Introduction to DPC++ Programming for FPGA

A part of the DPC++ Tutorial Series

Prof. Yan Luo

Acknowledgement
This work is supported by Intel Corporation 2020-2021

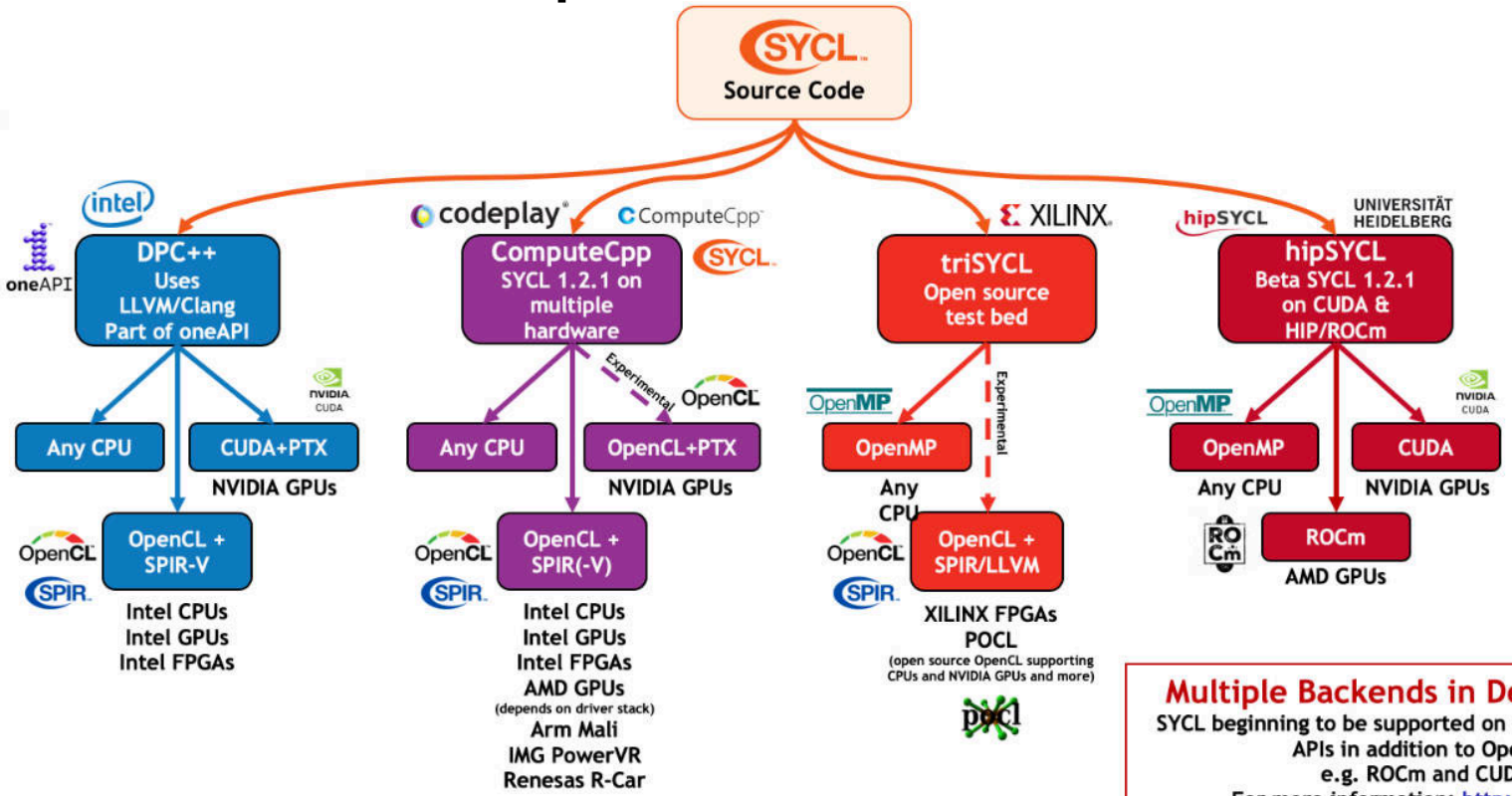
UMASS LOWELL
Learning with Purpose

Data Parallel C++

- A high-level language for data parallel programming
- Based on modern C++
- Single source for heterogeneous computing architectures
- Offloading computing to accelerators (e.g. FPGA and GPU)
- Speedup on data parallel workloads
  - Algorithm and parallelism analysis
  - Data/task decomposition
  - Architecture oriented performance optimization (e.g. for FPGA)

UMASS LOWELL
Learning with Purpose

DPC++: an Implementation of SYCL



Source: https://www.khronos.org/sycl/

A DPC++ Example

```
1  #include <CL/sycl.hpp>
2  #include <iostream>
3
4  constexpr int num=16;
5  using namespace sycl;
6
7  int main() {
8    auto r = range{num};
9    buffer<int> a{r};
10
11   queue{}.submit([&](handler& h) {
12     accessor out{a, h};
13     h.parallel_for(r, [=](item<1> idx) {
14       out[idx] = idx;
15     });
16   });
17
18   host_accessor result{a};
19   for (int i=0; i<num; ++i)
20     std::cout << result[i] << "\n";
21 }
```

Header Files

namespace scope

device queue & command

SYCL buffer declaration

kernel function on device

Lambda function

host access results in buffer

Matrix Multiplication : How to Think in Parallel?

A part of the DPC++ Tutorial Series

Prof. Yan Luo

Acknowledgement
This work is supported by Intel Corporation 2020-2021

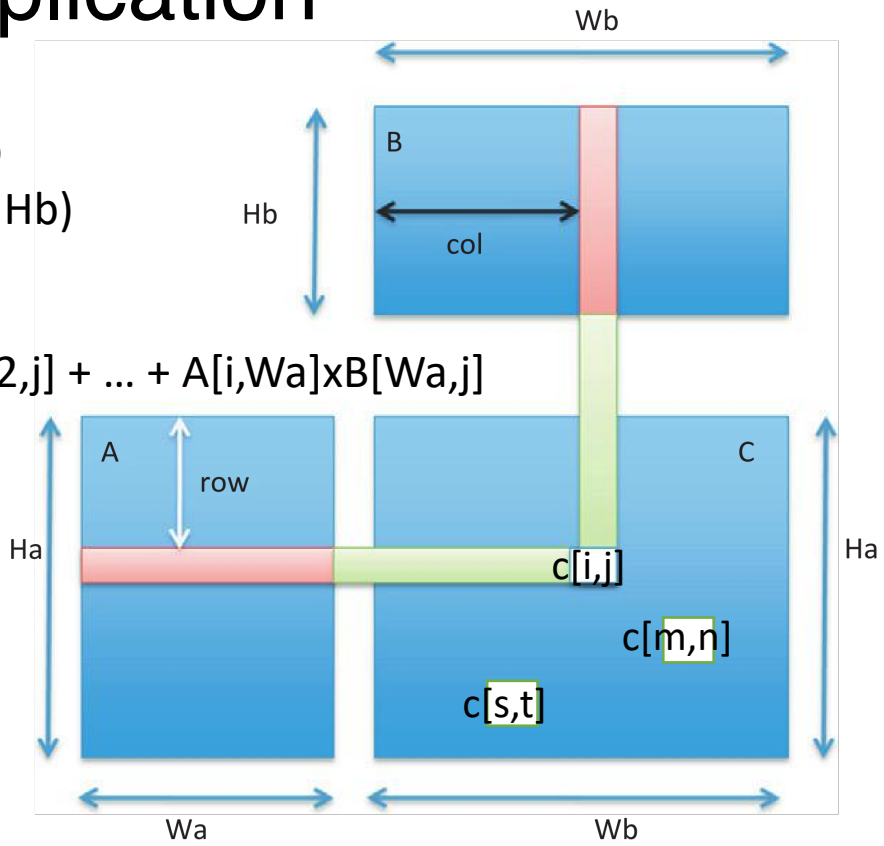
UMASS LOWELL
Learning with Purpose

Matrix Multiplication

A x B = C

A is Wa x Ha
B is Wb x Hb
(Note: Wa = Hb)
C is Wb x Ha

C[i,j] = A[i,1]xB[1,j]+A[i,2]xB[2,j] + … + A[i,Wa]xB[Wa,j]



A C++ Implementation

```
// iterate over the rows of Matrix A
for (int i = 0; i < Ha; i++)
    // iterate over the columns of Matrix B
    for (int j = 0; i < Wb; j++) {
        C[i][j] = 0;
        // element-wise multiplication and accumulation
        for (int k = 0; k < Wa; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
```

A DPC++ Implementation

```
q.submit([&](handler &h) {
    auto A = a_buf.get_access<access::mode::read>(h);
    auto B = b_buf.get_access<access::mode::read>(h);
    auto C = sum_buf.get_access<access::mode::write>(h);

    range<2> num_items{a_rows, b_columns};

    h.parallel_for<class MMpara>(num_items, [=](id<2> i) {
       size_t row = i[0], col = i[1];

       C[row][col] = 0;
       for (size_t k = 0; k < Wa; k++)
           C[row][col] += A[row * Wa + k] * B[k * Wb + col];
    });
});
```

create accessors

set up problem size

lambda function for device

UMASS LOWELL
Learning with Purpose

Demonstration

Compilation and execution of Matrix Multiplication example on Intel FPGA DevCloud

UMASS LOWELL
Learning with Purpose

A Very Brief Introduction to FPGA Design Concepts

A part of the DPC++ Tutorial Series

Prof. Yan Luo

Acknowledgement
This work is supported by Intel Corporation 2020-2021

UMASS LOWELL
Learning with Purpose

Agenda

- Introduction to FPGA Architecture
- Concepts of FPGA Hardware Design
- Mapping Source Code to Hardware Datapath
- Scheduling
- Parallelism Models to FPGA Hardware
- Memory Types

Some materials used in this presentation are based on Intel® OneAPI DPC++ FPGA Optimization Guide

UMASS LOWELL
Learning with Purpose

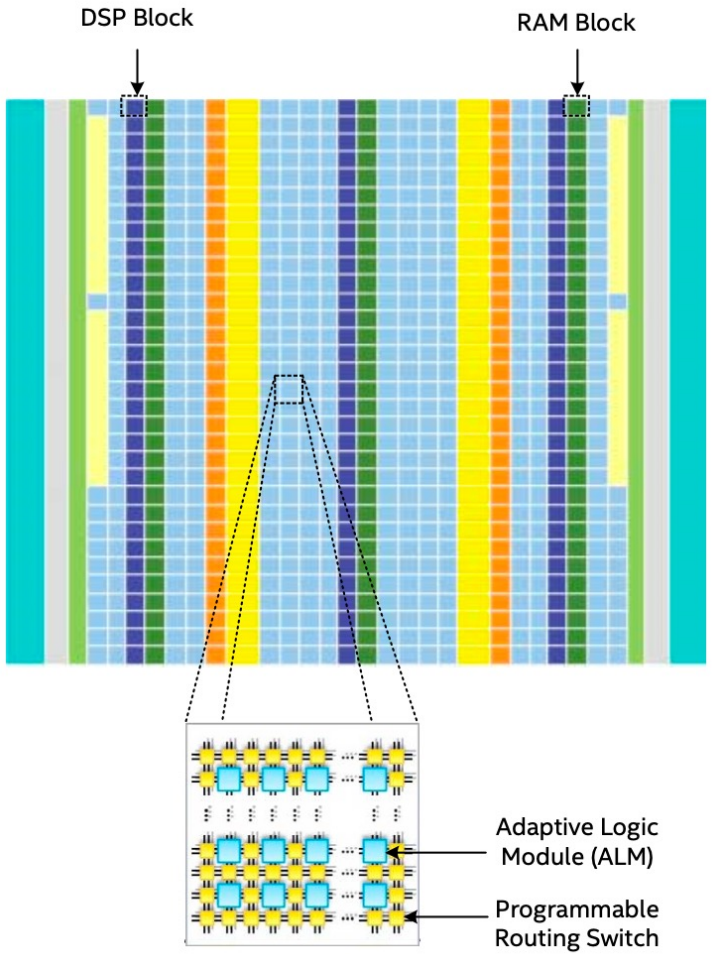
FPGA vs CPU

- FPGA does not have a fixed datapath
  - that is why it is “field programmable” !
- “Program” the hardware resources
  - Your have a lot more control on how your design is mapped
- Function as accelerators to offload intensive computing
- Design methodologies
  - Hardware Description Language
  - High level language (like DPC++)

UMASS LOWELL
Learning with Purpose

FPGA Architecture

- Adaptive Logic Module
- RAM block
- DSP block
- Programmable routing switch



FPGA Hardware Design Concepts

- Maximum frequency f_{MAX}
- Latency
- Pipelining
- Throughput
- Datapath
- Control path
- Occupancy

UMASS LOWELL
Learning with Purpose

f_{MAX}

- f_{MAX} is the max clock frequency a digital circuit operates at
- The max rate at which the outputs or registers are updated
- Clock speed is limited by physical propagation delay
  - Signal propagation across combination logic between two consecutive register stages
- The propagation delay is
  - a function of the complexity of the combinational logic
- Critical path
  - The path with the most combinational logic elements
  - Limits the speed of the entire circuit

UMASS LOWELL
Learning with Purpose

Latency

- Measures how long it takes to complete one or more operations
  - Latency of a single operation within
  - Or latency of the entire circuit
- Measure in time (ms) or cycles
  - Cycles are preferred
- Lowering latency ?
  - Yes, shorter latency is desirable in general
  - But shorter latency may decrease f_{MAX}
  - So, caution on the design alternatives

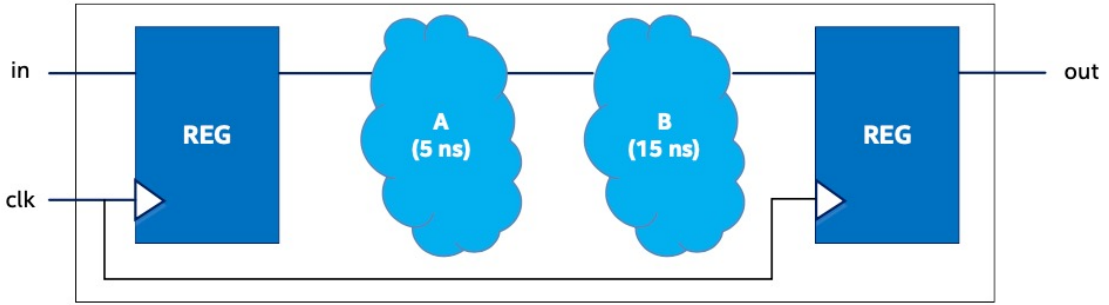
UMASS LOWELL
Learning with Purpose

Pipelining to improve f_{MAX}

unpipelined

f_{MAX} = 50MHz

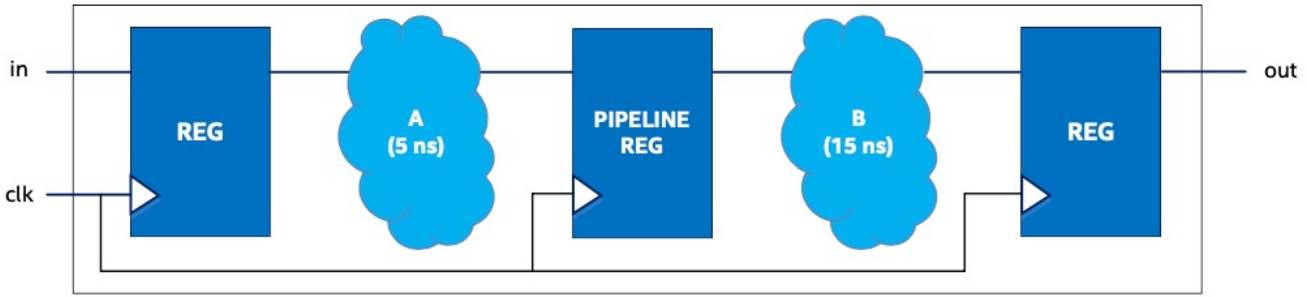
Delay of 2 cycles



pipelined

f_{MAX} = 66.7MHz

delay of 3 cycles



Throughput

- The rate at which data is processed
- Higher fMAX the higher throughput
- Throughput is NOT the inverse of delay
  - Parallel processing or overlapped data processing

UMASS LOWELL
Learning with Purpose

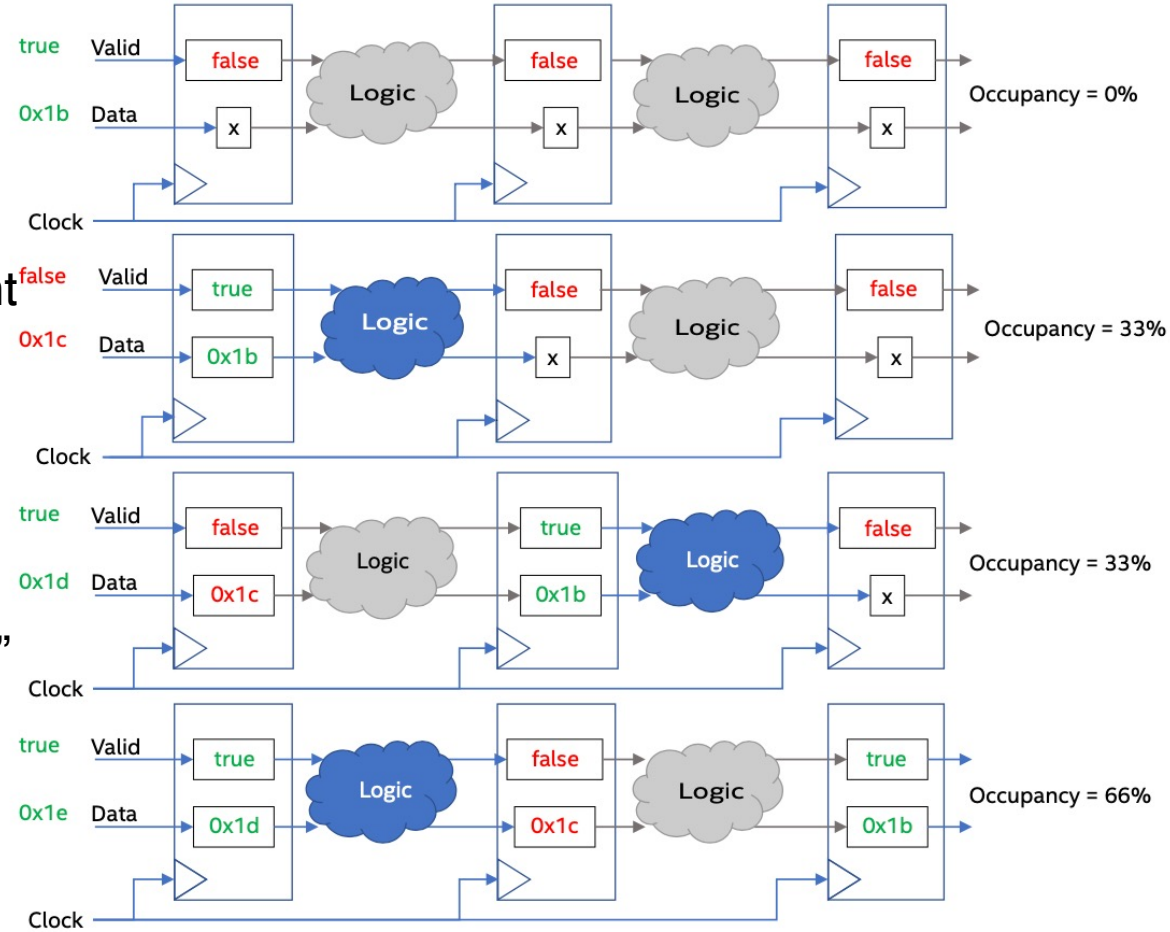
Datapath and Control Path

- Datapath
  - A chain of registers and combinational logic
  - Memory blocks or offchip memory units are considered as outside of the datapath
- Control Path
  - Path of the signals that control the operation of the circuit
  - Logic added for
    - Handshaking flow control
    - Loop control
    - Branch control

UMASS LOWELL
Learning with Purpose

Occupancy

- Datapath Occupancy: At a point in time the proportion of the datapath that contains **valid** data
- Circuit Occupancy: average occupancy over time from program start to end
- Unoccupied portion = ”bubbles”
- Decreasing bubbles increases occupancy



DPC++ Design Analysis (I): Analyze FPGA Early Image

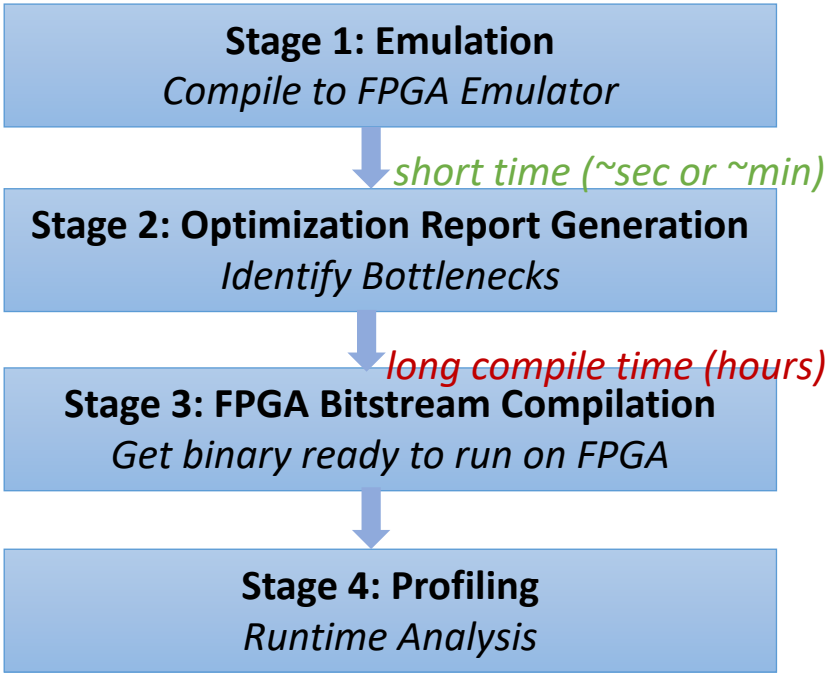
A part of the DPC++ Tutorial Series

Prof. Yan Luo

Acknowledgement
This work is supported by Intel Corporation 2020-2021

UMASS LOWELL
Learning with Purpose

Analyze your Design before Optimization



1. Make sure the design is functionally correct
2. Look for bottlenecks through compilation reports and revise design if necessary
3. Generate FPGA binary for execution on hardware
4. Run profiling to analyze runtime performance

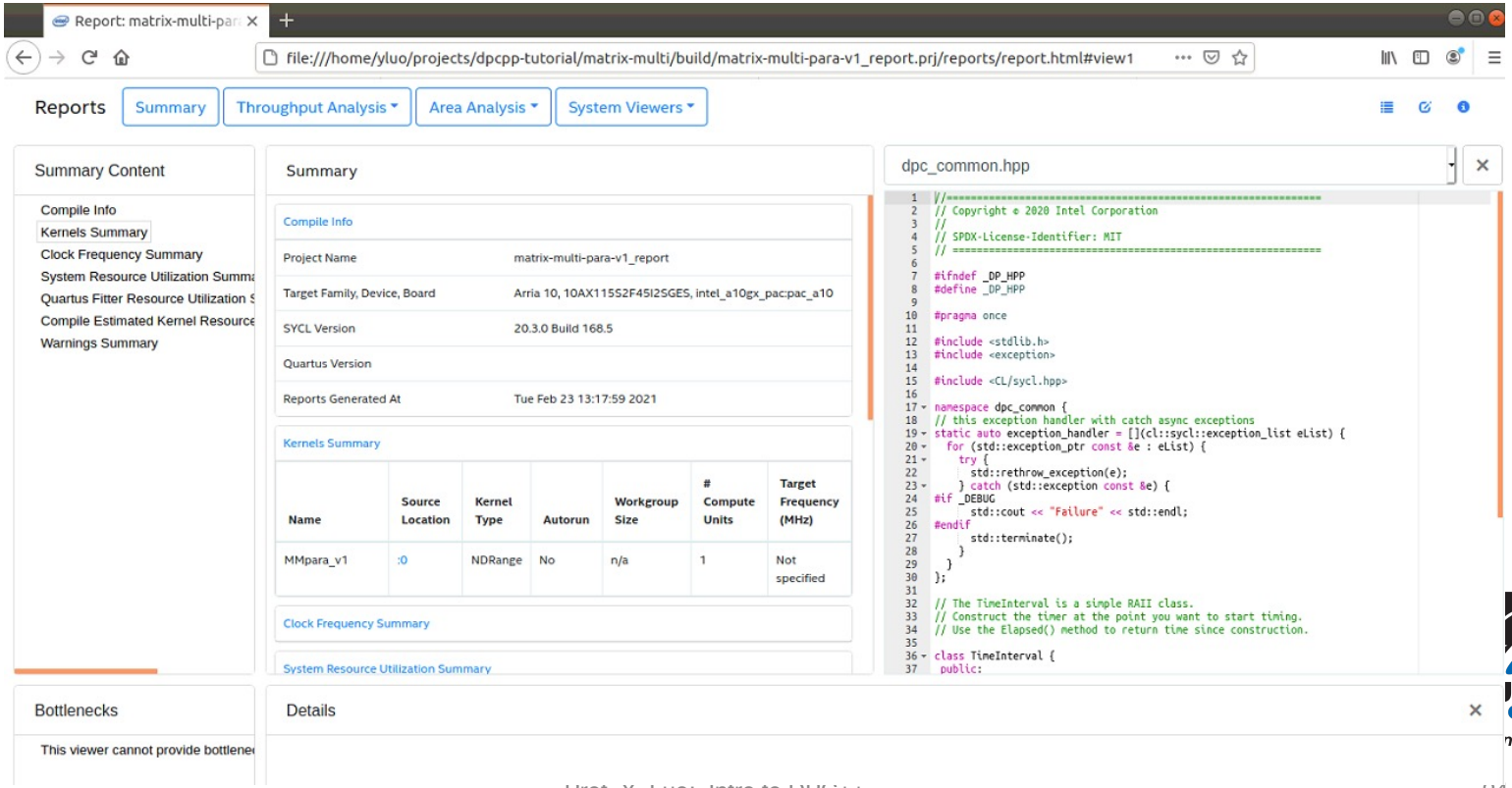
UMASS LOWELL
Learning with Purpose

Demo: Compilation Report Generation and Analysis

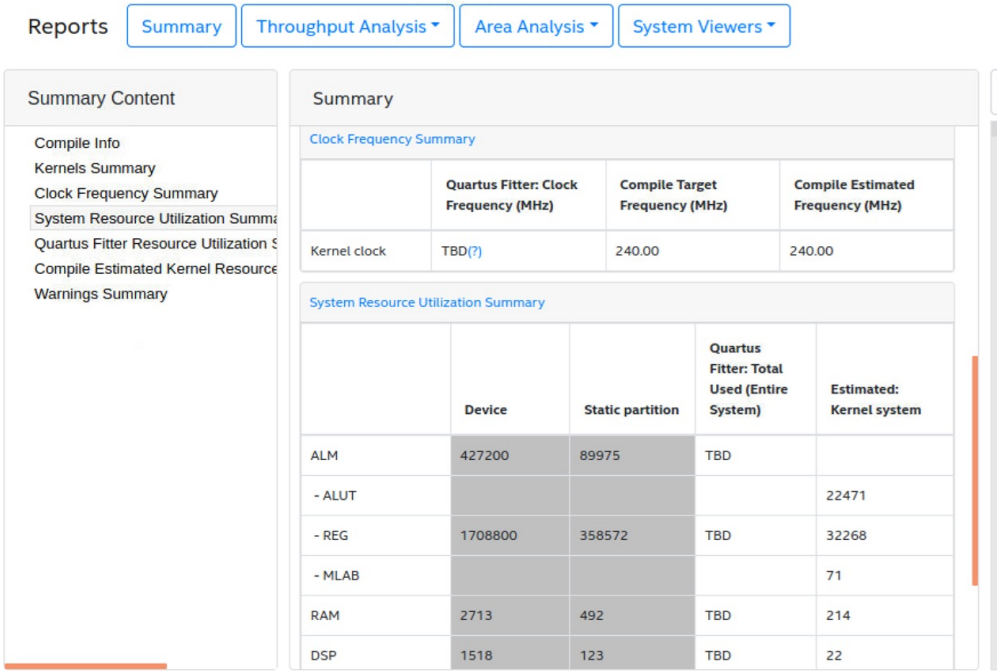
Look through compilation report of Matrix Multiplication example

UMASS LOWELL
Learning with Purpose

Static Analysis Report



Clock Frequency and System Resource Utilization Summary



Reports | Summary | Throughput Analysis | Area Analysis | System Viewers

Summary Content

- Compile Info
- Kernels Summary
- Clock Frequency Summary
- System Resource Utilization Summa
- Quartus Fitter Resource Utilization S
- Compile Estimated Kernel Resource
- Warnings Summary

Summary

Clock Frequency Summary

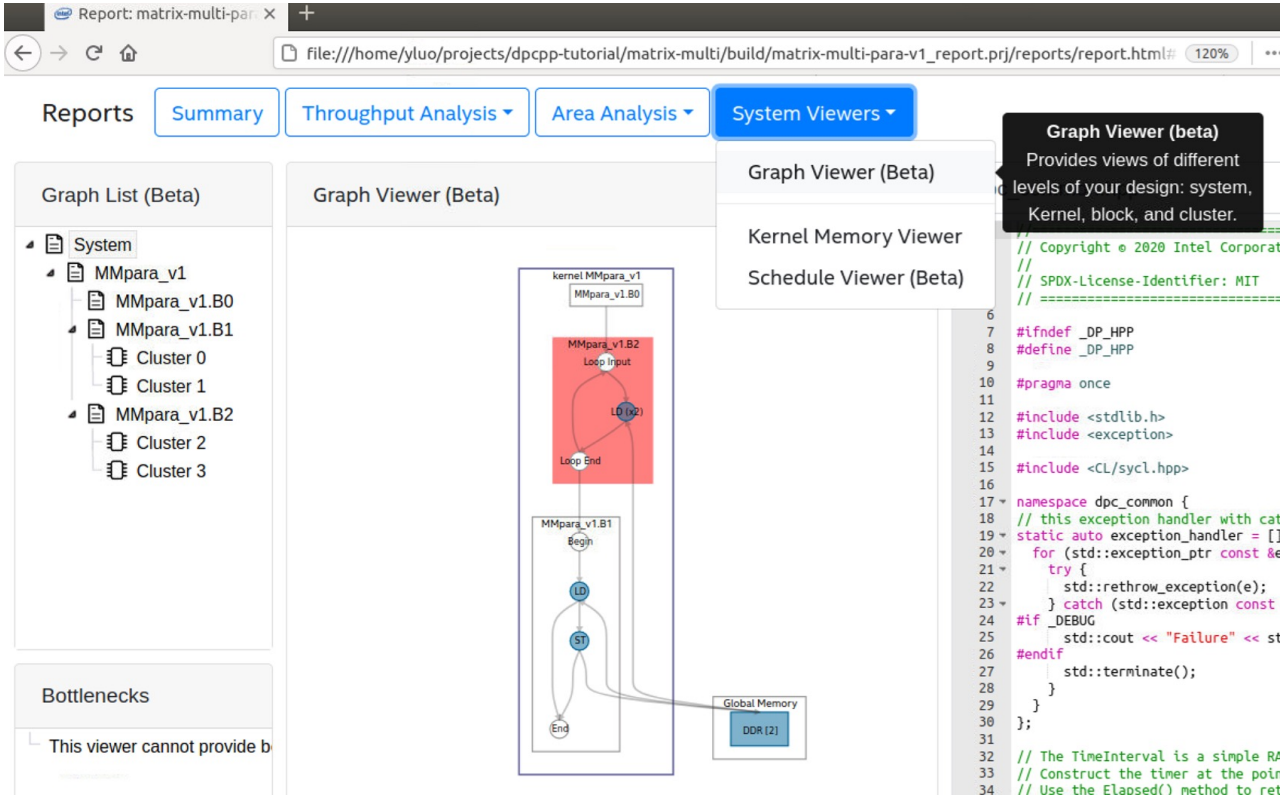
	Quartus Fitter: Clock Frequency (MHz)	Compile Target Frequency (MHz)	Compile Estimated Frequency (MHz)
Kernel clock	TBD(?)	240.00	240.00

System Resource Utilization Summary

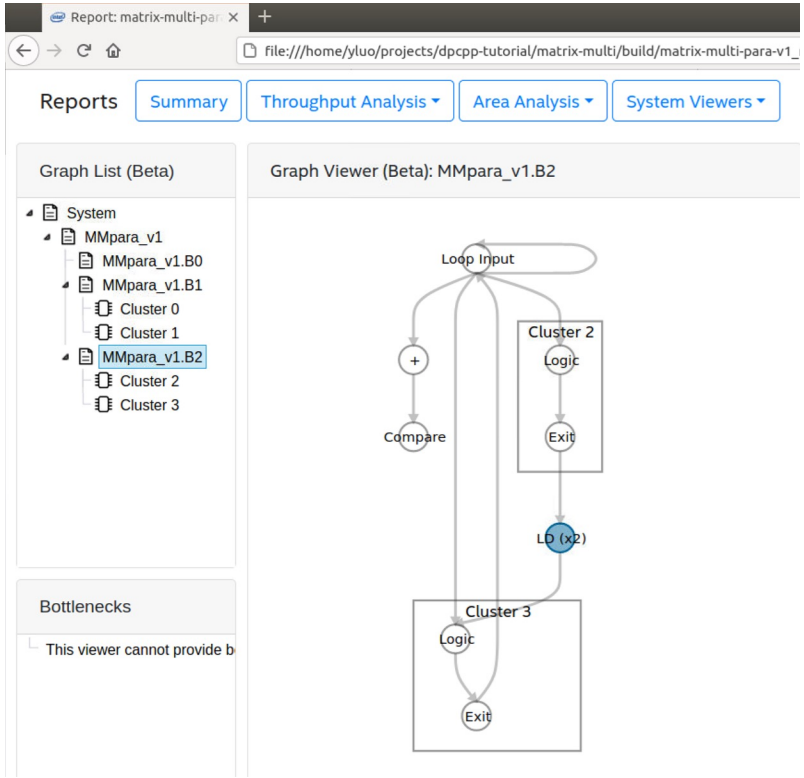
	Device	Static partition	Quartus Fitter: Total Used (Entire System)	Estimated: Kernel system
ALM	427200	89975	TBD	
- ALUT				22471
- REG	1708800	358572	TBD	32268
- MLAB				71
RAM	2713	492	TBD	214
DSP	1518	123	TBD	22

UMASS LOWELL
Learning with Purpose

System Graph Viewer

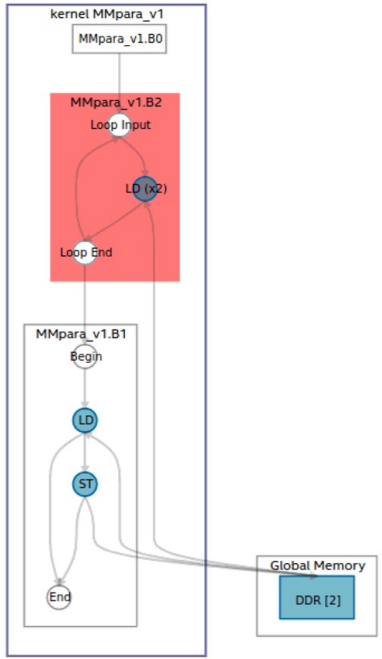


Zoom into a Module



parallel for() v1

Graph Viewer (Beta)

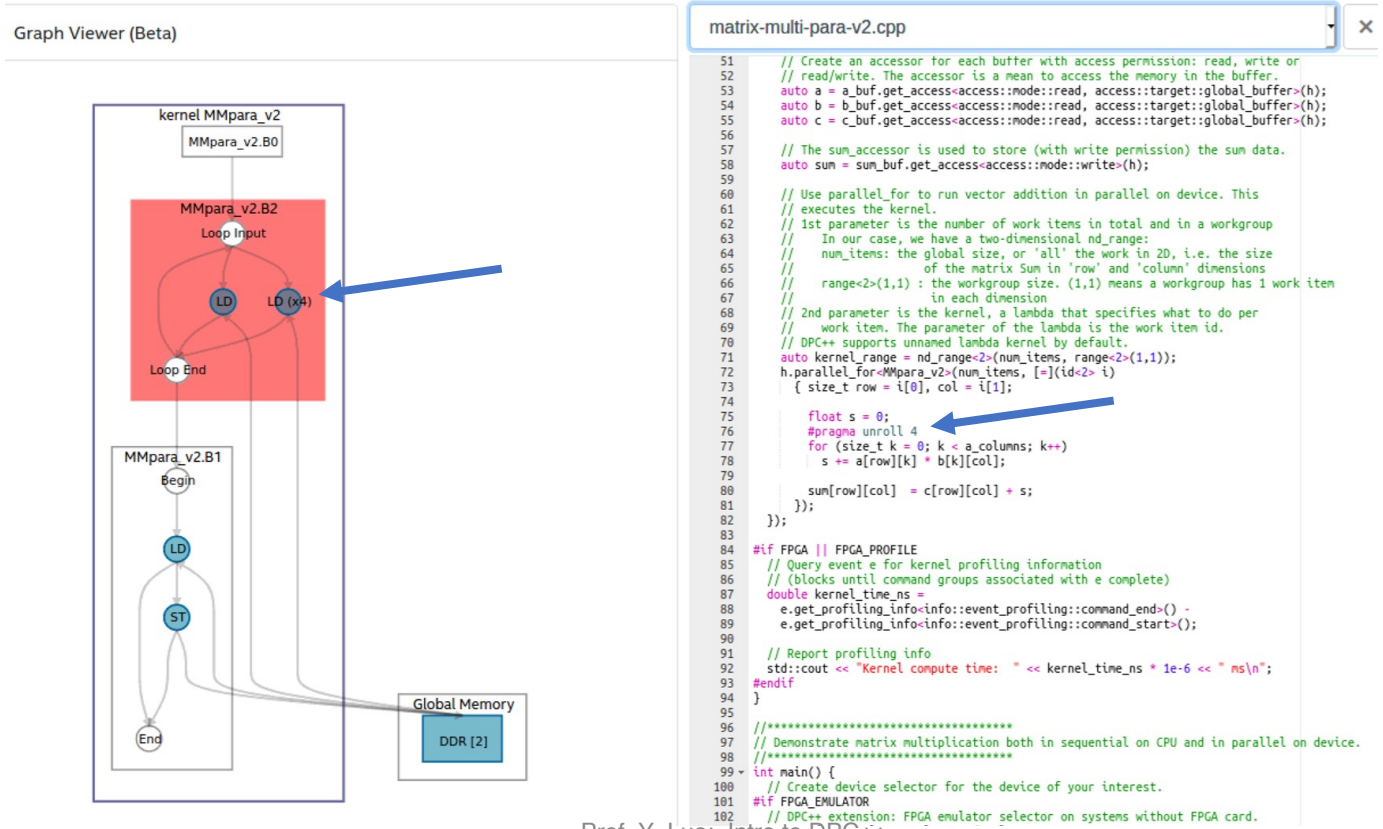


matrix-multi-para-v1.cpp

```
63      //    In our case, we have a two-dimensional nd_range:
64      //    num_items: the global size, or 'all' the work in 2D, i.e. the
          size
65      //                of the matrix Sum in 'row' and 'column'
          dimensions
66      //    range<2>(1,1) : the workgroup size. (1,1) means a workgroup has
          1 work item
67      //                  in each dimension
68      // 2nd parameter is the kernel, a lambda that specifies what to do
          per
69      //    work item. The parameter of the lambda is the work item id.
70      // DPC++ supports unnamed lambda kernel by default.
71      auto kernel_range = nd_range<2>(num_items, range<2>(1,1));
72      h.parallel_for<MMpara_v1>(num_items, [=](id<2> i)
73        { size_t row = i[0], col = i[1];
74
75          float s = 0;
76          //#pragma unroll 2
77          for (size_t k = 0; k < a_columns; k++)
78            s += a[row][k] * b[k][col];
79
80          sum[row][col]  = c[row][col] + s;
81        });
82    });
83
84  #if FPGA || FPGA_PROFILE
85    // Query event e for kernel profiling information
86    // (blocks until command groups associated with e complete)
87    double kernel_time_ns =
88      e.get_profiling_info<info::event_profiling::command_end>() -
89      e.get_profiling_info<info::event_profiling::command_start>();
90
91    // Report profiling info
92    std::cout << "Kernel compute time:  " << kernel_time_ns * 1e-6 << "
          ms\n";
93  #endif
94  }
95
96  //************************************
97  // Demonstrate matrix multiplication both in sequential on CPU and in
       parallel on device.
98  //************************************
99  int main() {
100   // Create device selector for the device of your interest
```

UMASS LOWELL
Learning with Purpose

parallel for() v2

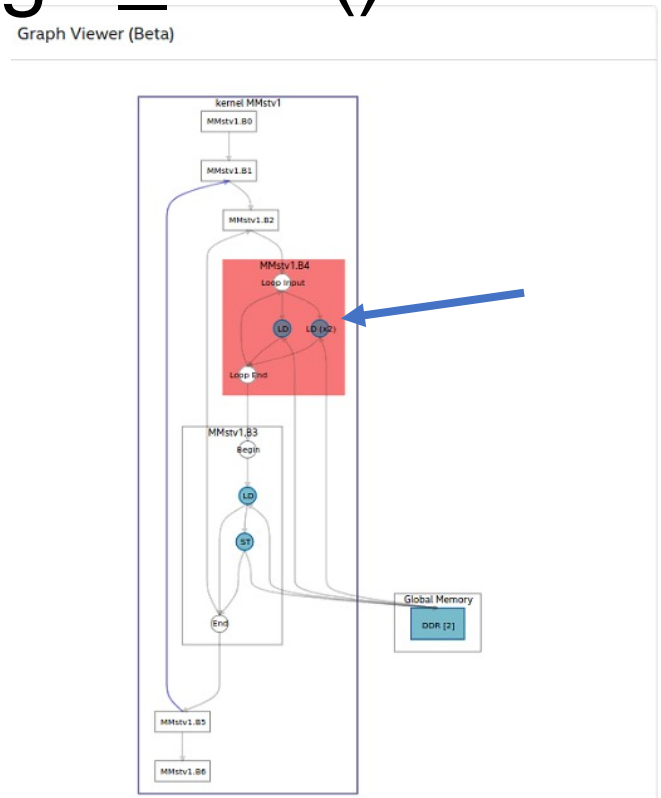


matrix-multi-para-v2.cpp

```
51      // Create an accessor for each buffer with access permission: read, write or
52      // read/write. The accessor is a mean to access the memory in the buffer.
53      auto a = a_buf.get_access<access::mode::read, access::target::global_buffer>(h);
54      auto b = b_buf.get_access<access::mode::read, access::target::global_buffer>(h);
55      auto c = c_buf.get_access<access::mode::read, access::target::global_buffer>(h);
56
57      // The sum_accessor is used to store (with write permission) the sum data.
58      auto sum = sum_buf.get_access<access::mode::write>(h);
59
60      // Use parallel_for to run vector addition in parallel on device. This
61      // executes the kernel.
62      // 1st parameter is the number of work items in total and in a workgroup
63      //    In our case, we have a two-dimensional nd_range:
64      //    num_items: the global size, or 'all' the work in 2D, i.e. the size
65      //                of the matrix Sum in 'row' and 'column' dimensions
66      //    range<2>(1,1) : the workgroup size. (1,1) means a workgroup has 1 work item
67      //                    in each dimension
68      // 2nd parameter is the kernel, a lambda that specifies what to do per
69      //    work item. The parameter of the lambda is the work item id.
70      // DPC++ supports unnamed lambda kernel by default.
71      auto kernel_range = nd_range<2>(num_items, range<2>(1,1));
72      h.parallel_for<MMpara_v2>(num_items, [=](id<2> i)
73        { size_t row = i[0], col = i[1];
74
75          float s = 0;
76          #pragma unroll 4
77          for (size_t k = 0; k < a_columns; k++)
78            s += a[row][k] * b[k][col];
79
80          sum[row][col]  = c[row][col] + s;
81        });
82    });
83
84  #if FPGA || FPGA_PROFILE
85    // Query event e for kernel profiling information
86    // (blocks until command groups associated with e complete)
87    double kernel_time_ns =
88      e.get_profiling_info<info::event_profiling::command_end>() -
89      e.get_profiling_info<info::event_profiling::command_start>();
90
91    // Report profiling info
92    std::cout << "Kernel compute time:  " << kernel_time_ns * 1e-6 << " ms\n";
93  #endif
94  }
95
96  //************************************
97  // Demonstrate matrix multiplication both in sequential on CPU and in parallel on device.
98  //************************************
99  int main() {
100   // Create device selector for the device of your interest.
101 #if FPGA_EMULATOR
102   // DPC++ extension: FPGA emulator selector on systems without FPGA card.
```

UMASS LOWELL
Learning with Purpose

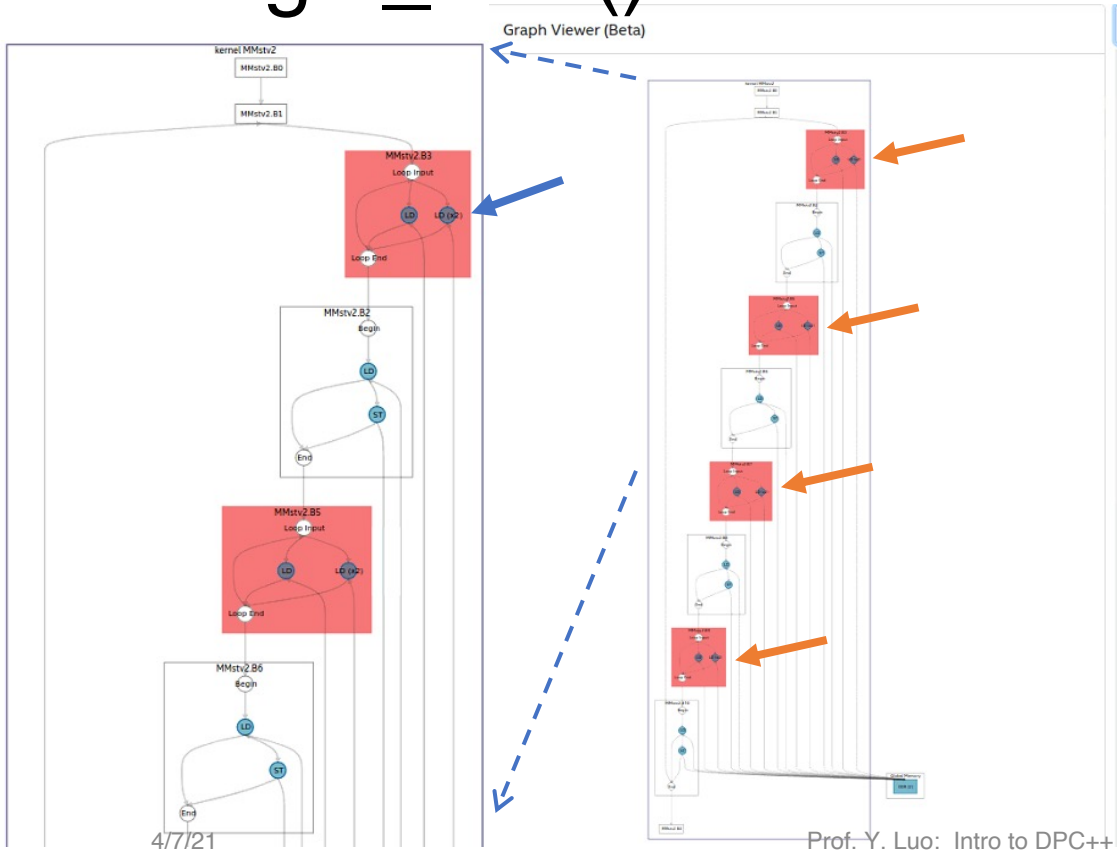
single_task() v1



matrix-multi-st-v1.cpp

```
50   event e = q.submit([&](handler &h) {
51     // Create an accessor for each buffer with access permission: read, write or
52     // read/write. The accessor is a mean to access the memory in the buffer.
53     auto a = a_buf.get_access<access::mode::read, access::target::global_buffer>(h);
54     auto b = b_buf.get_access<access::mode::read, access::target::global_buffer>(h);
55     auto c = c_buf.get_access<access::mode::read, access::target::global_buffer>(h);
56
57     // The sum_accessor is used to store (with write permission) the sum data.
58     auto sum = sum_buf.get_access<access::mode::write>(h);
59
60     // Use single_task to run vector addition in parallel on device. This
61     // executes the kernel.
62     // A kernel that is executed on one thread using NDRange(1,1,1) is enqueued
63     // using the cl::sycl::single_task API:
64     //   single_task<typename kernel_lambda_name>([=](){});
65     h.single_task<MMstv1>([=]() [[intel::kernel_args_restrict]]
66       {
67         size_t row, col;
68         float s = 0;
69         for (row = 0; row < a_rows; row++)
70           for (col = 0; col < b_columns; col++) {
71             #pragma unroll 2
72             for (size_t k = 0; k < a_columns; k++)
73               s += a[row][k] * b[k][col];
74             sum[row][col]  = c[row][col] + s;
75           }
76       });
77   });
78
79 #if FPGA || FPGA_PROFILE
80   // Query event e for kernel profiling information
81   // (blocks until command groups associated with e complete)
82   double kernel_time_ns =
83     e.get_profiling_info<info::event_profiling::command_end>() -
84     e.get_profiling_info<info::event_profiling::command_start>();
85
86   // Report profiling info
87   std::cout << "Kernel compute time:  " << kernel_time_ns * 1e-6 << " ms\n";
88 #endif
89
90 }
91
92
93 //************************************
94 // Demonstrate matrix multiplication both in sequential on CPU and in parallel on device.
95 //************************************
96 int main() {
97   // Create device selector for the device of your interest.
98 #if FPGA_EMULATOR
99   // DPC++ extension: FPGA emulator selector on systems without FPGA card.
100  INTEL::fpga_emulator_selector d_selector;
101 #elif defined(FPGA) || defined(FPGA_PROFILE)
102   // DPC++ extension: FPGA selector on systems with FPGA card.
```

single_task() v2



matrix-multi-st-v2.cpp

```
  // Submit a command group to the queue by a lambda function that contains the
  // data access permission and device computation (kernel).
  event e = q.submit([&](handler &h) {
    // Create an accessor for each buffer with access permission: read, write or
    // read/write. The accessor is a mean to access the memory in the buffer.
    auto a = a_buf.get_access<access::mode::read, access::target::global_buffer>(h);
    auto b = b_buf.get_access<access::mode::read, access::target::global_buffer>(h);
    auto c = c_buf.get_access<access::mode::read, access::target::global_buffer>(h);

    // The sum_accessor is used to store (with write permission) the sum data.
    auto sum = sum_buf.get_access<access::mode::write>(h);

    // Use single_task to run vector addition in parallel on device. This
    // executes the kernel.
    // A kernel that is executed on one thread using NDRange(1,1,1) is enqueued
    // using the cl::sycl::single_task API:
    //   single_task<typename kernel_lambda_name>([=](){});
    h.single_task<MMstv2>([=]() [[intel::kernel_args_restrict]]
      {
        float s = 0;
        #pragma unroll 4
        for (size_t i = 0; i < a_rows * b_columns; i++) {
          size_t row, col;
          row = i / widthC;
          col = i % widthC;
          #pragma unroll 2
          for (size_t k = 0; k < a_columns; k++)
            s += a[row][k] * b[k][col];
          sum[row][col]  = c[row][col] + s;
        }
      });
  });

#if FPGA || FPGA_PROFILE
  // Query event e for kernel profiling information
  // (blocks until command groups associated with e complete)
  double kernel_time_ns =
    e.get_profiling_info<info::event_profiling::command_end>() -
    e.get_profiling_info<info::event_profiling::command_start>();

  // Report profiling info
  std::cout << "Kernel compute time:  " << kernel_time_ns * 1e-6 << " ms\n";
#endif
}


//************************************
// Demonstrate matrix multiplication both in sequential on CPU and in parallel on device.
//************************************
int main() {
  // Create device selector for the device of your interest.
#if FPGA_EMULATOR
  // DPC++ extension: FPGA emulator selector on systems without FPGA card.
  INTEL::fpga_emulator_selector d_selector;
#elif defined(FPGA) || defined(FPGA_PROFILE)
  // DPC++ extension: FPGA selector on systems with FPGA card.
  INTEL::fpga_selector d_selector;
#else
  // The default device selector will select the most performant device.
  default_selector d_selector;
  //cpu selector d selector;
```

DPC++ Design Analysis (II): Analyze Runtime Profiling Data

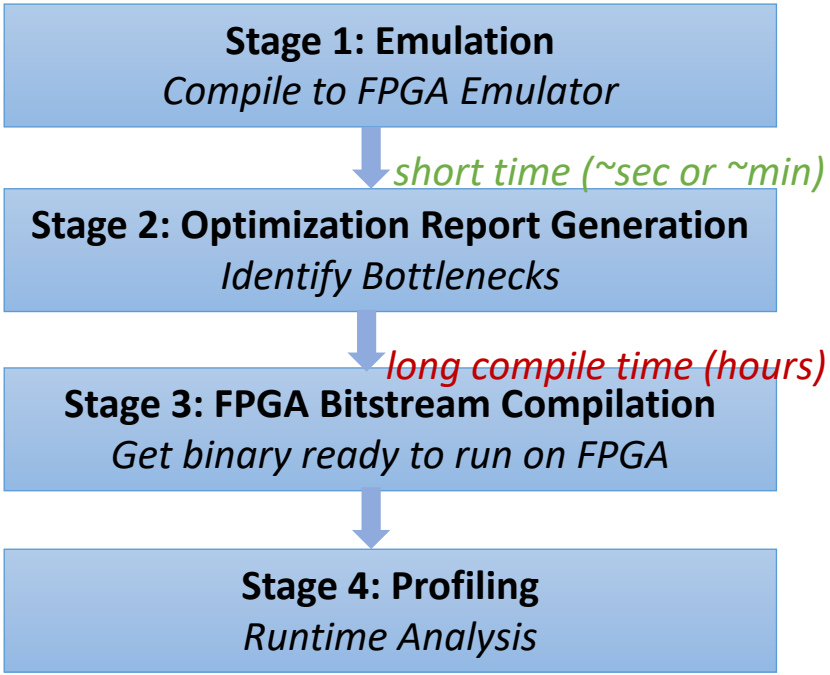
A part of the DPC++ Tutorial Series

Prof. Yan Luo

Acknowledgement
This work is supported by Intel Corporation 2020-2021

UMASS LOWELL
Learning with Purpose

Analyze your Design before Optimization



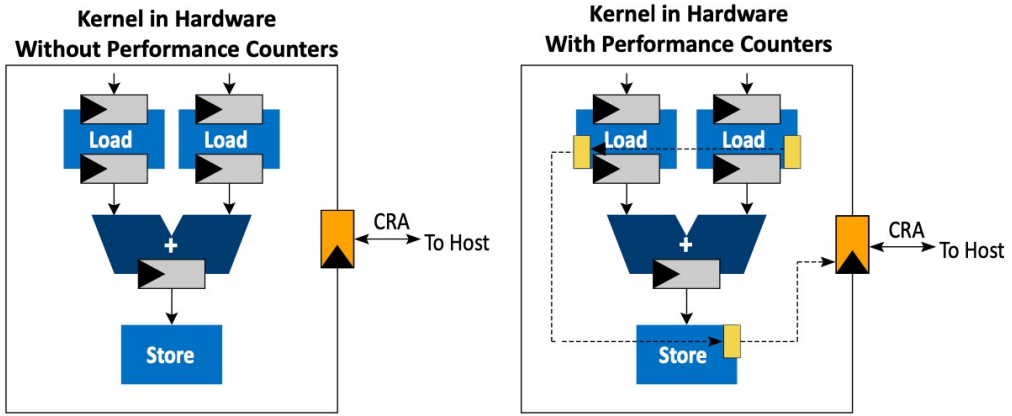
1. Make sure the design is functionally correct
2. Look for bottlenecks through compilation reports and revise design if necessary
3. Generate FPGA binary for execution on hardware
4. Run profiling to analyze runtime performance

UMASS LOWELL
Learning with Purpose

Compile for profiling

- Intel FPGA Dynamic Profiler
  - User performance counters to collect kernel performance data
    - Stall, occupancy, idle, bandwidth usage, etc.
  - The data can be viewed in Intel VTune Profiler

To enable profiling, add `–Xsprofile` flag to compiler command



UMASS LOWELL
Learning with Purpose

Profiling data collection

- Execute FPGA binary with profiling tool. Do not generate json file

```
aocl profile -no-json ./matrix-multi-para-v1.fpga_profile
```

- Generate json results file using profile.mon

```
aocl profile -no-run profile.mon ./matrix-multi-para-v1.fpga_profile
```

- Load profile results folder to Intel VTune toolkit
  - Need to put the files (profile.mon and profile.json) in a folder
  - Check “import multiple trace files from a directory” option when importing profiling data in VTune.

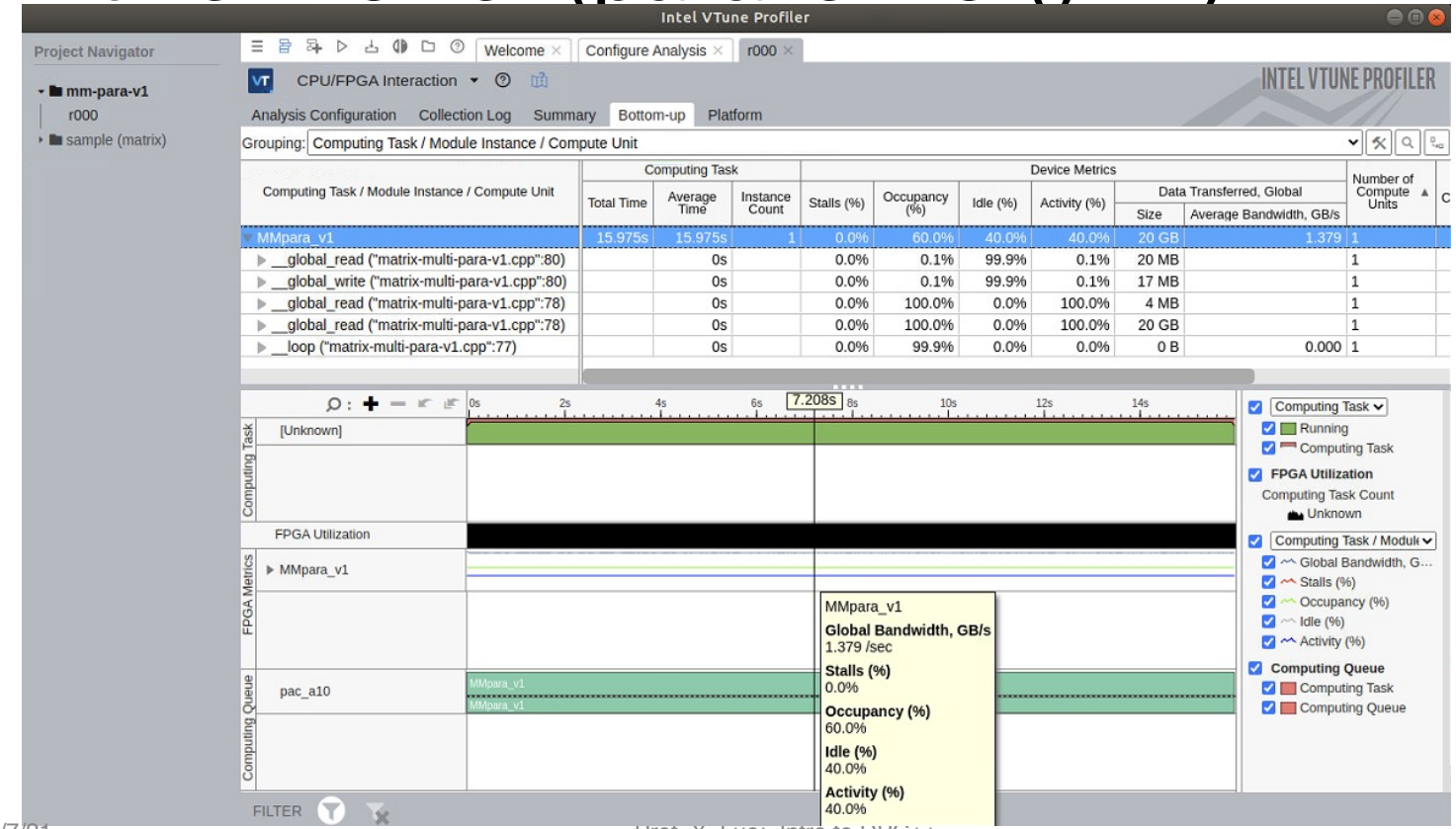
UMASS LOWELL
Learning with Purpose

Demo: Profiling Analysis

Run-time profiling analysis on Matrix Multiplication example

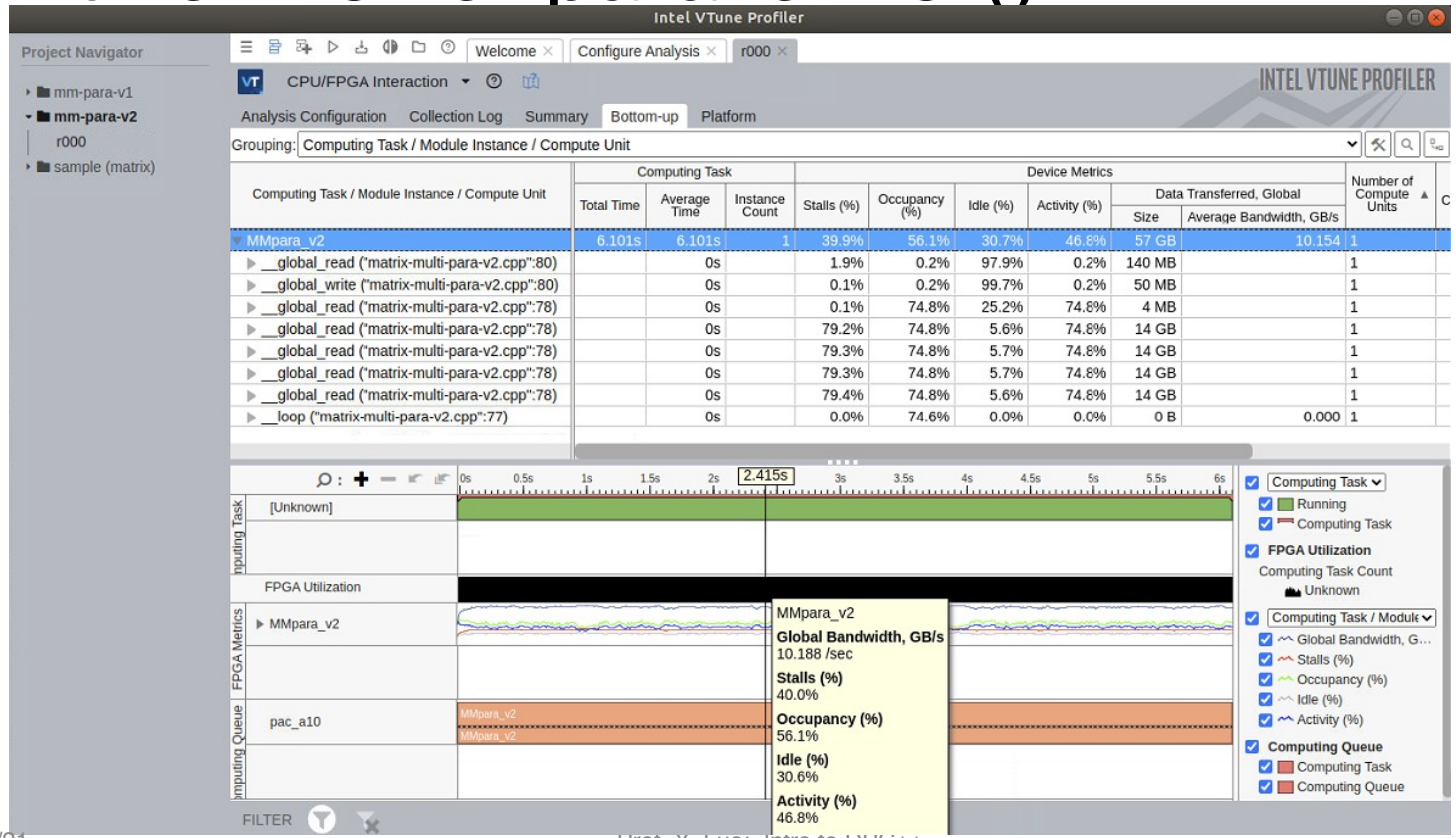
UMASS LOWELL
Learning with Purpose

VTune Profiler (parallel_for() v1)



UMASS LOWELL
Learning with Purpose

VTune Profiler parallel_for() v2



UMASS LOWELL
Learning with Purpose

DPC++ Design Optimization: Reducing Global Memory Access with Local Memory

A part of the DPC++ Tutorial Series

Prof. Yan Luo

Acknowledgement
This work is supported by Intel Corporation 2020-2021

UMASS LOWELL
Learning with Purpose

Motivation

- We observe a large amount of global memory access
  - Matrix-multi in those implementations are a memory-bound
  - The total sizes of matrice A, B = 25.6MB

    800x1600 and 1600x3200 floating point numbers in A, B respectively
  - We read 20GB from global memory!
- The matrix-multi calculation use the same element from source matrices multiple times
  - Reads an entire row of A and an entire column of B to calculate **each** value of C
  - Adjacent elements in C require reading much of the same data (row from matrix A or a column from matrix B)

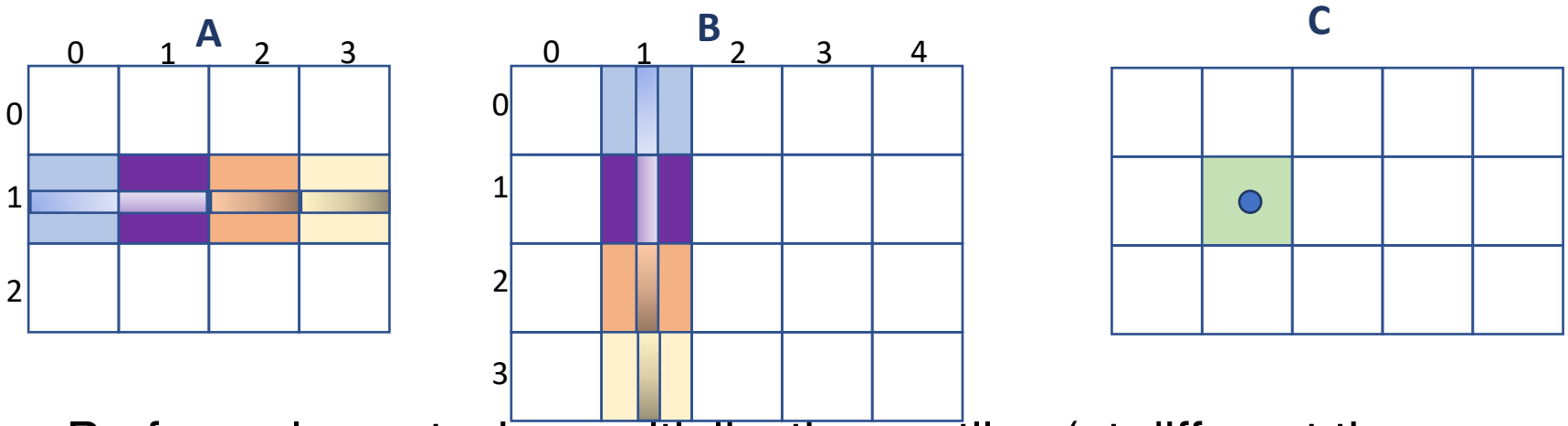
UMASS LOWELL
Learning with Purpose

Tiling

- Tiling is buffering data onto fast on-chip storage where it will be repeatedly accessed (caching)
  - Very common technique
  - Used when multiple instances need to access overlapping parts of data set
- Data must be partitioned into blocks to fit into local memory
  - Only work-items within a workgroup can share data
  - Local memory size and geometry set at compile time
  - Workgroup sizes (block sizes) must be known at compile time

UMASS LOWELL
Learning with Purpose

Tiling Illustration



- Perform element-wise multiplication on tiles (at different times or in parallel)

$$A_{10} \otimes B_{01} \quad A_{11} \otimes B_{11} \quad A_{12} \otimes B_{21} \quad A_{13} \otimes B_{31}$$

- The accumulated sum is from multiple tile-wise multiplications

$$C_{11} = A_{10} \otimes B_{01} + A_{11} \otimes B_{11} + A_{12} \otimes B_{21} + A_{13} \otimes B_{31}$$

UMASS LOWELL
Learning with Purpose

Overview of the Design

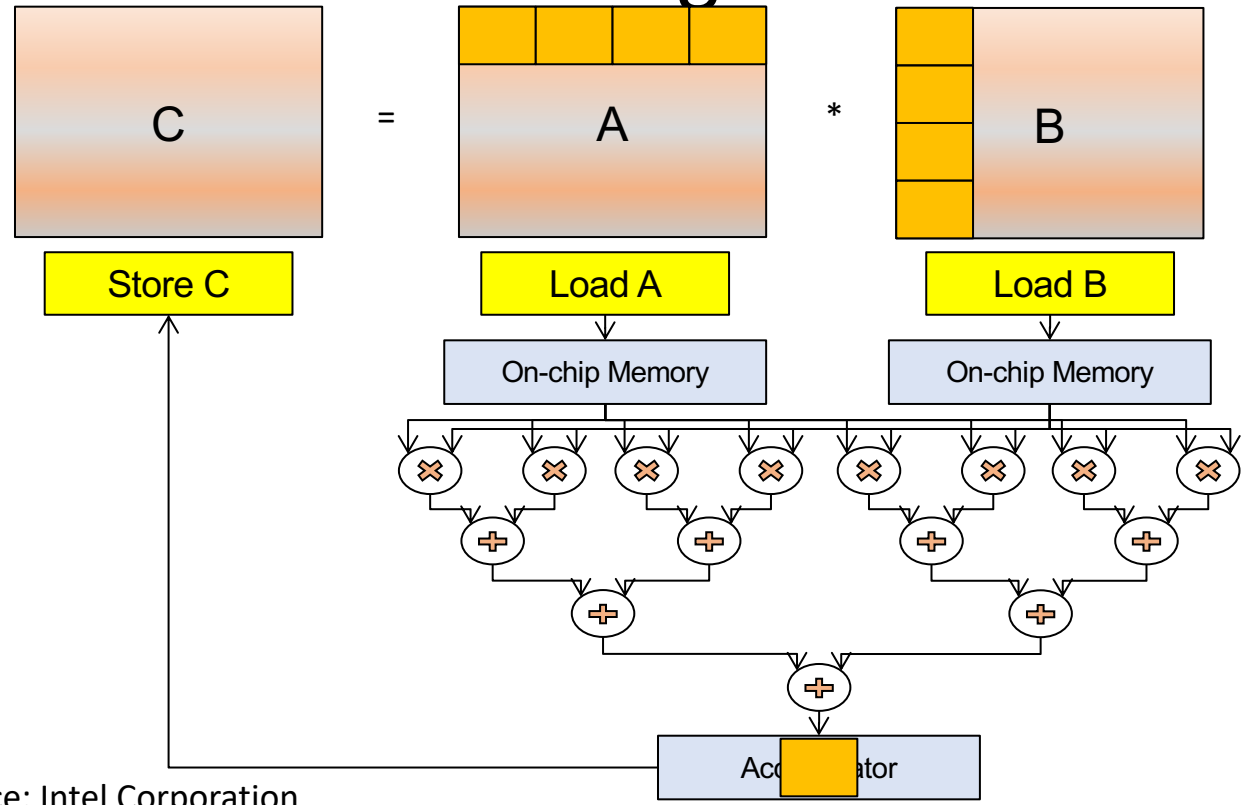


Diagram source: Intel Corporation

Implementation Details

- Calculate the number of rounds for tile-wise multiplication
- Create a single-task kernel
- Allocate local memory for tiles
- Loop unrolling optimization

UMASS LOWELL
Learning with Purpose

Tile-wise multiplication kernel (step 1)

- Calculate the number of tiles, title row and column ID

```
for(int i=0; i < a_rows*a_columns/(BLOCK_SIZE*BLOCK_SIZE); i++) {
  // the block indices of the block in A
  auto block_row_a = i / (a_columns/BLOCK_SIZE);
  auto block_col_a = i % (a_columns/BLOCK_SIZE);
  // we need to calculate dot-product with all the blocks in B where
  // the row number is equal to block_col_a
  auto block_row_b = block_col_a;
  for(int j=0; j<b_columns/BLOCK_SIZE; j++)
  {
```

……

UMASS LOWELL
Learning with Purpose

Tile-wise multiplication kernel (step 2)

- Allocation of local memory for tile
  - BLOCK_SIZE x BLOCK_SIZE floats
  - Need one local memory block to hold a tile from A, B (and D)

```
// allocate local memory to hold a block of data from A, B
[[intel::numbanks(NUM_BANKS), intel::bankwidth(BANK_WIDTH)]] float local_mem_a[BLOCK_SIZE][BLOCK_SIZE];
[[intel::numbanks(NUM_BANKS), intel::bankwidth(BANK_WIDTH)]] float local_mem_b[BLOCK_SIZE][BLOCK_SIZE];
[[intel::numbanks(NUM_BANKS), intel::bankwidth(BANK_WIDTH)]] float local_mem_d[BLOCK_SIZE][BLOCK_SIZE];
```

- Note NUMBANKS and BANK_WIDTH
  - Large numbers help but are limited by the FPGA's MLAB resources.

UMASS LOWELL
Learning with Purpose

Tile-wise multiplication kernel (step 3)

- Then load the tile data from global memory to local memory for computation
  - DPC++ compiler optimize with pipelined Load-Store units

```
// load blocks of data to local memory from global memory
for (m=0; m < BLOCK_SIZE; m++)
  for ( n=0; n < BLOCK_SIZE; n++)
  {
    local_mem_a[m][n] = a[block_row_a*BLOCK_SIZE + m][block_col_a*BLOCK_SIZE + n];
    local_mem_b[m][n] = b[block_row_b*BLOCK_SIZE + m][j*BLOCK_SIZE + n];
    local_mem_d[m][n] = d[block_row_a*BLOCK_SIZE + m][j*BLOCK_SIZE + n];
  }
```

UMASS LOWELL
Learning with Purpose

Tile-wise multiplication kernel (step 4)

- Element wise multiplication and accumulation within a Tile

```
// element-wise multiplication and accumulation
for (m=0; m < BLOCK_SIZE; m++)
  for ( n=0; n < BLOCK_SIZE; n++) {
    s = 0;
    #pragma unroll
    for (k=0; k < BLOCK_SIZE; k++)
      s += local_mem_a[m][k] * local_mem_b[k][n];
    // add to Matrix D
    // the corresponding row and col in D
    row = block_row_a * BLOCK_SIZE + m;
    col = j * BLOCK_SIZE + n;
    //d[row][col] += s;
    local_mem_d[m][n] += s;
  }
```

  - Note that we are doing D = A x B + C. so the final accumulation is D.
  - We will add C to D at a later step.

UMASS LOWELL
Learning with Purpose