("kkk"), jjj("jjj"), iii("iii"), kk("kk"), jj("jj"), ii("ii"), k("k"), j("j"), i("i");
Func A("A", TYPE, {P}, Place::Device),
   B("B", TYPE, {P}, Place::Device),
   C("C", TYPE, {P}, Place::Device),
   c("c", Place::Device);
A(P) = select(jjj == 0, a(total_k, total_i), A(P_jjj_minus_1));
B(P) = select(iii == 0, b(total_j, total_k), B(P_iii_minus_1));
C(P) = select(kkk == 0 && kk == 0 && k == 0,
            0,
            select(kk == 0, C(P_k_minus_1), C(P_kk_minus_1)),
            C(P_kkk_minus_1))
        + A(P) * B(P);
c(P_c) = select((kkk == KKK - 1) && (kk == KK -1) && (k == K - 1), C(P));
A.merge_ures(B, C, c);
.set_bounds(kkk, 0, KKK, jjj, 0, JJJ, iii, 0, III)
.set_bounds(kk, 0, KK, jj, 0, JJ, ii, 0, II)
.set_bounds(k, 0, K, j, 0, J, i, 0, I);
Intermediate results are allocated space in global memory

Sequential loops. No parallelism.

Access global memory for every intermediate result
Issues

• Very inefficient using global memory for intermediate results of function A, B, and C

• No optimization of memory sizes
  • Each Func is allocated a space of size $K * J * I$, i.e. the product of the extents of all the loops
  • When the input sizes are big, these intermediate results can waste a huge amount of memory, e.g. with input matrices of sizes $2^K * 4^K$ and $4^K * 2^K$
Space-time transform and vectorization

A.space_time_transform(kkk, jjj, iii)
   .vectorize(kkk);

- Fully unroll loop jjj and iii. Every iteration turns into a hardware PE.
  - PEs execute in parallel, subject only to the dependences between them
- Vectorize loop kkk
  - Enables data parallelism: KKK number of data from matrix a and b will be loaded together every cycle
- Allocate minimal shift registers for intermediate results.
Generated code looks like…

```c
__kernel void kernel_c_WAIT_FINISH_(...)
{
    float _C_shreg[16][4][2]; // Constant size. Not related with the (dynamic) extents of the outermost loops
    float4 _B_shreg[4][2]; // 4 values will be loaded together from matrix a and b, respectively
    float4 _A_shreg[4][2];
}
```
### Loops Analysis

<table>
<thead>
<tr>
<th>Loop: kernel_c_WAIT_FINISH_B7</th>
<th>Block: kernel_c_WAIT_FINISH_B7</th>
<th>Not specified</th>
<th>240.0</th>
<th>1</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: kernel_c_WAIT_FINISH_B10 (a.cl:123)</td>
<td>Block: kernel_c_WAIT_FINISH_B10</td>
<td>Not specified</td>
<td>240.0</td>
<td>15</td>
<td>422</td>
</tr>
<tr>
<td>Block: kernel_c_WAIT_FINISH_B8</td>
<td>Not specified</td>
<td>240.0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Block: kernel_c_WAIT_FINISH_B6</td>
<td>Not specified</td>
<td>240.0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Block: kernel_c_WAIT_FINISH_B3</td>
<td>Not specified</td>
<td>240.0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

---

**Static estimate of performance: fMax II report**
Static estimate of performance: Loop analysis

kernel_c_WAIT_FINISH_B10:
- Compiler failed to schedule this loop with smaller II due to data dependency on variable(s):
  - _65 (Unknown location)
  - _74 (Unknown location)
  - _C_shreg (a.cl: 84)
__kernel void kernel_c_WAIT_FINISH_(...){
    float _C_shreg[16][4][2]; ...
    for (int _A_s0_i = 0; _A_s0_i < 0 + _0; _A_s0_i++) { ...
        for (int _A_s0_j = 0; _A_s0_j < 0 + _1; _A_s0_j++) {
            for (int _A_s0_k = 0; _A_s0_k < 0 + 2; _A_s0_k++) { ...
                for (int _A_s0_ii_jj = 0; _A_s0_ii_jj < 0 + 16; _A_s0_ii_jj++) { ...
                    #pragma unroll for (int _dummy__1_s0_iii=0;_dummy__1_s0_iii < 0 + 2; _dummy__1_s0_iii++) { ...
                        #pragma unroll for (int _dummy_s0_jjj = 0; _dummy_s0_jjj < 0 + 4; _dummy_s0_jjj++) {
                            float _4 = _C_shreg[_15][_dummy_s0_jjj][_dummy__1_s0_iii];
                            _C_temp[_dummy_s0_jjj][_dummy__1_s0_iii] = _4;
                            #pragma unroll for (int _dummy__2_s0_l1=0;_dummy__2_s0_l1 < 0 + 15; _dummy__2_s0_l1++) {
                                int _5 = 15 - _dummy__2_s0_l1;
                                int _6 = 14 - _dummy__2_s0_l1;
                                float _8 = _C_shreg[_6][_dummy_s0_jjj][_dummy__1_s0_iii];
                                _C_shreg[_5][_dummy_s0_jjj][_dummy__1_s0_iii] = _8;
                            }
                            float _9 = _C_temp[_dummy_s0_jjj][_dummy__1_s0_iii];
                            _C_shreg[0][_dummy_s0_jjj][_dummy__1_s0_iii] = _9;
                        }
                    }
                }
            }
        }
    }
}

Look at the generated code

- C is allocated shift registers, and its size is constant ✔
- Rotate the shift registers of C in each PE ✔
- Dependence cycles across kk iterations ✗
Reordering

#define P kkk, jjj, iii, kk, jj, ii, k, j, i
#define P_iii_minus_1 kkk, jjj, iii - 1, kk, jj, ii, k, j, i
#define P_jjj_minus_1 kkk, jjj - 1, iii, kk, jj, ii, k, j, i
#define P_kkk_minus_1 kkk - 1, jjj, iii, kk, jj, ii, k, j, i
#define P_kkk_minus_1 kkk + KKK - 1, jjj, iii, Kk + KK - 1, jj, ii, k - 1, j, i
__kernel void kernel_c_WAIT_FINISH_(...) {

    float _C_shreg[16][4][2];
    
    for (int _A_s0_i = 0; _A_s0_i < 0 + _0; _A_s0_i++){
        for (int _A_s0_j = 0; _A_s0_j < 0 + _1; _A_s0_j++){
            for (int _A_s0_k = 0; _A_s0_k < 0 + _2; _A_s0_k++){
                for (int _A_s0_kk_ii_jj = 0; _A_s0_kk_ii_jj < 0 + 4096; _A_s0_kk_ii_jj++){

                    #pragma unroll for (int _dummy__1_s0_iii = 0; _dummy__1_s0_iii < 0 + 2; _dummy__1_s0_iii++){
                        #pragma unroll for (int _dummy_s0_jjj = 0; _dummy_s0_jjj < 0 + 4; _dummy_s0_jjj++){
                            float _4 = _C_shreg[15][_dummy_s0_jjj][_dummy__1_s0_iii];
                            _C_temp[_dummy_s0_jjj][_dummy__1_s0_iii] = _4;
                            #pragma unroll for (int _dummy__2_s0_l1 = 0; _dummy__2_s0_l1 < 0 + 15; _dummy__2_s0_l1++){
                                int _5 = 15 - _dummy__2_s0_l1; int _6 = 14 - _dummy__2_s0_l1;
                                float _8 = _C_shreg[_6][_dummy_s0_jjj][_dummy__1_s0_iii];
                                _C_shreg[_5][_dummy_s0_jjj][_dummy__1_s0_iii] = _8;
                                float _9 = _C_temp[_dummy_s0_jjj][_dummy__1_s0_iii];
                                _C_shreg[0][_dummy_s0_jjj][_dummy__1_s0_iii] = _9;
                        }
                    }
                }
            }
        }
    }
}

Loop kk moved outside of jj and ii (actually flattened with them)

Rotate the shift regs of C in each PE

Dependence cycles not crossing kk loop, since registers rotated before the accesses
Static estimate of performance: fMax II report
Static estimate of performance: Loop analysis

<table>
<thead>
<tr>
<th>Loops Analysis</th>
<th>Pipelined</th>
<th>II</th>
<th>Speculated iterations</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kernel_c_WAIT_FINISH_B5 (a:ct:97)</code></td>
<td>Yes</td>
<td>&gt;1</td>
<td>0</td>
<td>Serial ex: Me...</td>
</tr>
<tr>
<td><code>kernel_c_WAIT_FINISH_B8 (a:ct:99)</code></td>
<td>Yes</td>
<td>~221</td>
<td>1</td>
<td>Memory depe...</td>
</tr>
</tbody>
</table>

- Fully unrolled loop (a:ct:102): n/a n/a n/a Unrolled by #...
- Fully unrolled loop (a:ct:105): n/a n/a n/a Unrolled by #...
- Fully unrolled loop (a:ct:110): n/a n/a n/a Unrolled by #...

Details:

**kernel_c_WAIT_FINISH_B8:**

- Compiler failed to schedule this loop with smaller II due to memory dependency:
  - From: Store Operation (a:ct: 261)
  - To: Store Operation (a:ct: 261)
__kernel void kernel_c_WAIT_FINISH_(...) { ...

float _C_shreg[16][4][2]; ...

for (int _A_s0_i = 0; _A_s0_i < 0 + _0; _A_s0_i++){
for (int _A_s0_j = 0; _A_s0_j < 0 + _1; _A_s0_j++){
for (int _A_s0_k = 0; _A_s0_k < 0 + _2; _A_s0_k++){ ...

for (int _A_s0_kk_ii_jj = 0; _A_s0_kk_ii_jj < 0 + 4096; _A_s0_kk_ii_jj++){ ...

#pragma unroll for (int _dummy__1_s0_iii = 0; _dummy__1_s0_iii < 0 + 2; _dummy__1_s0_iii++){
#pragma unroll for (int _dummy_s0_jjj = 0; _dummy_s0_jjj < 0 + 4; _dummy_s0_jjj++){

float _4 = _C_shreg[15][_dummy_s0_jjj][_dummy__1_s0_iii];
_C_temp[_dummy_s0_jjj][_dummy__1_s0_iii] = _4;

#pragma unroll for (int _dummy__2_s0_l1 = 0; _dummy__2_s0_l1 < 0 + 15; _dummy__2_s0_l1++){

int _5 = 15 - _dummy__2_s0_l1; int _6 = 14 - _dummy__2_s0_l1;
float _8 = _C_shreg[_6][_dummy_s0_jjj][_dummy__1_s0_iii];
_C_shreg[_5][_dummy_s0_jjj][_dummy__1_s0_iii] = _8;

float _9 = _C_temp[_dummy_s0_jjj][_dummy__1_s0_iii];
_C_shreg[0][_dummy_s0_jjj][_dummy__1_s0_iii] = _9;

#pragma unroll for (int _A_s0_iii = 0; _A_s0_iii < 0 + 2; _A_s0_iii++){
#pragma unroll for (int _A_s0_jjj = 0; _A_s0_jjj < 0 + 4; _A_s0_jjj++){
...

...  #pragma unroll for (int _A_s0_kkk = 0; _A_s0_kkk < 0 + 4; _A_s0_kkk++){
...

... if (...) {
  float _79 = _C_shreg[0][_dummy_s0_jjj][_dummy__1_s0_iii];
...
...  _c[103] = _79;
...  

The code corresponds to this line of the specification:
c(P_c)=select((kkk==KKK-1)&&(kk==KK-1)&&(k==K-1),C(P))
A write happens only when a reduction is done. But the OpenCL compiler seems to be conservative, and assume a write happens every _A_s0_kk_ii_jj iteration. That is why there is a write-write dependence cycle across the loop.

Look at the generated code again
Isolating the output

Systolic array

Func drainer("drainer", Place::Device);
c.isolate_consumer(drainer);
drainer.space_time_transform(jjj, iii);
Bad II in the drainer now

<table>
<thead>
<tr>
<th>fMAX II Report</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Target II</strong></td>
</tr>
<tr>
<td>Block: kernel_drainer_WAIT_FINISH_B1</td>
</tr>
<tr>
<td>Loop: kernel_drainer_WAIT_FINISH_B2 (a.cl:257)</td>
</tr>
<tr>
<td>Block: kernel_drainer_WAIT_FINISH_B2</td>
</tr>
<tr>
<td>Loop: kernel_drainer_WAIT_FINISH_B3 (a.cl:260)</td>
</tr>
<tr>
<td>Block: kernel_drainer_WAIT_FINISH_B3</td>
</tr>
<tr>
<td>Loop: kernel_drainer_WAIT_FINISH_B5 (a.cl:262)</td>
</tr>
<tr>
<td>Block: kernel_drainer_WAIT_FINISH_B5</td>
</tr>
<tr>
<td>Block: kernel_drainer_WAIT_FINISH_B6</td>
</tr>
<tr>
<td>Block: kernel_drainer_WAIT_FINISH_B4</td>
</tr>
</tbody>
</table>

```
int _drainer_min_2,
int _drainer_min_3,
int _drainer_min_4,
int _drainer_min_5,
int _drainer_stride_1,
int _drainer_stride_2,
int _drainer_stride_3,
int _drainer_stride_4,
int _drainer_stride_5,
int _p0_extent_0,

_state_space_drainer float *restrict _drainer) {
  int _80 = _p0_extent_0 >> 3;
  for (int _drainer_s0_i = 0; _drainer_s0_i < _80; 
    _drainer_s0_i++)
    {
      int _81 = _p1_extent_0 >> 4;
      for (int _drainer_s0_j = 0; _drainer_s0_j < _81; 
        _drainer_s0_j++)
        {
          for (int _drainer_s0_ii jj = 0; _drainer_s0_ii jj < 0 + 16; 
            _drainer_s0_ii jj++)
            {
              #pragma unroll
              for (int _drainer_s0_iii = 0; _drainer_s0_iii < 0 + 2; 
                _drainer_s0_iii++)
            }
        }
    }
```

```
Drainer: loop analysis

### Loops Analysis

<table>
<thead>
<tr>
<th>Loop Description</th>
<th>Pipelined</th>
<th>II</th>
<th>Speculated Iterations</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel_drainer_WAIT_FINISH_B2 (a.ct:257)</td>
<td>Yes</td>
<td>&gt;=1</td>
<td>0</td>
<td>Serial exe: Me...</td>
</tr>
<tr>
<td>kernel_drainer_WAIT_FINISH_B3 (a.ct:260)</td>
<td>Yes</td>
<td>&gt;=1</td>
<td>0</td>
<td>Serial exe: Me...</td>
</tr>
<tr>
<td>kernel_drainer_WAIT_FINISH_B5 (a.ct:262)</td>
<td>Yes</td>
<td>~221</td>
<td>1</td>
<td>Memory dep...</td>
</tr>
</tbody>
</table>

**Fully unrolled loop (a.ct:265)**
- n/a
- n/a
- n/a
- Unrolled by #...

**Fully unrolled loop (a.ct:268)**
- n/a
- n/a
- n/a
- Unrolled by #...

### a.cl

```c
286 int _98 = _drainer_min_2 * _drainer_stride_2;
287 int _99 = _drainer_min_1 * _drainer_stride_1;
288 int _100 = _99 + _drainer_min_0;
289 int _101 = _98 + _100;
290 int _102 = _97 + _101;
291 int _103 = _96 + _102;
292 int _104 = _95 + _103;
293 int _105 = _94 - _104;
```

### Details

**kernel_drainer_WAIT_FINISH_B5:**
- Compiler failed to schedule this loop with smaller II due to memory dependency:
  - From: Store Operation (a.cl: 294)
  - To: Store Operation (a.cl: 294)
Look at the code

```c
channel float _c_channel[2][4] __attribute__((depth(0))) ;
__kernel void kernel_c(...){...
for (int _A_s0_i = 0; _A_s0_i < 0 + _0; _A_s0_i++){...
for (int _A_s0_j = 0; _A_s0_j < 0 + _1; _A_s0_j++){...
for (int _A_s0_k = 0; _A_s0_k < 0 + _2; _A_s0_k++){...
for (int _A_s0_kk_ii_jj = 0; _A_s0_kk_ii_jj < 0 + 4096; _A_s0_kk_ii_jj++){...
float _79 = _C_shreg[0][_A_s0_jjj][_A_s0_iii];
write_channel_intel(_c_channel[_A_s0_iii][_A_s0_jjj], _79); ...
} // kernel kernel_c
...
__kernel void kernel_drainer_WAIT_FINISH_(...){...
for (int _drainer_s0_i = 0; _drainer_s0_i < 0 + _80; _drainer_s0_i++){...
for (int _drainer_s0_j = 0; _drainer_s0_j < 0 + _81; _drainer_s0_j++){...
for (int _drainer_s0_ii_jj = 0; _drainer_s0_ii_jj < 0 + 16; _drainer_s0_ii_jj++){...
#pragma unroll for (int _drainer_s0_iii = 0; _drainer_s0_iii < 0 + 2; _drainer_s0_iii++){
#pragma unroll for (int _drainer_s0_jjj = 0; _drainer_s0_jjj < 0 + 4; _drainer_s0_jjj++){
float __82 = read_channel_intel(_c_channel[_drainer_s0_iii][_drainer_s0_jjj]);
...
```

The systolic array drains results through channels, instead of directly writing memory

✔️
Look at the code (Cont.)

__kernel void kernel_drainer_WAIT_FINISH_(...){...
for (int _drainer_s0_i = 0; _drainer_s0_i < 0 + _80; _drainer_s0_i++){
for (int _drainer_s0_j = 0; _drainer_s0_j < 0 + _81; _drainer_s0_j++){
for (int _drainer_s0_ii_jj = 0; _drainer_s0_ii_jj < 0 + 16; _drainer_s0_ii_jj++){
#pragma unroll for (int _drainer_s0_iii = 0; _drainer_s0_iii < 0 + 2; _drainer_s0_iii++){
#pragma unroll for (int _drainer_s0_jjj = 0; _drainer_s0_jjj < 0 + 4; _drainer_s0_jjj++){
float __82 = read_channel_intel(_c_channel[_drainer_s0_iii][_drainer_s0_jjj]);
int _83 = _drainer_s0_i * _drainer_stride_5;
int _84 = _drainer_s0_j * _drainer_stride_4;
int _85 = _drainer_s0_ii_jj >> 2;  int _86 = _85 * _drainer_stride_3;
int _87 = _drainer_s0_ii_jj & 3;   int _88 = _87 * _drainer_stride_2;
int _89 = _drainer_s0_iii * _drainer_stride_1; int _90 = _89 + _drainer_s0_jjj;
int _91 = _88 + _90; int _92 = _86 + _91; int _93 = _84 + _92;
int _94 = _83 + _93; int _95 = _drainer_min_5 * _drainer_stride_5;
int _96 = _drainer_min_4 * _drainer_stride_4;
int _97 = _drainer_min_3 * _drainer_stride_3;
int _98 = _drainer_min_2 * _drainer_stride_2;
int _99 = _drainer_min_1 * _drainer_stride_1;
int _100 = _99 + _drainer_min_0; int _101 = _98 + _100; int _102 = _97 + _101;
int _103 = _96 + _102; int _104 = _95 + _103; int _105 = _94 - _104;
_drainer[_105] = __82;
The complex address might have confused the OpenCL compiler, which then assumes dependency for safety.
Isolating for serialization and de-serialization

```
Func drainer("drainer", Place::Device),
deserializer("deserializer", Place::Host);
c.isolate_consumer(drainer);
drainer.space_time_transform(jjj, iii);
drainer.isolate_consumer(deserializer);
```
fMax II report

| Kernel/Block | Target || Scheduled MHz | Block ID | Latency | Max Interfacing Iterations |
|--------------|--------|-----------------|----------|---------|---------------------------|
| Block kernel.cBB | Not specified | 2400 | 1 | 0 | 1 |
| Block kernel.cBB | Not specified | 2400 | 1 | 0 | 1 |
| Block kernel.cBB | Not specified | 2400 | 1 | 0 | 1 |
| Kernel/driver_WAIT_FINISH | Target fMax - Not specified MHz | (1x3249) |
| Block kernel/driver_WAIT_FINISH.cBB | Not specified | 2400 | 1 | 2 | 1 |
| Block kernel/driver_WAIT_FINISH.cBB | Not specified | 2400 | 1 | 0 | 1 |
| Kernel/driver_WAIT_FINISH.B2 (cBBx249) | Not specified | 2400 | 1 | 0 | 1 |
| Loop kernel/driver_WAIT_FINISH.B2 | Not specified | 2400 | 1 | 0 | 1 |
| Block kernel/driver_WAIT_FINISH.B2 | Not specified | 2400 | 1 | 0 | 1 |
| Loop kernel/driver_WAIT_FINISH.B2 (cBBx3249) | Not specified | 2400 | 1 | 5 | 1 |
| Loop kernel/driver_WAIT_FINISH.B2 (cBBx3249) | Not specified | 2400 | 1 | 0 | 1 |

All II=1!

Much simpler address generation but still could be further optimized
Isolating full I/O paths

Func aSerializer("aSerializer", Place::Host), aLoader("aLoader", Place::Device),
aFeeder("aFeeder", Place::Device), bSerializer("bSerializer", Place::Host),
bLoader("bLoader", Place::Device), bFeeder("bFeeder", Place::Device),
drainer("drainer", Place::Device), collector("collector", Place::Device),
unloader("unloader", Place::Device),
deserializer("deserializer", Place::Host);

A.isolate_producer_chain(a, aSerializer, aLoader, aFeeder);
A.isolate_producer_chain(b, bSerializer, bLoader, bFeeder);
c.isolate_consumer_chain(drainer);
drainer.space_time_transform(jjj, iii);
drainer.isolate_consumer_chain(collector, unloader, deserializer);
<table>
<thead>
<tr>
<th>Function</th>
<th>Target II</th>
<th>Scheduled fMax</th>
<th>Block II</th>
<th>Latency</th>
<th>Max Interviewing Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>fMax II report</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Function Details**

- **Scheduled fMax**: Specifies the scheduled fMax value for each function.
- **Block II**: Indicates the block II value for each function.
- **Latency**: Represents the latency for each function.
- **Max Interviewing Iterations**: Shows the maximum interview iterations for each function.
Dynamic profile (2*4 PEs, each vectorized by 4)

FPGA GEMM exec time = 2.32507 s

# operations = 34359738368
Throughput: 14.77792 GFLOPS
Scaling up to medium size (8 \times 8 \text{ PEs}, each vectorized by 8)

FPGA GEMM exec time \quad = \quad 0.80683 \text{ s}

# operations = 34359738368

Throughput: 42.58627 \text{ GFLOPS}

Memory bandwidth consumed by the loadings of the input matrices is totally about 22 GB/s
Stalls in reading from bLoader

In short, the input paths become a bottleneck, which makes the design memory-bound.
Optimize input paths for memory bandwidth

- Remove redundant host-device data transfer
  - aSerializer.remove(jjj, jj, j);
  - bSerializer.remove(iii, ii, i);
  - aLoader.remove(jjj, jj);
  - aFeeder.buffer(aLoader, k);
  - bLoader.remove(iii, ii);
  - bFeeder.buffer(bLoader, k);
  - aFeeder.scatter(aLoader, iii);
  - bFeeder.scatter(bLoader, jjj);

- Remove reuse loops in loaders
  - aLoader.remove(jjj, jj);
  - aFeeder.buffer(aLoader, k);
  - bLoader.remove(iii, ii);
  - bFeeder.buffer(bLoader, k);

- Insert a buffer at a loop level that encloses all removed loops in a producer

- Scatter data across consumer PEs
Dynamic profile

FPGA GEMM exec time \(= 0.24568\) s

# operations = 34359738368
Throughput: 139.85691 GFLOPS

Almost an order of magnitude saving of the memory bandwidth
- Memory bandwidth consumed by the loaders are about 2.8 GB/s instead of 22GB/s
Still many stalls in the output paths

128 output channels, all stalled most of the time.

- 8 * 8 drainer PEs, communicating with 8 * 8 systolic array PEs directly
- 8 * 8 collector PEs, communicating with 8 * 8 drainer PEs directly
Simplifying the output paths

drainer.gather(c, iii);
collector.gather(drainer, jji);
collector.vectorize(jjj);
unloader.vectorize(jjj);

FPGA GEMM exec time = 0.18169 s

# operations = 34359738368
Throughput: 189.11542 GFLOPS
Now we have full I/O paths
Remaining bottlenecks

- 8 stalls in the drainer
- 8 stalls in the collector

Much less than before
Scaling up to a large array (10*8 PEs, each vectorized by 16).

### Graph

- **Equation**: \( y \text{ (GFLOPS)} = 34 \text{ (GB/S)} \times x \text{ (FLOP/B)} \)
- **Equation**: \( y \text{ (GFLOPS)} = 2 \times \text{DSPs} \times \text{fMax} = 2 \times 1299 \times 147 \times 0.001 = 384 \)

### Table

<table>
<thead>
<tr>
<th>LUTs</th>
<th>Registers</th>
<th>Logic</th>
<th>I/O pins</th>
<th>DSPs</th>
<th>Memory bits</th>
<th>BRAMs</th>
<th>fmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>188149</td>
<td>399,683</td>
<td>190,304/427,200 (45%)</td>
<td>310/826 (38%)</td>
<td>1,299/1,518 (86%)</td>
<td>32,490,792 / 55,562,240 (58%)</td>
<td>2,065/2,713 (76%)</td>
<td>147.73</td>
</tr>
</tbody>
</table>
Still working on

- Isolate out control signals to simplify the systolic array
- Further increase array size to 11 * 8 PEs, each vectorized by 16
- Add "-fpc -fp-relaxed" to the compilation flag for simpler logic.
- Add "-fmax=500" for possibly higher frequency.
- Add “-high-effort” to increase the chance of success in place and route.
- Seed sweeping.
Summary

- A tool for incremental, intuitive design space exploration
  - Guided by static profile, dynamic profile, and rooflines
  - Hosted on DevCloud, a free and well-maintained software and hardware environment for academics and researchers
- We commit to continual updating and maintenance
- Productivity comes from telling the compiler what to do
- Performance comes from sophisticated implementation of the compiler
  - Still a valuable tool even eventually you implement your design in RTL
  - Help quickly eliminate potential bottlenecks in your design before spending time on RTL