T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Tensor Computations

Nitish Srivastava[1], Hongbo Rong[2], Prithayan Barua[3], Guanyu Feng[4], Huanqi Cao[3], Zhiru Zhang[1], David Albonesi[1], Vivek Sarkar[3], Wenguang Chen[3], Paul Petersen[2], Geoff Lowney[2], Adam Herr[2], Christopher Hughes[2], Timothy Mattson[2], Pradeep Dubey[2]

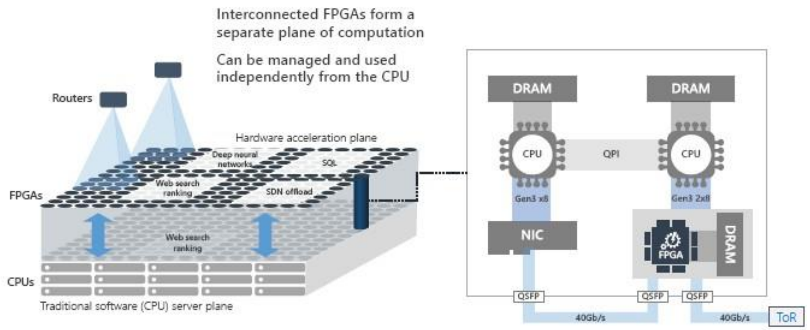
[1]Cornell University
[2]Intel Parallel Computing Labs
[3]Georgia Institute of Technology
[4]Tsinghua University

Tensor Computations

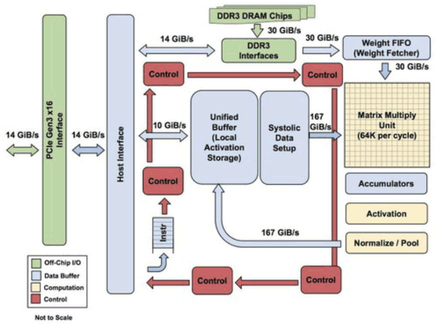
- Tensor computations are everywhere
  - Data analytics (e.g. graphs in social interactions)
  - Machine learning (e.g. CNN, recommender systems)
  - Scientific computing (e.g. finite element method)

- Increasing popularity of deploying tensor applications to spatial architectures

FPGAs



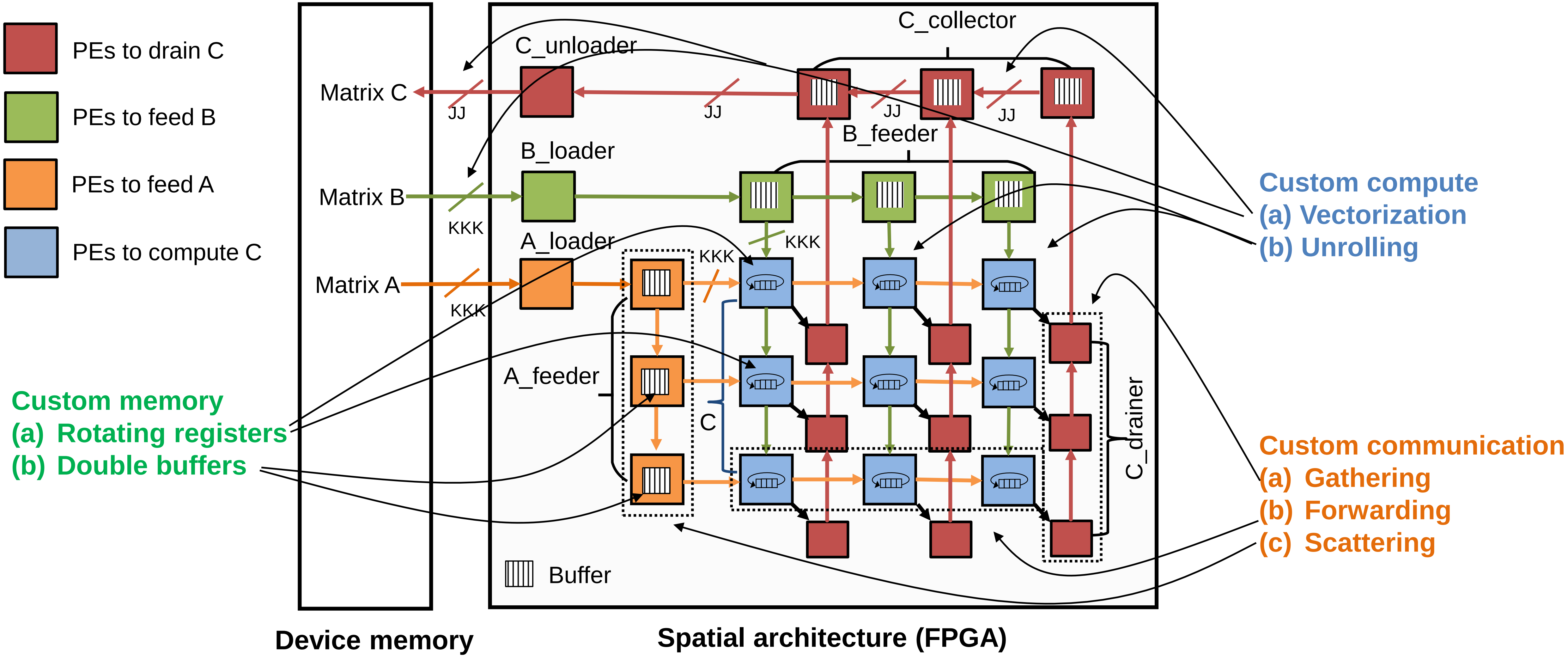
Microsoft Brainwave



ASICs (Google TPU)

- There exists a gap on how to map these applications on different spatial architectures

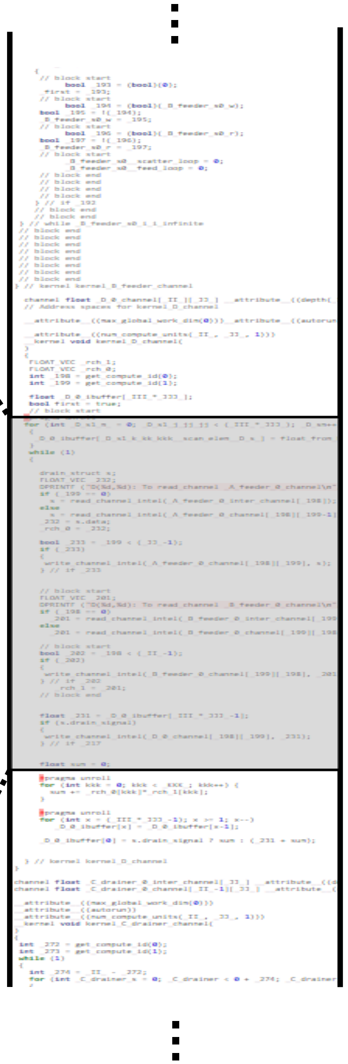
Driving Example – Matrix Multiplication (GEMM)



High-performance GEMM design on FPGA

High-Performance GEMM Systolic Array in HLS

```
unrolled for(int ii = 0; ii < II; ii++) {
unrolled for(int jj = 0; jj < JJ; jj++) {
 for(int i = 0; i < I/II*III; i++) {
   for(int j = 0; j < J/JJ*JJJ; j++) {
     for(int k = 0; k < K/KK; k++) {
      float buffer[IIII][JJJ]
    for(int iii = 0; iii < III; iii++) {
       for(int jjj = 0; jjj < JJJ; jjj++) {
         8xfloat a = RCH (chA[ii][jj])
         WCH (chA[ii+1][jj], a)
         8xfloat b = RCH (chB[ii][jj])
         WCH (chB[ii][jj+1], b)
         if (drain)
           WCH (chC[ii][jj], buf[iii][jjj])
         #pragma unroll
         for(int kkk = 0; kkk < KKK; kkk++)
            sum += a[kkk]*b[kkk]
         buffer[iii][jjj] += sum;
}}}}}}}
```



750 lines

of high-performance HLS code

High-Performance GEMM Systolic Array in HLS

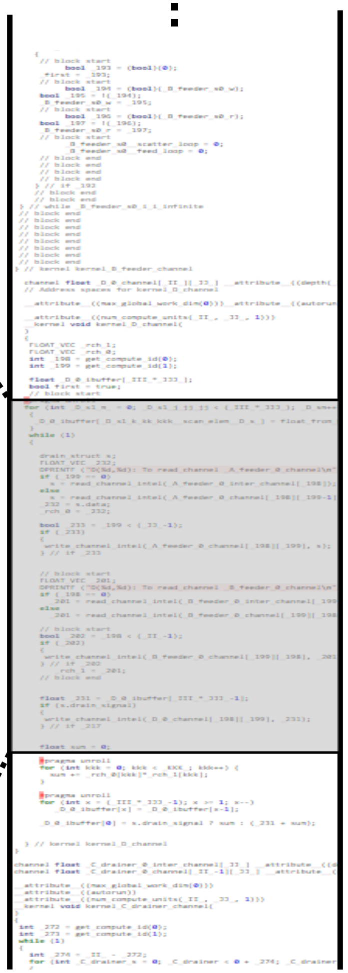
```
unrolled for(int ii = 0; ii < II; ii++) {
unrolled for(int jj = 0; jj < JJ; jj++) {
 for(int i = 0; i < I/II*III; i++) {
   for(int j = 0; j < J/JJ*JJJ; j++) {
     for(int k = 0; k < K/KK; k++) {
       float buffer[III][JJJ]
       for(int iii = 0; iii < III; iii++) {
         for(int jjj = 0; jjj < JJJ; jjj++) {
           8xfloat a = RCH (chA[ii][jj])
           WCH (chA[ii+1][jj], a)
           8xfloat b = RCH (chB[ii][jj])
           WCH (chB[ii][jj+1], b)
           if (drain)
             WCH (chC[ii][jj], buf[iii][jjj])
           #pragma unroll
           for(int kkk = 0; kkk < KKK; kkk++)
             sum += a[kkk]*b[kkk]
           buffer[iii][jjj] += sum;
}}}}}}}
```

Custom compute (Loop unrolling)

Custom compute (Loop tiling)

Custom compute (Vectorization)

⋮



⋮

750 lines
of high-performance HLS code

High-Performance GEMM Systolic Array in HLS

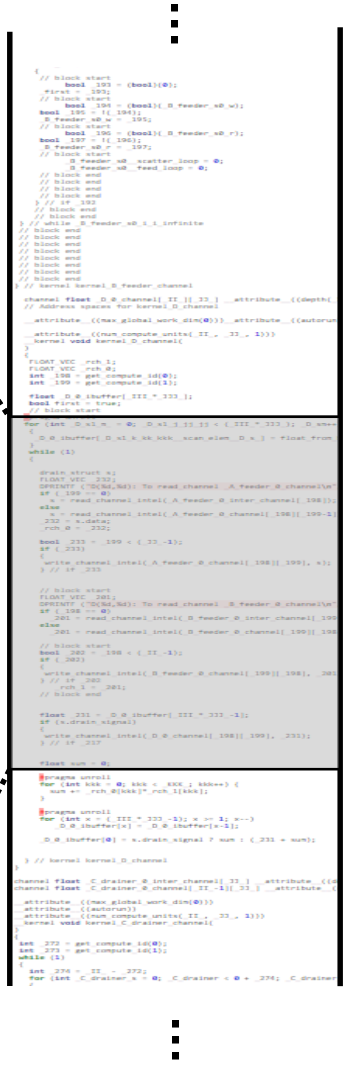
```
unrolled for(int ii = 0; ii < II; ii++) {
unrolled for(int jj = 0; jj < JJ; jj++) {
  for(int i = 0; i < I/II*III; i++) {
    for(int j = 0; j < J/JJ*JJJ; j++) {
      for(int k = 0; k < K/KK; k++) {
        float buffer[III][JJJ]
        for(int iii = 0; iii < III; iii++) {
          for(int jjj = 0; jjj < JJJ; jjj++) {
            8xfloat a = RCH (chA[ii][jj])
            WCH (chA[ii+1][jj], a)
            8xfloat b = RCH (chB[ii][jj])
            WCH (chB[ii][jj+1], b)
            if (drain)
              WCH (chC[ii][jj], buf[iii][jjj])
            #pragma unroll
            for(int kkk = 0; kkk < KKK; kkk++)
              sum += a[kkk]*b[kkk]
            buffer[iii][jjj] += sum;
}}}}}}}
```

Custom compute (Loop unrolling)

Custom compute (Loop tiling)

Custom memory (Buffer)

Custom compute (Vectorization)



750 lines
of high-performance HLS code

High-Performance GEMM Systolic Array in HLS

```
unrolled for(int ii = 0; ii < II; ii++) {
unrolled for(int jj = 0; jj < JJ; jj++) {
 for(int i = 0; i < I/II*III; i++) {
   for(int j = 0; j < J/JJ*JJJ; j++) {
     for(int k = 0; k < K/KK; k++) {
       float buffer[III][JJJ]
      for(int iii = 0; iii < III; iii++) {
         for(int jjj = 0; jjj < JJJ; jjj++) {
           8xfloat a = RCH (chA[ii][jj])
           WCH (chA[ii+1][jj], a)
           8xfloat b = RCH (chB[ii][jj])
           WCH (chB[ii][jj+1], b)
           if (drain)
               WCH (chC[ii][jj], buf[iii][jjj])
            #pragma unroll
            for(int kkk = 0; kkk < KKK; kkk++)
               sum += a[kkk]*b[kkk]
            buffer[iii][jjj] += sum;
}}}}}}}
```

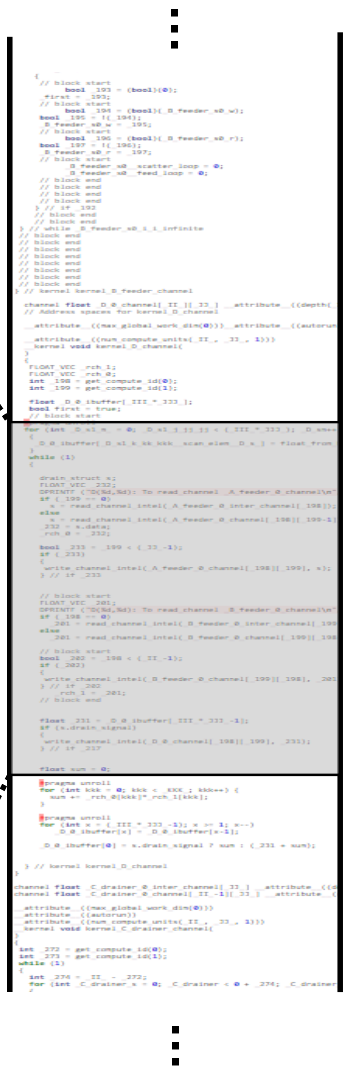
Custom compute (Loop unrolling)

Custom compute (Loop tiling)

Custom comm. (Forwarding)

Custom memory (Buffer)

Custom compute (Vectorization)

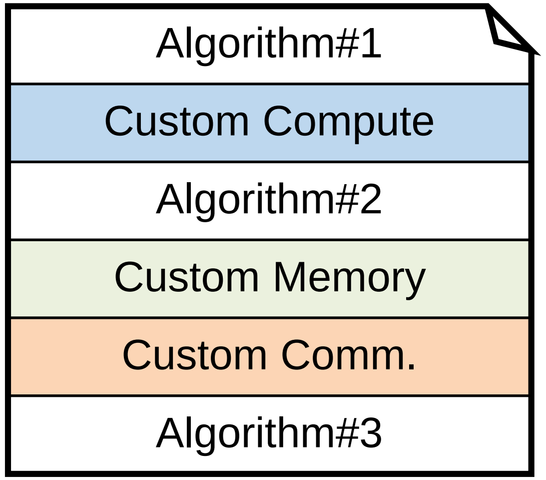


750 lines
of high-performance HLS code

Entangled Hardware Customization and Algorithm
Less Portable, Less Maintainable and Less Productive

Decoupling Hardware Customization and Algorithm

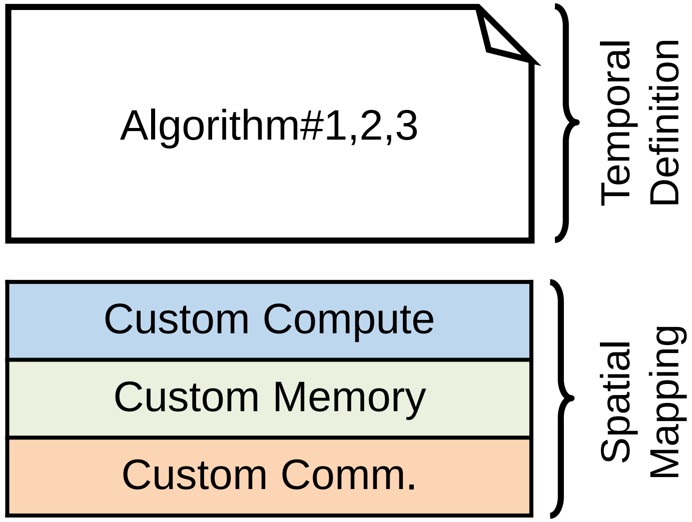
HLS C



Entangled algorithm specification & customization schemes [1,2,3]

[1] Intel HLS
[2] Xilinx Vivado HLS
[3] Canis, et al. FPGA'11

T2S



Decoupled customization & clean abstraction

```
C(i, j) = 0;
C(i, j) += A(i, k) * B(k, j);
C.update().tile(k, j, i, kk, jj, ii, KK, JJ, II);
            .isolate_producer_chain(A, A_loader, A_feeder)
            .isolate_producer_chain(B, B_loader, B_feeder)
            .isolate_consumer_chain(C, C_drainer, C_unloader);
A_loader.unroll(ii).remove(jj).vload(kk);
A_feeder.buffer(ii, Buffer::Double).unroll(ii);
B_loader.unroll(jj).remove(ii).vload(kk);
B_feeder.buffer(k, Buffer::Double).unroll(jj);
C.update().unroll(jj, ii)
           .forward(A_feeder, {1, 0}) .forward(B_feeder, {0, 1});
C_drainer.unroll(jj, ii).gather(C, {1, 0})
C_unloader.buffer(ii).unroll(ii).vstore(jj);
```

Temporal to Spatial ⟶ T2S

T2S is an extension over Halide for spatial architectures

Temporal Definition in T2S

```
Func C
C(i, j) = 0
C(i, j) += A(i, k) * B(k, j)
C.tile(i,j,k,ii,jj,kk,II,JJ,KK)
```

Algorithm

C

```
for i, j, k
   for ii, jj, kk
      i’ = i*II + ii
      j’ = j*JJ + jj
      k’ = k*KK + kk
      C[i’, j’] += A[i’,k’] * B[k’,j’]
```

C

Spatial Mapping in T2S

Func C	
C(i, j) = 0	Algorithm
C(i, j) += A(i, k) * B(k, j)	
C.tile(i,j,k,ii,jj,kk,II,JJ,KK)	
C.isolate_producer(A, A_feeder)	Spatial Mapping

C

```
for i, j, k
  for ii, jj, kk
    i',j',k' = …
    C[i', j'] += A[i',k'] * B[k',j']
```

Spatial Mapping in T2S

```
Func C
C(i, j) = 0
C(i, j) += A(i, k) * B(k, j)
C.tile(i,j,k,ii,jj,kk,II,JJ,KK)
C.isolate_producer(A, A_feeder)
```

Algorithm

Spatial Mapping

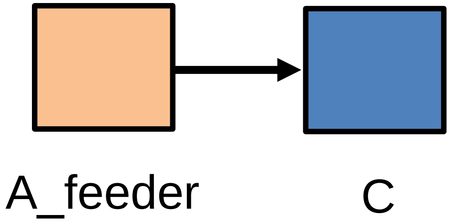
A_feeder

```
for i, j, k
  for ii, jj, kk
    i',j',k' = …
    WCH (ch1, A[i',k'])
```

C

```
for i, j, k
  for ii, jj, kk
    i',j',k' = …
    C[i', j'] +=          * B[k',j']
```

channel1 (ch1)



Spatial Mapping in T2S

```
Func C
C(i, j) = 0
C(i, j) += A(i, k) * B(k, j)
C.tile(i,j,k,ii,jj,kk,II,JJ,KK)
C.isolate_producer(A, A_feeder)
C.unroll(ii, jj)
```

Algorithm

Spatial Mapping

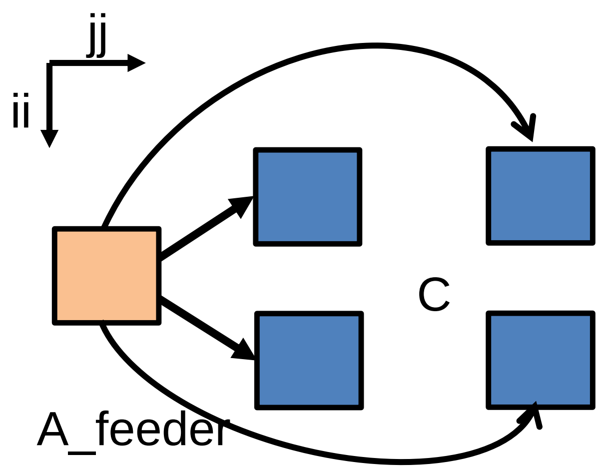
A_feeder

```
for i, j, k
   for ii, jj, kk
      i',j',k' = …
      WCH (ch1[ii][jj], A[i',k'])
```

C

```
unrolled for ii, jj
for i, j, k
   for kk
      i',j',k' = …
      C[i', j'] =             * B[k',j']
```

ch1[ii][jj]



Spatial Mapping in T2S

```
Func C
C(i, j) = 0
C(i, j) += A(i, k) * B(k, j)
C.tile(i,j,k,ii,jj,kk,II,JJ,KK)
C.isolate_producer(A, A_feeder)
C.unroll(ii, jj)
A_feeder.isolate_producer(A, A_loader)
```

Algorithm

Spatial Mapping

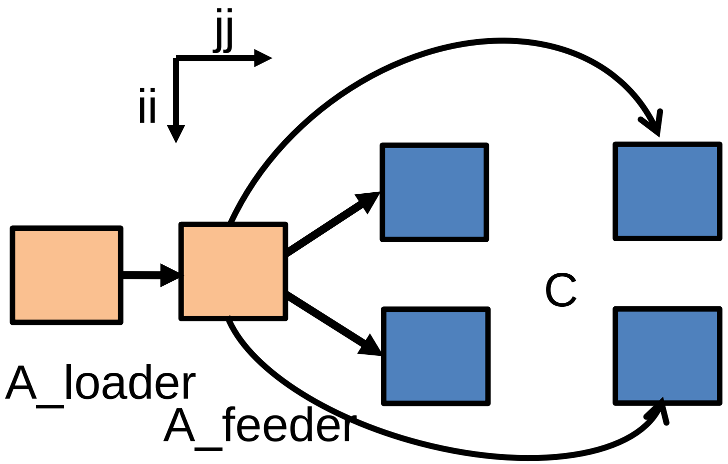
A_loader

```
for i, j, k
  for ii, jj, kk
    i',j',k' = …
    WCH (ch2, A[i',k'])
```

A_feeder

```
for i, j, k
  for ii, jj, kk
    i',j',k' = …
    WCH (ch1[ii][jj],        )
```

channel2 (ch2)



Spatial Mapping in T2S

Func C

C(i, j) = 0

C(i, j) += A(i, k) * B(k, j)

C.tile(i,j,k,ii,jj,kk,II,JJ,KK)

Algorithm

C.isolate_producer(A, A_feeder)

C.unroll(ii, jj)

A_feeder.isolate_producer(A, A_loader)

A_loader.remove(jj)

A_feeder.buffer(ii, DOUBLE)

Spatial Mapping

A_loader

for i, j, k

  for ii, ~~jj,~~ kk

    i’,j’,k’ = …

    WCH (ch2, A[i’,k’])

A_feeder

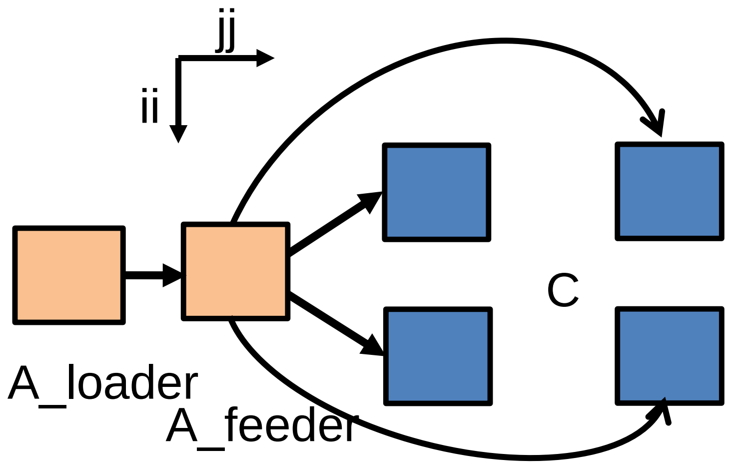
for i, j, k

  for ii, jj, kk

    i’,j’,k’ = …

    WCH (ch1[ii][jj],   )

channel2 (ch2)



Spatial Mapping in T2S

Algorithm

```
Func C
C(i, j) = 0
C(i, j) += A(i, k) * B(k, j)
C.tile(i,j,k,ii,jj,kk,II,JJ,KK)
```

Spatial Mapping

```
C.isolate_producer(A, A_feeder)
C.unroll(ii, jj)
A_feeder.isolate_producer(A, A_loader)
A_loader.remove(jj)
A_feeder.buffer(ii, DOUBLE)
```

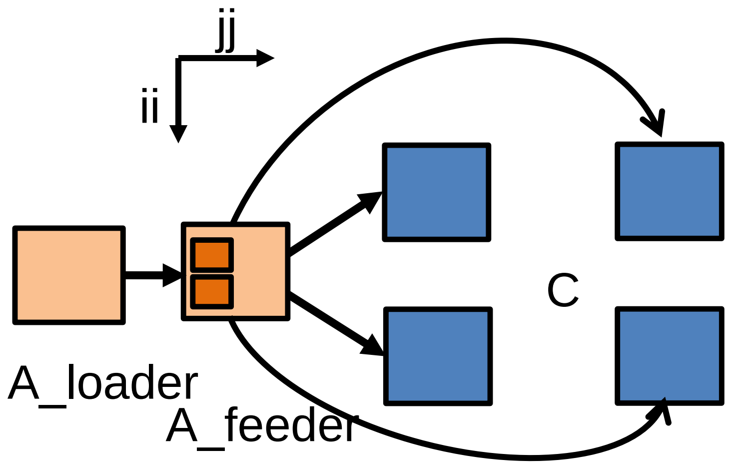
A_feeder

```
for i, j, k, ii
  float buf [KK]
  for kk = 0 .. KK
    buf [kk] = RCH(...)
  for jj, kk
    i',j',k' = …
    WCH (ch1[ii][jj], buf[kk])
```

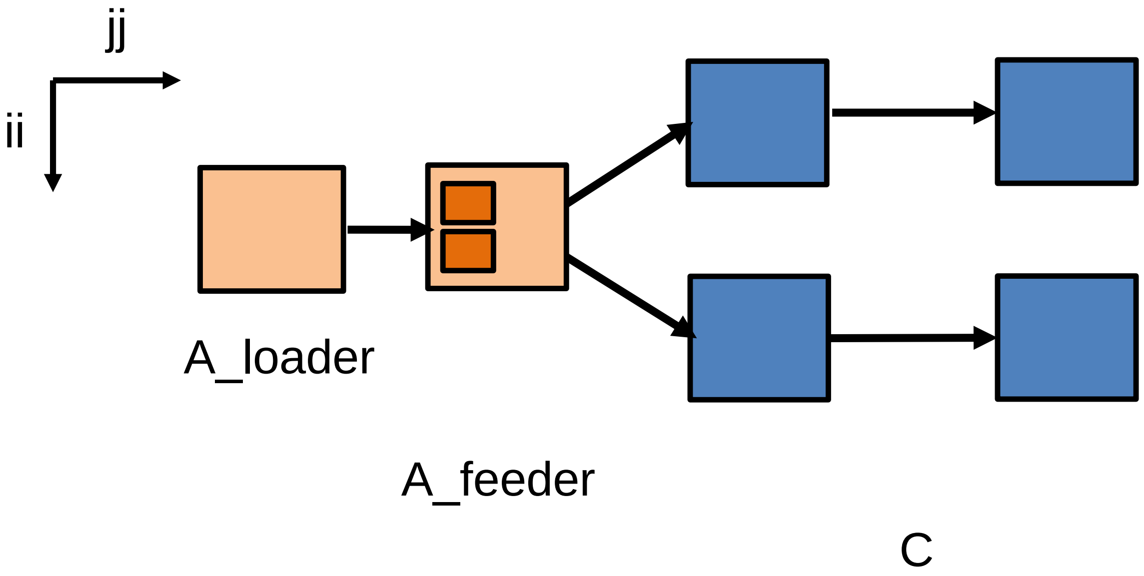
A_loader

```
for i, j, k
  for ii, jj, kk
    i',j',k' = …
    WCH (ch2, A[i',k'])
```

ch2



Spatial Mapping in T2S



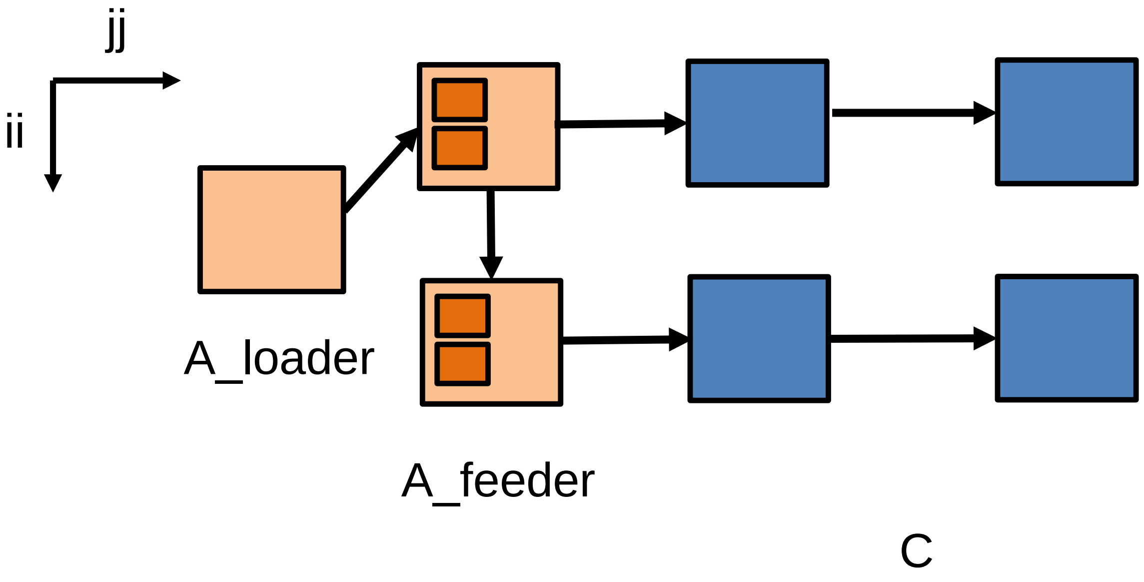
Algorithm

```
Func C
C(i, j) = 0
C(i, j) += A(i, k) * B(k, j)
C.tile(i,j,k,ii,jj,kk,II,JJ,KK)
```

Spatial Mapping

```
C.isolate_producer(A, A_feeder)
C.unroll(ii, jj)
A_feeder.isolate_producer(A, A_loader)
A_loader.remove(jj)
A_feeder.buffer(ii, DOUBLE)
C.forward(A_feeder, +jj)
```

Spatial Mapping in T2S



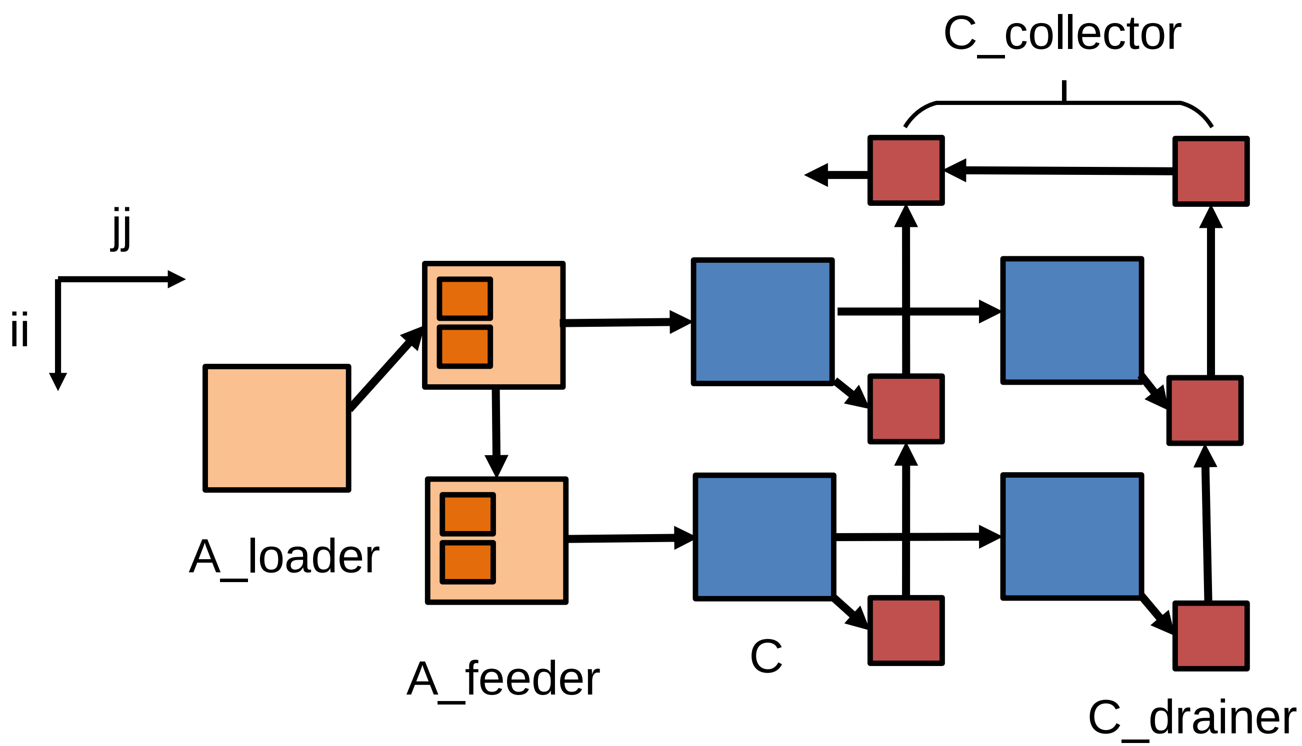
Algorithm

```
Func C
C(i, j) = 0
C(i, j) += A(i, k) * B(k, j)
C.tile(i,j,k,ii,jj,kk,II,JJ,KK)
```

Spatial Mapping

```
C.isolate_producer(A, A_feeder)
C.unroll(ii, jj)
A_feeder.isolate_producer(A, A_loader)
A_loader.remove(jj)
A_feeder.buffer(ii, DOUBLE)
C.forward(A_feeder, +jj)
A_feeder.unroll(ii).scatter(A, +ii)
```

Spatial Mapping in T2S



Algorithm

Func C

C(i, j) = 0

C(i, j) += A(i, k) * B(k, j)

C.tile(i,j,k,ii,jj,kk,II,JJ,KK)

Spatial Mapping

C.isolate_producer(A, A_feeder)

C.unroll(ii, jj)

A_feeder.isolate_producer(A, A_loader)

A_loader.remove(jj)
A_feeder.buffer(ii, DOUBLE)

C.forward(A_feeder, +jj)

A_feeder.unroll(ii).scatter(A, +ii)

C.isolate_consumer(C, C_drainer)

C_drainer.isolate_consumer_chain(C, C_collector, C_unloader)

C_drainer.unroll(ii).unroll(jj).gather(C, -ii)
C_collector.unroll(jj).gather(C, -jj)

T2S Code

~20 LOC vs 750 lines of HLS code

```
C(j, i) = 0.0f;
C(j, i) += A(k, i) * B(j, k);
C.tile(j, i, jj, ii, JJ, II).tile(jj, ii, jjj, iii, JJJ, III);
C.update(0).tile(k, j, i, kk, jj, ii, KK, JJ, II).tile(kk, jj, ii, kkk, jjj, iii, KKK, JJJ, III);
C.update(0).isolate_producer_chain(A, A_serializer, A_loader, A_feeder)
           .isolate_producer_chain(B, B_serializer, B_loader, B_feeder)
           .isolate_consumer_chain(C, C_drainer, C_collector, C_unloader, C_deserializer);
A_serializer.sread().swrite();
B_serializer.sread().swrite();
C.update(0).vread({A,B});
C.update(0).unroll(ii)
           .unroll(jj)
C.update(0).forward(A_feeder, { 0, 1 })
           .forward(B_feeder, { 1, 0 });
A_serializer.remove(jjj, jj, j);
A_loader.remove(jjj, jj);
A_feeder.buffer(ii, true).unroll(ii);
B_serializer.remove(iii, ii, i);
B_loader.remove(iii, ii);
B_feeder.buffer(k, true).unroll(jj);
A_feeder.scatter(A, {1});
B_feeder.scatter(B, {1});
C_drainer.unroll(ii).unroll(jj).gather(C, jjj, { 1, 0 });
C_collector.unroll(jj).gather(C_drainer, jjj, { 1 });
```

T2S Code

~20 LOC vs 750 lines of HLS code

```
C(j, i) = 0.0f;
C(j, i) += A(k, i) * B(j, k);
C.tile(j, i, jj, ii, JJ, II).tile(jj, ii, jjj, iii, JJJ, III);
C.update(0).tile(k, j, i, kk, jj, ii, KK, JJ, II).tile(kk, jj, ii, kkk, jjj, iii, KKK, JJJ, III);
C.update(0).isolate_producer_chain(A, A_serializer, A_loader, A_feeder)
           .isolate_producer_chain(B, B_serializer, B_loader, B_feeder)
           .isolate_consumer_chain(C, C_drainer, C_collector, C_unloader, C_deserializer);
A_serializer.sread().swrite();
B_serializer.sread().swrite();
C.update(0).vread({A,B});
C.update(0).unroll(ii)
           .unroll(jj)
C.update(0).forward(A_feeder, { 0, 1 })
           .forward(B_feeder, { 1, 0 });
A_serializer.remove(jjj, jj, j);
A_loader.remove(jjj, jj);
A_feeder.buffer(ii, true).unroll(ii);
B_serializer.remove(iii, ii, i);
B_loader.remove(iii, ii);
B_feeder.buffer(k, true).unroll(jj);
A_feeder.scatter(A, {1});
B_feeder.scatter(B, {1});
C_drainer.unroll(ii).unroll(jj).gather(C, jjj, { 1, 0 });
C_collector.unroll(jj).gather(C_drainer, jjj, { 1 });
```

Custom compute (Loop Tiling)

Custom compute (Compute Partitioning)

Custom compute (Data Vectorization)

Custom compute (Loop Unrolling)

T2S Code

~20 LOC vs 750 lines of HLS code

```
C(j, i) = 0.0f;
C(j, i) += A(k, i) * B(j, k);
C.tile(j, i, jj, ii, JJ, II).tile(jj, ii, jjj, iii, JJJ, III);
C.update(0).tile(k, j, i, kk, jj, ii, KK, JJ, II).tile(kk, jj, ii, kkk, jjj, iii, KKK, JJJ, III);
C.update(0).isolate_producer_chain(A, A_serializer, A_loader, A_feeder)
           .isolate_producer_chain(B, B_serializer, B_loader, B_feeder)
           .isolate_consumer_chain(C, C_drainer, C_collector, C_unloader, C_deserializer);
A_serializer.sread().swrite();
B_serializer.sread().swrite();
C.update(0).vread({A,B});
C.update(0).unroll(ii)
           .unroll(jj)
C.update(0).forward(A_feeder, { 0, 1 })
           .forward(B_feeder, { 1, 0 });
A_serializer.remove(jjj, jj, j);
A_loader.remove(jjj, jj);
A_feeder.buffer(ii, true).unroll(ii);
B_serializer.remove(iii, ii, i);
B_loader.remove(iii, ii);
B_feeder.buffer(k, true).unroll(jj);
A_feeder.scatter(A, {1});
B_feeder.scatter(B, {1});
C_drainer.unroll(ii).unroll(jj).gather(C, jjj, { 1, 0 });
C_collector.unroll(jj).gather(C_drainer, jjj, { 1 });
```

Custom compute (Loop Tiling)

Custom compute (Compute Partitioning)

Custom compute (Data Vectorization)

Custom compute (Loop Unrolling)

Custom memory (Buffer Insertion)

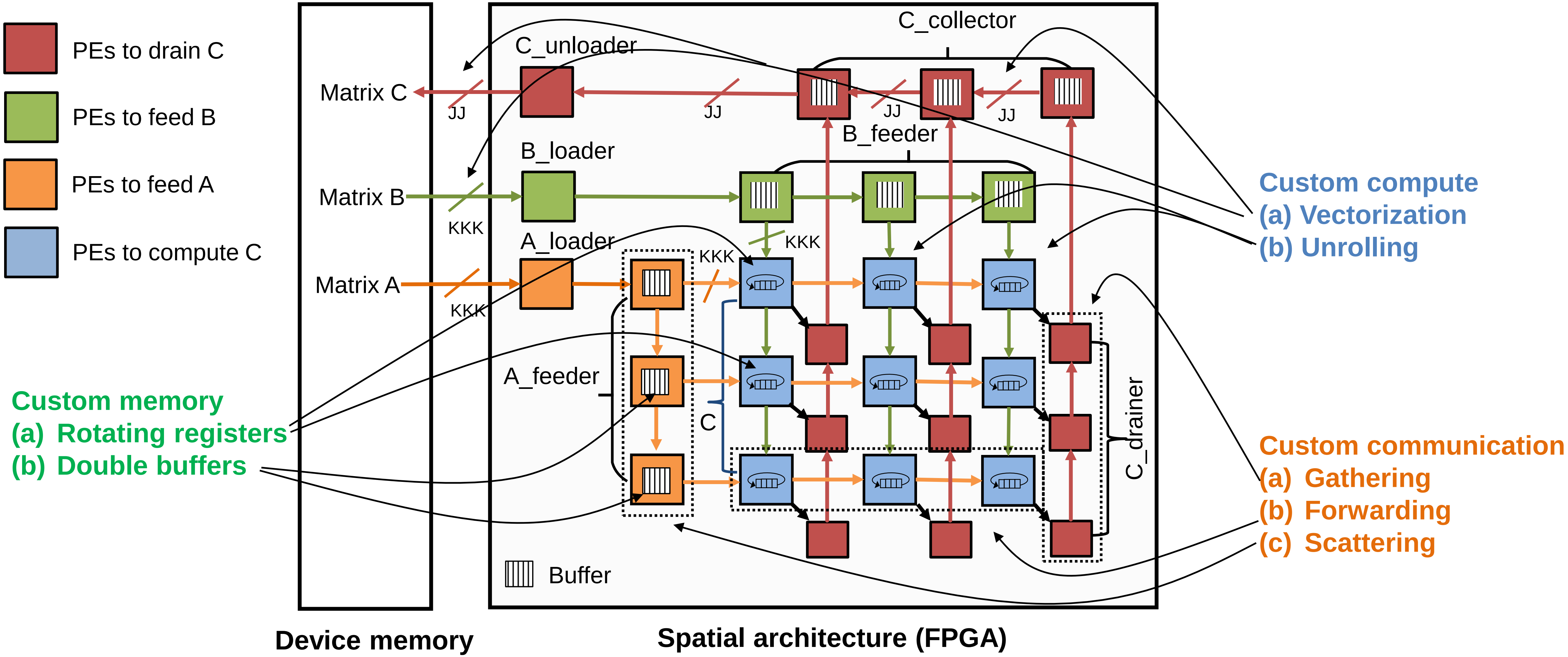
T2S Code

~20 LOC vs 750 lines of HLS code

```
C(j, i) = 0.0f;
C(j, i) += A(k, i) * B(j, k);
C.tile(j, i, jj, ii, JJ, II).tile(jj, ii, jjj, iii, JJJ, III);
C.update(0).tile(k, j, i, kk, jj, ii, KK, JJ, II).tile(kk, jj, ii, kkk, jjj, iii, KKK, JJJ, III);
C.update(0).isolate_producer_chain(A, A_serializer, A_loader, A_feeder)
           .isolate_producer_chain(B, B_serializer, B_loader, B_feeder)
           .isolate_consumer_chain(C, C_drainer, C_collector, C_unloader, C_deserializer);
A_serializer.sread().swrite();
B_serializer.sread().swrite();
C.update(0).vread({A,B});
C.update(0).unroll(ii)
           .unroll(jj)
C.update(0).forward(A_feeder, { 0, 1 })
           .forward(B_feeder, { 1, 0 });
A_serializer.remove(jjj, jj, j);
A_loader.remove(jjj, jj);
A_feeder.buffer(ii, true).unroll(ii);
B_serializer.remove(iii, ii, i);
B_loader.remove(iii, ii);
B_feeder.buffer(k, true).unroll(jj);
A_feeder.scatter(A, {1});
B_feeder.scatter(B, {1});
C_drainer.unroll(ii).unroll(jj).gather(C, jjj, { 1, 0 });
C_collector.unroll(jj).gather(C_drainer, jjj, { 1 });
```

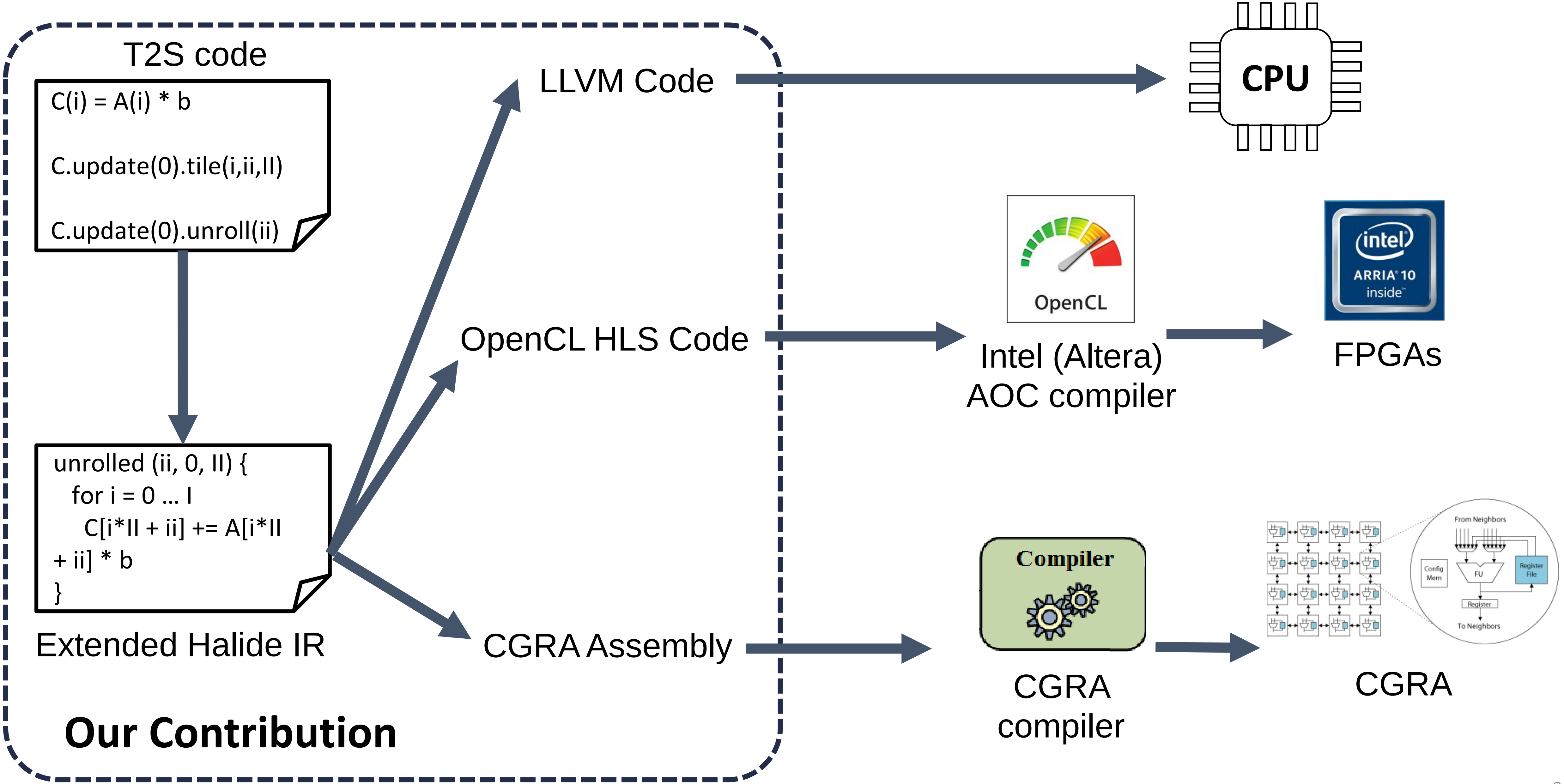
Code lines	Category
C.tile … / C.update(0).tile …	**Custom compute (Loop Tiling)**
isolate_producer_chain / isolate_consumer_chain	**Custom compute (Compute Partitioning)**
sread().swrite() / vread({A,B})	**Custom compute (Data Vectorization)**
unroll(ii) / unroll(jj)	**Custom compute (Loop Unrolling)**
forward(…)	**Custom comm. (Data Forwarding)**
remove(…) / buffer(…)	**Custom memory (Buffer Insertion)**
scatter(…)	**Custom comm. (Data Scattering)**
gather(…)	**Custom comm. (Data Gathering)**

Driving Example – Matrix Multiplication (GEMM)



High-performance GEMM design on FPGA

T2S Compilation Flow



GEMM on Arria 10

- Baseline
  - Open-source NDRange-style OpenCL code, tuned on the specific FPGA
- Ninja
  - handwritten and manually optimized design from industry

	Baseline	T2S	Ninja
LOC	70	**20**	750
Systolic array size	--	10 x 8	10 x 8
Vector length	16 x float	16 x float	16 x float
# Logic elements	131 K (31%)	214 K (50%)	230 K (54%)
# DSPs	1,032 (68%)	**1,282 (84%)**	1,280 (84%)
# RAMs	1,534 (57%)	1,384 (51%)	1,069 (39%)
Frequency (MHz)	189	215	245
Throughput (GFLOPS)	311	**549**	626

1.8x speedup over the baseline with 3.5x less code
82% performance of ninja implementation with 3% code

Tensor Decomposition Kernels on FPGA & CGRA

- Tensor decomposition kernels
  - $\boldsymbol{MTTKRP}$: $D(i,j) \mathrel{+}= A(i,k,l) * B(k,j) * C(l,j)$
  - $\boldsymbol{TTM}$: $C(i,j,k) \mathrel{+}= A(i,j,l) * B(l,k)$
  - $\boldsymbol{TTMc}$: $D(i,j,k) \mathrel{+}= A(i,l,m) * B(l,j) * C(m,k)$

Evaluation on CGRA

	LOC	Throughput wrt. Ninja GEMM	FMA Usage
GEMM	40	92 %	**100 %**
MTTKRP	32	99 %	**100 %**
TTM	47	104 %	**100 %**
TTMc	38	103 %	**95 %**

Evaluation on Arria-10 FPGA

Benchmark	LOC	Systolic Array Size	Logic Usage	DSP Usage	RAM Usage	Frequency (MHz)	Throughput (GFLOPS)
MTTKRP	28	8 x 9	53 %	81 %	56 %	204	**700**
TTM	30	8 x 11	64 %	93 %	88 %	201	**562**
TTMc	37	8 x 10	54 %	90 %	62 %	205	**738**

~100 % FMA usage and ~100 % throughput compared to ninja GEMM for CGRA
~80-90 % DSP utilization and 560-740 GFLOPS for FPGA

Conclusions

- T2S (Temporal to Spatial)
  - Provides a concise yet expressive programming abstraction that decouples spatial mapping from temporal definition
- Identifies a set of key compiler optimizations
  - essential for creating high-performance spatial hardware for tensor kernels
- Demonstrated high-performance (close to ninja) for
  - GEMM, MTTKRP, TTM and TTMc for both FPGA and CGRA

T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Tensor Computations

Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao,
Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, Geoff Lowney, Adam Herr, Christopher Hughes, Timothy Mattson, Pradeep Dubey

Question?