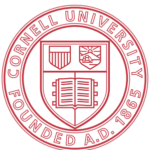
Domain-Specific Programming



Cornell University

Announcements

- Final project
  - In-depth exploration of a research topic
    - (1) Designing new accelerators with HLS; OR
      (2) Developing new automation algorithms/tools
  - **3-4 students / team** (up to 12 teams in total)
    - 11 teams = 11 * 4 students
    - 12 teams = 4 * 3 students + 8 * 4 students
  - Weekly meetings with the instructor start next week on Thu 11/2
    - A Google sheet will be created for meeting scheduling
  - Abstract due Wed 11/8

Review: CNN Acceleration on FPGAs

```
9          for (ki=0; ki<K; ki++) {
10           for (kj=0; kj<K; kj++) {
5              for (trr=row; trr<min(row+Tr, R); trr++) {
6                for (tcc=col; tcc<min(col+Tc, C); tcc++) {
                   #pragma HLS pipeline
7                  for (too=to; too<min(to+To, O); too++) {
                     #pragma HLS unroll
8                    for (tii=ti; tii<(ti+Ti, I); tii++) {
                       #pragma HLS unroll
                       output_fm[too][trr][tcc] +=
                       weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
           }}}}}}
```

Parallelize across input (Ti) and output (To) channels

L is the pipeline depth (# of pipeline stages, II=1)

Number of cycles to execute the above loop nest

$$\approx K \times K \times Tr \times Tc + L \approx Tr \times Tc \times K^2$$

Agenda

- A brief intro to domain-specific languages (DSLs)

- DSLs for accelerator design

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

Donald Knuth on Multicore Architectures

Q: Vendors of multicore processors have expressed frustration at the difficulty of moving developers to this model. As a former professor, what thoughts do you have on this transition and how to make it happen?

> I might as well flame a bit about my personal unhappiness with the current trend toward multicore architecture. To me, **it looks more or less like the hardware designers have run out of ideas, and that they're trying to pass the blame for the future demise of Moore's Law to the software writers by giving us machines that work faster only on a few key benchmarks!** I won't be surprised at all if the whole multithreading idea turns out to be a flop, worse than the "Itanium" approach that was supposed to be so terrific — until it turned out that the wished-for compilers were basically impossible to write.

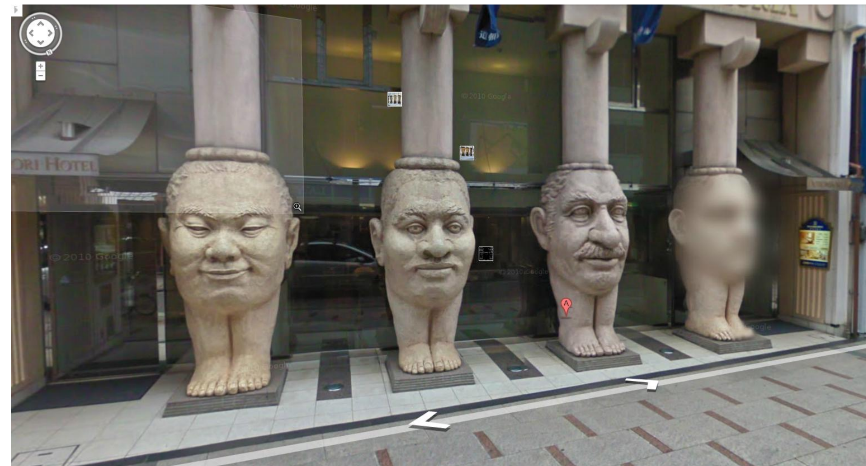
Source: http://www.informit.com/articles/article.aspx?p=1193856, 2008

Blur Filter: Original C++ Code

```
void blur_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height()); // allocate blurx array
  for (int x = 0; x < in.width(); x++)
        for (int y = 0; y < in.height(); y++)
      blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int x = 0; x < in.width(); x++)
        for (int y = 0; y < in.height(); y++)
      blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
```

The blurred face of KFC Mascot



Blur Filter: Optimized C++ Code for Multicore

```
void blur_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
        __m128i *blurxPtr = blurx;
        for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                        a = _mm_loadu_si128((__m128i*)(inPtr-1));
                        b = _mm_loadu_si128((__m128i*)(inPtr+1));
                        c = _mm_load_si128((__m128i*)(inPtr));
                        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                        avg = _mm_mulhi_epi16(sum, one_third);
                        _mm_store_si128(blurxPtr++, avg);
                        inPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
                a = _mm_load_si128(blurxPtr+(2*256)/8);
                b = _mm_load_si128(blurxPtr+256/8);
                c = _mm_load_si128(blurxPtr++);
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_store_si128(outPtr++, avg);
}}}}}
```

11X faster on quad core x86 processor

+ Tiling
+ Vectorization
+ Multithreading

More for Less – Domain-Specific Languages (DSLs)

- Programming languages that are tailored for a specific application domain
  - More accessible and productive for domain experts
  - Restricted expressiveness facilitates more automated optimization and verification
  - Examples: SQL, MATLAB, OpenGL, HTML, …

- Embedded DSLs (eDSLs)
  - A DSL built on a host, typically general-purpose language
  - Examples: Halide (in C++), PyTorch (in Python), TVM (in Python), Chisel (in Scala), …

Case Study: Halide, an eDSL for Image Processing

Main Idea: Separate algorithm (what to compute) from schedule (how to compute it)

```
// Algorithm of Blur Filter
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
```

Algorithm
- Write and test once
- Portable across platforms

```
// Schedule
blurx.compute_at(blur_y, y).unroll(x);
blury.tile(x, y, xi, yi, 256, 32);
```

Schedule
- Specify optimizations
- Explore combinations
- Target different back-ends

Scheduling functions encode common program transformations

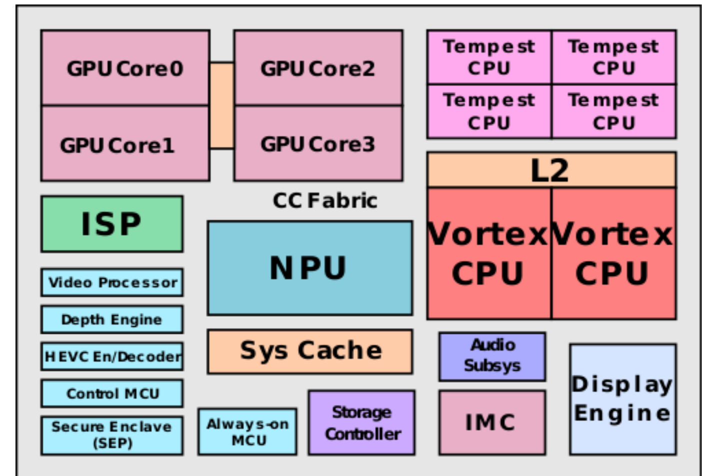
tile: loop tiling
unroll: loop unrolling
compute_at: change order of computation

[1] J. Ragan-Kelley et al. Halide: a Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. PLDI'2013.

[2] https://halide-lang.org

What about Accelerator-Rich Architectures?

- Modern computer systems embrace specialization to improve performance and energy efficiency
- Both hardware and software are increasingly customized for dedicated application domains

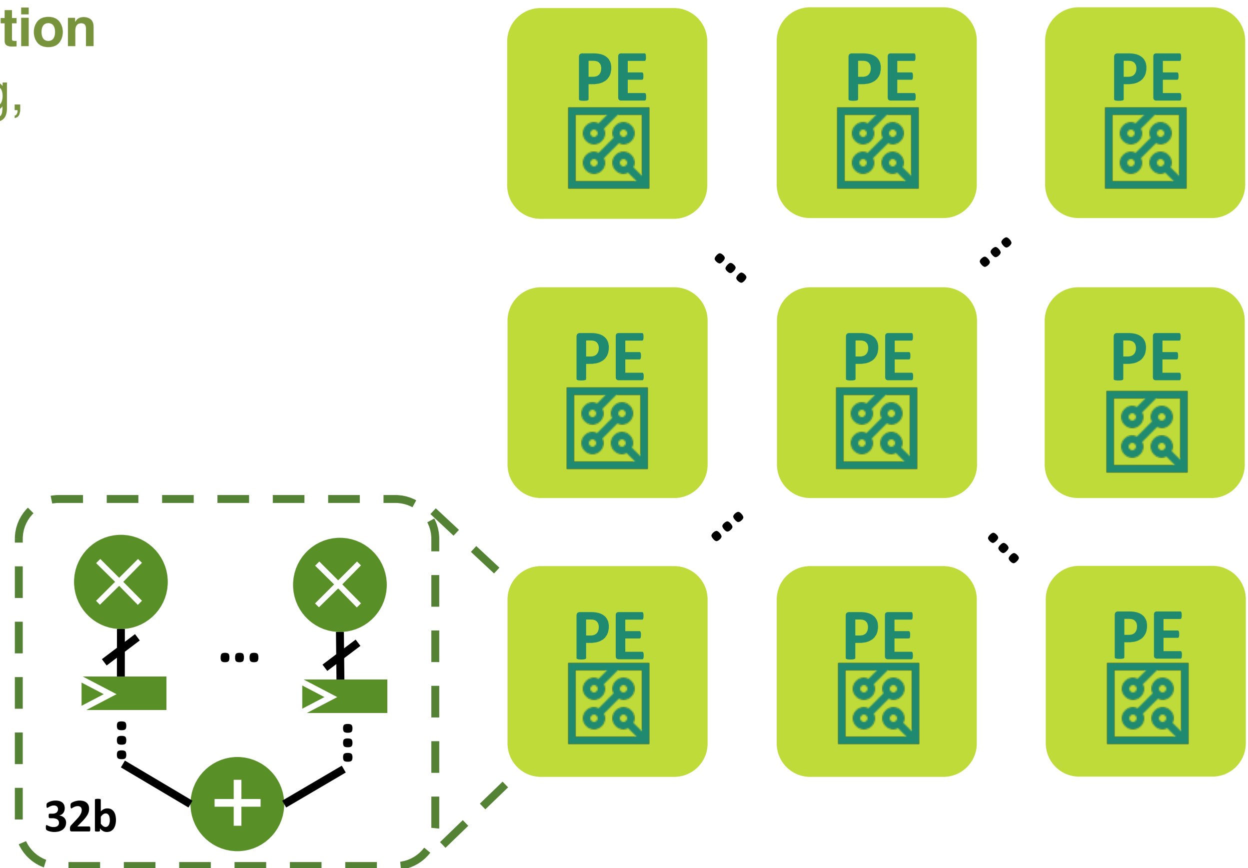


Apple 12 (iPhone X)

Even more challenging to design and program!

Essential Techniques for Hardware Specialization

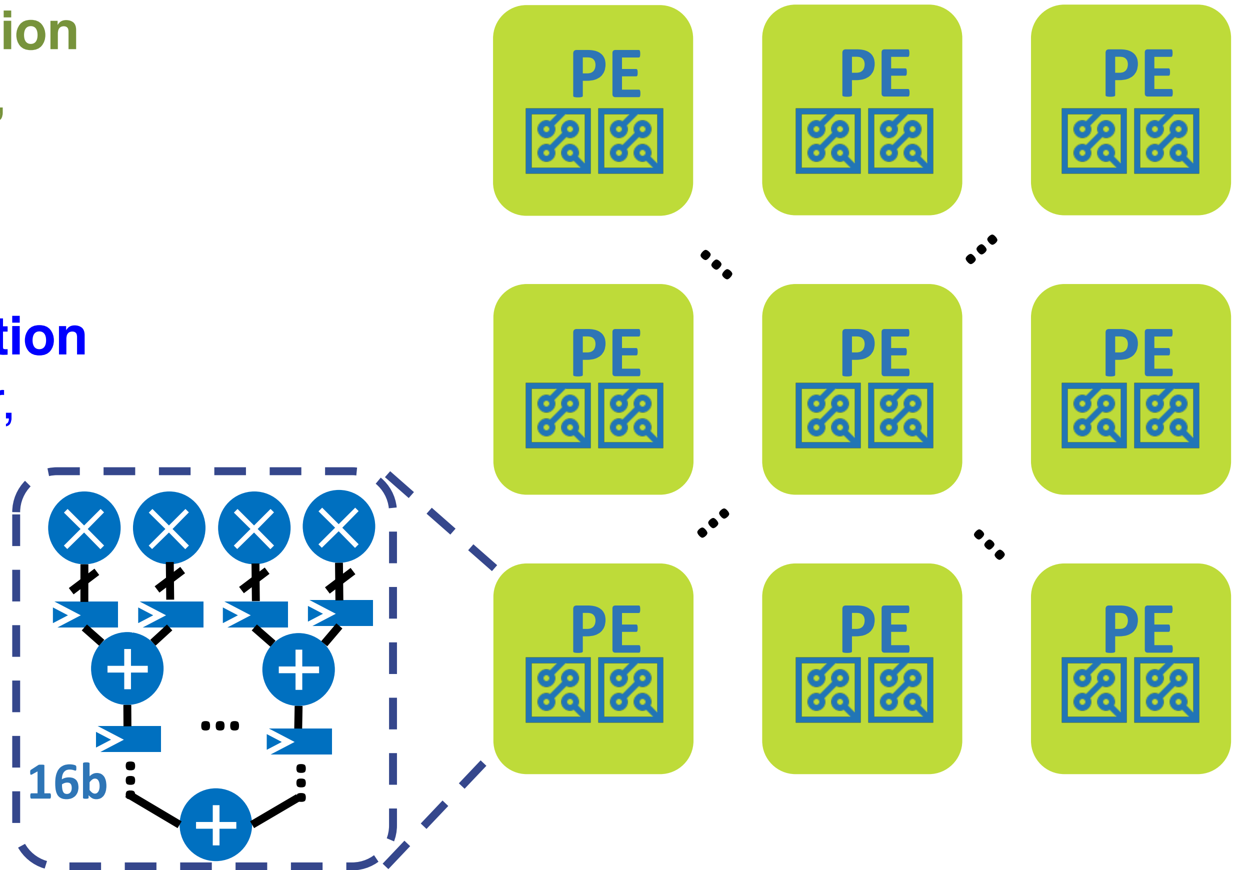
Compute customization
- parallel processing, pipelining …



Essential Techniques for Hardware Specialization

Compute customization
- parallel processing, pipelining …

Data type customization
- low-bitwidth integer, fixed point ...

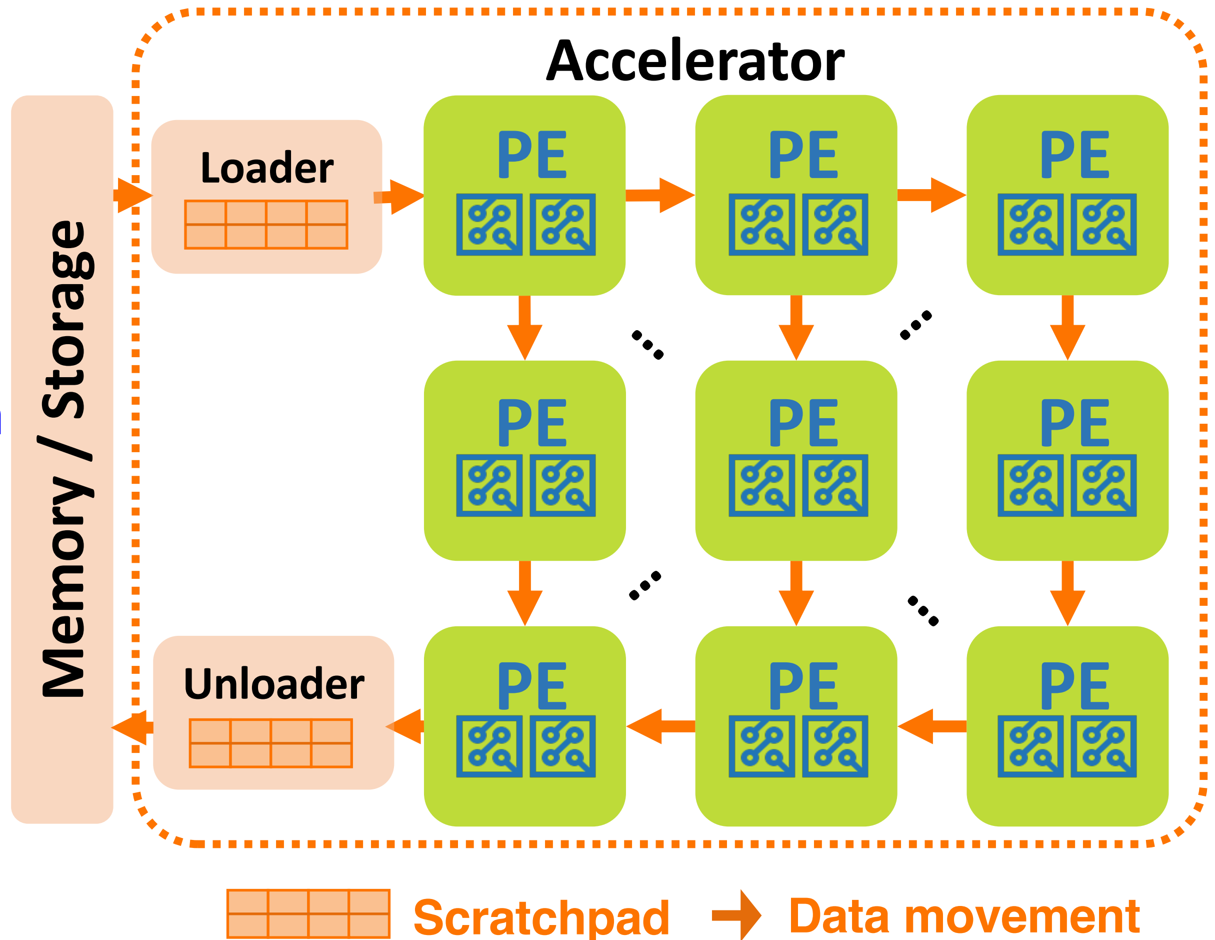


Essential Techniques for Hardware Specialization

Compute customization
- parallel processing, pipelining …

Data type customization
- low-bitwidth integer, fixed point ...

Memory customization
- banking, data reuse, streaming ...

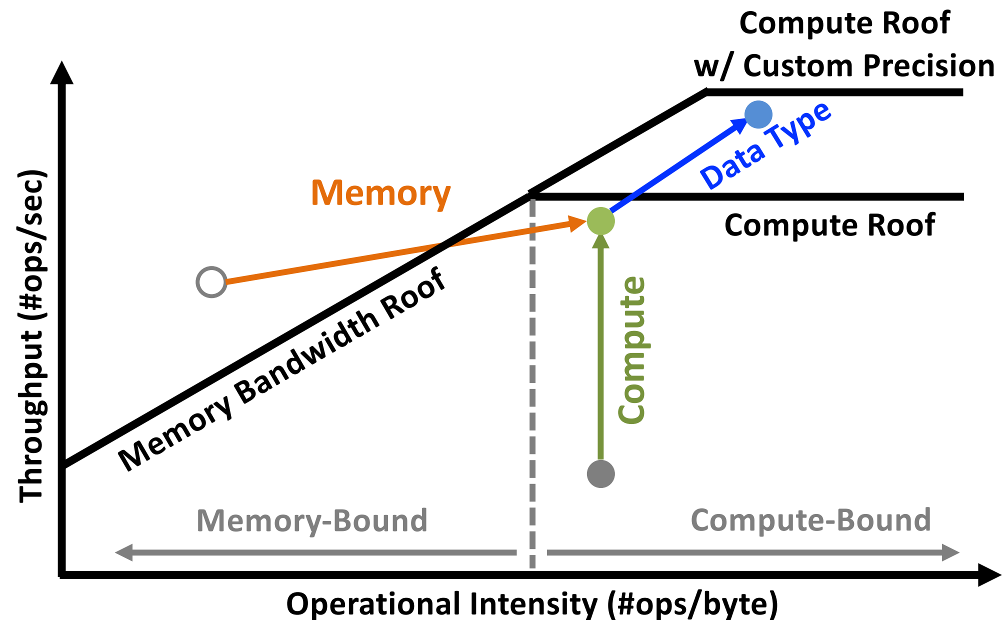


A Roofline View of Customization Techniques

Compute customization
- parallel processing, pipelining …

Data type customization
- low-bitwidth integer, fixed point ...

Memory customization
- banking, data reuse, streaming ...



Building Accelerator with HLS C/C++

<u>Example: Convolution</u>

```
for (int y = 0; y < N; y++)
  for (int x = 0; x < N; x++)
    for (int r = 0; r < 3; r++)
      for (int c = 0; c < 3; c++)
        out[x, y] += image[x+r, y+c] * kernel[r, c]
```

Algorithm#1
Compute Customization
Algorithm#2
Data Type Customization
Memory Customization
Algorithm#3

Entangled hardware customization & algorithm

- Less portable
- Less maintainable
- Less productive

```
#pragma HLS array_partition variable=filter dim=0
 hls::LineBuffer<3, N, ap_fixed<8,4> > buf;
 hls::Window<3, 3, ap_fixed<8,4> > window;
 for(int y = 0; y < N; y++) {
  for(int xo = 0; xo < N/M; xo++) {
#pragma HLS pipeline II=1
   for(int xi = 0; xi < M; xi++) {
    int x = xo*M + xi;
    ap_fixed<8,4> acc = 0;
    ap_fixed<8,4> in = image[y][x];
    buf.shift_up(x);
    buf.insert_top(in, x);
    window.shift_left();
    for(int r = 0; r < 2; r++)
     window.insert(buf.getval(r,x), i, 2);
    window.insert(in, 2, 2);
    if (y >= 2 && x >= 2) {
     for(int r = 0; r < 3; r++) {
      for(int c = 0; c < 3; c++) {
       acc += window.getval(r,c) * kernel[r][c];
     }}
     out[y-2][x-2] = acc;
}}}}
```

Custom compute (Loop tiling)

Custom data type (Quantization)

Custom memory (Reuse buffers)

HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing

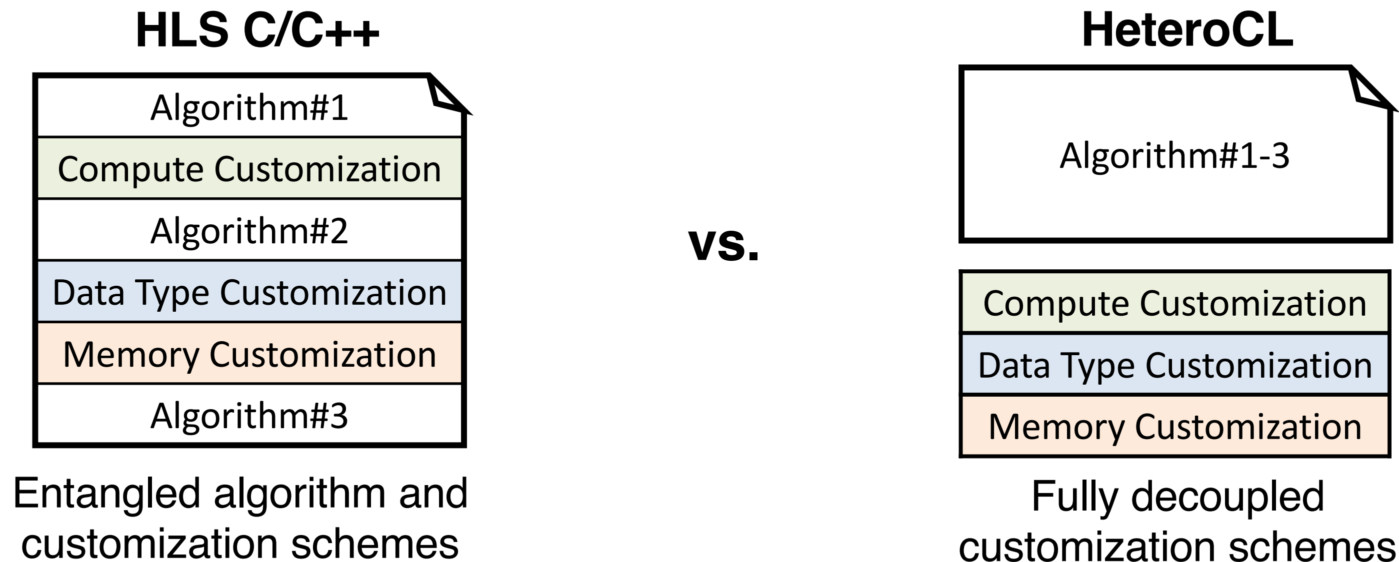
Yi-Hsiang Lai[1], Yuze Chi[2], Yuwei Hu[1], Jie Wang[2], Cody Hao Yu[2], Yuan Zhou[1], Jason Cong[2], Zhiru Zhang[1]

[1]Cornell University, [2]UCLA

FPGA'2019 (Best Paper Award)

HeteroCL Overview

- A Python-based embedded DSL and compilation framework for productive hardware specialization
  - **Portable**: Clean decoupling of algorithm & hardware customizations
  - **Flexible**: Mixed declarative & imperative programming
  - **Efficient**: Mapping to high-performance spatial architecture templates



https://github.com/cornell-zhang/heterocl

Decoupled Compute Customization

- The tensor DSL (built on TVM) separates algorithm from scheduling via declarative programming

Algorithm

HeteroCL code

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c]))
```

Declarative programming

Corresponding HLS code in C

```
for (int y = 0; y < N; y++)
  for (int x = 0; x < N; x++)
    for (int r = 0; r < 3; r++)
      for (int c = 0; c < 3; c++)
        out[x, y] += image[x+r, y+c] * kernel[r, c]
```

Decoupled customization

```
s = hcl.create_schedule()
xo, xi = s.split(out.x, factor=M)
s.reorder(xi, xo, out.y)
```

```
for (int xi = 0; xi < M; xi++)
  for (int xo = 0; xo < N/M; xo++)
    for (int y = 0; y < N; y++)
      for (int r = 0; r < 3; r++)
        for (int c = 0; c < 3; c++)
          out[xi+xo*M, y] +=
            image[xi+xo*M+r, y+c] * kernel[r, c]
```

Tiled loop

Reorder loops

Decoupled Data Type Customization

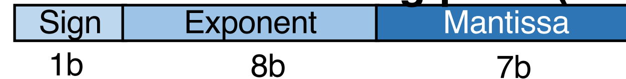
- HeteroCL further enables decoupled algorithm spec and data quantization schemes
  - Provides bit-accurate data type support (e.g., Int(15), Fixed(7,4))

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
  lambda y, x:
    hcl.sum(image[x+r, y+c]*kernel[r, c],
      axis=[r, c]))
```

```
for i in range(2, 8):
  s = hcl.create_scheme()
  s.quantize(out, Fixed(i, i-2))
```

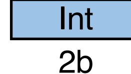
32-bit Floating-point

16-bit Brain Floating-point (bfloat)



8-bit Fixed-point Fixed(8, 6)

2-bit Integer Int(2)



Quantize/ downsize

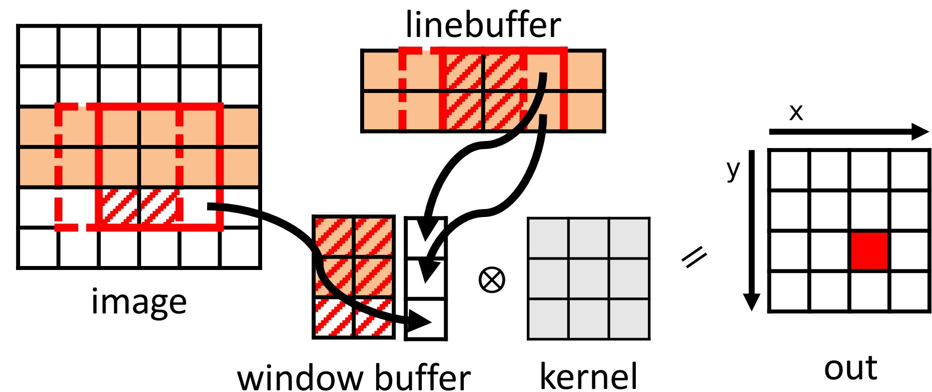
Decoupled Memory Customization

- Inferring custom on-chip storage with .reuse_at()

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c]))
```

```
s = hcl.create_schedule()
linebuf = s[image].reuse_at(out, out.y)
winbuf = s[linebuf].reuse_at(out, out.x)
```

```
for (int y = 0; y < N; y++)
  for (int x = 0; x < N; x++)
    for (int r = 0; r < 3; r++)
      for (int c = 0; c < 3; c++)
        out[x, y] += image[x+r, y+c] * kernel[r, c]
```



Customization Primitives in HeteroCL (a subset)

Compute customization

.split(i, v)	Split loop i of operation C into a two-level nest loop with v as the factor of the inner loop.
.fuse(i, j)	Fuse two sub-loops i and j of operation C in the same nest loop into one.
.reorder(i, j)	Switch the order of sub-loops i and j of operation C in the same nest loop.
.compute_at(C,i)	Merge loop i of the operation P to the corresponding loop level in operation C.
.unroll(i, v)	Unroll loop i of operation C by factor v.
.parallel(i)	Schedule loop i of operation C in parallel.
.pipeline(i, v)	Schedule loop i of operation C in pipeline manner with a target initiation interval v.

Memory customization

.partition(i,v)	Partition dim i of tensor C with a factor v.
.reshape(i,v)	Pack dim i of tensor C into words with a factor v.
.buffer_at(C,i)	Create an intermediate buffer at dim i of operation C to store the results of tensor P.
.reuse_at(C,i)	Create a reuse buffer storing the values of tensor P, where the values are reused at dim i of operation C.
.to(t, d, m)	Move a list of tensors t to destination d with mode m.

Data type customization

.downsize(t, d)	Downsize a list of tensors t to type d.
.quantize(t, d)	Quantize a list of tensors t to type d.

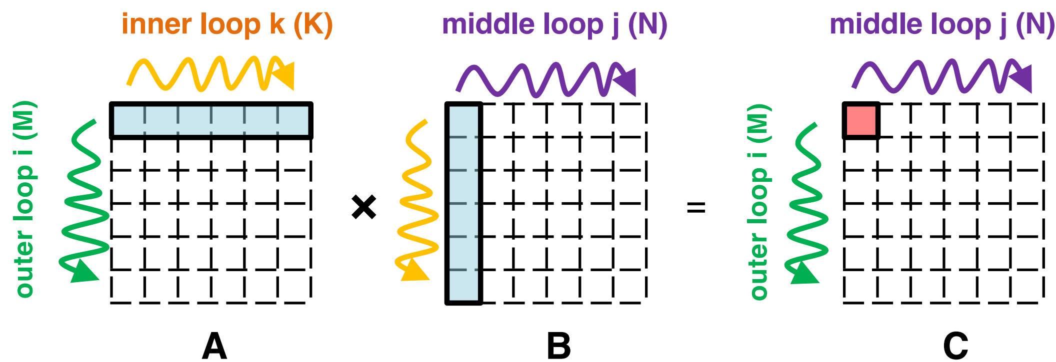
Macros for spatial architecture templates

C.stencil()	Specify operation C to be implemented with stencil with dataflow architectures using the SODA framework.
C.systolic()	Specify operation C to be implemented with systolic arrays using the AutoSA framework.

Case Study: Matrix Multiplication (MM)

- A vanilla MM implementation performs inner product to produce *one output element*
  - **Floating-point accumulation** introduces carried dependency, slowing down the pipeline (II>1)

MatMul via inner product



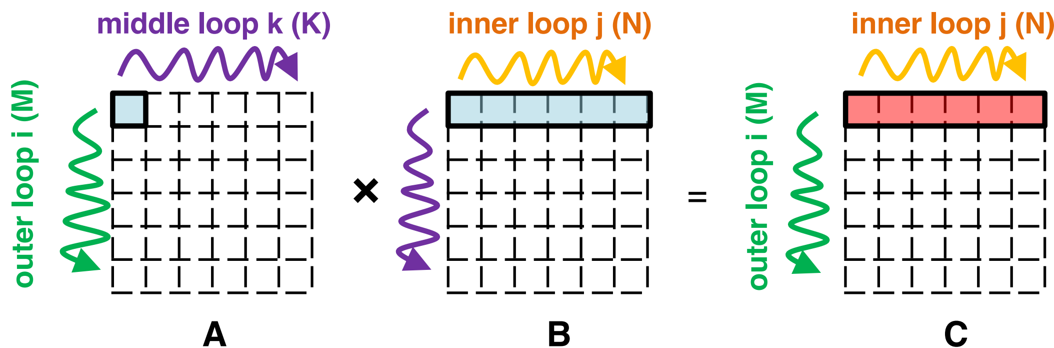
$$C[i,j] = A[i,:] \cdot B[:,j]$$

```
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    C[i, j] = 0;
    for (int k = 0; k < K; k++)
      #pragma pipeline II=??
      C[i, j] += A[i, k] * B[k, j]
```

Case Study: Optimized MM to Achieve II=1

- The row-wise product approach performs a sequence of scalar-vector products to produce *one output row*
  - An additional buffer is added to store the intermediate results (i.e., c_vec)

MatMul via row-wise product



$$C[i,:] = \sum_{k} A[i,k] \cdot B[k,:]$$

```
for (int i = 0; i < M; i++) {
  float C_vec[N];
  for (int j = 0; j < N; j++)
    C_vec[j] = 0.0;

  for (int k = 0; k < K; k++)
    for (int j = 0; j < N; j++)
      #pragma pipeline II=1
      C_vec[j] += A[i, k] * B[k, j];

  for (int j = 0; j < N; j++)
    C[i, j] = C_vec[j];
}
```

Case Study: Optimized MM in HeteroCL

- Optimizations via decoupled primitives
  - .buffer_at() creates an intermediate buffer at a given axis
  - .reorder() swaps the order of the k and j loops
  - Algorithm code stays unchanged

```
def MM_v2(M=1024, N=1024, K=512):
  hcl.init(hcl.Float())
  A = hcl.placeholder((M, K), name="A")
  B = hcl.placeholder((K, N), name="B")
  k = hcl.reduce_axis(0, K, name="k")
  C = hcl.compute((M, N), lambda i, j :
        hcl.sum(A[i, k] * B[k, j], axis=k), "C")
```

```
# customizations
s = hcl.create_schedule([A, B])
s.reorder(k, j)
s.buffer_at(C, i)
s.pipeline(j)
```

	II	**Latency (cycles)**	**Speedup**
Vanilla MM	8	4295M	1x
Optimized MM	**1**	539M	**7.97x**

SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs

Yi-Hsiang Lai[1], Hongbo Rong[2], Size Zheng[3], Weihao Zhang[4], Xiuping Cui[3], Yunshan Jia[3], Jie Wang[5], Brendan Sullivan[1], Zhiru Zhang[1], Yun Liang[3], Youhui Zhang[4], Jason Cong[5], Nithin George[2], Jose Alvarez[2], Christopher Hughes[2], Pradeep Dubey[2]

[1]Cornell University, [2]Intel, [3]Peking University, [4]Tsinghua University, [5]UCLA

ICCAD'2020

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

Systolic Arrays (for VLSI)

H. T. Kung† and Charles E. Leiserson†

And now I see with eye serene
The very pulse of the machine.
--William Wordsworth

Abstract

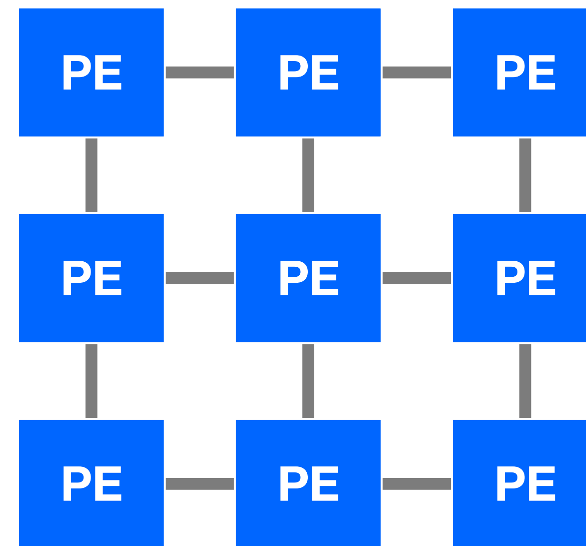
A systolic system is a network of processors which rhythmically compute and pass data through the system. Physiologists use the word "systole" to refer to the rhythmically recurrent contraction of the heart and arteries which pulses blood through the body. In a systolic computing system, the function of a processor is analogous to that of the heart. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network.

Many basic matrix computations can be pipelined elegantly and efficiently on systolic networks having an array structure. As an example, hexagonally connected processors can optimally perform matrix multiplication. Surprisingly, a similar systolic array can compute the LU-decomposition of a matrix. These systolic arrays enjoy simple and regular communication paths, and almost all processors used in the networks are identical. As a result, special purpose hardware devices based on systolic arrays can be built inexpensively using the VLSI technology.

1. Introduction

Developments in microelectronics have revolutionized computer design. Integrated circuit technology has increased the number and complexity of components that can fit on a chip or a printed circuit board. Component density has been doubling every one-to-two years and already, a multiplier can fit on a very large scale integrated

In Sparse Matrix Proceedings, 1978



parallel processing + pipelining

+ Simple & regular design
+ Massive parallelism
+ Short interconnection
+ Balancing compute with I/O

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

$$\mathbf{y} = \mathbf{A} * \mathbf{x}$$

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

Matrix Vector Multiplication (MV) in UREs

$$Z[i,j] = 0, when\ j = 0$$
$$Z[i,j] = Z[i,j-1] + A[i,j] \cdot x[j], when\ j > 0$$
$$y[i] = Z[i,N-1]$$

$$\mathbf{C} = \mathbf{A} * \mathbf{B}$$

```
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] * B[k, j]
```

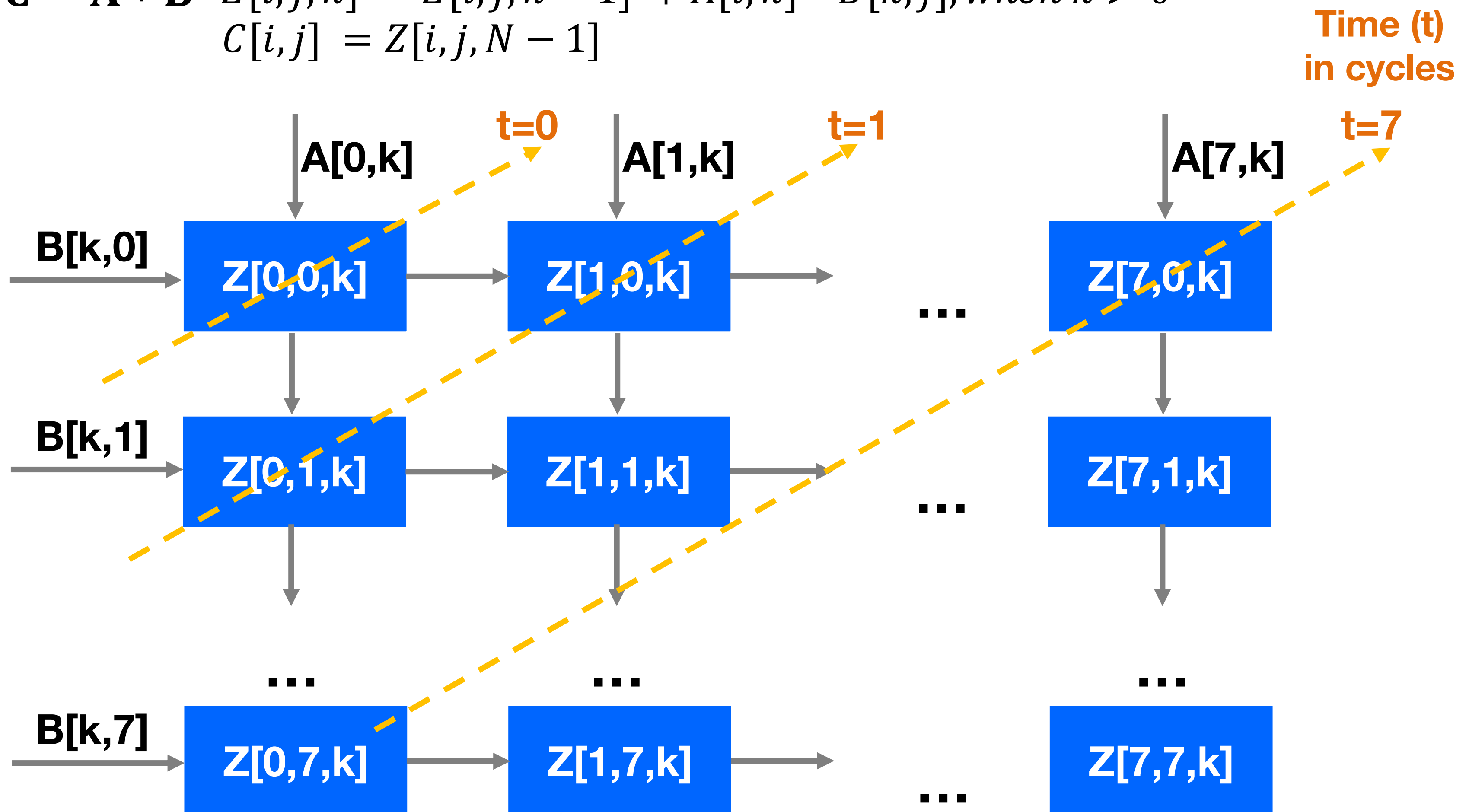
Matrix Matrix Multiplication (MM) in UREs

$$Z[i,j,k] = 0, when\ k = 0$$
$$Z[i,j,k] = Z[i,j,k-1] + A[i,k] \cdot B[k,j], when\ k > 0$$
$$C[i,j] = Z[i,j,N-1]$$

Mapping MM to a Systolic Array

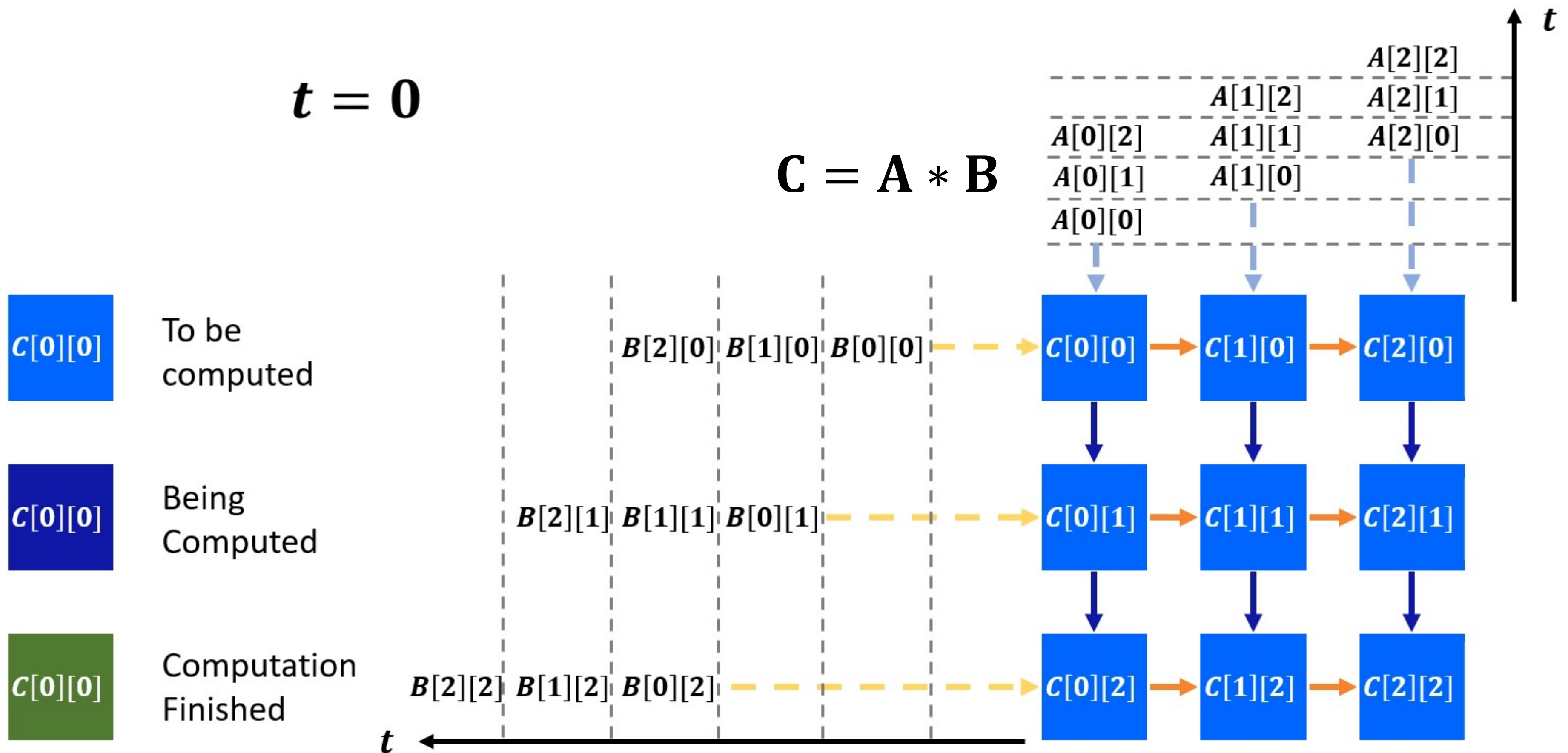
- Map the n-dimensional iteration space into a physical array of PEs

$$\mathbf{C} = \mathbf{A} * \mathbf{B} \quad \begin{aligned} Z[i,j,k] &= 0, when\ k = 0 \\ Z[i,j,k] &= Z[i,j,k-1] + A[i,k] \cdot B[k,j], when\ k > 0 \\ C[i,j] &= Z[i,j,N-1] \end{aligned}$$



MM Running on a Systolic Array

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



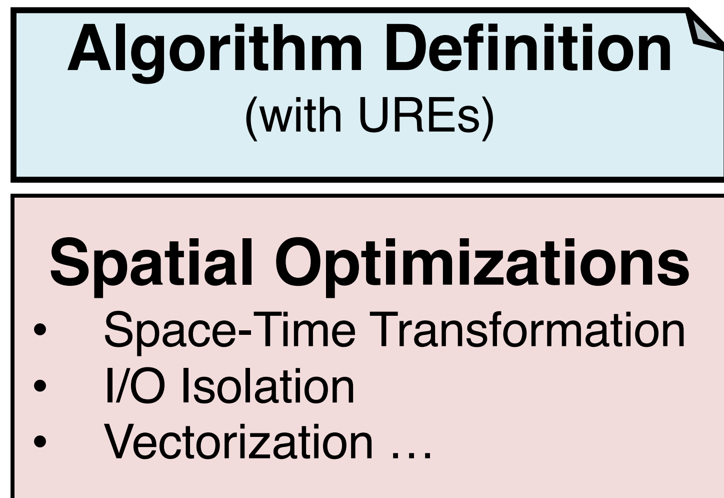
An eDSL for Constructing Systolic Arrays

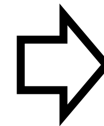
Decoupled algorithm definition and spatial optimizations

Explicitly represent optimizations such as **s**pace-time transformation

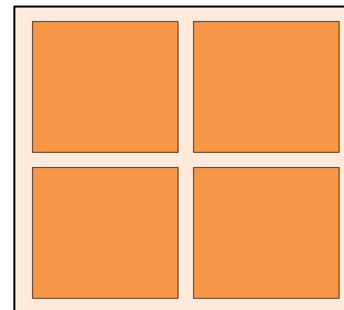
Concisely describe a systolic algorithm with uniform recurrence equations (**U**REs)

A programming model for accelerating systolic algorithms

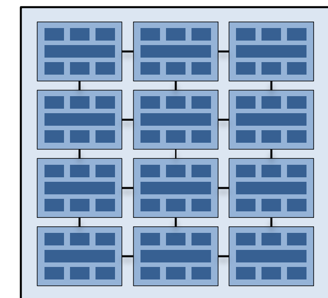




Processors + Accelerators



CPUs



FPGAs

Algorithm Specification with 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

$$\mathbf{C} = \mathbf{A} * \mathbf{B}$$

```
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] * B[k, j]
```

Matrix Matrix Multiplication (MM) in UREs

$$Z[i,j,k] = 0, when\ k = 0$$
$$Z[i,j,k] = Z[i,j,k-1] + A[i,k] \cdot B[k,j], when\ k > 0$$
$$C[i,j] = Z[i,j,N-1]$$

Algorithm Definition in SuSy

```
// Iteration space
Var i, j, k;
// UREs
Z(i, j, k) = select(k==0, 0, Z(i, j, k-1)) + A(i, j, k) * B(i, j, k);
C(i, j) = select(k == N-1, Z(i, j, k));
```

Declarative Programming (builds on Halide)

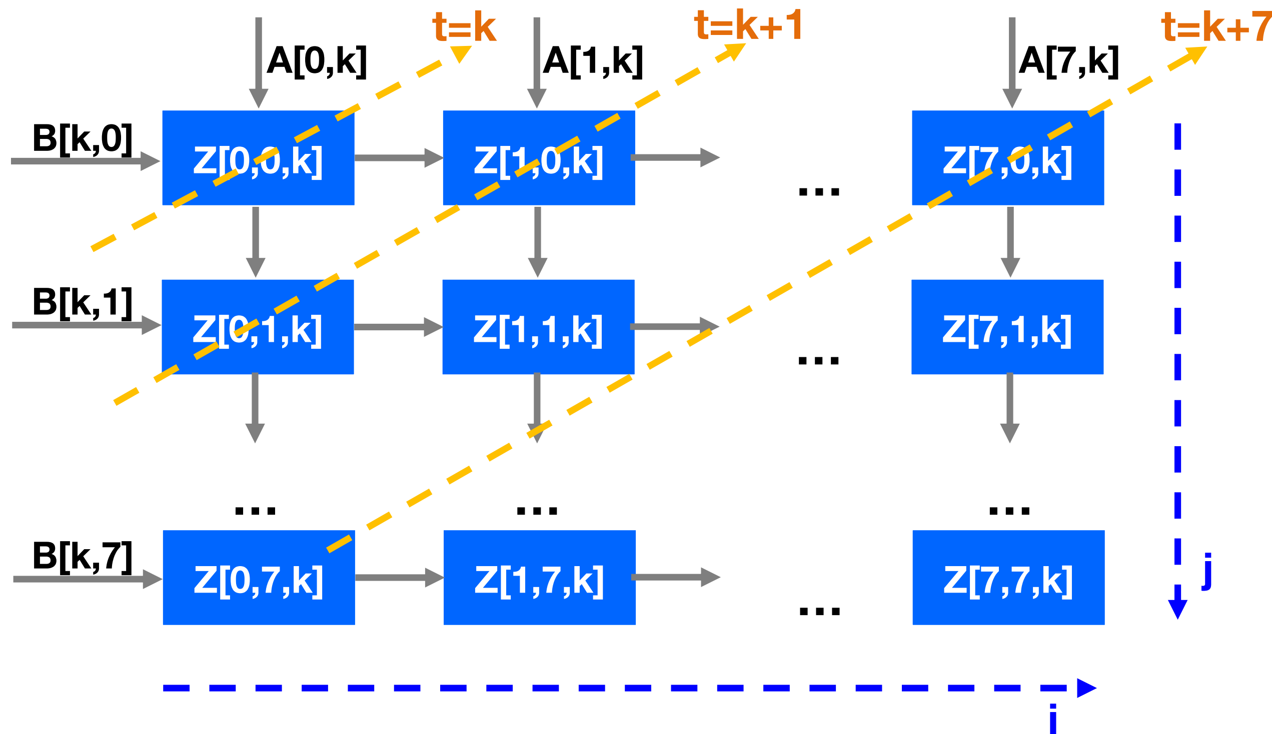
Space-Time Transformation

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \hdashline 1 & 1 & 1 \end{pmatrix} \begin{matrix} \Pi \\ \\ \tau \end{matrix}$$

Transformation Matrix

Π **Space dimensions** $\Pi \times (i, j, k)^T = (i, j)$

τ **Time schedule** $\tau \times (i, j, k)^T = i + j + k$



Supported Spatial Optimizations

Optimizations for custom I/O

F.merge_ures(U_1, U_2, ..., U_n)	Define the set of UREs F, U_1, U_2, ..., U_n to optimize.
F.space_time_transform(space, tau)	Specify the space-time transformation that will be applied to F, where space is the set of space loops, and tau is the scheduling vector.
F.vectorize(var)	Vectorize the specified loop variable var of F.
F.reorder(var_1, var_2, ..., var_n)	Reorder the loop nest for F according to the specified order, starting from the innermost level.
F.isolate_producer({E_1, E_2, ...}, P)	Isolate a list of expressions {E_1, E_2, ...} (usually inputs) in F to a separate producer kernel P.
F.isolate_consumer(E, C)	Isolate an expression E (usually an output) in F to a separate consumer kernel C.
F.remove(var)	Remove loop var of F.
F.buffer(E, v, mode)	Insert a reuse buffer at loop v for expression E with mode (either Buffer::Single or Buffer::Double).
F.scatter(E, var)	Reduce data communication overhead (i.e., data broadcast) by scattering the expression E to the consumer along loop var.
F.gather(E, var)	Reduce data communication overhead (i.e., data broadcast) by gathering the expression E from the producer along loop var.

https://github.com/IntelLabs/t2sp

Acknowledgements

- This lecture contains/adapts materials developed by
  - Yi-Hsiang Lai (Cornell ECE PhD, now AWS AI)
  - Authors of the following papers
    - HeteroCL: A Multi-Paradigm Programming Infrastructure for Sofware-Defined Reconfigurable Computing (FPGA'19)
    - SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs (ICCAD'20)