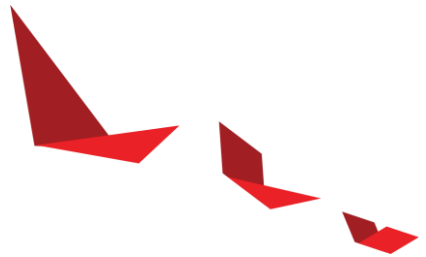
XILINX

Design Methodology for Developing Accelerated Applications

Agenda

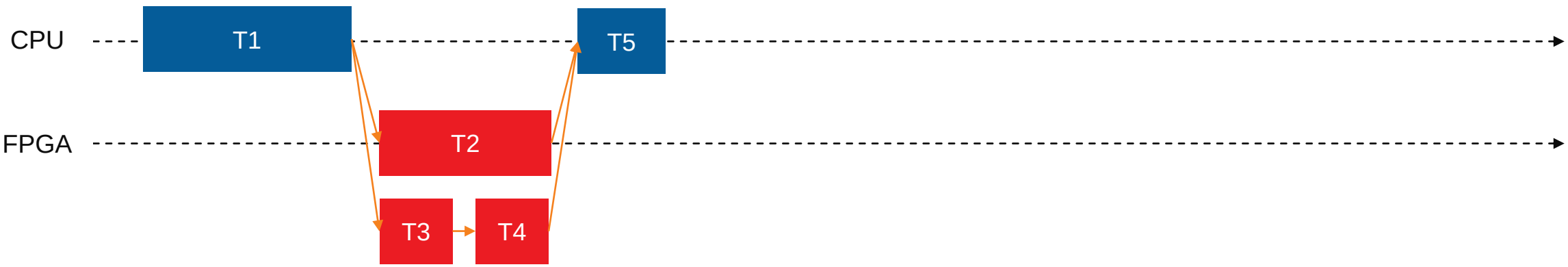


- Design Methodology
- Architecting the Application
- Developing Efficient Kernels
- Example Walk-Through

FPGA Acceleration : Boosting Application Performance

Without acceleration – Serial execution

With FPGA acceleration – Massively parallel execution, within and across functions



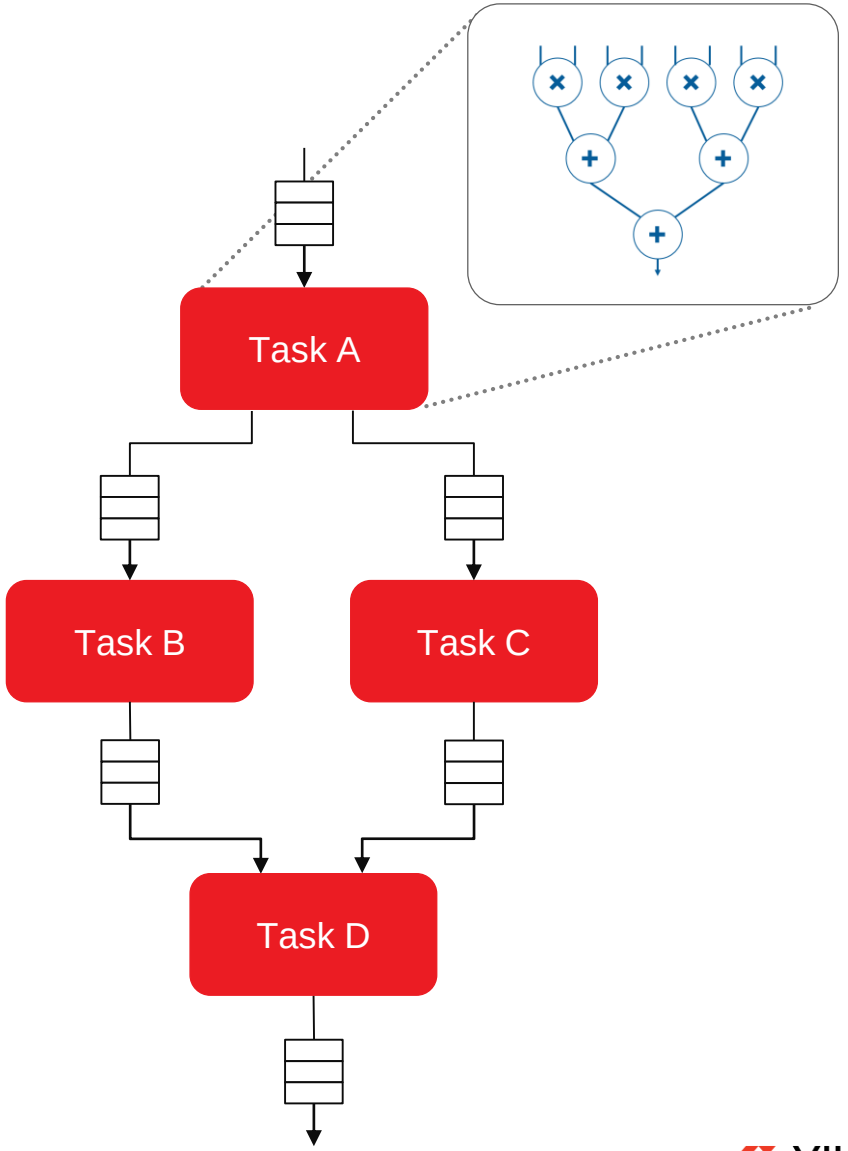
XILINX.

Adaptable Architectures – The FPGA Advantage

CPU	GPU	FPGA/ACAP
Fixed architecture	Fixed architecture	Adaptable Architecture
Predefined instruction set	Predefined instruction set	No fixed instruction set
Fixed memory hierarchy	Fixed memory hierarchy	Customizable memory hierarchy
Thread-level parallelism	SIMD parallelism	Excels at all types of parallelism

High-Performance FPGA Applications: Think “Parallel”

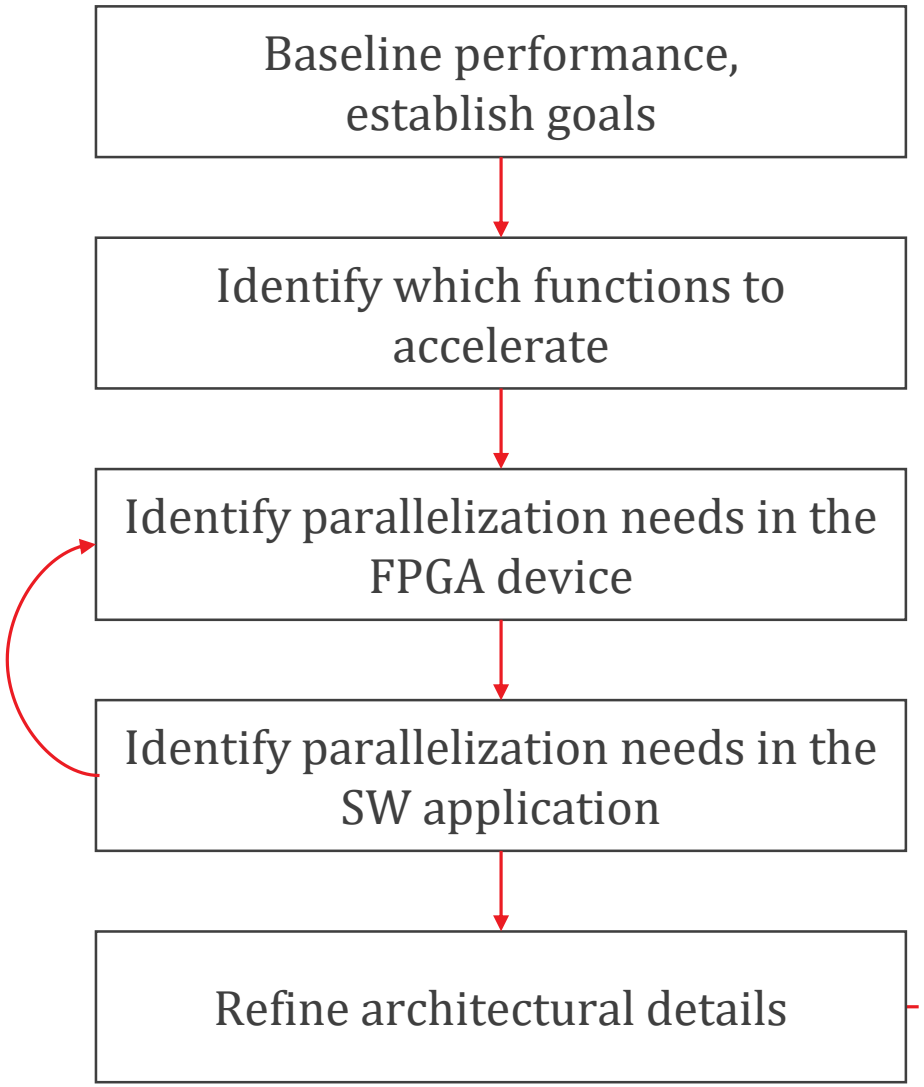
- Data-level parallelism
  - Processing different blocks of a data set in parallel
- Task-level parallelism
  - Executing different tasks in parallel
  - Executing different tasks in a pipelined fashion
- Instruction-level parallelism
  - Parallel instructions (superscalar)
  - Pipelined instructions
- Bit-level parallelism
  - Custom word width



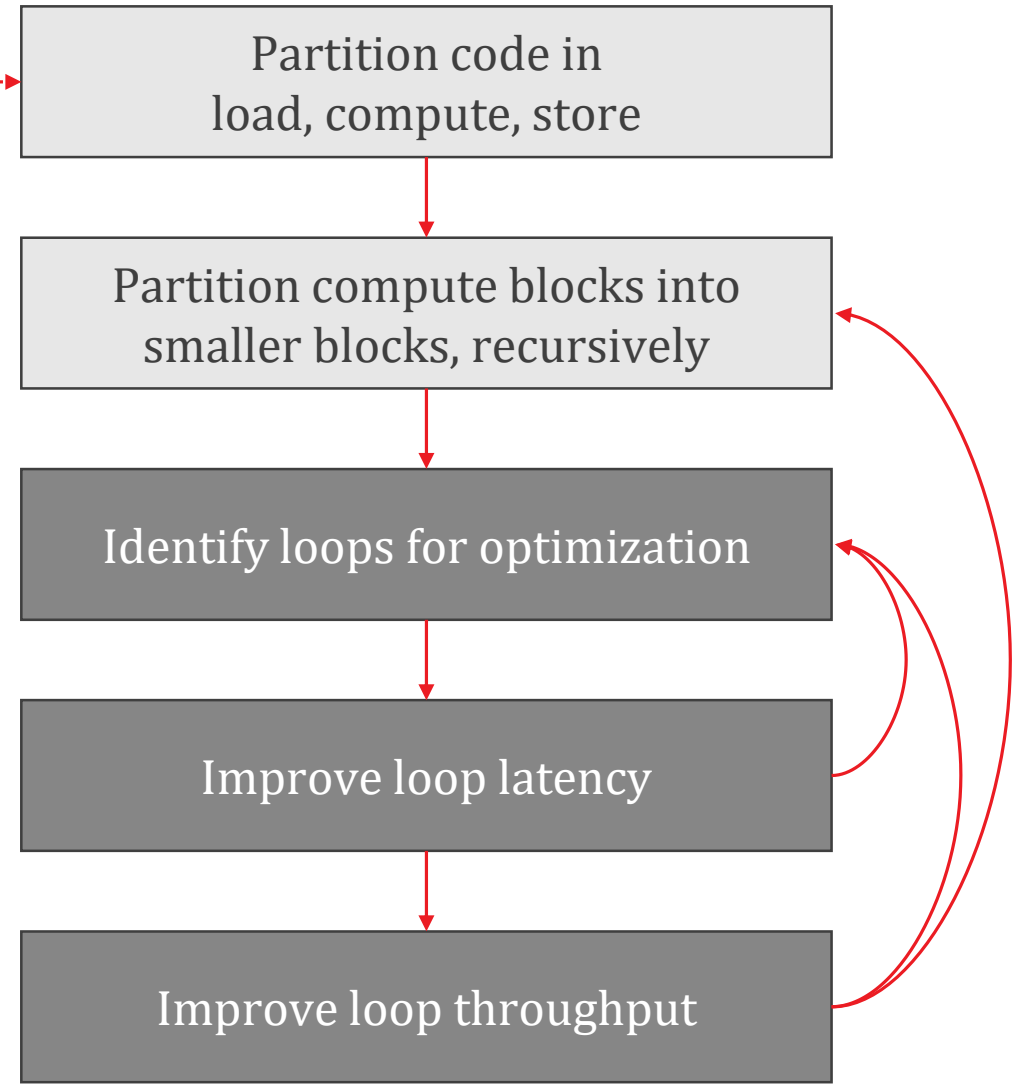
Methodical Approach Needed for Successful Results

- FPGA Acceleration ≠ Traditional SW Development
  - Programming the desired functionality on a pre-defined architecture
  - Programming an architecture to implement the desired functionality
- Xilinx provides a practical, step-by-step, methodology to achieve acceleration
  - Architecting the Application
  - Developing Efficient Kernels
- Available on-line and supported by comprehensive tutorials

Architecting the Application



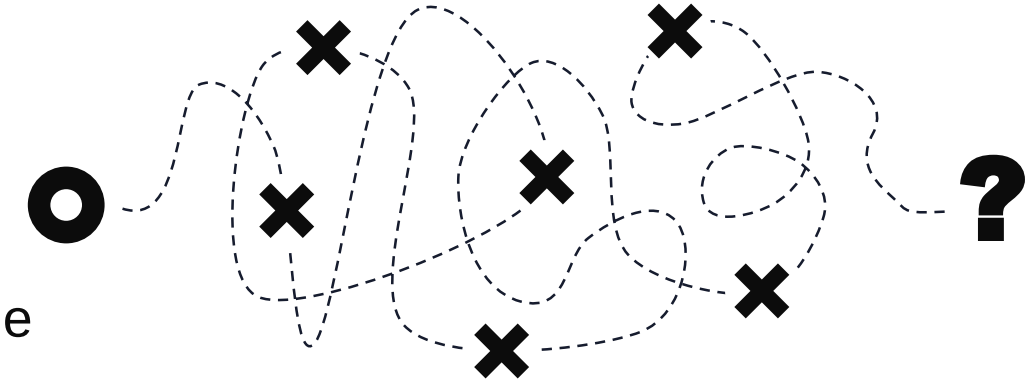
Developing C/C++ Accelerators



XILINX.

Know Your Application, Plan For Success

- Vitis generates optimized hardware from C/C++… but will not transform an algorithm into another one
  - eg: won't implement "bubble sort" from "quick sort"
- Algorithm selection has a major impact on achievable performance
  - Directly influences data access locality and potential for computational parallelism
- Understand your code to chart a path to success
  - Understand data dependencies
  - Determine where and how to achieve parallelism
  - Determine how data should flow for optimal performance
  - Application-level and Kernel-level



Trial and Error

Analysis and Planning

Leverage Libraries Whenever Possible

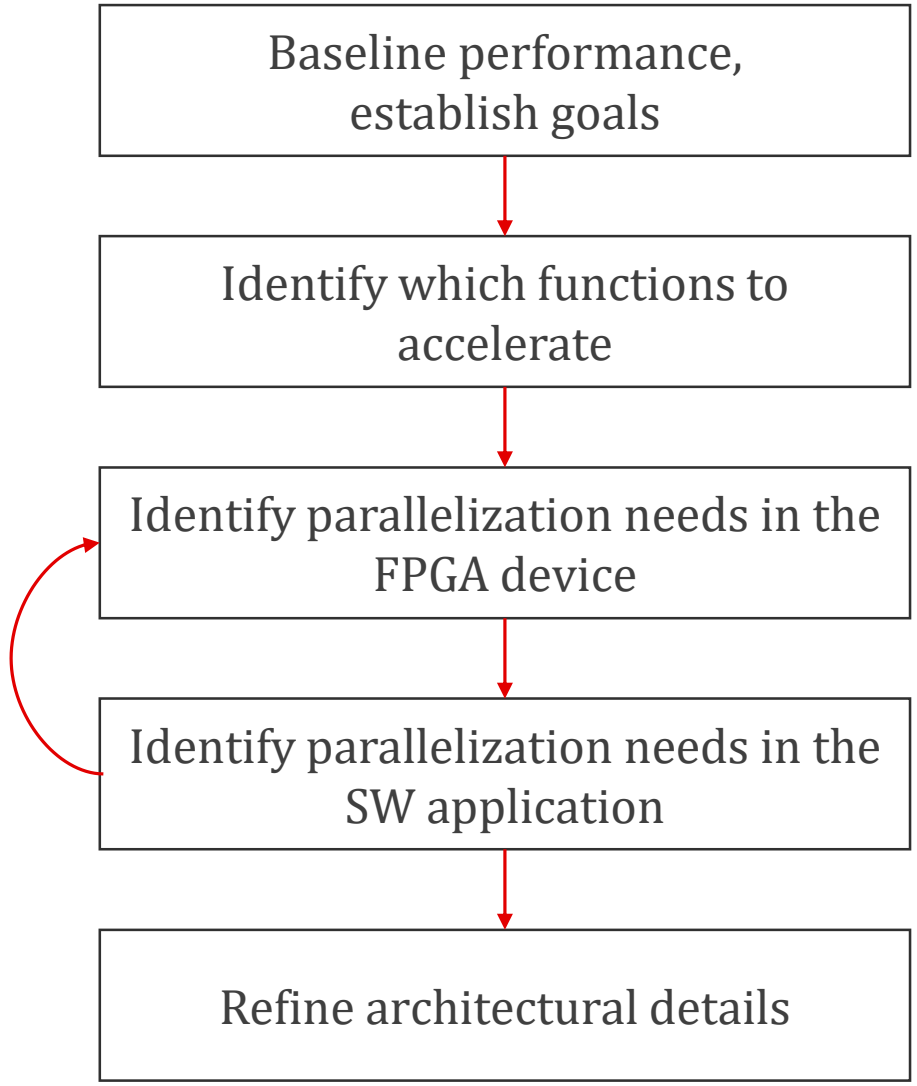
- Vitis Acceleration Libraries facilitate development of accelerated applications
- Rich set of acceleration functions
- Carefully optimized by Xilinx experts
- Comprehensive documentation
- Open source, Apache 2.0 license

✓	Linear algebra
✓	Matrix decomposition
✓	Linear and eigen value solvers
✓	Fast Fourier Transforms
✓	Encryption and hashing
✓	Compression algorithms
✓	Database functions
✓	Quantitative finance
✓	OpenCV computer vision
✓	Efficient I/O accesses

Architecting the Application

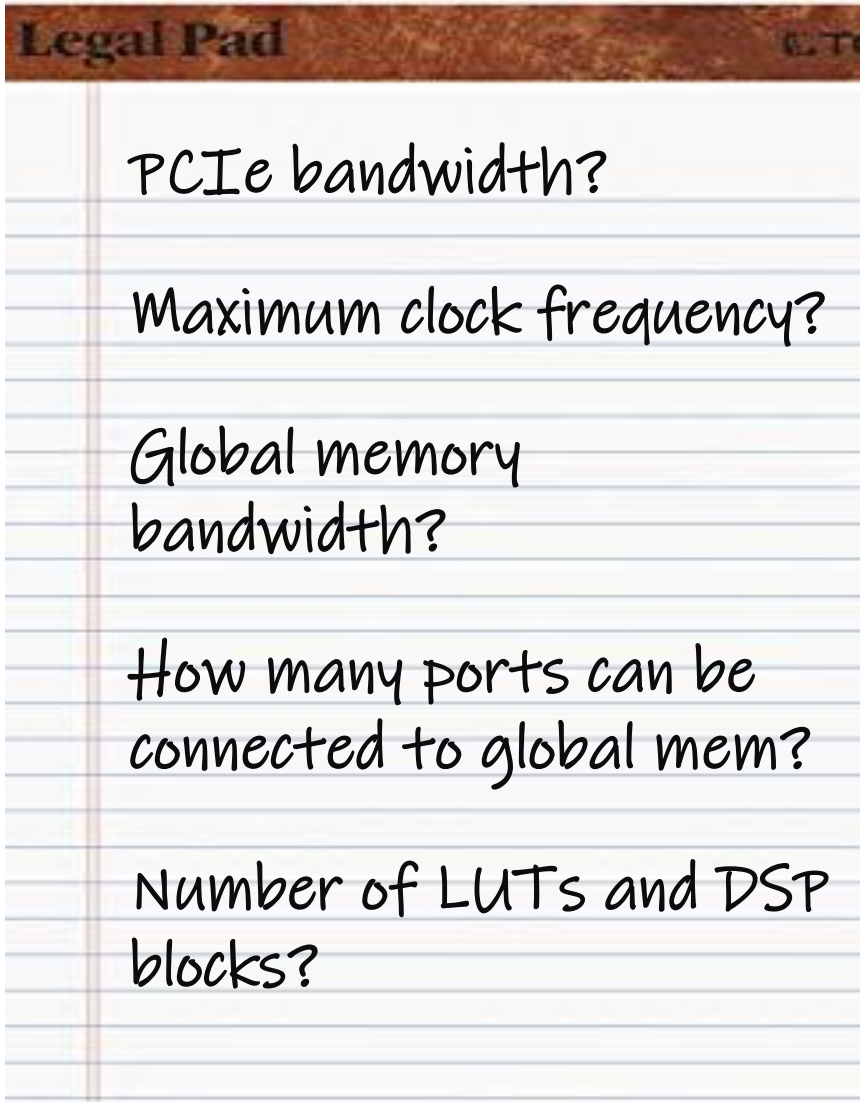
Architecting the Application

- Determine which software functions should be mapped to FPGA kernels
- Determine how much parallelism is needed, and how it should be delivered
  - Parallelism within compute units
  - Parallelism across of compute units
- Architect the SW application for performance
  - Minimize CPU idle time
  - Keep FPGA accelerators utilized
  - Optimize data transfers to and from the FPGA



Know the Constraints

- FPGA are highly configurable, but not infinite
- Start by learning the specs of your FPGA card
- Datasheets, xbutil and Github examples are valuable resources to learn more about the card
- Know what will be the performance ceiling of your application
  - eg: On Alveo, PCIe bandwidth is the upper bound for throughput



Identify Functions with Acceleration Potential

- Profile your application and measure the Throughput and Computational Intensity of key functions

$$Computational\ Intensity = \frac{Volume\ of\ Operations}{Volume\ of\ Memory\ Accesses}$$

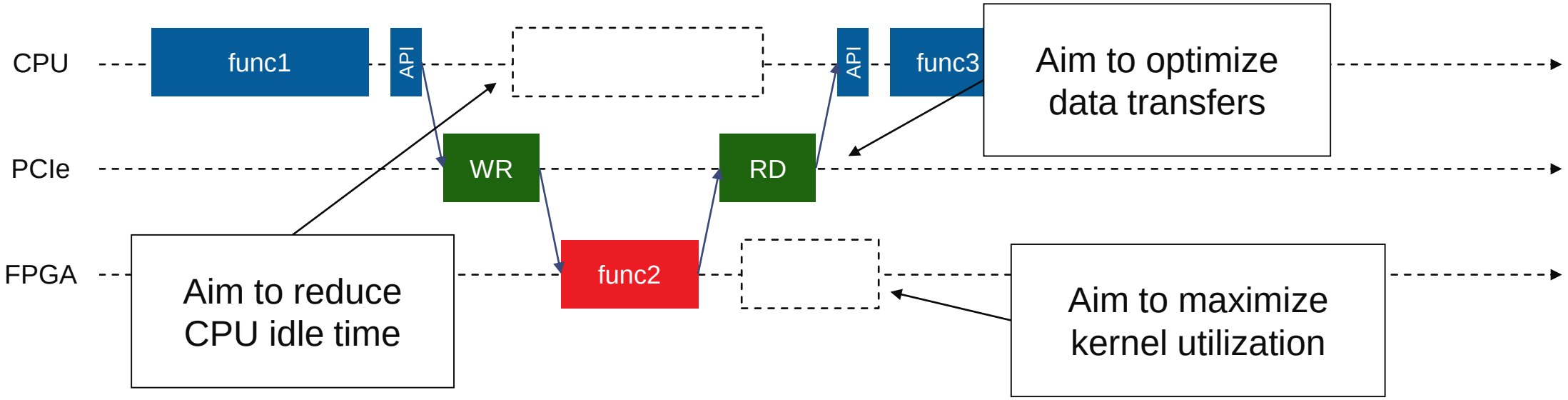
- Good candidates for accelerations are functions with high computational intensity
- The higher the computational intensity, the less the overhead cost of moving data for processing (relative to overall running time)

Estimate How Much Hardware Parallelism is Needed

$$Est.Parallelism\ Needed = \frac{Throughput\ Goal\ *\ Computational\ Intensity}{Fmax}$$

- Primary ways to achieve parallelism in FPGAs
  - Executing operations in parallel within a kernel
  - Executing multiple kernel in parallel within the system
  - Overlapping execution of tasks within a kernel and/or within the system
- Determine which combination of approaches is best suited to your application
  - This is design specific and mostly determined by data access profiles

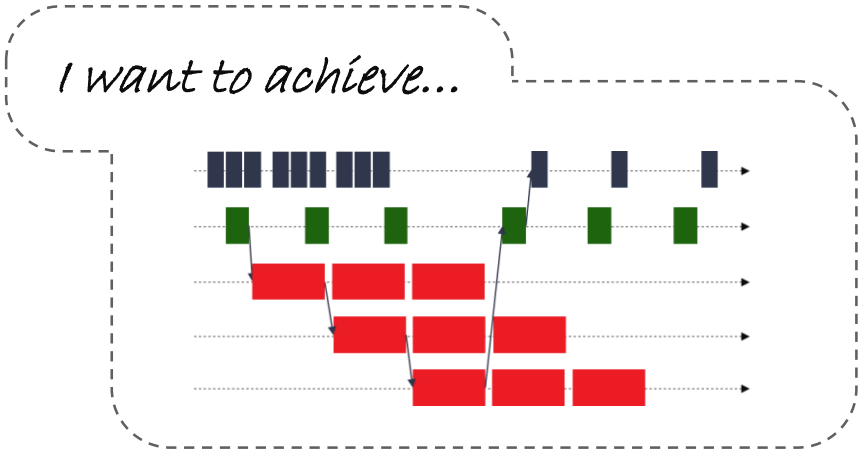
Architect the Software Application for Parallelism

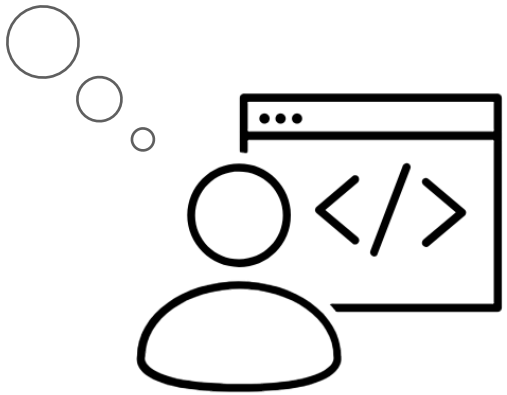


- Minimize CPU idle time and do other tasks while the FPGA kernels are running
- Keep the kernels active, performing new computations as often as possible
- Optimize data transfers to and from the FPGA
- Threads and asynchronous programming are helpful to achieve this

Conceptualize the Desired Application Timeline

- You should now have a good understanding of
  - What functions needs to be accelerated
  - What parallelism is needed to meet performance goals
  - How parallelism will be delivered
- Summarize this information in the form of an expected application timeline
  - Represent performance and parallelization in action
- Vitis generates timeline views from application runs
- Having expected results to compare against is important for the optimization process

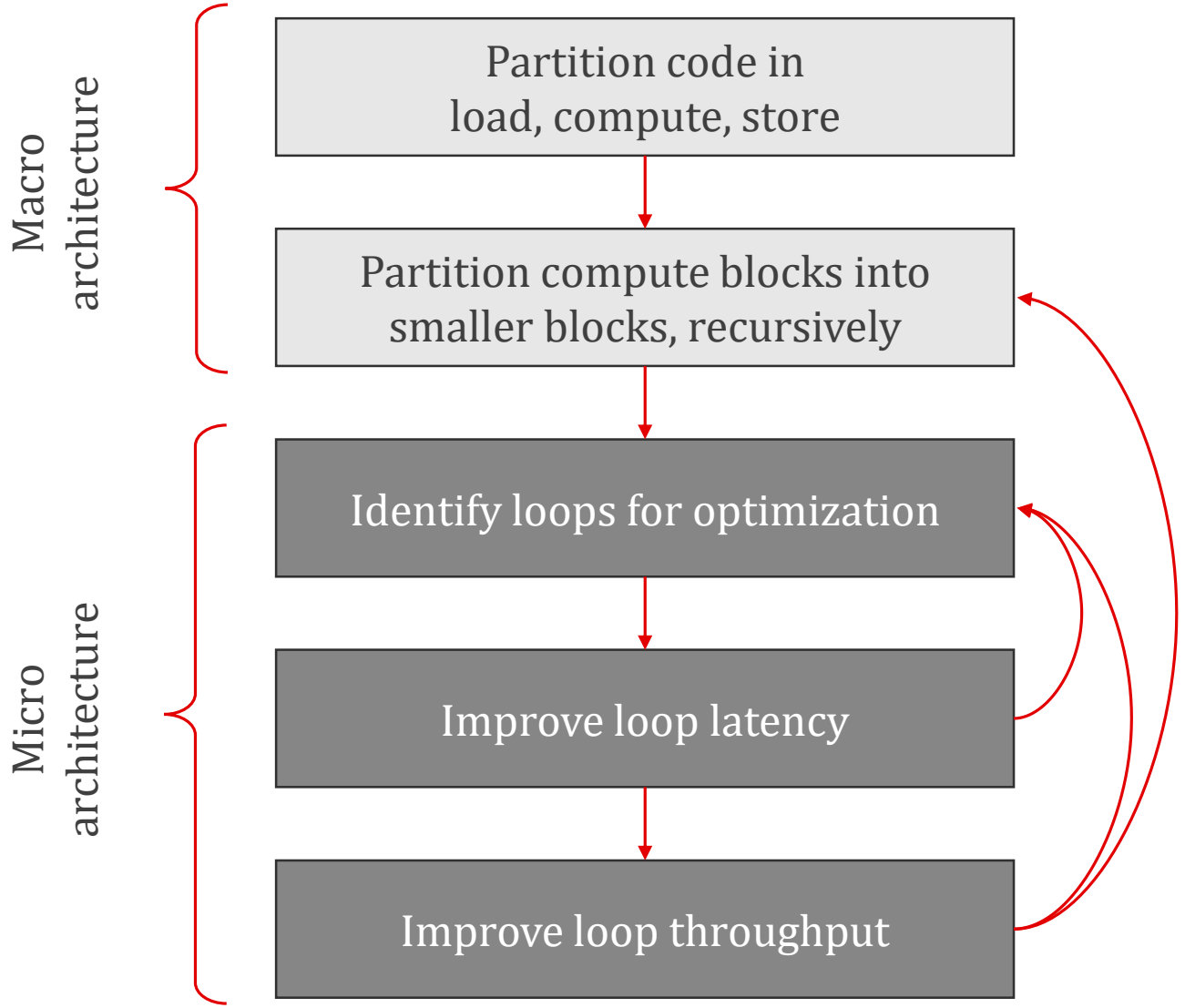




Developing Efficient Kernels

Developing Efficient Kernels

- Start with a clear understanding of where and how you will implement parallelism
- Structure the code for efficient data movement
- Structure the code for task-level parallelism and pipelining
- Optimize loops for latency and throughput

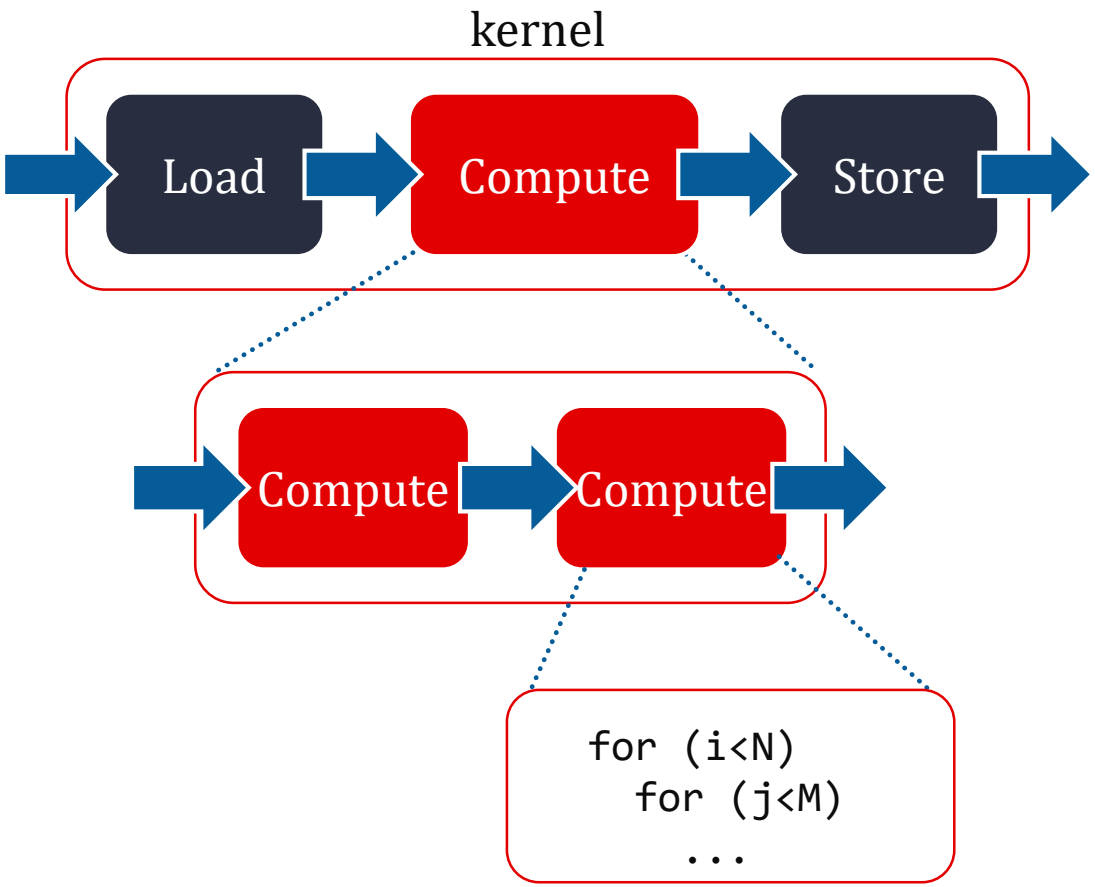


The Principles of HLS Code Development

I/O Accesses	I/O accesses are expensive, encapsulate them in dedicated functions for optimization purposes
Task-level Parallelism	Sequential Function calls can be made to execute in parallel in HW, if data dependencies allow it Sequential Loops will execute sequentially in HW
Instruction Parallelism	Instruction level parallelism is achieved at the loop level, using pragmas for unrolling and pipelining

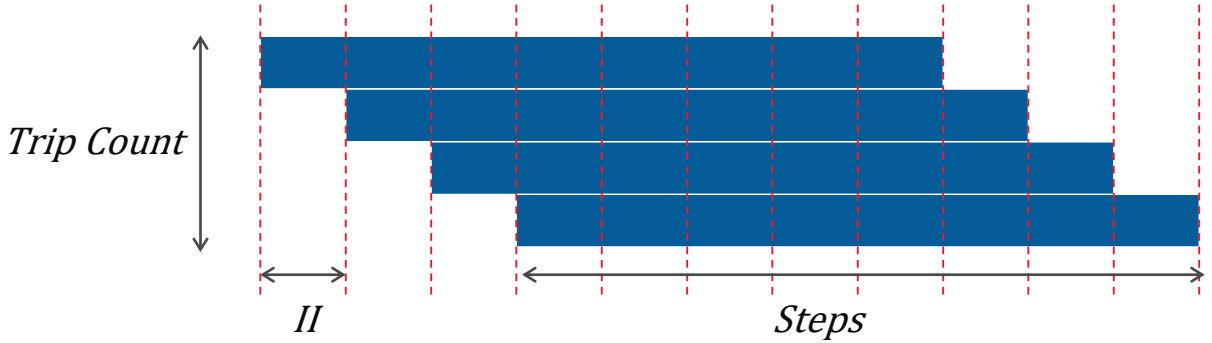
Code the Macro-Architecture of the Kernel

- Partition kernel code design into load, compute, store stages
- For load/store blocks, write tightly nested loops to infer bursting transactions to global memory
- Partition compute block into smaller blocks, recursively
- Estimate throughput of each blocks
  - Should be greater than or equal to throughput goal
- Lowest-level block should only contains a single loop nest or loops that need to run sequentially



Optimize Loops to Meet Performance Target (1/2)

```
for (i<N)
  for (j<M)
    ...
```



$$Loop\ Latency = \left((Trip\ Count\ - 1) * II + Steps\right) * Clock\ Period$$

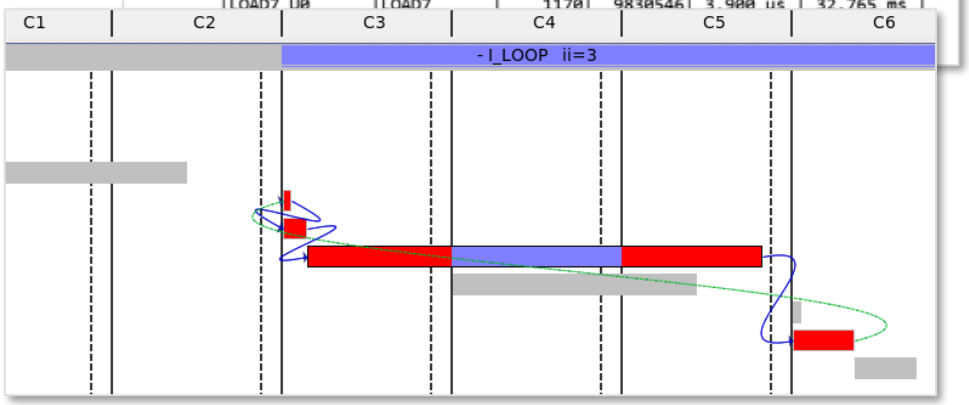
- Ways to improve performance of a loop (assuming fixed clock period) :
  - Reduce the Trip Count, so that the loop performs fewer iterations
  - Reduce the Initiation Interval (II), so that loop iterations can start earlier
  - Reduce the number of Steps in the loop to take less time to perform one iteration

Optimize Loops to Meet Performance Target (2/2)

- Use Vitis reports and visualization tools to analyze loop profiles and identify bottlenecks
- Unroll loops to reduce Trip Count and leverage instruction-level parallelism
- Eliminate data access contentions to improve II
  - Partition and reshape arrays when possible
  - Create internal caching structures
- Eliminate loop-carried dependencies to improve II
  - Use pragmas to disambiguate memory dependencies
  - Look for algorithmic ways of breaking long feedback paths
- Move to next block when throughput goal is met

```
for (i<N)
  for (j<M)
    ...
```

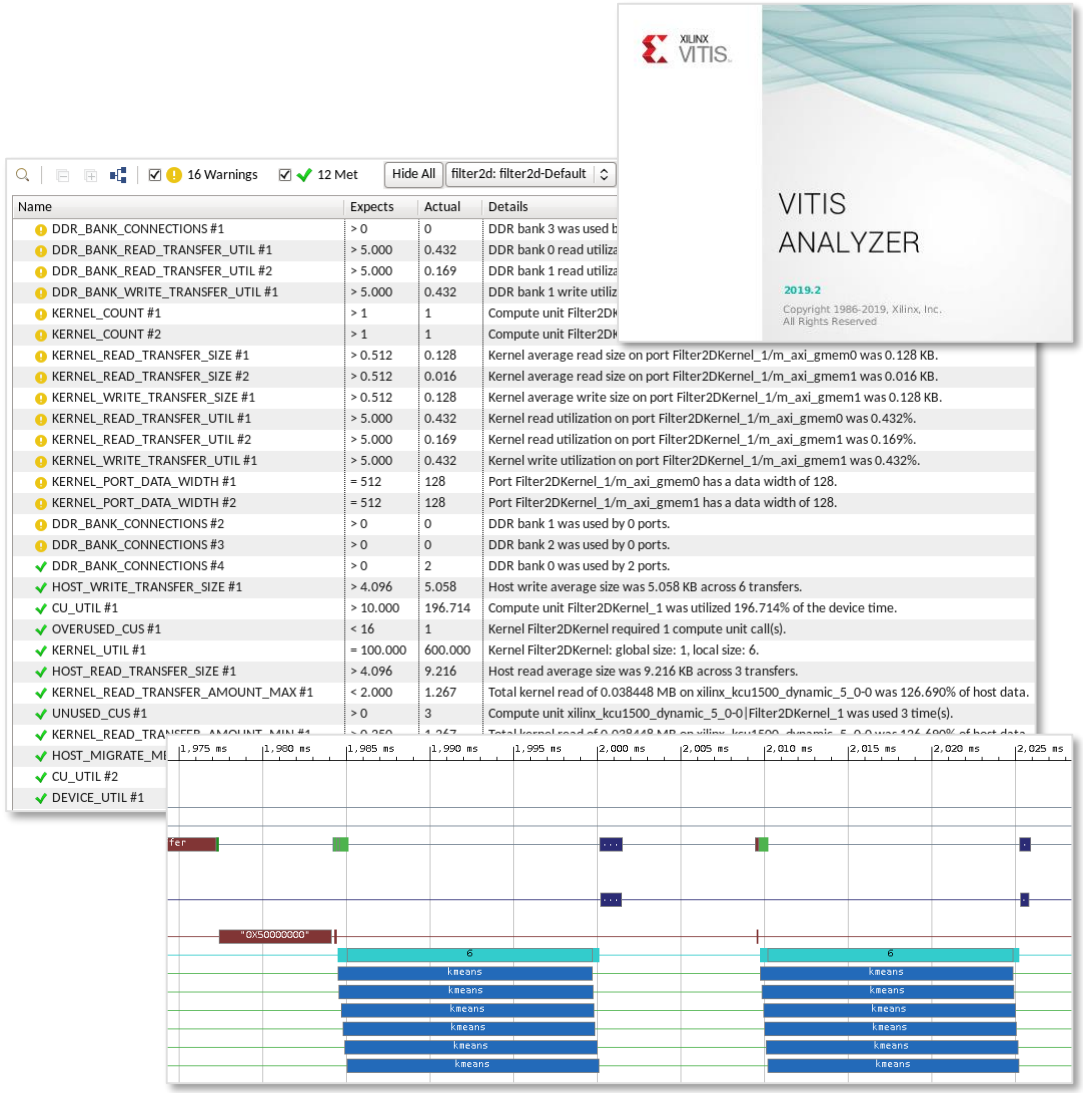
```
+ Latency:
    * Summary:
    +---------+----------+----------+-----------+
    | Latency (cycles)   | Latency (absolute)   |
    | min     |   max    |    min   |    max    |
    +---------+----------+----------+-----------+
    | 1496    | 9830874  | 4.986 us | 32.766 ms |
    +---------+----------+----------+-----------+

    + Detail:
        * Instance:
        +-------------+----------+-------+--------+----------+-----------+
        |             |          | Latency (cycles) | Latency (absolute)   |
        |   Instance  |  Module  |   min |   max  |    min   |    max    |
        +-------------+----------+-------+--------+----------+-----------+
        |processDU_U0 |processDU |   1450|  9830828| 4.833 us | 32.766 ms |
        |RGB2YUV_U0   |RGB2YUV   |   1057|  9830433| 3.523 us | 32.765 ms |
        |STORE_U0     |STORE     |   1029|  9830408| 3.430 us | 32.765 ms |
        |LOAD7_U0     |LOAD7     |   1170|  9830546| 3.900 us | 32.765 ms |
```



XILINX.

Converge on Desired Results with Vitis Analyzer

- Use Vitis Guidance, a built-in expert system
  - Checks for design suboptimalities
  - Recommends improvements
- Verify kernels using Vitis Emulation flow
  - Test HW/SW system integration
  - Validate performance estimates with simulations
  - Detailed debug visibility
- Leverage Vitis Profile Summary and Application Timeline Trace to visualize system performance
- Iterate and optimize interactively

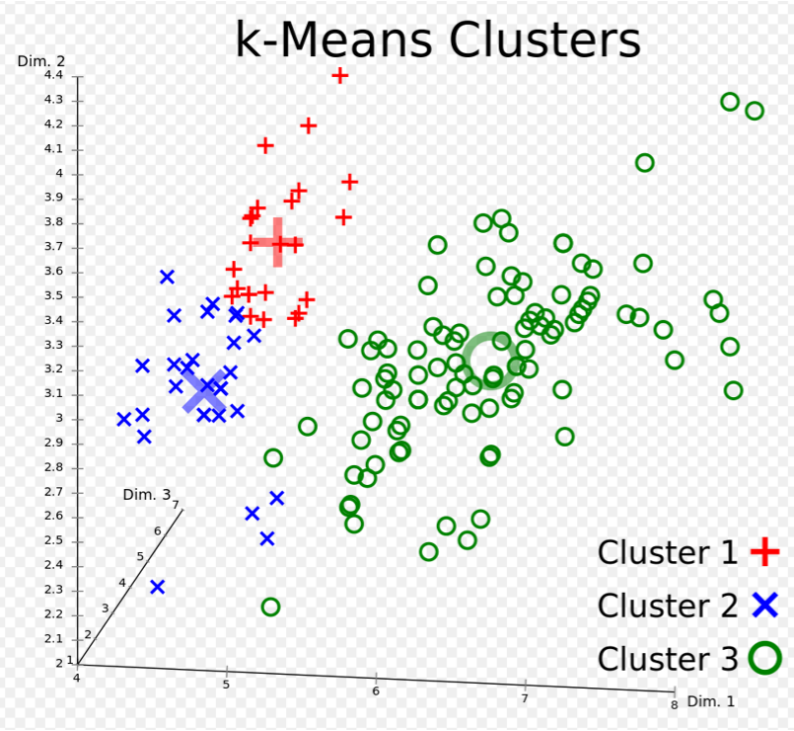


XILINX

Example Walk-Through

Example – K-means Clustering Algorithm Optimization

- K-means clustering is a method of vector quantization
  - Popular for cluster analysis in data mining
  - Aims to partition "n" observations into "k" clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster.
- Optimize K-Means clustering algorithm for FPGA acceleration and get the performance much higher than CPU
  - Explains how a CPU based K-means application is optimized for FPGA
  - Presents application optimization in step-by-step manner which are directly applicable and relevant to FPGA acceleration



Source: https://en.wikipedia.org/wiki/K-means_clustering

Baseline Design

- Port CPU code to FPGA design
- Performance is much worse than CPU design due to sequential loop implementation

CPU Time	FPGA Time (*)	Speed up
1 hour 11 minutes	42 hours	0.024x

**Note: Data based on Vivado HLS estimation number*

- Code Snippet

```
#define CONST_NPOINTS 492040
#define CONST_NCLUSTERS 100
#define CONST_NFEATURES 34
const unsigned int c_npoints = CONST_NPOINTS;
const unsigned int c_nfeatures = CONST_NFEATURES;
const unsigned int c_nclusters = CONST_NCLUSTERS
kmeans_ops: for(unsigned int point_id = 0 ; point_id < npoints ; point_id++){
    #pragma HLS LOOP_TRIPCOUNT min=c_npoints max=c_npoints
...
kmeans_nclusters: for (int i=0; i < nclusters; i++){
        #pragma HLS LOOP_TRIPCOUNT min=c_nclusters max=c_nclusters
...
kmeans_nfeatures: for (int l = 0; l < nfeatures; l++){
            #pragma HLS LOOP_TRIPCOUNT min=c_nfeatures max=c_nfeatures
```

Local Caching and Burst Read

- Store data in local buffer and reuse it
- Burst read the data into FPGA to maximize the throughput
- Got some improvements but still far from the CPU data

CPU Time	FPGA Time (*)	Speed up
1 hour 11 minutes	13 hours	0.077x

**Note: Data based on Vivado HLS estimation number*

- Code Snippet

```
cluster_cache: for (int i = 0; i < total_cluster_size; i++, fIdx++){
#pragma HLS PIPELINE II=1
    l_cluster_features[i] = clusters[clustIdx * nfeatures + fIdx];
    if(fIdx >= nfeatures){
        fIdx = 0;
        clustIdx++;
    }
}
```

```
read_features: for(int idx = 0 ; idx < nfeatures; idx ++){
 #pragma HLS PIPELINE II=1
     l_point_features[idx] = feature[point_id * nfeatures + idx];
 }
kmeans_nclusters: for (int i=0; i < nclusters; i++)
    {
        kmeans_nfeatures: for (int l = 0; l < nfeatures; l++)
        {
        #pragma HLS PIPELINE II=1 // II mentioned 1 but given 11
            float point_value   = l_point_features[l];
            float cluster_value = l_cluster_features[i * nfeatures + l];
            float diff = (point_value - cluster_value);
            dist += diff * diff;
        }
        if (dist < min_dist) {
            min_dist = dist;
            index    = i;
        }
        dist = 0;
    }
```

Data type Optimization

- Use fixed point data instead of floating point data
- Overall latency is reduced due to better II for fixed point data operations

CPU Time	FPGA Time (*)	Speed up
1 hour 11 minutes	1 hour 6 minutes	~1x

**Note: Data based on Vivado HLS estimation number*

- Code Snippet

```
#define DATA_TYPE unsigned int
#define VDATA_TYPE unsigned long
#define MAX_VALUE 0xFFFFFFFFFFFFFFFF
```

Data Parallel Processing

- Process multiple data points in parallel
- Partition data buffers to support more access bandwidth
- Unroll the loop for full parallel implementation

CPU Time	FPGA Time (*)	Speed up
1 hour 11 minutes	4.9 minutes	14.4x

**Note: Data based on Vivado HLS estimation number*

- Code Snippet

```
float l_point_features[PARALLEL_POINTS][MAX_FEATURES];

#pragma HLS ARRAY_PARTITION variable=l_point_features complete dim=1

kmeans_itr: for(unsigned int point_id = 0 ; point_id < npoints ; point_id+=PARALLEL_POINTS) {
    read_features: for(int idx = 0 ; idx < total_points * nfeatures ; idx++, fIdx++){}
    point_init: for (int i = 0; i < PARALLEL_POINTS; i++){}
    kmeans_ops: for (int i=0; i < nclusters * nfeatures; i++){
        square_and_add: for(int idx = 0; idx < PARALLEL_POINTS; idx++){}
        if((fIdx + 1) < nfeatures){
                fIdx++;
            }
        else {
        update_min: for (int pointIdx = 0; pointIdx < PARALLEL_POINTS; pointIdx ++) {}
        }
    }
    result_write: for(int i = 0; i < total_points; i++){}
}
```

Maximize DDR Data Throughput

- Restructure the data channel to read/write 512-bit data from DDR
- DDR access latency can be large, maximizing DDR transfer efficiency helps on overall performance

CPU Time	FPGA Time (*)	Speed up
1 hour 11 minutes	33 seconds	129x

**Note: Data based on actual hardware test*

- Code Snippet

```
VECTOR_SIZE = BUS_WIDTH/DATA_TYPE_SIZE
VECTOR_SIZE = 512/32 = 16

typedef struct MEMBERSHIP{
    int data[VECTOR_SIZE];
}MEMBERSHIP_TYPE;

typedef struct VDATA{
    unsigned int data[VECTOR_SIZE];
}VDATA_TYPE;

typedef struct VMULT{
    unsigned int data[VECTOR_SIZE];
}VMULT_TYPE;

 #pragma HLS data_pack variable=feature
 #pragma HLS data_pack variable=clusters
 #pragma HLS data_pack variable=membership
```

Use Multiple Compute Units

- Use multiple CUs to process on different set of data to increase degree of parallelism

CPU Time	FPGA Time (*)	Speed up
1 hour 11 minutes	16.7 seconds	255x

**Note: Data based on actual hardware test*

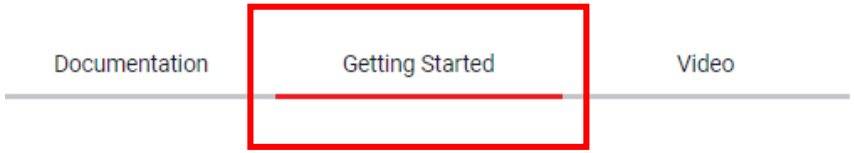
- Code Snippet

```
void kmeans(
                VDATA_TYPE         * feature,
                VDATA_TYPE         * clusters,
                MEMBERSHIP_TYPE    * membership,
                int                  n_points,
                int                  nclusters,
                int                  nfeatures,
                int                  cu_id,
                int                  num_cu
              )

   for (int i = 0; i < NUM_CU; i++){
       int narg = 0;
       OCL_CHECK(err, err = g_kernel_kmeans.setArg(narg++, d_feature));
       OCL_CHECK(err, err = g_kernel_kmeans.setArg(narg++, d_cluster));
       OCL_CHECK(err, err = g_kernel_kmeans.setArg(narg++, d_membership));
       OCL_CHECK(err, err = g_kernel_kmeans.setArg(narg++, sizeof(cl_int), (void*) &n_points));
       OCL_CHECK(err, err = g_kernel_kmeans.setArg(narg++, sizeof(cl_int), (void*) &n_clusters));
       OCL_CHECK(err, err = g_kernel_kmeans.setArg(narg++, sizeof(cl_int), (void*) &n_features));
       OCL_CHECK(err, err = g_kernel_kmeans.setArg(narg++, i));
       OCL_CHECK(err, err = g_kernel_kmeans.setArg(narg++, NUM_CU));

       OCL_CHECK(err, err = g_q.enqueueTask(g_kernel_kmeans, NULL, &wait_events[i]));
   }

   g_q.enqueueWaitForEvents(wait_events);
   g_q.finish();
```

Taking the Next Step

Start Development with Vitis Unified Software Platform

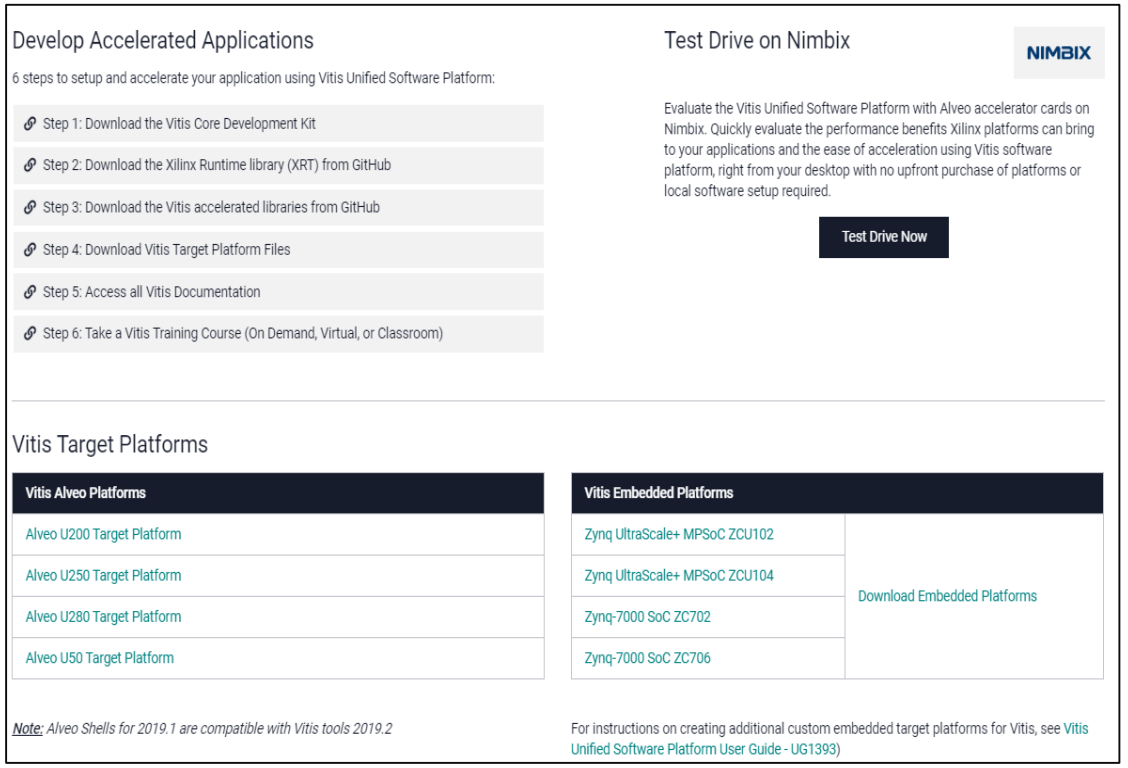
- Download Vitis Unified Software Platform – Free !



XILINX VITIS™

www.xilinx.com/vitis

- Design Methodology Documentation



Develop Accelerated Applications

6 steps to setup and accelerate your application using Vitis Unified Software Platform:

- Step 1: Download the Vitis Core Development Kit
- Step 2: Download the Xilinx Runtime library (XRT) from GitHub
- Step 3: Download the Vitis accelerated libraries from GitHub
- Step 4: Download Vitis Target Platform Files
- Step 5: Access all Vitis Documentation
- Step 6: Take a Vitis Training Course (On Demand, Virtual, or Classroom)

Test Drive on Nimbix

NIMBIX

Evaluate the Vitis Unified Software Platform with Alveo accelerator cards on Nimbix. Quickly evaluate the performance benefits Xilinx platforms can bring to your applications and the ease of acceleration using Vitis software platform, right from your desktop with no upfront purchase of platforms or local software setup required.

Test Drive Now

Vitis Target Platforms

Vitis Alveo Platforms
Alveo U200 Target Platform
Alveo U250 Target Platform
Alveo U280 Target Platform
Alveo U50 Target Platform

Vitis Embedded Platforms	
Zynq UltraScale+ MPSoC ZCU102	Download Embedded Platforms
Zynq UltraScale+ MPSoC ZCU104	
Zynq-7000 SoC ZC702	
Zynq-7000 SoC ZC706	

Note: Alveo Shells for 2019.1 are compatible with Vitis tools 2019.2

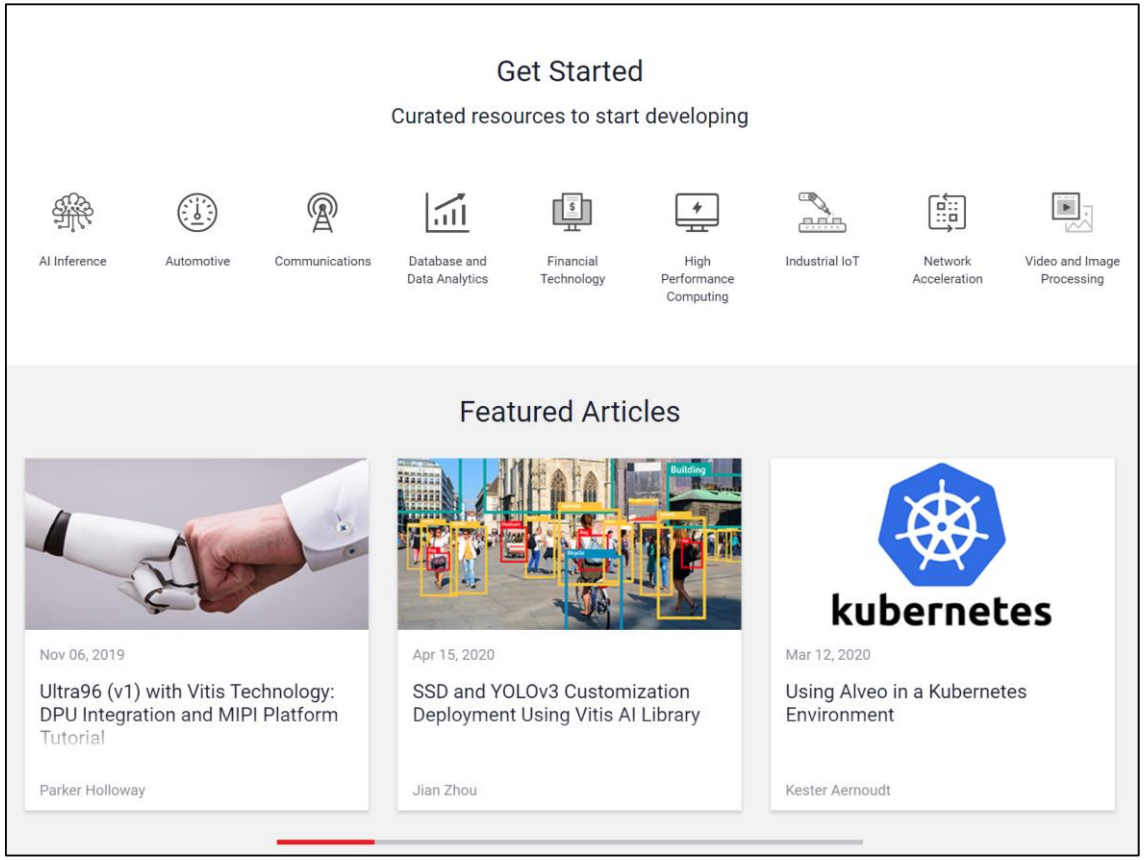
For instructions on creating additional custom embedded target platforms for Vitis, see Vitis Unified Software Platform User Guide - UG1393)

XILINX.

Developer Articles & Resources

- Get Started with “How-To” Developer Articles & Tutorials

https://developer.xilinx.com/

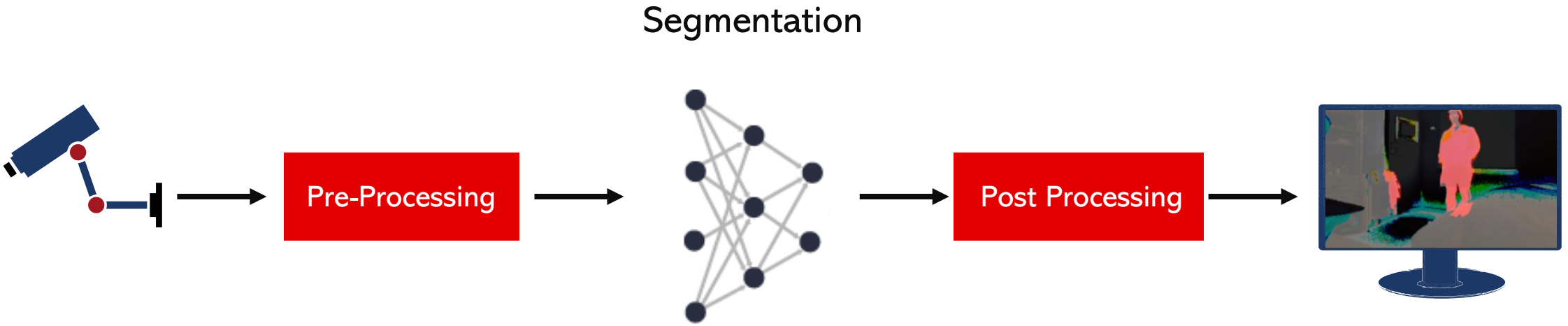


- Getting Started Tutorials

https://github.com/Xilinx/Vitis-Tutorials

Whole Application Acceleration : AI-enabled Systems

Tune in Next Thursday, May 7



XILINX VITIS™ AI

Accelerate the Whole Application on Xilinx Platforms (AI + non-AI)

XILINX

Thank You