Embedding a DSL in SYCL for Productive and Performant Tensor Computing on Heterogeneous Devices

Abenezer Wudenhe

University of California, Riverside

Parallel Computing Lab, Intel

2022

Overview of T2S

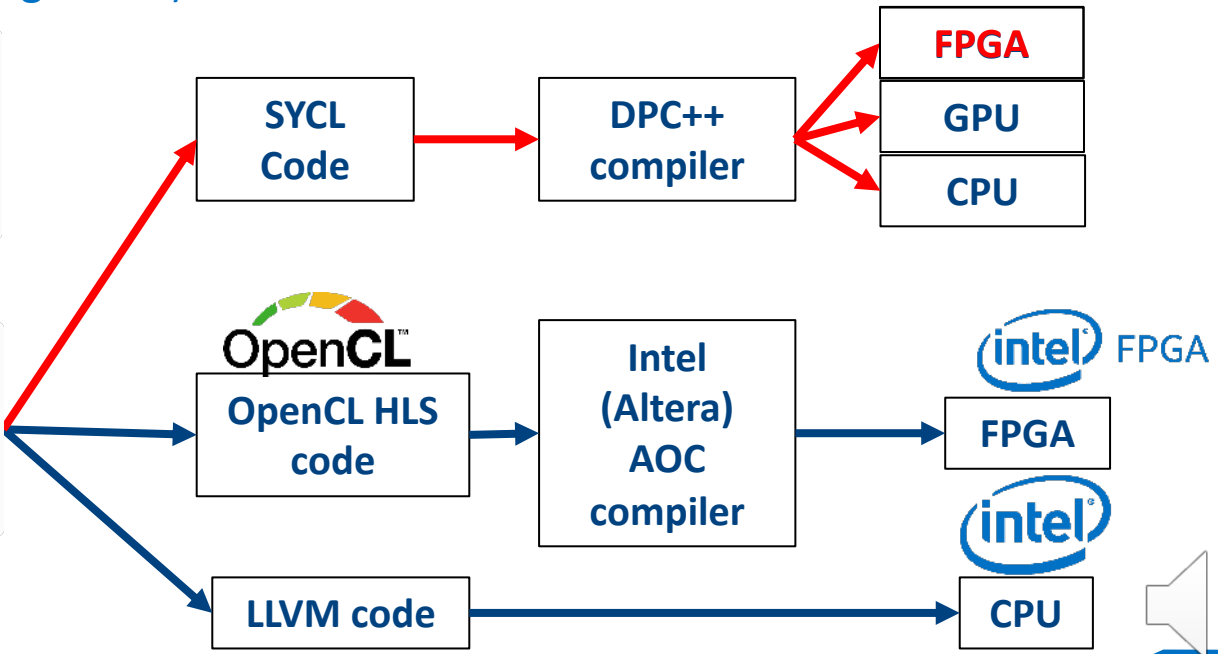
- A constructive programming approach for spatial architectures (e.g. FPGAs) and vector architectures (e.g. GPUs)

```
1 C(i) = A(i)*b
2 C.update(0).tile(i,ii,II)
3 C.update(0).unroll(ii)
```

T2S code

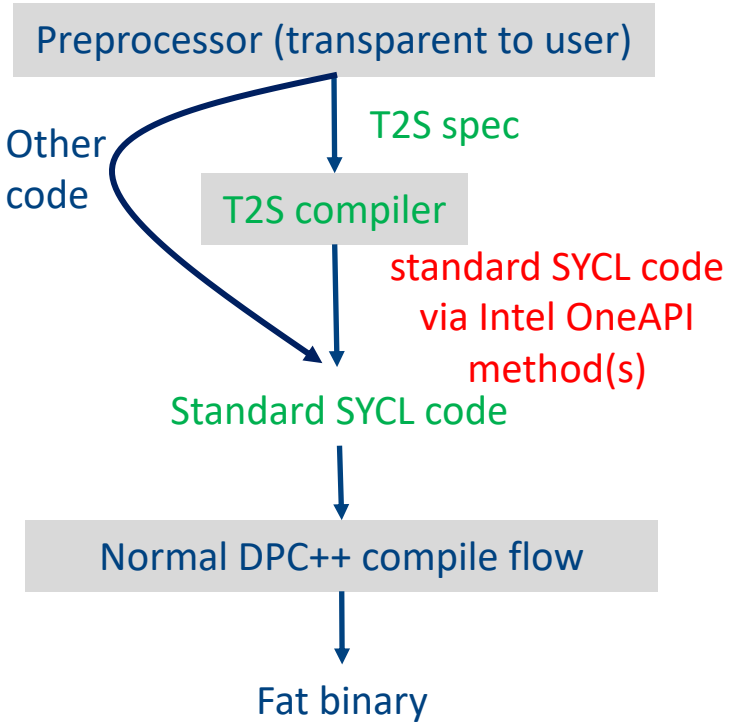
```
1 unrolled(ii,0,II){
2     for(i = 0; /*...*/){
3         C[i*II-ii] += A[i*II+ii]*b;
4     }
5 }
```

Intermediate Representation (IR)



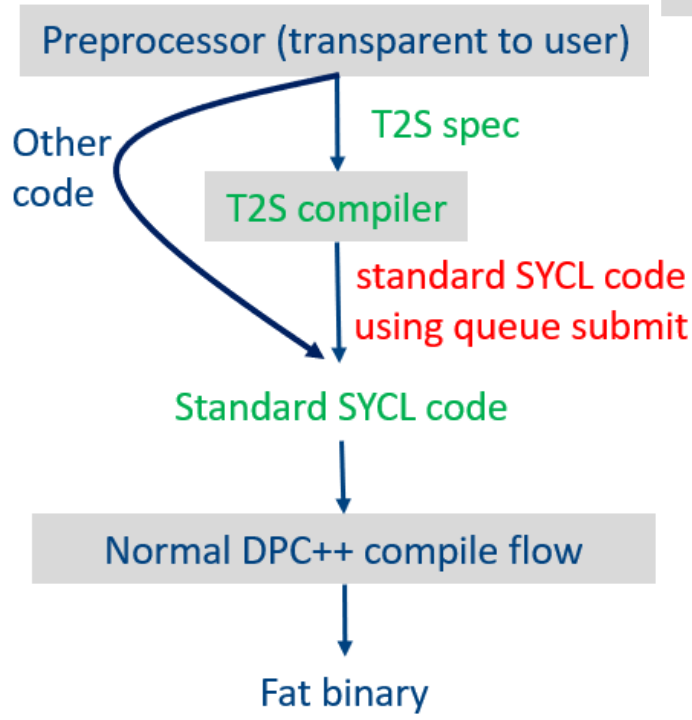
Integrating T2S with SYCL

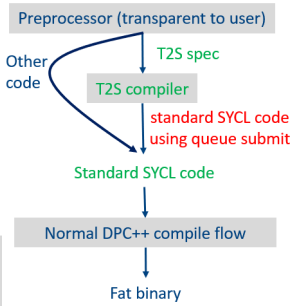
```
/* gemm.cpp */

// The only header file needed for including T2S.
#include "HalideBuffer.h"

int main(){

    // A T2S specification that would create a matrix multiply accelerator on an FPGA.
    #pragma t2s_spec_start
        // Inputs
        ImageParam A("A", TTYPE, 2), B("B", TTYPE, 2);
        // ...
        // Synthesize the spec into a function "gemm", which is OneAPI/SYCL device code for FPGA
        C.compile_to_oneapi( { A, B }, "gemm", IntelFPGA);
    #pragma t2s_spec_end

    // Initalize data
    const int TOTAL_I = III * II * I;
    const int TOTAL_J = JJJ * JJ * J;
    const int TOTAL_K = KKK * KK * K;
    float *a = new float[TOTAL_K * TOTAL_I];
    float *b = new float[TOTAL_J * TOTAL_K];
    float *c = new float[JJJ * III * JJ * II * J * I];
    for(unsigned int i = 0; i < (TOTAL_K * TOTAL_I); i++){ a[i] = random(); }
    for(unsigned int i = 0; i < (TOTAL_J * TOTAL_K); i++){ b[i] = random(); }
    for(unsigned int i = 0; i < (JJJ * III * JJ * II * J * I); i++){ c[i] = 0.0f; }

    // Dimensions of the data
    int a_dim[2] = {TOTAL_K, TOTAL_I};
    int b_dim[2] = {TOTAL_J, TOTAL_K};
    int c_dim[6] = {JJJ, III, JJ, II, J, I};

    // Below we invoke the generated matrix multiple accelerator

    // Call a function "gemm: that is to be generated by the embedded T2S specification above (C2)
    // Pass in the data and dimensions. For example, "A(A, a, a_dim)" means that "A" in the
    // spec above (C2) corresponds to array "a" with dimension "a_dim".
    #pragma t2s_submit gemm (A, a, a_dim) (C, c, c_dim) (B, b, b_dim)

    return 0;
}
```



Integrating T2S with SYCL

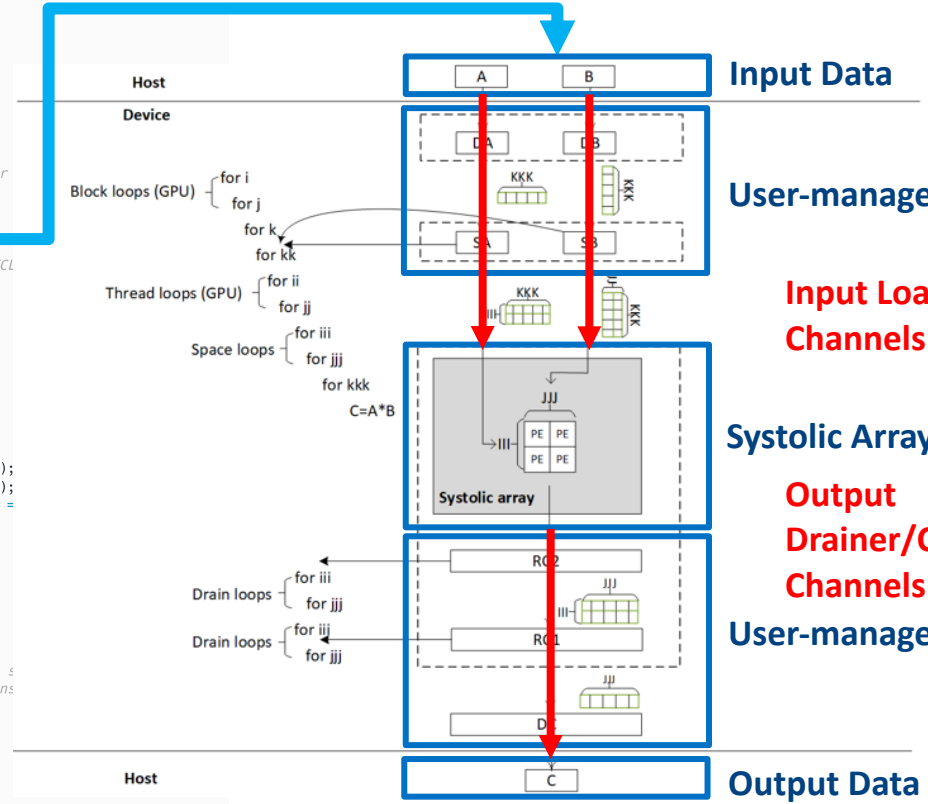
```
/* gemm.cpp */

// The only header file needed for including T2S.
#include "HalideBuffer.h"

int main(){

    // A T2S specification that would create a matrix multiply accelerator on an FPGA.
    #pragma t2s_spec_start
        // Inputs
        ImageParam A("A", TTYPE, 2), B("B", TTYPE, 2);
        // ...
        // Synthesise the spec into a function "gemm", which is OneAPI/SYCL device code, for FPGA
        C.compile_to_oneapi( { A, B }, "gemm", IntelFPGA);
    #pragma t2s_spec_end

    // Initalize data
    const int TOTAL_I = III * II * I;
    const int TOTAL_J = JJJ * JJ * J;
    const int TOTAL_K = KKK * KK * K;
    float *a = new float[TOTAL_K * TOTAL_I];
    float *b = new float[TOTAL_J * TOTAL_K];
    float *c = new float[JJJ * III * JJ * II * J * I];
    for(unsigned int i = 0; i < (TOTAL_K * TOTAL_I); i++){ a[i] = random(); }
    for(unsigned int i = 0; i < (TOTAL_J * TOTAL_K); i++){ b[i] = random(); }
    for(unsigned int i = 0; i < (JJJ * III * JJ * II * J * I); i++){ c[i] = 0.0f; }

    // Dimensions of the data
    int a_dim[2] = {TOTAL_K, TOTAL_I};
    int b_dim[2] = {TOTAL_J, TOTAL_K};
    int c_dim[6] = {JJJ, III, JJ, II, J, I};

    // Below we invoke the generated matrix multiple accelerator

    // Call a function "gemm: that is to be generated by the embedded T2S specification above (C2)
    // Pass in the data and dimensions. For example, "A(A, a, a_dim)" means that "A" in the
    // spec above (C2) corresponds to array "a" with dimension "a_dim".
    #pragma t2s_submit gemm (A, a, a_dim) (C, c, c_dim) (B, b, b_dim)

    return 0;
}
```

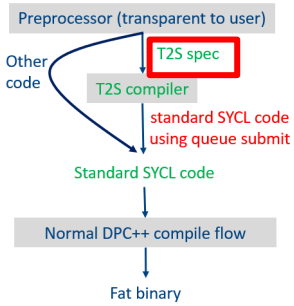




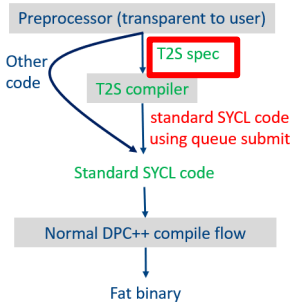
Integrating T2S with SYCL

```
/* gemm.cpp */

// The only header file needed for including T2S.
#include "HalideBuffer.h"

int main(){

    // A T2S specification that would create a matrix multiply accelerator
    #pragma t2s_spec_start
        // Inputs
        ImageParam A("A", TTYPE, 2), B("B", TTYPE, 2);
        // ...
        // Synthesise the spec into a function "gemm", which is OneAPI/SYCL
        C.compile_to_oneapi( { A, B }, "gemm", IntelFPGA);
    #pragma t2s_spec_end

    // Initalize data
    const int TOTAL_I = III * II * I;
    const int TOTAL_J = JJJ * JJ * J;
    const int TOTAL_K = KKK * KK * K;
    float *a = new float[TOTAL_K * TOTAL_I];
    float *b = new float[TOTAL_J * TOTAL_K];
    float *c = new float[JJJ * III * JJ * II * J * I];
    for(unsigned int i = 0; i < (TOTAL_K * TOTAL_I); i++){ a[i] = random();
    for(unsigned int i = 0; i < (TOTAL_J * TOTAL_K); i++){ b[i] = random();
    for(unsigned int i = 0; i < (JJJ * III * JJ * II * J * I); i++){ c[i] =

    // Dimensions of the data
    int a_dim[2] = {TOTAL_K, TOTAL_I};
    int b_dim[2] = {TOTAL_J, TOTAL_K};
    int c_dim[6] = {JJJ, III, JJ, II, J, I};

    // Below we invoke the generated matrix multiple accelerator

    // Call a function "gemm: that is to be generated by the embedded T2S s
    // Pass in the data and dimensions. For example, "A(A, a, a_dim)" means
    // spec above (C2) corresponds to array "a" with dimension "a_dim".
    #pragma t2s_submit gemm (A, a, a_dim) (C, c, c_dim) (B, b, b_dim)

    return 0;
}
```





Integrating T2S with SYCL

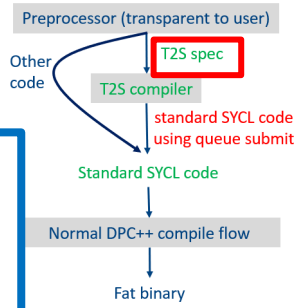



```cpp
/* gemm.cpp */

// The only header file needed for including T2S.
#include "HalideBuffer.h"

int main(){

    // A T2S specification that would create a matrix multiply accelerator
    #pragma t2s_spec_start
        // Inputs
        ImageParam A("A", TTYPE, 2), B("B", TTYPE, 2);
        // ...
        // Synthesise the spec into a function "gemm", which is OneAPI/FPGA
        C.compile_to_oneapi( { A, B }, "gemm", IntelFPGA);
    #pragma t2s_spec_end

    // Initalize data
    const int TOTAL_I = III * II * I;
    const int TOTAL_J = JJJ * JJ * J;
    const int TOTAL_K = KKK * KK * K;
    float *a = new float[TOTAL_K * TOTAL_I];
    float *b = new float[TOTAL_J * TOTAL_K];
    float *c = new float[JJJ * III * JJ * II * J * I];
    for(unsigned int i = 0; i < (TOTAL_K * TOTAL_I); i++){ a[i] = random(
    for(unsigned int i = 0; i < (TOTAL_J * TOTAL_K); i++){ b[i] = random(
    for(unsigned int i = 0; i < (JJJ * III * JJ * II * J * I); i++){ c[i]

    // Dimensions of the data
    int a_dim[2] = {TOTAL_K, TOTAL_I};
    int b_dim[2] = {TOTAL_J, TOTAL_K};
    int c_dim[6] = {JJJ, III, JJ, II, J, I};

    // Below we invoke the generated matrix multiple accelerator

    // Call a function "gemm: that is to be generated by the embedded T2S
    // Pass in the data and dimensions. For example, "A(A, a, a_dim)" means that "A" in the
    // spec above (C2) corresponds to array "a" with dimension "a_dim".
    #pragma t2s_submit gemm (A, a, a_dim) (C, c, c_dim) (B, b, b_dim)

    return 0;
}
```

```cpp
// Dependences
#define P               cii,      cooo,   yyy,   xxx, coo, yy, xx,  ky,      kx,      ci,   y, x, co, n
#define P_cii_minus_1   cii-1,    cooo,   yyy,   xxx, coo, yy, xx,  ky,      kx,      ci,   y, x, co, n
#define P_ky_minus_1    cii+CII-1, cooo,  yyy,   xxx, coo, yy, xx,  ky-1,    kx,      ci,   y, x, co, n
#define P_kx_minus_1    cii+CII-1, cooo,  yyy,   xxx, coo, yy, xx,  ky+KY-1, kx-1,    ci,   y, x, co, n
#define P_ci_minus_1    cii+CII-1, cooo,  yyy,   xxx, coo, yy, xx,  ky+KY-1, kx+KX-1, ci-1, y, x, co, n
#define P_cooo_minus_1  cii,      cooo-1, yyy,   xxx, coo, yy, xx,  ky,      kx,      ci,   y, x, co, n
#define P_yyy_minus_1   cii,      cooo,   yyy-1, xxx, coo, yy, xx,  ky,      kx,      ci,   y, x, co, n
#define P_Out                     cooo,   yyy,   xxx, coo, yy, xx,                              y, x, co, n
// Linearized addresso
#define total_iy        (yyy + YYY*yy + YYY*YY*y + ky)
#define total_ix        (xxx + XXX*xx + XXX*XX*x + kx)
#define total_oy        (yyy + YYY*yy + YYY*YY*y)
#define total_ox        (xxx + XXX*xx + XXX*XX*x)
#define total_co        (cooo + COOO*coo + COOO*COO*co)
#define total_ci        (cii + CII*ci)

// Type of the data to process in C and T2S
#define CTYPE float
#define TTYPE Float(32)

// Inputs
#ifdef GPU
    ImageParam I("I", TTYPE, 2), K("K", TTYPE, 2);
    #define P_I     total_ci + (TOTAL_CI) * n,  total_iy + (TOTAL_IY) * total_ix
    #define P_K     total_co + (TOTAL_CO) * kx, total_ci + (TOTAL_CI) * ky
    #define P_O     total_co + (TOTAL_CO) * n,  total_oy + (TOTAL_OY) * total_ox
    #define UN      (I.dim(0).extent() / TOTAL_CI)
#else
    ImageParam I("I", TTYPE, 4), K("K", TTYPE, 4);
    #define P_I     total_iy, total_ix, total_ci, n
    #define P_K     ky, kx, total_ci, total_co
    #define P_O     P_Out
    #define UN      (I.dim(3).extent())
#endif
```

```cpp
    // UREs
    Var cii("cii"), ci("ci"), cooo("cooo"), coo("coo"), co("co"), ky("ky"), kx("kx"), yyy("yyy"), xxx("xxx"), yy("yy"), xx("xx"), y("y"), x("x"), n("n");
    URE A("A", TTYPE, {P}), B("B", TTYPE, {P}), C("C", TTYPE, {P}), Out("Out");
    A(P) = select(cooo == 0, I(P_I), A(P_cooo_minus_1));
    B(P) = select(yyy == 0, K(P_K), B(P_yyy_minus_1));
    C(P) = select(cii == 0 && ky == 0 && kx == 0 && ci == 0, 0,
              select(cii == 0, select(ky == 0, select(kx == 0, C(P_ci_minus_1), C(P_kx_minus_1)), C(P_ky_minus_1)), C(P_cii_minus_1)))
              + A(P) * B(P);
    Out(P_Out) = select(cii == CII-1 && ky == KY-1 && kx == KX-1 && ci == CI-1, C(P));

    // Put all the UREs inside the same loop nest of X.
    A.merge_ures(B, C, Out);

    // Explicitly set the loop bounds
    A.set_bounds(cooo,  0, COOO, coo,  0, COO, co, 0, CO)
     .set_bounds(ky,    0, KY,   kx,   0, KX)
     .set_bounds(cii,   0, CII,  ci,   0, CI)
     .set_bounds(yyy,   0, YYY,  xxx,  0, XXX)
     .set_bounds(yy,    0, YY,   xx,   0, XX)
     .set_bounds(y,     0, Y,    x,    0, X)
     .set_bounds(n,     0, UN);

    // Create a systolic array
    A.space_time_transform(cooo, yyy);

    // GPU can have many threads running in parallel.
#ifdef GPU
    A.gpu_blocks(x, co, n).gpu_threads(yy, xx);
#endif

    // I/O network
    Stensor DI("iLoader", DRAM), SI("iFeeder", SRAM), DK("kLoader", DRAM), SK("kFeeder", SRAM);
    Stensor RO2("drainer", REG), RO1("collector", REG), DO("unloader", DRAM), O("deserializer");
    I >> DI.out(cii) >> FIFO(128)
      >> SI.scope(kx).out(cii, yyy) >> FIFO(128);
    K >> DK.out(cii) >> FIFO(128)
      >> SK.scope(kx).out(cii, cooo) >> FIFO(128);
    Out >> FIFO(1024) >> RO2.scope(xxx).out(cooo, yyy)
        >> FIFO(128)  >> RO1.scope(yyy).out(cooo)
        >> FIFO(128)  >> DO >> O(P_O);

    // Compile the kernel to an FPGA bitstream, and expose a C interface for the host to invoke
#ifdef GPU
    // O.compile_to_host("conv-interface", { I, K }, "conv", IntelGPU);
    // O.compile_to_oneapi({ I, K }, "conv", IntelGPU);
    std::cout << "(NOTE) T2S w/ OneAPI on GPU is Not implemented at this time\n";
#else
    // O.compile_to_host("conv-interface", { I, K }, "conv", IntelFPGA);
    O.compile_to_oneapi({ I, K }, "conv", IntelFPGA);
#endif
```

Unified programming model: T2S



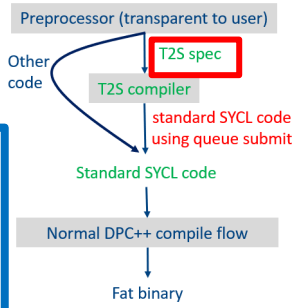
Defining Parameters

```
29    // Dependences
30    #define P               cii,       cooo,   yyy,   xxx, coo, yy, xx,  ky,      kx,      ci,   y, x, co, n
31    #define P_cii_minus_1   cii-1,     cooo,   yyy,   xxx, coo, yy, xx,  ky,      kx,      ci,   y, x, co, n
32    #define P_ky_minus_1    cii+CII-1, cooo,   yyy,   xxx, coo, yy, xx,  ky-1,    kx,      ci,   y, x, co, n
33    #define P_kx_minus_1    cii+CII-1, cooo,   yyy,   xxx, coo, yy, xx,  ky+KY-1, kx-1,    ci,   y, x, co, n
34    #define P_ci_minus_1    cii+CII-1, cooo,   yyy,   xxx, coo, yy, xx,  ky+KY-1, kx+KX-1, ci-1, y, x, co, n
35    #define P_cooo_minus_1  cii,       cooo-1, yyy,   xxx, coo, yy, xx,  ky,      kx,      ci,   y, x, co, n
36    #define P_yyy_minus_1   cii,       cooo,   yyy-1, xxx, coo, yy, xx,  ky,      kx,      ci,   y, x, co, n
37    #define P_Out                      cooo,   yyy,   xxx, coo, yy, xx,                             y, x, co, n
38    // Linearized addresso
39    #define total_iy        (yyy + YYY*yy + YYY*YY*y + ky)
40    #define total_ix        (xxx + XXX*xx + XXX*XX*x + kx)
41    #define total_oy        (yyy + YYY*yy + YYY*YY*y)
42    #define total_ox        (xxx + XXX*xx + XXX*XX*x)
43    #define total_co        (cooo + COOO*coo + COOO*COO*co)
44    #define total_ci        (cii + CII*ci)
45
46    // Type of the data to process in C and T2S
47    #define CTYPE float
48    #define TTYPE Float(32)
```

Defining Inputs

```
49
50    // Inputs
51  #if 0 CPU
52      ImageParam I("I", TTYPE, 2), K("K", TTYPE, 2);
53      #define P_I     total_ci + (TOTAL_CI) * n,  total_iy + (TOTAL_IY) * total_ix
54      #define P_K     total_co + (TOTAL_CO) * kx, total_ci + (TOTAL_CI) * ky
55      #define P_O     total_co + (TOTAL_CO) * n,  total_oy + (TOTAL_OY) * total_ox
56      #define UN      (I.dim(0).extent() / TOTAL_CI)
57  #else
58      ImageParam I("I", TTYPE, 4), K("K", TTYPE, 4);
59      #define P_I     total_iy, total_ix, total_ci, n
60      #define P_K     ky, kx, total_ci, total_co
61      #define P_O     P_Out
62      #define UN      (I.dim(3).extent())
63  #endif
```

Unified programming model: T2S



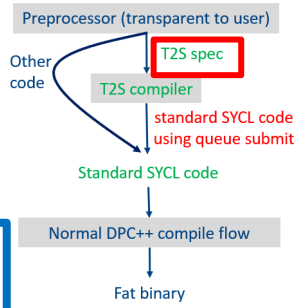

```
65      // UREs
66      Var cii("cii"), ci("ci"), cooo("cooo"), coo("coo"), co("co"), ky("ky"), kx("kx"), yyy("yyy"), xxx("xxx"), yy("yy"), xx("xx"), y("y"), x("x"), n("n");
67      URE A("A", TTYPE, {P}), B("B", TTYPE, {P}), C("C", TTYPE, {P}), Out("Out");
68      A(P) = select(cooo == 0, I(P_I), A(P_cooo_minus_1));
69      B(P) = select(yyy == 0, K(P_K), B(P_yyy_minus_1));
70      C(P) = select(cii == 0 && ky == 0 && kx == 0 && ci == 0, 0,
71                  select(cii == 0, select(ky == 0, select(kx == 0, C(P_ci_minus_1), C(P_kx_minus_1)), C(P_ky_minus_1)), C(P_cii_minus_1)))
72                  + A(P) * B(P);
73      Out(P_Out) = select(cii == CII-1 && ky == KY-1 && kx == KX-1 && ci == CI-1, C(P));
74
75      // Put all the UREs inside the same loop nest of X.
76      A.merge_ures(B, C, Out);
77
78      // Explicitly set the loop bounds
79      A.set_bounds(cooo,  0, COOO, coo,  0, COO, co, 0, CO)
80       .set_bounds(ky,    0, KY,   kx,   0, KX)
81       .set_bounds(cii,   0, CII,  ci,   0, CI)
82       .set_bounds(yyy,   0, YYY,  xxx,  0, XXX)
83       .set_bounds(yy,    0, YY,   xx,   0, XX)
84       .set_bounds(y,     0, Y,    x,    0, X)
85       .set_bounds(n,     0, UN);
86
87      // Create a systolic array
88      A.space_time_transform(cooo, yyy);
89
90      // GPU can have many threads running in parallel.
91  #ifdef GPU
92      A.gpu_blocks(x, co, n).gpu_threads(yy, xx);
93  #endif
```

Uniform recurrence equations

Space-time transform

SIMT

Unified programming model: T2S



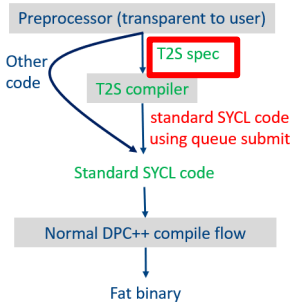

```
95      // I/O network
96      Stensor DI("iLoader", DRAM), SI("iFeeder", SRAM), DK("kLoader", DRAM), SK("kFeeder", SRAM);
97      Stensor RO2("drainer", REG), RO1("collector", REG), DO("unloader", DRAM), O("deserializer");
98      I >> DI.out(cii) >> FIFO(128)
99        >> SI.scope(kx).out(cii, yyy) >> FIFO(128);
100     K >> DK.out(cii) >> FIFO(128)
101       >> SK.scope(kx).out(cii, cooo) >> FIFO(128);
102     Out >> FIFO(1024) >> RO2.scope(xxx).out(cooo, yyy)
103         >> FIFO(128)  >> RO1.scope(yyy).out(cooo)
104         >> FIFO(128)  >> DO >> O(P_O);
```

Abstract memory hierarchy

```
105
106     // Compile the kernel to an FPGA bitstream, and expose a C interface for the host to invoke
107 #ifdef GPU
108     // O.compile_to_host("conv-interface", { I, K }, "conv", IntelGPU);
109     // O.compile_to_oneapi({ I, K }, "conv", IntelGPU);
110     std::cout << "(NOTE) T2S w/ OneAPI on GPU is Not implemented at this time\n";
111 #else
112     // O.compile_to_host("conv-interface", { I, K }, "conv", IntelFPGA);
113     O.compile_to_oneapi({ I, K }, "conv", IntelFPGA);
114 #endif
```

Generate OneAPI/SYCL Code

OneAPI Code Generator

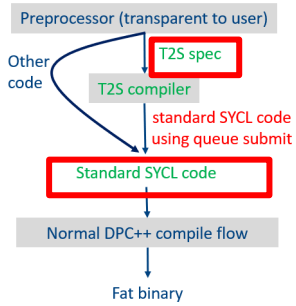



```
1 void compile_to_oneapi(
2     const std::vector< Argument > &,
3     const std::string & fn_name = "",
4     const Target & target = get_target_from_enviornment()
5 );
```

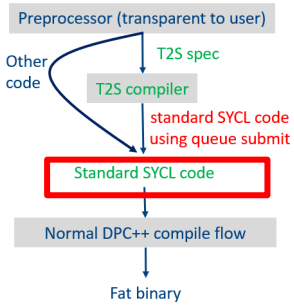
- **`const std::vector<Argument> &:`** A set of inputs to the Func or Stensor pipeline stage
- **`const std::string & fn_name:`** Name of the compiled function and file prefix
- **`const Target & target:`** Set of Halide Targets
  - i.e **`Target::IntelFPGA`** or **`Target::IntelGPU`**

Integrating T2S with OneAPI and SYCL

```
/* gemm.cpp */

// The only header file needed for including T2S.
#include "HalideBuffer.h"

int main(){

    // A T2S specification that would create a matrix multiply accelerator on an FPGA.
    #pragma t2s_spec_start
        // Inputs
        ImageParam A("A", TTYPE, 2), B("B", TTYPE, 2);
        // ...
        // Synthesise the spec into a function "gemm", which is OneAPI/SYCL device code, for FPGA
        C.compile_to_oneapi( { A, B }, "gemm", IntelFPGA);
    #pragma t2s_spec_end

    // Initalize data
    const int TOTAL_I = III * II * I;
    const int TOTAL_J = JJJ * JJ * J;
    const int TOTAL_K = KKK * KK * K;
    float *a = new float[TOTAL_K * TOTAL_I];
    float *b = new float[TOTAL_J * TOTAL_K];
    float *c = new float[JJJ * III * JJ * II * J * I];
    for(unsigned int i = 0; i < (TOTAL_K * TOTAL_I); i++){ a[i] = random(); }
    for(unsigned int i = 0; i < (TOTAL_J * TOTAL_K); i++){ b[i] = random(); }
    for(unsigned int i = 0; i < (JJJ * III * JJ * II * J * I); i++){ c[i] = 0.0f; }

    // Dimensions of the data
    int a_dim[2] = {TOTAL_K, TOTAL_I};
    int b_dim[2] = {TOTAL_J, TOTAL_K};
    int c_dim[6] = {JJJ, III, JJ, II, J, I};

    // Below we invoke the generated matrix multiple accelerator

    // Call a function “gemm: that is to be generated by the embedded T2S specification above (C2)
    // Pass in the data and dimensions. For example, “A(A, a, a_dim)” means that “A” in the
    // spec above (C2) corresponds to array ”a” with dimension “a_dim”.
    #pragma t2s_submit gemm (A, a, a_dim) (C, c, c_dim) (B, b, b_dim)

    return 0;
}
```



Integrating T2S with OneAPI and SYCL




```
/* gemm.run.cpp */
#include "HalideBuffer.h"
#include "gemm.sycl.h"

// Loop bounds
#define KKK         16
#define JJJ         8
#define III         10
#define JJ          32
#define II          32
#define KK          32
#define K           32
#define J           32
#define I           32

int main(){

    // Initalize data
    const int TOTAL_I = III * II * I;
    const int TOTAL_J = JJJ * JJ * J;
    const int TOTAL_K = KKK * KK * K;
    float *a = new float[TOTAL_K * TOTAL_I];
    float *b = new float[TOTAL_J * TOTAL_K];
    float *c = new float[JJJ * III * JJ * II * J * I];
    for(unsigned int i = 0; i < (TOTAL_K * TOTAL_I); i++){ a[i] = random(); }
    for(unsigned int i = 0; i < (TOTAL_J * TOTAL_K); i++){ b[i] = random(); }
    for(unsigned int i = 0; i < (JJJ * III * JJ * II * J * I); i++){ c[i] = 0.0f; }

    // Dimensions of the data
    int a_dim[2] = {TOTAL_K, TOTAL_I};
    int b_dim[2] = {TOTAL_J, TOTAL_K};
    int c_dim[6] = {JJJ, III, JJ, II, J, I};

    // Call a function "gemm: that is to be generated by the embedded T2S specification above (C2)
    // Pass in the data and dimensions. For example, "A(A, a, a_dim)" means that "A" in the
    // spec above (C2) corresponds to array "a" with dimension "a_dim".

    // Preprocessor has wrapped input/output memory in DSL usable Runtime Buffer
    Halide::Runtime::Buffer<float, (sizeof(a_dim)/sizeof(int))> A_h(
        a, std::vector<int>(std::begin(a_dim), std::end(a_dim)));
    Halide::Runtime::Buffer<float, (sizeof(c_dim)/sizeof(int))> C_h(
        c, std::vector<int>(std::begin(c_dim), std::end(c_dim)));
    Halide::Runtime::Buffer<float, (sizeof(b_dim)/sizeof(int))> B_h(
        b, std::vector<int>(std::begin(b_dim), std::end(b_dim)));

#if defined(FPGA_EMULATOR)
    std::cout << "USING FPGA EMULATOR\n";
    sycl::ext::intel::fpga_emulator_selector device_selector; // (NOTE) for emulation
#else
    std::cout << "USING REAL FPGA\n";
    sycl::ext::intel::fpga_selector device_selector; // (NOTE) for full compile and hardware profiling
#endif

    // Below we invoke the generated matrix multiple accelerator

    // Preprocessor has invoded the "gemm" function defined by the developers T2S spec
    // the generated "gemm.sycl.h" file
    std::cout << "Start Run\n";
    double exec_time = 0;
    exec_time = gemm(device_selector, A_h.raw_buffer(), B_h.raw_buffer(), C_h.raw_buffer());
    std::cout << "Run completed!\n";
    std::cout << "kernel exec time: " << exec_time << "\n";

    return 0;
}
```

Included OneAPI generated GEMM header file

Wrapped input argument inside T2SP DSL structures

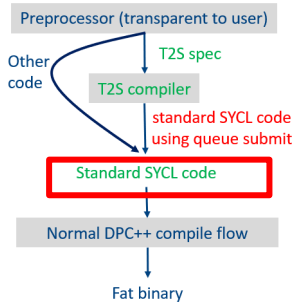
Define OneAPI FPGA device selector

Execute generated GEMM function

Integrating T2S

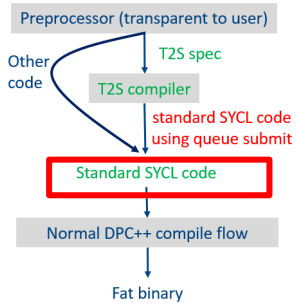
1. Wrapper
2. Host/Device Kernels
3. Memory Management
4. Performance Metrics

1. Wrapper
2. Host/Device Kernels
3. Memory Management
4. Performance metrics



Integrating T2S

1. Wrapper
2. Host/Device Kernels
3. Memory Management
4. Performance metrics

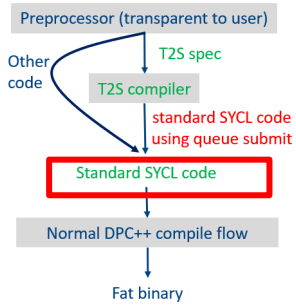


- Generated GEMM code takes in 2 types of arguments
  - **sycl::ext::intel::fpga_selector device_selector**
    - Used to either use FPGA emulation or synthesize to real hardware
  - **Halide::Runtime::Buffer**
    - T2SP's Halide Runtime Buffers. Used to manage device and host memory copies and management

```
1 /* gemm.sycl.h */
2 // Function wrapper
3 double gemm(const sycl::device_selector &deviceSelector,
4             Halide::Runtime::Buffer<float,2> *A,
5             Halide::Runtime::Buffer<float,2> *B,
6             Halide::Runtime::Buffer<float,2> *C) {
7     // …
8 }
```

Integrating T2S

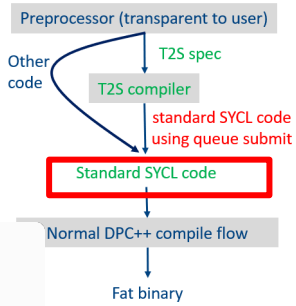
1. Wrapper
2. Host/Device Kernels
3. Memory Management
4. Performance metrics



- Host Kernels
  - Host kernels are executed as normal C++ code on host memory
- Device Kernels
  - Device kernels are submitted into device queues and kernel events are recorded in a **std::vector** for performance analysis at the completion of the application

```
1 /* gemm.sycl.h */
2 // Device Kernel
3 oneapi_kernel_events.push_back( q_device.submit([&](sycl::handler &h){
4   h.single_task<class kernel_1>([=](){
5     // ...
6   }); //  h.single_task kernel_1
7 }) ); // q_device.submit
```

Integrating T2S

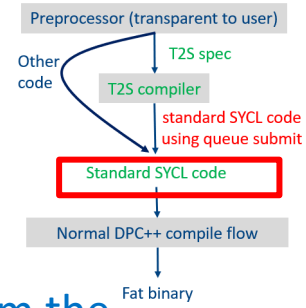


- Memory management
  - Device and Host memory are allocated though **sycl::malloc_device()** and **std::malloc()** respectively
- Memory copies (memcpy) are explicit and executed depending on the memory operation of that kernel
  - If kernel requires load operation, memcpy host->device before kernel submission
  - If kernel requires store operation, memcpy device->host after kernel execution is complete
- SYCL Pipes are used to communicate between data between kernels on the device
- Currently there is no implement of device-to-device memory copies

```
1  /* gemm.sycl.h */
2  // Memcpy
3  std::cout << "//   memcpy device->host A\n";
4  q_device.submit([&](handler& h){
5      h.memcpy((void *)A->host,
6               (void *)A->device,
7               A->size_in_bytes()
8      );
9  });
10
11 std::cout << "//   memcpy host->device A\n";
12 q_device.submit([&](handler& h){
13     h.memcpy((void *)A->device,
14              (void *)A->host,
15              A->size_in_bytes()
16     );
17 });
```

Integrating T2S

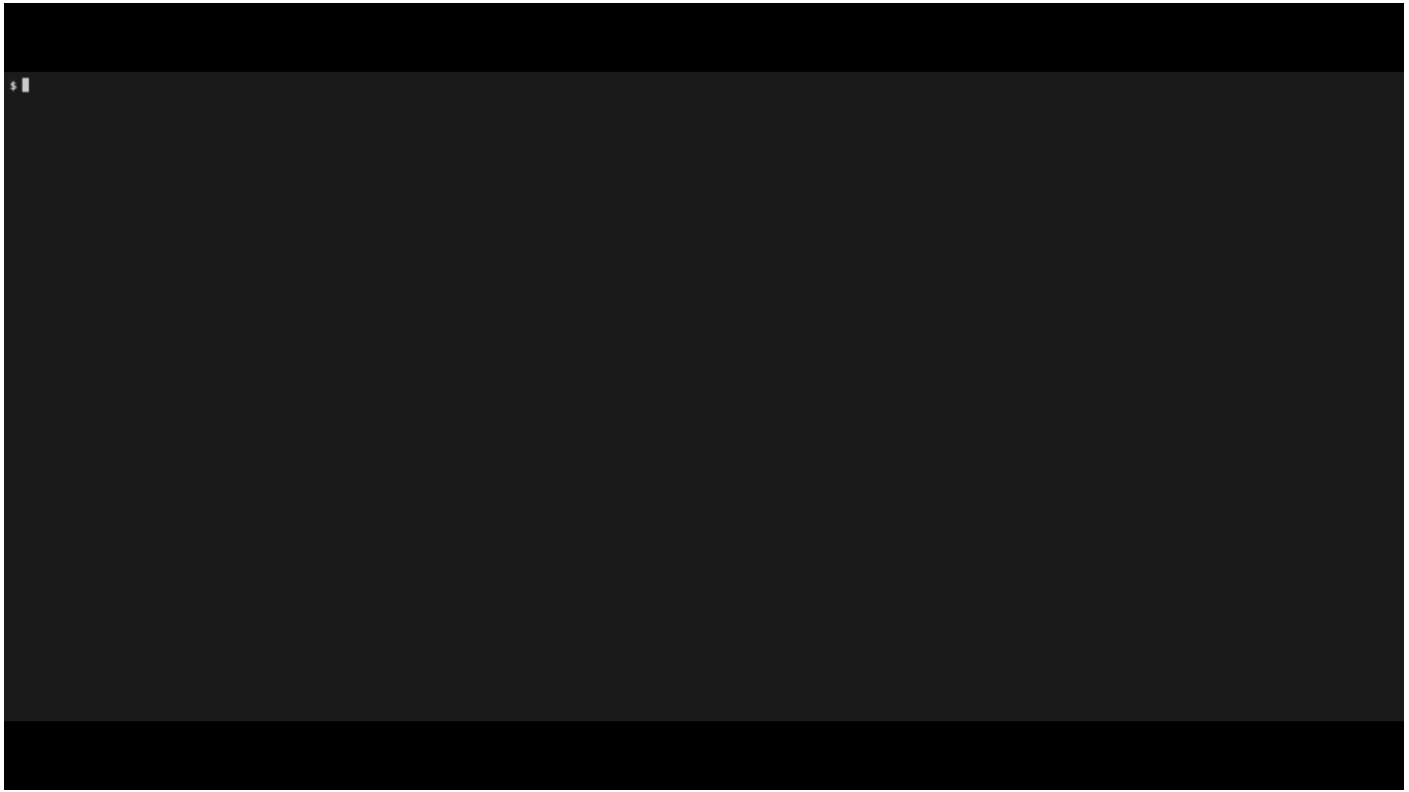
1. Wrapper
2. Host/Device Kernels
3. Memory Management
4. Performance metrics

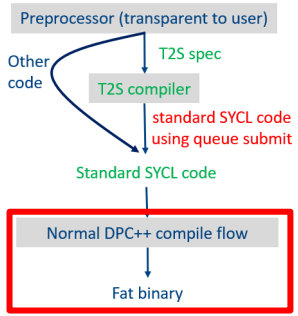


- The generated returns a double value of the kernel execution time i.e. from the earliest start to the latest end of the device kernels

```
/* gemm.sycl.h */
// Performance
double k_earliest_start_time =
oneapi_kernel_events.at(0).get_profiling_info<sycl::info::event_profiling::command_start>();
double k_latest_end_time =
oneapi_kernel_events.at(0).get_profiling_info<sycl::info::event_profiling::command_end>();
for (unsigned i = 1; i < oneapi_kernel_events.size(); i++) {
  double tmp_start =
    oneapi_kernel_events.at(i).get_profiling_info<sycl::info::event_profiling::command_start>();
  double tmp_end =
    oneapi_kernel_events.at(i).get_profiling_info<sycl::info::event_profiling::command_end>();
  if (tmp_start < k_earliest_start_time) {
    k_earliest_start_time = tmp_start;
  }
  if (tmp_end > k_latest_end_time) {
    k_latest_end_time = tmp_end;
  }
}
// Get time in ns
double events_time = (k_latest_end_time - k_earliest_start_time);
return events_time;
```

Recorded demo: T2S Full Compiling Flow GEMM





Experiments/Results

- Benchmark Applications
  - General Matrix Multiply (GEMM)
  - 2-D convolution (CONV)
  - Capsule convolution (CAPSULE)

Experiments/Results

- GEMM/CONV/CAPSULE FPGA Synthesis Results for OpenCL and Generated OneAPI

Benchmark	ALUT	Registers	DSP blocks	RAM Blocks	Clock Freq.
GEMM	105151	180,874	89 / 1,518 (6 %)	462 / 2,713 (17 %)	264
CONV	98648	168,673	64 / 1,518 (4 %)	499 / 2,713 (18 %)	276
CAPSULE	119038	198,909	112 / 1,518 (7 %)	607 / 2,713 (22 %)	275

Experiments/Results

- GEMM/CONV/CAPSULE FPGA Performance results for OpenCL and Generated OneAPI

Benchmark	GFLOPS
GEMM	302.402
CONV	285.301
CAPSULE	231.877

Conclusion

- Expanding Intel's T2SP to leverage SYCL implementation to extend the T2SP domain specific language to one uniform compiling flow for agnostic hardware acceleration
  - Developed Code generator to produce SYCL Device & Host Code
  - Implemented Clang source-to-source code for file preprocessing
  - FPGA implementation is able to non-trivial performance

Future Work

- Investigation as to performance bottlenecks to achieve faster tensor computing
- Extend to GPU, CPU, and other tensor computing accelerators