NDRange and Single-work items kernel improvements for FPGA

Emmanuel Kieffer

High Performance
Computing &
Big Data Services

hpc.uni.lu
hpc@uni.lu
@ULHPC

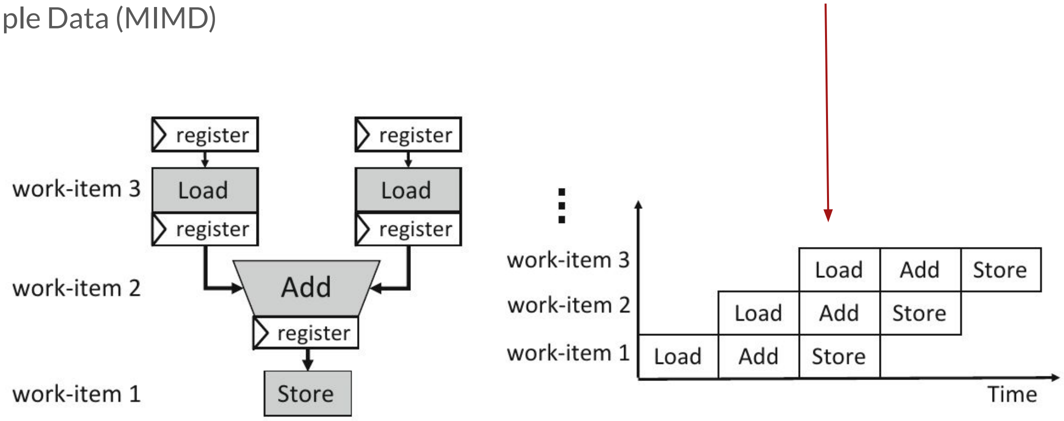
UNIVERSITÉ DU
LUXEMBOURG

NDRange Kernels V.S. Single-Work-Item Kernels

NDRange kernels

- It may looks like as GPU-like programming but:
  - Work-items are not launched simultaneously
  - No Single Instruction Multiple Data (SIMD) like GPU
  - More like Multiple Instruction Multiple Data (MIMD)
  - Work-items are pipelined
- SIMD on FPGA → yes
  - Vectorization
  - *“num_simd_work_items(N)”*

Executed in parallel

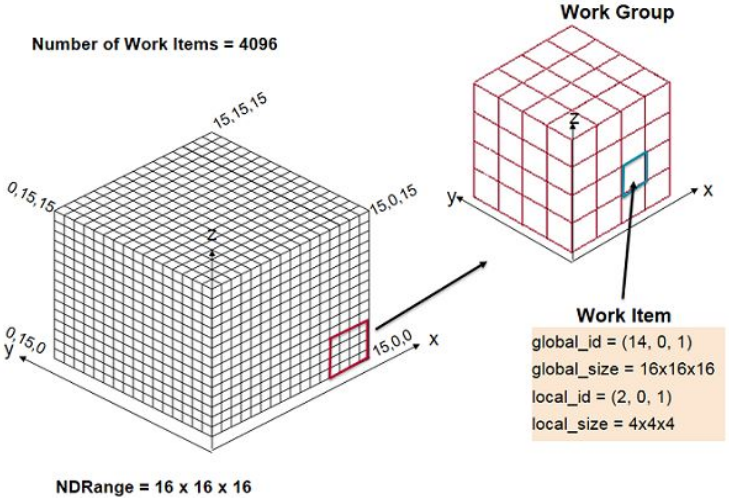


work-item 3			Load	Add	Store
work-item 2		Load	Add	Store	
work-item 1	Load	Add	Store		

(source:Design of FPGA-Based Computing Systems with OpenCL)

NDRange kernels

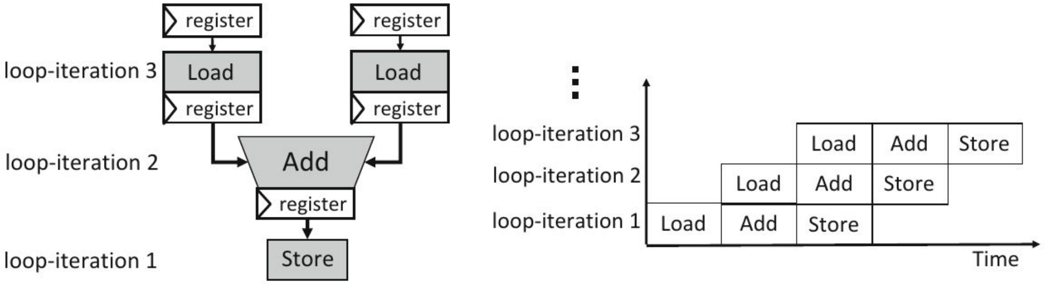
- With data dependencies, data needs to be shared
    - Local or global memories
    - Need for barriers functions
- Barriers function has a huge penalty cost for FPGA
    - Due to pipelining



global_id = (14, 0, 1)
global_size = 16x16x16
local_id = (2, 0, 1)
local_size = 4x4x4

Single Work-item

- Equivalent to launch kernels with NDRange of (1,1,1)
- Define as a Task in OpenCL
- Loops in single-work items are automatically parallelized by the Intel FPGA compiler
- Multiple loop-iterations are computed in different pipeline stages
- Nothing special needed to preserve data dependency

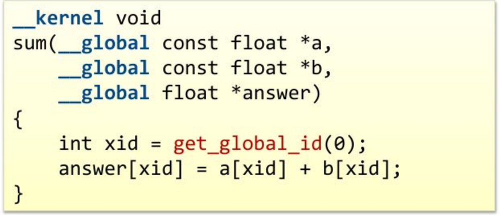


(source:Design of FPGA-Based Computing Systems with OpenCL)

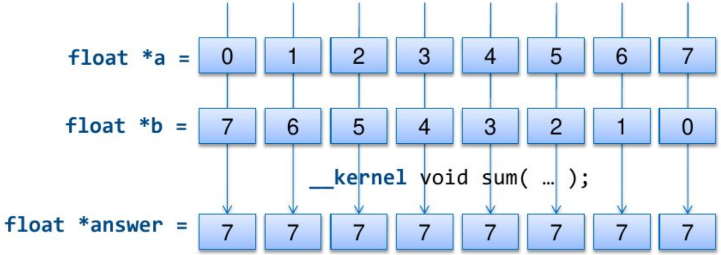
In a nutshell

NDRange

- No Data sharing between work-items
- Replicate kernels on FPGA and GPU



```
__kernel void
sum(__global const float *a,
    __global const float *b,
    __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}
```



float *a = 0 1 2 3 4 5 6 7

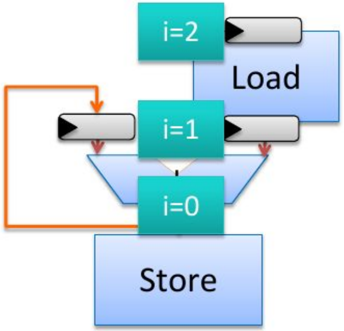
float *b = 7 6 5 4 3 2 1 0

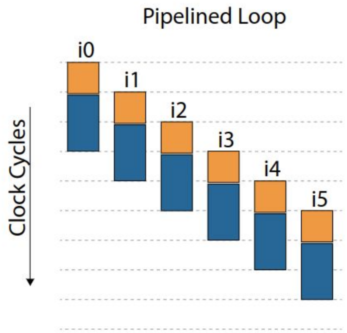
__kernel void sum(…);

float *answer = 7 7 7 7 7 7 7 7

Single Work-item

- Data dependencies
- Porting CPU code to FPGA



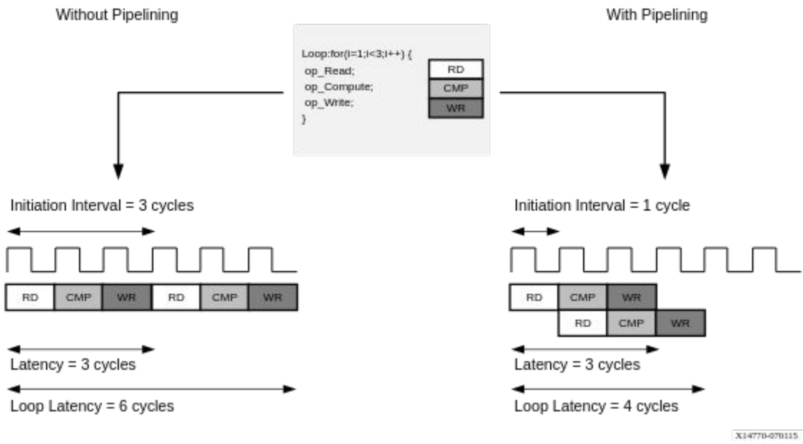


General Improvement Strategies

UNIVERSITÉ DU LUXEMBOURG

Loop unrolling

- Improve degree of parallelism
- More data can be processed in one clock cycle
- Add `#pragma unroll <N>`
- `<N>` is the unroll factor
- Limits the number of iterations to unroll
- Ex:
  - `#pragma unroll 1` : prevent a loop in your kernel from unrolling
  - `#pragma unroll` : let the offline compiler decide how to unroll the loop



```
Loop:for(i=1;i<3;i++) {
 op_Read;
 op_Compute;
 op_Write;
}
```

(source:Loop Optimization in HLSL)

Loop unrolling

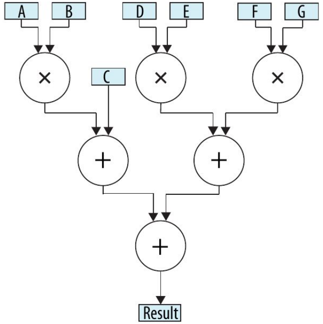
```
#define M 512

__kernel void loop_unrolling(__global int *restrict x,
                             __global int *restrict z){

for(unsigned int i=0; i<M; i++){
      #pragma unroll 1
      for(unsigned int j=0; j<8; j++)
            z[i] += x[i*M+j];
      }
}
```

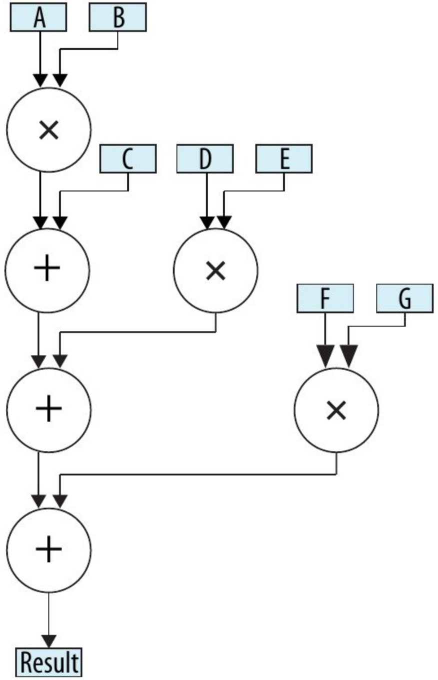
Strategy	Kernel processing time (ms)	Clock frequency (MHz)
Without unrolling	0.073	454
With N=100	0.054	454

- Large unrolling factor may reduce the clock frequency
- Avoid unrolling outer-loops to avoid huge resource utilisation
- Unrolling loops fails if you do not have enough resources on the FPGA
- Loops boundaries should be know at compilation time (~ constant)

Optimizing Floating-Point Operations

- Manually influence floating-point operations order
  - result = (((A * B) + C) + (D * E)) + (F * G);
- By default, operations are unbalanced (see Figure)
- By default, implementation is consistent with IEEE Standard 754-2008
- Using the aoc compiler option “`-ffp-reassociate`”:
  - Balance the tree but may be inconsistent with IEEE Standard 754-2008
  - May cause small floating-point differences





(source:Intel)

Rounding Operations

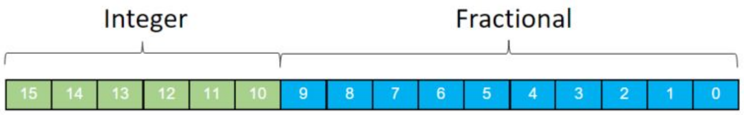
- Reducing intermediate rounding operations
  - Help reducing hardware resources
  - Need to be sure if your application can tolerate small deltas (differences) in results
- As previously, you can instruct the aoc offline compiler:
  - To remove floating-point rounding operations and conversions whenever possible
  - With the option “`-ffp-contract=fast` ”
  - Called “fused floating-point operations”
  - Round a floating-point operation only once—at the end of the tree of the floating-point operations

(source:Intel)

Fixed-Point Operations

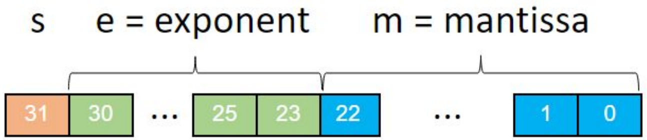
- **Floating-point operations** consume more logic on FPGA
- Increase the amount of hardware resources available using **fixed-point operations**
  - Hardware necessary to implement it is smaller
  - Drawback is that you have to anticipate the possible data range
- OpenCL standard does not support fixed-point representation
- Need to use existing types (char, short, int, long) which have predefined number of bits
- Solution: Using **bit masking.** The offline compiler disregards unnecessary bits at compilation

Fixed-Point



(source:Imperix)

Floating-Point



number= $(-1)^s * (1.m) * 2^{e-127}$

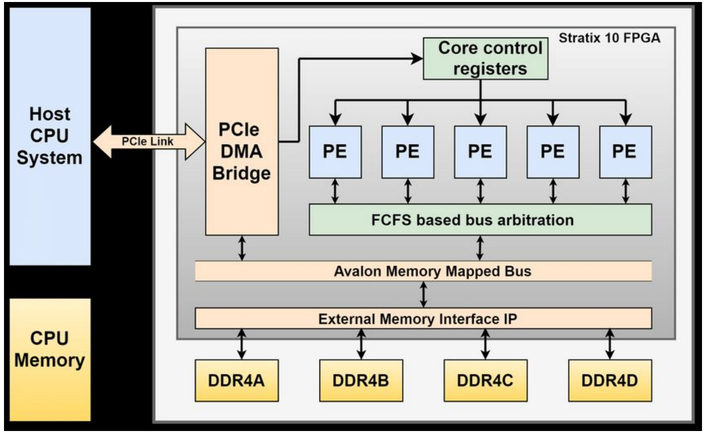
Fixed-Point Operations with OpenCL

```
__kernel fixed_point_masking_add (__global const unsigned int * restrict x,
                                  __global const unsigned int * restrict y,
                                  __global unsigned int * restrict z)
{
      size_t gid = get_global_id(0);
      // Temporary unsigned int (32 bits)
      unsigned int temp;
      // 0x1FFFF ->  1_1111_1111_1111_1111 (17 bits)
      // 0x3FFFF -> 11_1111_1111_1111_1111 (18 bits)
      temp = 0x3FFFF & ((0x1FFFF & a[gid]) + ((0x1FFFF & b[gid]));
      // Mask the first 14 bits --> results on 18 bits (carry)
      result[gid] = temp & 0x3_FFFF;
}
```

Results of adding the two 17 bits will never exceed 18 bits

Aligning memory

- Enable Direct Access Memory (DMA) by using at least 64-byte aligned memory
- Why ?
  - Frees the host processor to perform other calculations during data transfer
- Later, we will see how to use **Memory Access Coalescing**



(source:Communication-avoiding micro-architecture to compute Xcorr scores for peptide identification)

```
// For Windows
#define AOCL_ALIGNMENT 64
#include <malloc.h>
void *ptr = _aligned_malloc (size,
AOCL_ALIGNMENT);
_aligned_free (ptr);
// For Linux
#define AOCL_ALIGNMENT 64
#include <stdlib.h>
void *ptr = NULL;
posix_memalign (&ptr, AOCL_ALIGNMENT, size);
free (ptr);
```

Avoiding Pointer Aliasing

- Pointer aliasing is a hidden kind of data dependency
- Make sure that kernel inputs/outputs do not aliases other pointer
- Use the “`restrict`“ keyword to inform the offline compiler
- Prevents the offline compiler from creating unnecessary memory dependencies

```
__kernel void myKernel (__global int * restrict x, __global int * restrict y)
```

Performance Improvement for NDRange kernels

Work-Group Size

- The offline compiler can perform strong optimization for hardware resources
- You can specify:
  - A maximum group size with "`__attribute__((max_work_group_size(<N>)))` " N being the number of work-items.
  - A required group size "`__attribute__((req_work_group_size(<N`$_{\texttt{Dim1}}$`>,<N`$_{\texttt{Dim2}}$`>,<N`$_{\texttt{Dim3}}$`>)))`" where (<NDim1>,<NDim2>,<NDim3>) define the number of work-items in each dimension
- In both case, you need to instruct the "**clEnqueueNDRangeKernel**" function about the local_work_size:
  - Smaller than or equal to the maximum work-group size if using `__attribute__((max_work_group_size(<N>)))`
  - Equal to the number of work-items defined by `__attribute__((req_work_group_size(<N`$_{\texttt{Dim1}}$`>,<N`$_{\texttt{Dim2}}$`>,<N`$_{\texttt{Dim3}}$`>)))`

Kernel vectorisation

- Higher throughput can be achieved using kernel vectorization
- Multiple work-items can be then executed in a Single Instruction Multiple Data (SIMD)
- The offline compiler combines multiple scalar operations such as addition, multiplication, etc ...
- More hardware resources are required and therefore the require memory bandwidth will be higher
- The user kernel design may become **memory-bound**

Automatic kernel vectorisation

- **Automatic vectorization** can be achieved using the “`__attribute__((num_simd_work_items(<N>)))` ”
- `<N>` identical operation will be vectorized by the offline compiler
- Note that `<N>` should be a **power of 2** and the number of work-group items divisible by `<N>`

```
__attribute__((num_simd_work_items(8)))
__attribute__((reqd_work_group_size(128,1,1)))
__kernel void mult_kernel( __global const float * restrict X,
                           __global const float * restrict Y,
                           __global float * restrict Z)
{
int id = get_global_id(0);
Z[id] = X[id] + Y[id];
}
```

8 SIMD lanes

128/8 = 16 wide vector operation

The offline compiler coalesces 8 loads to optimize (reduce) the access to memory in case there are no data dependencies

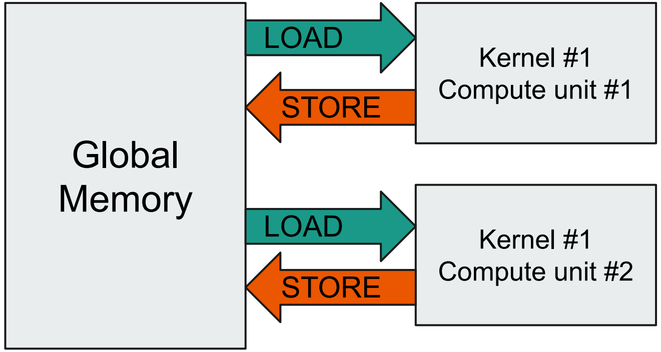
Vector data types

- Using vector data types can also increase the number of computations per clock cycles
- For example, `float16` represents buffers of 16 floats
- They allow better memory access using coalescing

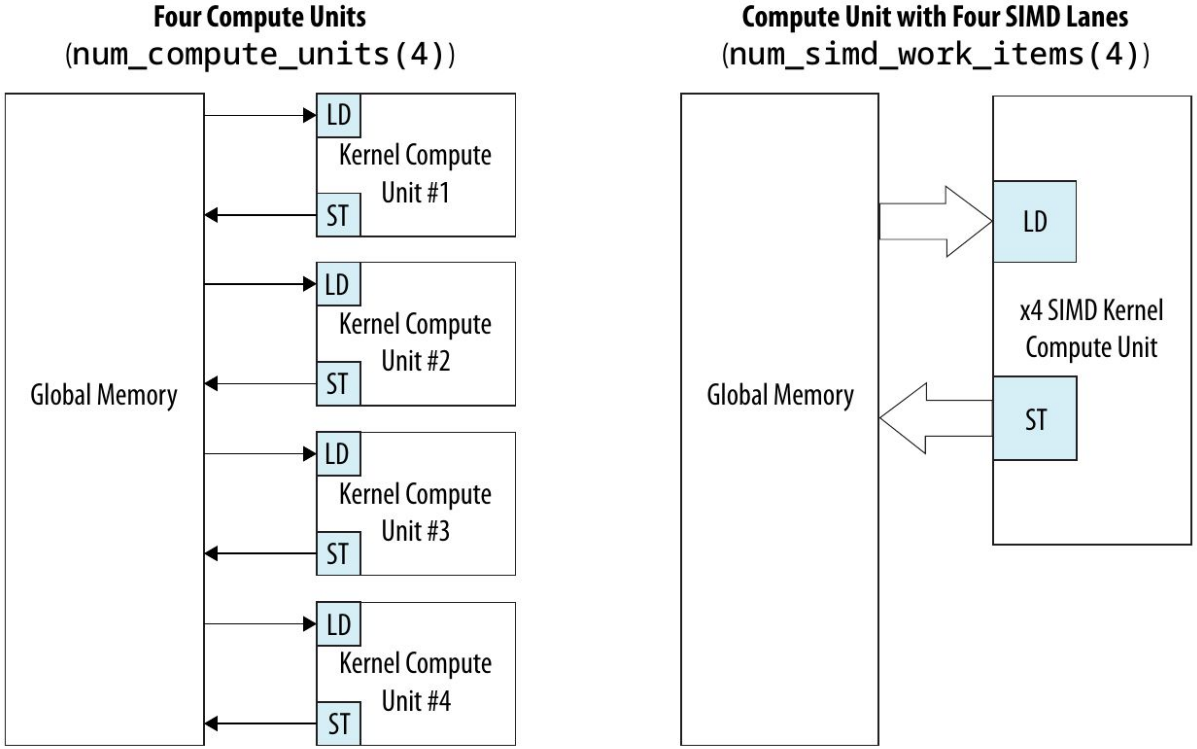
```
__attribute__((reqd_work_group_size(128,1,1)))
__kernel void mult_kernel( __global const float16 * restrict X,
                           __global const float16 * restrict Y,
                           __global float16 * restrict Z)
{
int id = get_global_id(0);
Z[id] = X[id] + Y[id];
}
```

Multiple compute units

- You can increase the number of compute units to improve throughput
- Multiple compute units can be executed multiple work-groups simultaneously for each single kernel
- Work-groups are then **dispatched automatically** by the hardware scheduler to available compute units
- Obviously, this will require **more hardware resources**
- Contrary to vectorization, memory cannot be coalesced :
  - Each compute units accesses the memory concurrently
- The number of compute units can be defined using:
  - “`__attribute__((num_compute_units(<N>)))`”
  - With `<N>` the number of compute units



Compute units vs SIMD



(source: Intel documentation)

Compute units vs SIMD

- Prefer SIMD to compute units
- Both improve throughput but have different efficiency to access memory
- "`__attribute__((num_compute_units(<N>)))` " modifies the number of compute units to which work-groups will be scheduled.
- "`__attribute__((num_simd_work_items(<N>)))` " modifies the amount of work a compute unit can perform in parallel on a single work-group. The datapath of the compute unit is duplicated by sharing the control logic across each SIMD vector lane.
- In summary, SIMD leads to more efficient hardware than using more compute units when trying to achieve the same objective

Compute units vs SIMD

Vectorisation	● Smaller hardware utilization ● Coalesced memory	● SIMD factor a multiple of 2 ● Generally limited to 16
Compute units	● Number of compute units if synthesizable ● Allows to maximize resources usage	● Memory access competition between compute units

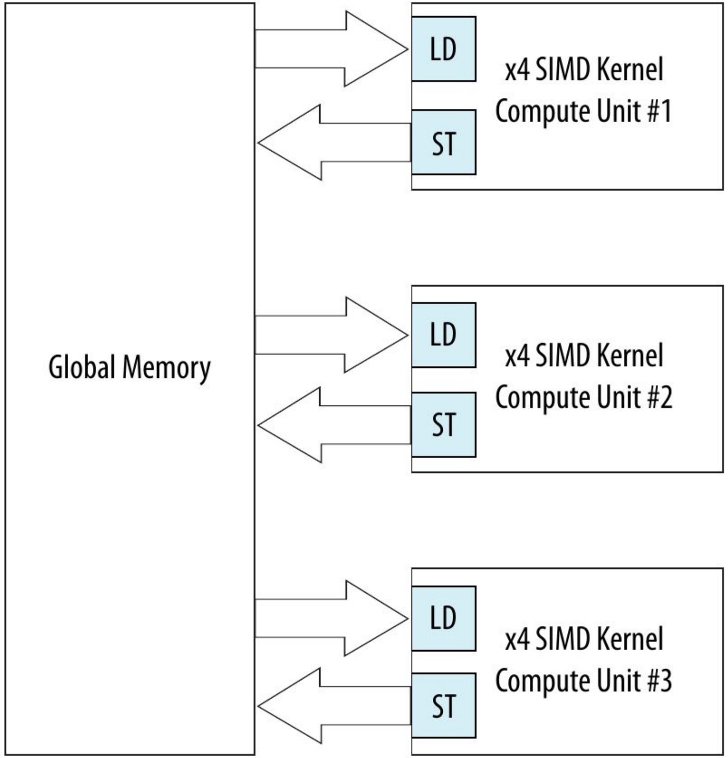
Hybridizing compute units with vectorisation

- To take advantage of both strategies
- Combine kernel vectorization with multiple compute units
- Fully utilize the resources
- Ex by Intel:
  - Suppose you cannot increase the number of SIMD lanes to 16
  - Combining 3 compute units with each 4 SIMD lanes may be fit to the FPGA

```
__attribute__((num_simd_work_items(4)))
__attribute__((num_compute_units(3)))
__attribute__((reqd_work_group_size(8,8,1)))
__kernel void matrixMult(...){}
```

Hybridizing compute units with vectorisation

- 12 operations per clock cycle
- Filling the FPGA will take significant amount of time
- Remove the compute units and SIMD attributes when you need to recompile several time



(source: Intel documentation)

Performance Improvement for Single work-item kernels

UNIVERSITÉ DU
LUXEMBOURG

Remove nested loops

- With a nested loop:
    - Outer-loop must wait till the inner-loop is finished
    - Extra cycle delay due to control overhead

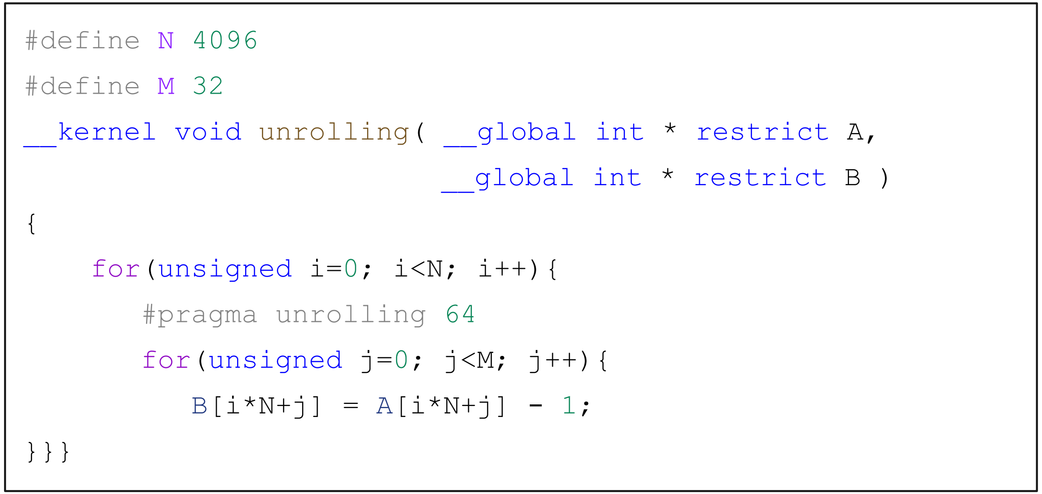
```
#define N 4096
__kernel void nestedloop1( __global int * restrict
A, __global int * restrict B ){

      for(unsigned i=0; i<N; i++){
            for(unsigned j=0; j<N; j++)
      {
            B[i*N+j] = A[i*N+j] - 1;
      }}}
```

```
#define N 4096
__kernel void nestedloop2( __global int * restrict
A, __global int * restrict B ){

      for(unsigned i=0; i<N*N; i++){
            B[i*N+j] = A[i*N+j] - 1;
      }
}
```

(source: Intel documentation)

Unroll loops

- Unrolling to remove nested loops
- Use only when:
  - Inner-loop has small number of iterations
  - Simple inner-loop single iteration
- Unrolling increase resource utilization
- Should be used with care



```
#define N 4096
#define M 32
__kernel void unrolling( __global int * restrict A,
                         __global int * restrict B )
{
    for(unsigned i=0; i<N; i++){
       #pragma unrolling 64
       for(unsigned j=0; j<M; j++){
          B[i*N+j] = A[i*N+j] - 1;
}}}
```

(source:Design of FPGA-Based Computing Systems with OpenCL)

Data dependencies causing serial executions

(source:Design of FPGA-Based Computing Systems with OpenCL)

```
#define N 4096

__kernel void
data_dependency( __global int * restrict A,
                 __global int * restrict B,
                 __global int * restrict res)
{
  int sum = 0;
  for(unsigned i=0; i<N; i++){
    for(unsigned j=0; j<N; j++){
        sum += A[i*N+j];
    }
    sum += B[i];
  }
  *res=sum;
}
```

Dependency leading to serial executions (Data-dependent region)

```
#define N 4096

__kernel void data_dependency( __global int * restrict A,
                                __global int * restrict B,
                              __global int * restrict res)
{
  int sum = 0;
  for(unsigned i=0; i<N; i++){
    int tmpsum = 0;
    for(unsigned j=0; j<N; j++){
        tmpsum += A[i*N+j];
    }
    sum += tmpsum;
    sum += B[i];
  }
  *res=sum;}
```

Introduce a local temp variable

Avoid Conditional Loops

(source: Intel documentation)

```
if (condition) {
      for (int i = 0; i < m; i++) {
      // statements
      }
}else {
      for (int i = 0; i < m; i++) {
      // statements
      }
}
```

```
for (int i = 0; i < m; i++) {
      if (condition) {
      // statements
      }else {
       // statements
       }
}
```

The loop should contain the conditions, not the other way around !!!

Deepest possible variable scope

(source: Intel documentation)

```
int a[N];
int b[N];
for (int i = 0; i < m; ++i) {
for (int j = 0; j < n; ++j) {
// statements
}
}
```

```
int a[N];
for (int i = 0; i < m; ++i) {
int b[N];
for (int j = 0; j < n; ++j) {
// statements
}
}
```

Declare variables just before you need them to reduce resources !!

Use local memory

(source: Intel documentation)

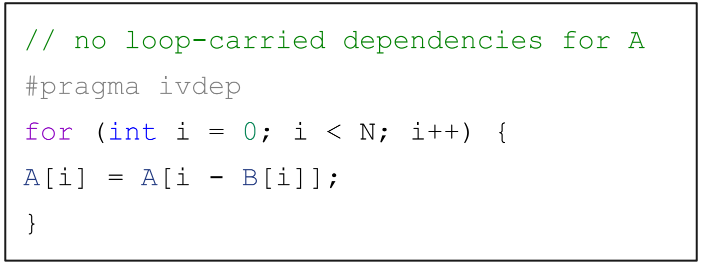
```
#define N 128

__kernel void unoptimized( __global int*
restrict A )
{
      for (unsigned i = 0; i < N; i++)
            A[N-i] = A[i];
}
```

```
#define N 128

__kernel void optimized( __global int* restrict A ){
  int temp[N];
  for(unsigned i=0; i<N; i++)
    temp[i]=A[i];
  for (unsigned i = 0; i < N; i++)
    temp[N-i] = temp[i];
  for(unsigned i=0; i<N; i++)
    A[i]=temp[i];
}
```

- **Global memory accesses have longer latencies than local memory**
- **Transferring to local memory**
- **Loop-carried dependency is now on temp**

Loop-Carried Dependencies Caused by Memory Arrays Access

- Add `#pragma ivdep` in your single work-item kernel
- The offline compiler generate hardware ensuring load and store instructions operate within dependency constraints
- If the updated data is not needed for another iteration of the loop, the next iteration can be initiated without waiting for the data from the previous iteration to be written to the global memory.
- `ivdep` pragma instructs the offline compiler to remove this extra hardware

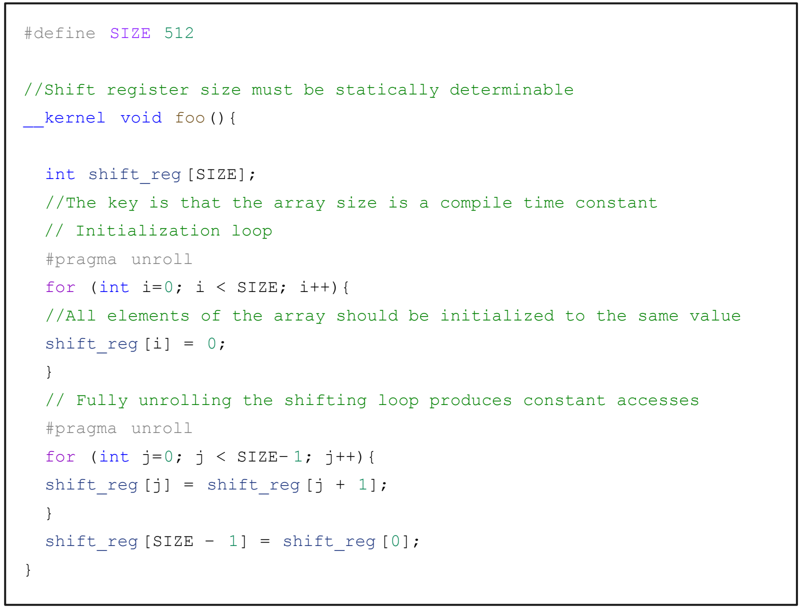


```
// no loop-carried dependencies for A
#pragma ivdep
for (int i = 0; i < N; i++) {
A[i] = A[i - B[i]];
}
```

(source: Intel documentation)

Shift Registers

- Shift register pattern is very important in FPGA
- Shift register must have a known size at compilation
- Shift register elements must be initialized with same value
- Constant access when fully unrolling
- A shift register into a block RAM allows to handle multiple access points into the array



```
#define SIZE 512

//Shift register size must be statically determinable
__kernel void foo(){

  int shift_reg[SIZE];
  //The key is that the array size is a compile time constant
  // Initialization loop
  #pragma unroll
  for (int i=0; i < SIZE; i++){
  //All elements of the array should be initialized to the same value
  shift_reg[i] = 0;
  }
  // Fully unrolling the shifting loop produces constant accesses
  #pragma unroll
  for (int j=0; j < SIZE-1; j++){
  shift_reg[j] = shift_reg[j + 1];
  }
  shift_reg[SIZE - 1] = shift_reg[0];
}
```

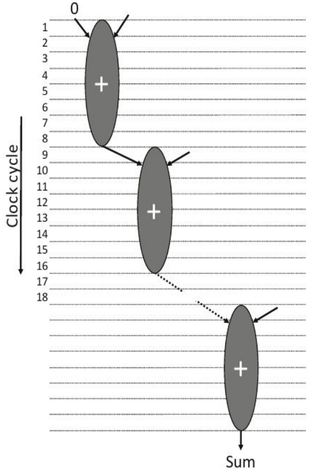
(source: Intel documentation)

Relaxing Loop-Carried Dependency using Shift Registers

- Handling single work-item kernels that carry out double precision floating-point operations
- Each add operation requires the result of its previous operation
- Latency of the floating-point addition is several clock-cycles
- To relax the data dependency, infer the array as a shift register.

```
__kernel void double_add_1 (__global double *arr,int N, __global double
*result)
{

double temp_sum = 0;
for (int i = 0; i < N; ++i){
temp_sum += arr[i];
}
*result = temp_sum;
}
```

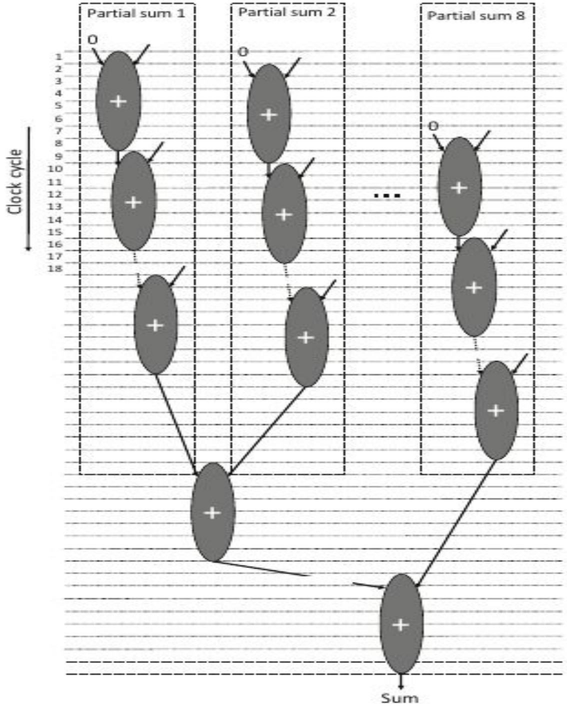
(source: Intel documentation)



(source:Design of FPGA-Based Computing Systems with OpenCL)

Relaxing Loop-Carried Dependency using Shift Registers

- Create a shift register holding partial sums
- Partial sums computed in parallel
- Latency of the floating-point addition is several clock-cycles
- So Initialisation Interval (II) > 1



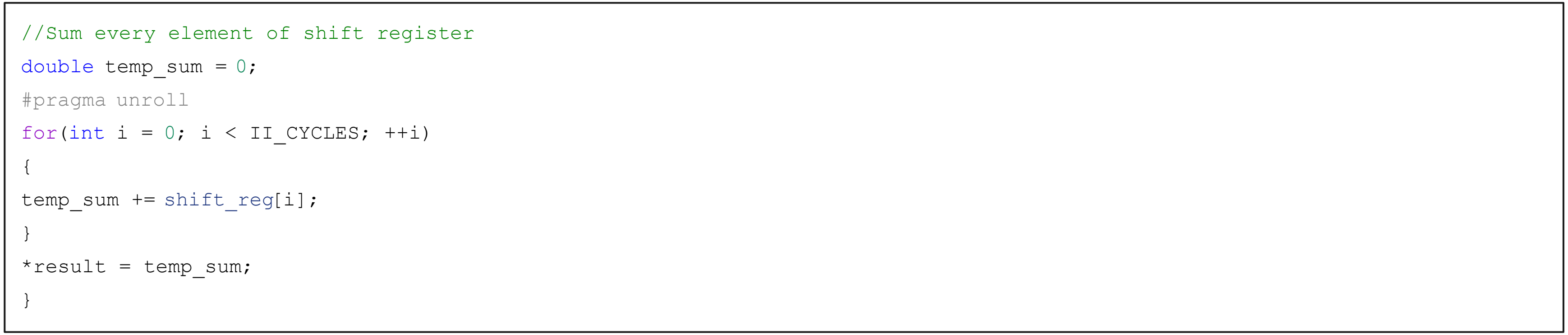
(source:Design of FPGA-Based Computing Systems with OpenCL)

Relaxing Loop-Carried Dependency using Shift Registers

```
//Shift register size must be statically determinable
#define II_CYCLES 12

__kernel void double_add_2 (__global double *arr, int N,__global double *result){
//Create shift register with II_CYCLE+1 elements
double shift_reg[II_CYCLES+1];
//Initialize all elements of the register to 0
for (int i = 0; i < II_CYCLES + 1; i++){
shift_reg[i] = 0;
}
//Iterate through every element of input array
for(int i = 0; i < N; ++i){
//Load ith element into end of shift register
//if N > II_CYCLE, add to shift_reg[0] to preserve values
shift_reg[II_CYCLES] = shift_reg[0] + arr[i];
#pragma unroll
//Shift every element of shift register
for(int j = 0; j < II_CYCLES; ++j)
{
shift_reg[j] = shift_reg[j + 1];
}
```

(source: Intel documentation)

Relaxing Loop-Carried Dependency using Shift Registers



```
//Sum every element of shift register
double temp_sum = 0;
#pragma unroll
for(int i = 0; i < II_CYCLES; ++i)
{
temp_sum += shift_reg[i];
}
*result = temp_sum;
}
```

(source: Intel documentation)

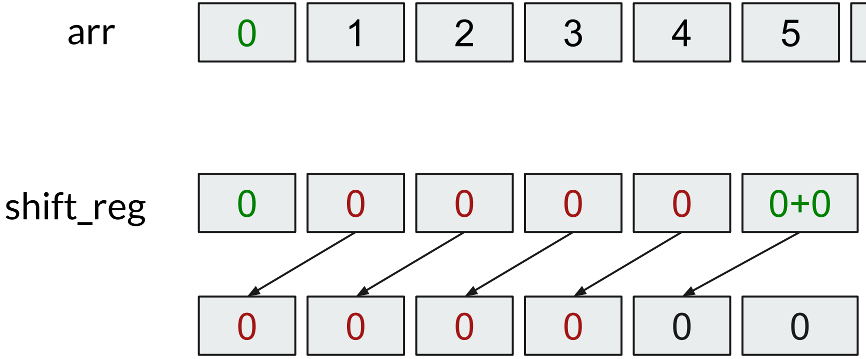
Example N=10, II_CYCLES=5

arr[N]	0	1	2	3	4	5	6	7	8	9
shift_reg[II_CYCLES+1]	0	0	0	0	0	0				

// All elements initialized to 0

(source: Intel documentation)

Iteration 0



arr

0	1	2	3	4	5	6	7	8	9

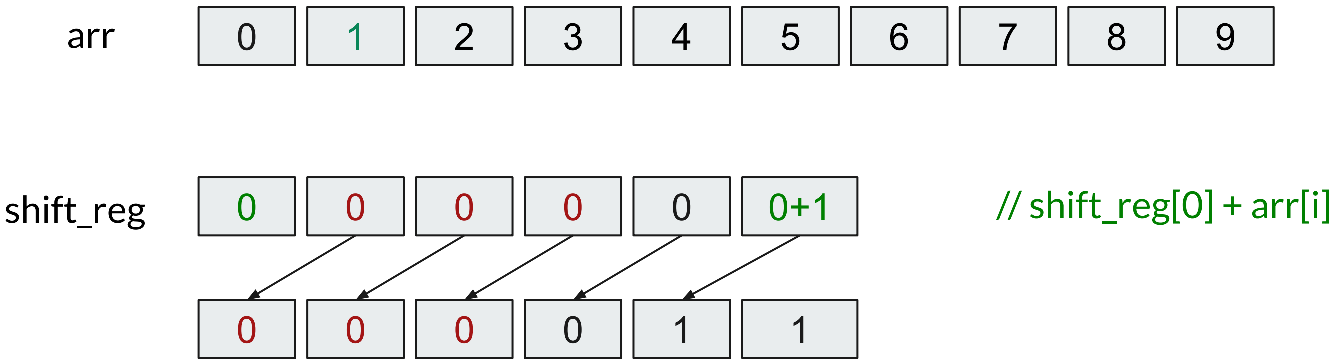
shift_reg

0	0	0	0	0	0+0
0	0	0	0	0	0

// shift_reg[0] + arr[i]

(source: Intel documentation)

Iteration 1



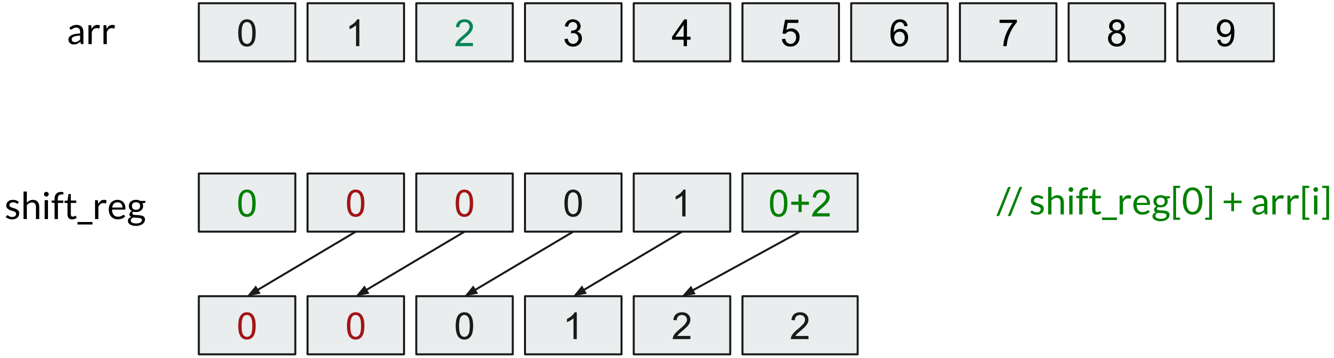
arr	0	1	2	3	4	5	6	7	8	9

shift_reg	0	0	0	0	0	0+1
	0	0	0	0	1	1

// shift_reg[0] + arr[i]

(source: Intel documentation)

Iteration 2



arr

0	1	2	3	4	5	6	7	8	9

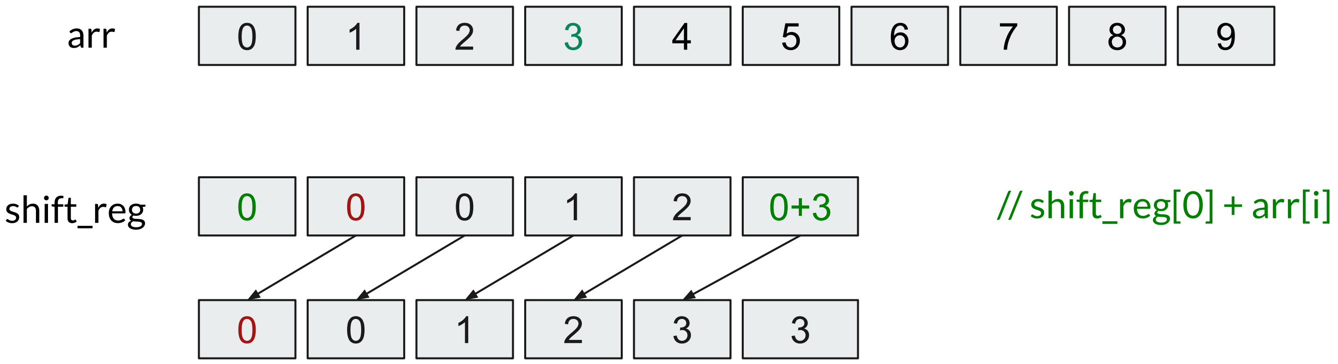
shift_reg

0	0	0	0	1	0+2
0	0	0	1	2	2

// shift_reg[0] + arr[i]

(source: Intel documentation)

Iteration 3



arr 0 1 2 3 4 5 6 7 8 9

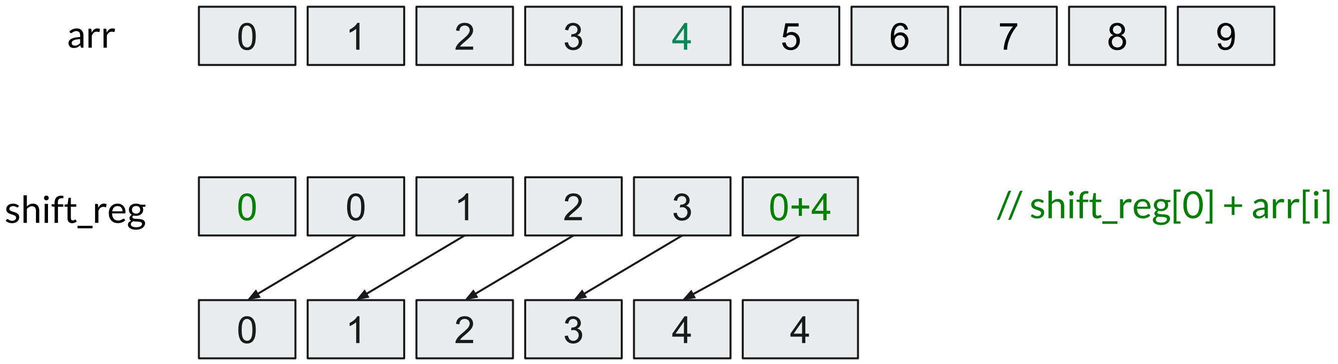
shift_reg 0 0 0 1 2 0+3

0 0 1 2 3 3

// shift_reg[0] + arr[i]

(source: Intel documentation)

Iteration 4



arr

0	1	2	3	4	5	6	7	8	9

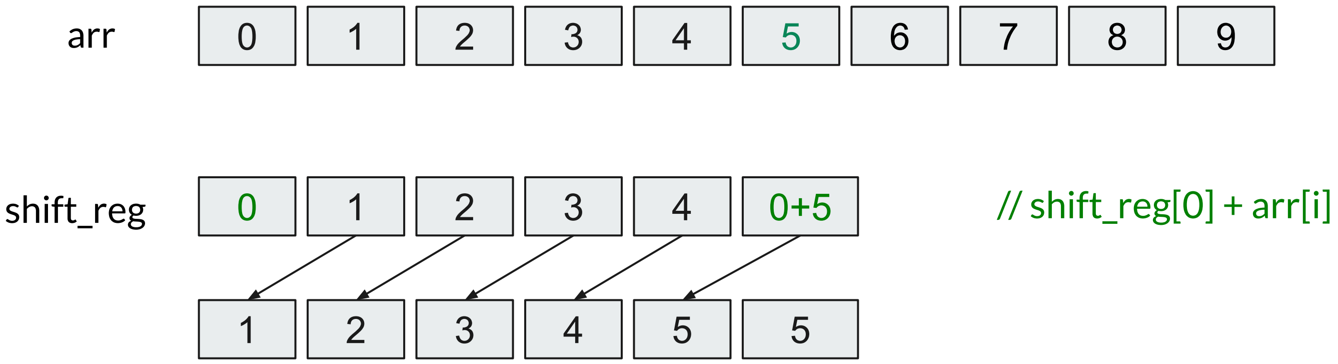
shift_reg

0	0	1	2	3	0+4
0	1	2	3	4	4

// shift_reg[0] + arr[i]

(source: Intel documentation)

Iteration 5



arr: 0 1 2 3 4 5 6 7 8 9

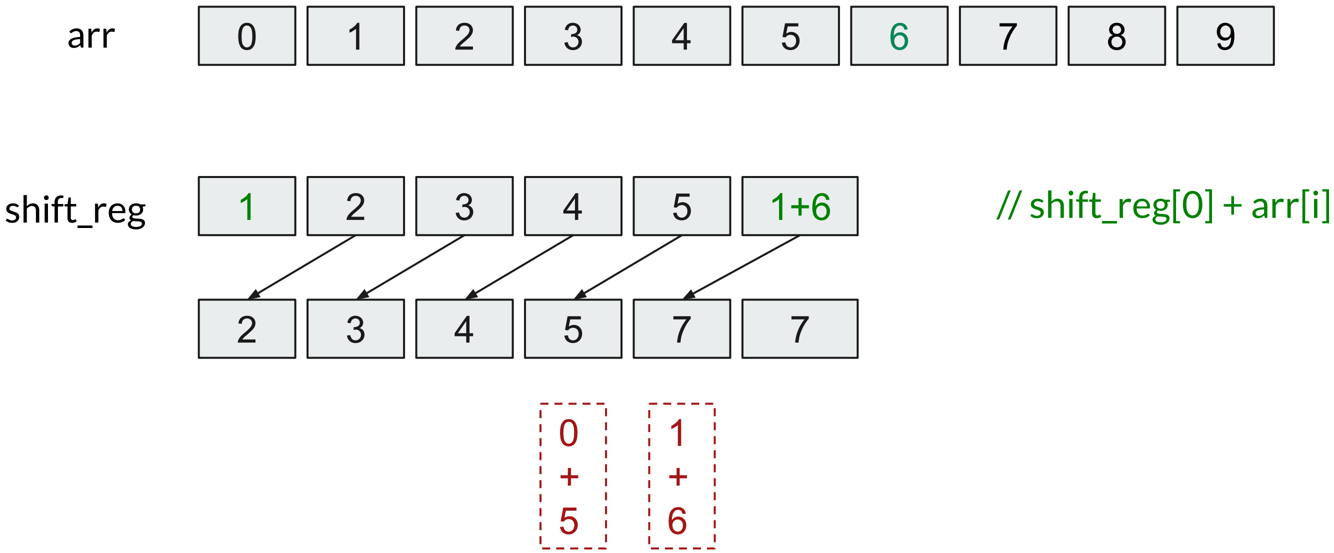
shift_reg: 0 1 2 3 4 0+5

1 2 3 4 5 5

// shift_reg[0] + arr[i]

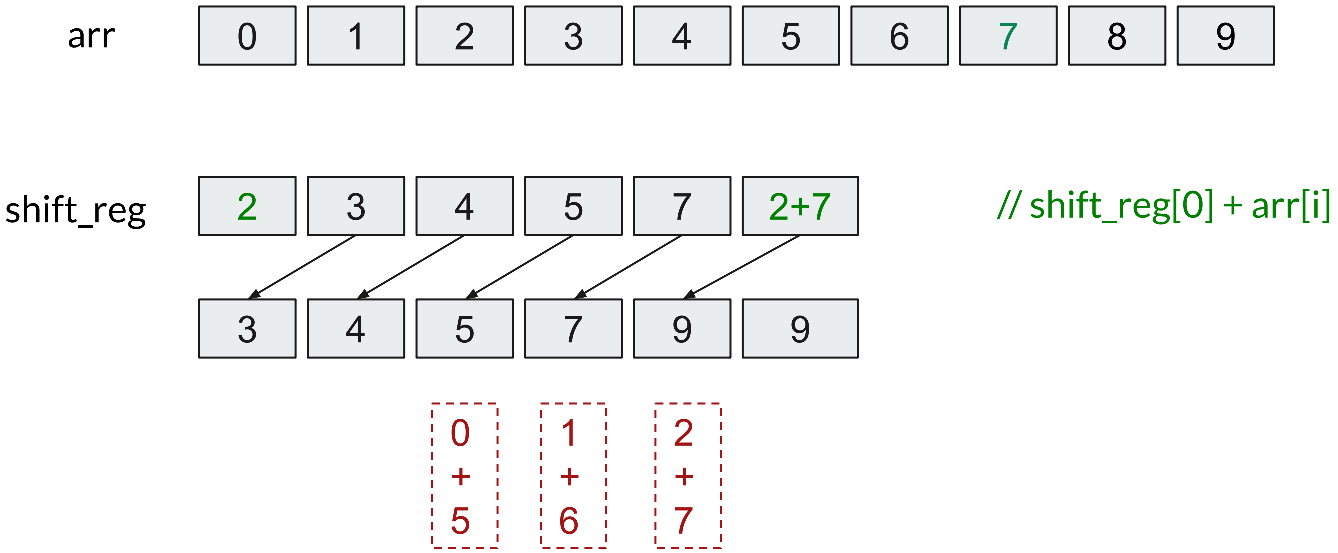
(source: Intel documentation)

Iteration 6



(source: Intel documentation)

Iteration 7



arr 0 1 2 3 4 5 6 7 8 9

shift_reg 2 3 4 5 7 2+7

// shift_reg[0] + arr[i]

3 4 5 7 9 9

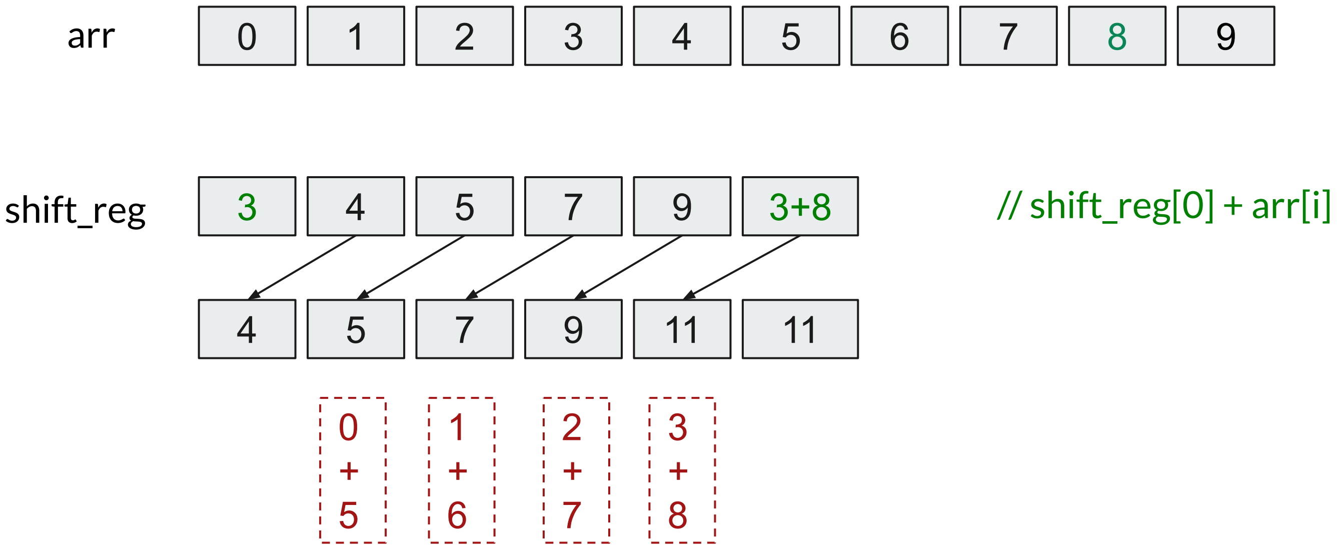
0 + 5

1 + 6

2 + 7

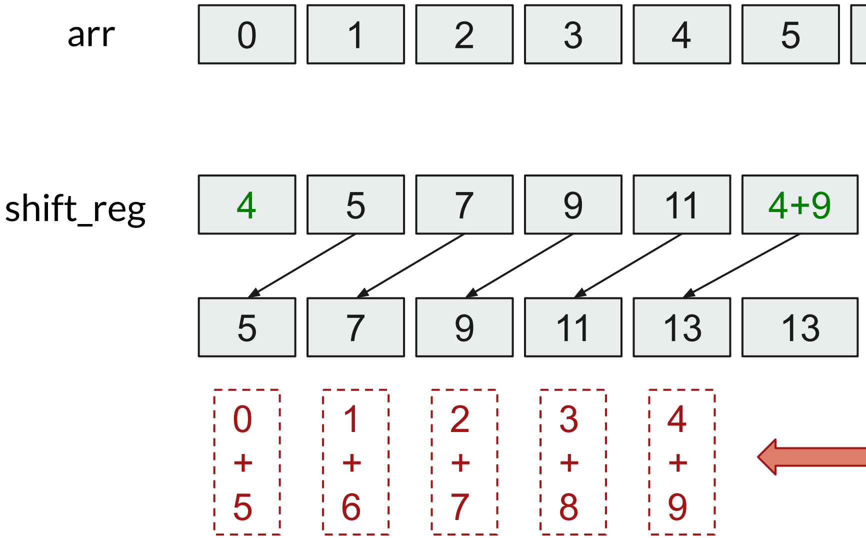
(source: Intel documentation)

Iteration 8



(source: Intel documentation)

Iteration 9



arr	0	1	2	3	4	5	6	7	8	9

shift_reg	4	5	7	9	11	4+9
	5	7	9	11	13	13
	0 + 5	1 + 6	2 + 7	3 + 8	4 + 9	

// shift_reg[0] + arr[i]

Partial sums computed in // (last unrolled loop)

(source: Intel documentation)

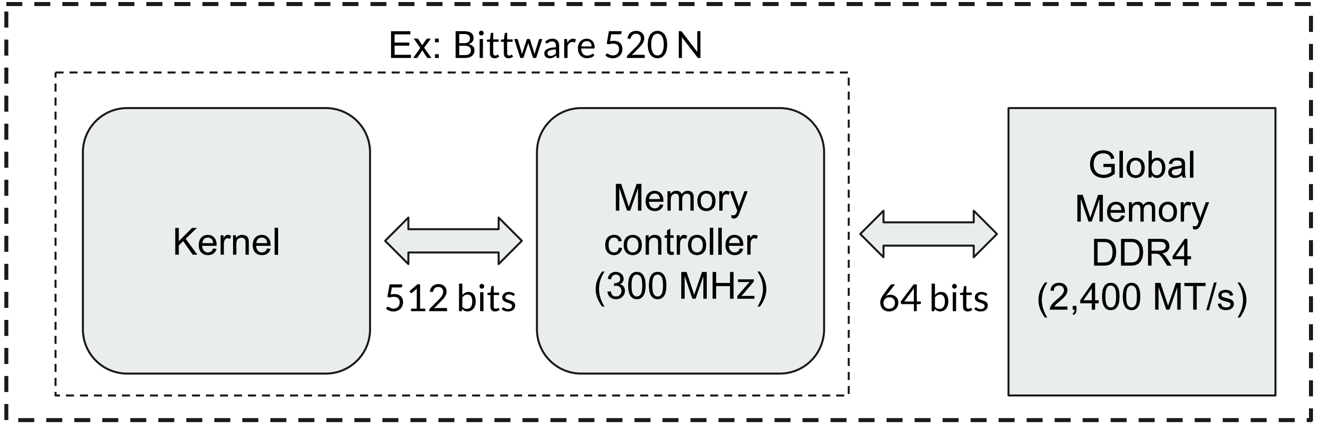
Memory Improvements

Memory Hierarchy

Type	Access	Hardware
Host memory	read/write only by host	DRAM
Global memory (device)	read/write by host and work-items	FPGA DRAM (DDR,QDR)
Local memory (device)	read/write only by work-group	RAM blocks
Constant memory (device)	read/write by host read only by work-items	FPGA DRAM RAM blocks
Private memory device	read/write by single work-item only	RAM blocks Registers

Memory Coalescing

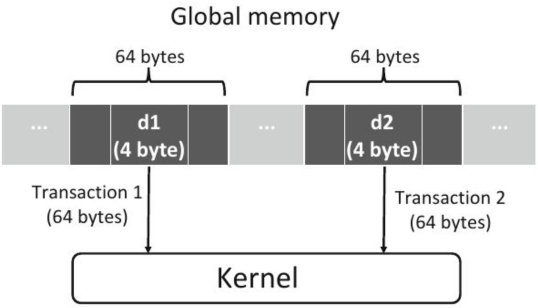
- Global memory
  - A "***transaction***" corresponds to one read or write access to or from memory
  - Data bus of the memory controller is wider (controller memory freq. is lower)
  - Data with between DDR and PHY is 64 bits
  - When we requests data, we need to have efficient transactions
  - To take advantage of the memory controller bandwidth, **the frequency of your kernels should exceed it**



Memory Coalescing

- Each “***transaction***” access 64 bytes
- If you are passing kernel arguments distant more than 64 bytes
- More transactions are therefore needed to read global memory
- Strategy:
    - Make sure that global memory access are coalesced to merge transactions
    - Simplify the data path of memory access
    - Built-in vector type (e.g., int<N>, float<N>) when you have support a single type
    - Or custom data types using structures when mixing multiple types

Contiguous memory region !!!



(source: Design of FPGA-Based Computing Systems with OpenCL)

Non-coalesced memory access

```
#define N 4096
#define IT 4194304

__kernel void kernel_ncoalesced (__global double * restrict A,  __global double * restrict B,
                                  __global double * restrict C, __global double * restrict D,
                                   __global double * restrict E, __global double * restrict F,
                                   __global double * restrict G, __global double * restrict H,
                                   __global double * restrict out){
  unsigned int k = 0;
  for(unsigned int it = 0; it<IT; it++){
     out[k] = A[k] + B[k] + C[k] + D[k] + E[k] + F[k] + G[k] + H[k];
     if(k < N){
        k++;
      }else{
        k=0;
     }
  }}
```

Coalesced memory access

```
#define N 4096
#define IT 4194304
struct StructEx
{
double A, B, C, D, E, F, G, H;
};
__kernel void kernel_coalesced (__global struct StructEx * restrict all_inputs,
                                 __global double * restrict out){
  unsigned int k = 0;
  for(unsigned int i = 0; i<N; i++){
    out[k] = all_inputs[k].A + all_inputs[k].B + all_inputs[k].C + all_inputs[k].D +
             all_inputs[k].E + all_inputs[k].F + all_inputs[k].G + all_inputs[k].H;
    if(k < N){
       k++;
    }else{
       k=0;
    }
}}
```

Comparison

Strategy	Kernel processing time (ms)	Clock frequency (MHz)
Non-coalesced	45.523	454
Coalesced	0.555	454

Structure alignment

- Structure alignment will improve performance
- General Alignment for structure (CPU):
  - The alignment must be a power of two.
  - The alignment must be a multiple of the **least-common-multiple word-width** of he structure member sizes.
- Example:
  - 1: 1, 2, 3, 4
  - 4: 4, 8, 12, 16
- Structure example will be 4 bytes aligned
- The compiler will therefore add 3 bytes of padding between a and b..

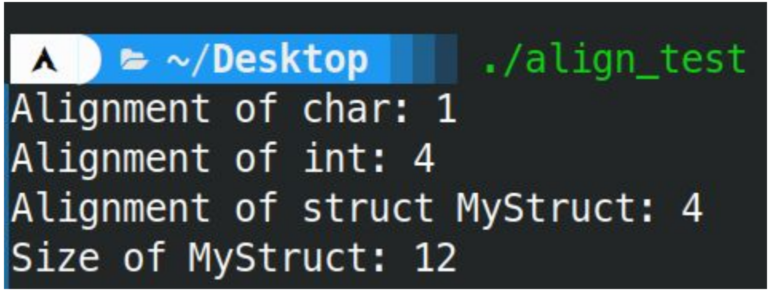
```
struct MyStruct
{
char a;\\ 1 byte
int b;\\ 4 bytes
int c;\\ 4 bytes
};
```

Alignment hint

- If you struggle to remind how to compute the alignment, use the following C code.

align.c

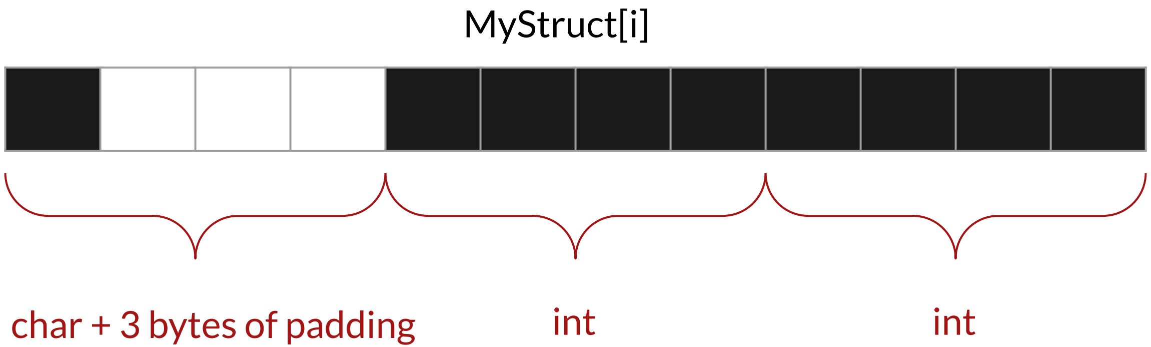
```
#include <stdio.h>

struct MyStruct {
char a ;
int b ;
int c ;

};
int main() {
printf("Alignment of char: %zu\n", _Alignof(char));
printf("Alignment of int: %zu\n", _Alignof(int));
printf("Alignment of double: %zu\n", _Alignof(double));
printf("Alignment of struct MyStruct: %zu\n", _Alignof(struct MyStruct));
printf("Size of MyStruct: %zu\n",sizeof(struct MyStruct));
return 0;}
```



```
~/Desktop ./align_test
Alignment of char: 1
Alignment of int: 4
Alignment of struct MyStruct: 4
Size of MyStruct: 12
```

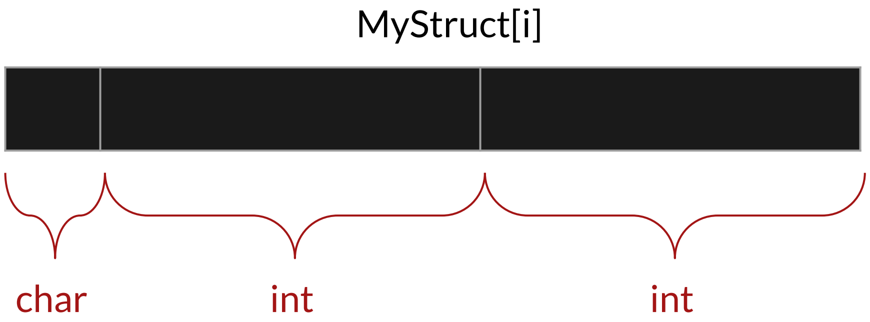
Arrays of Structures

- Each element of MyStruct has 12 bytes due to padding
- Recall that each transaction between the user kernel design and the memory controller is **512 bits wide**
- We have therefore **64/12 = 5.333** => alignment is far from optimal as the 6th element of MyStruct will be split between two 64-byte regions



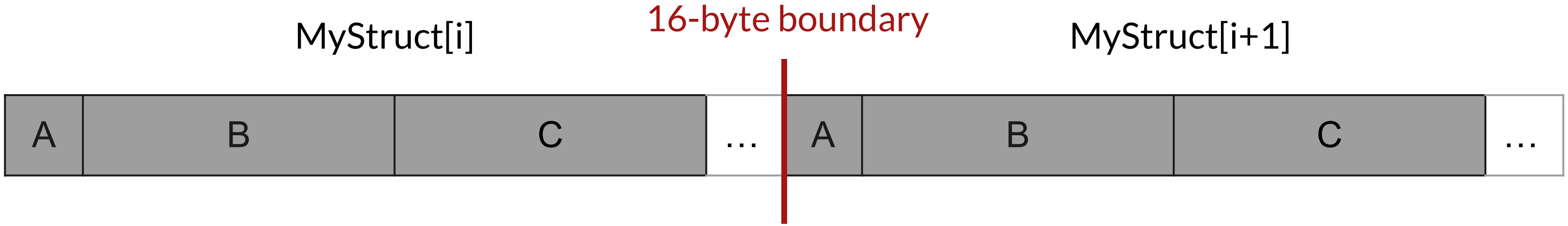
Removing padding

- Removing all padding will definitely reduce the size
- Padding can be removed by adding the “packed” attribute, i.e, “`__attribute__((packed))` ” in your kernel
- Each element of MyStruct will have therefore 9 bytes
- However, **64/9 = 7.111** => we still have some elements in multiple 64-bytes region and the alignment is sub-optimal



Change alignment and padding

- To improve performance, align structure such all elements belongs to a single 64-byte regions
- Padding can still be removed by adding the “packed” attribute, i.e, “`__attribute__((packed))` ”
- Transaction size is 64 bytes, the minimum alignment which is also a multiple of the transaction size is **16**
- Enforce a 16-byte alignment with “`__attribute__((aligned(16)))` ”

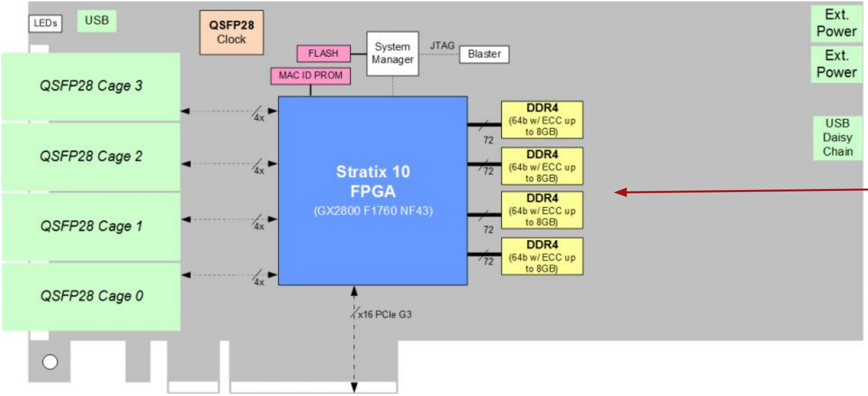


Comparison

Strategy	Kernel processing time (ms)	Clock frequency (MHz)
With default padding and misaligned	50.706	454
Without padding and aligned	40.789	454

Optimize Global Memory Accesses

- Modern FPGA cards have multiple memory modules that can be accessed in parallel
- These modules are called “memory banks”
- The offline compiler works in burst-interleaved configuration to evenly allocated onto multiple banks
- The goal is to avait data concentration on a particular bank

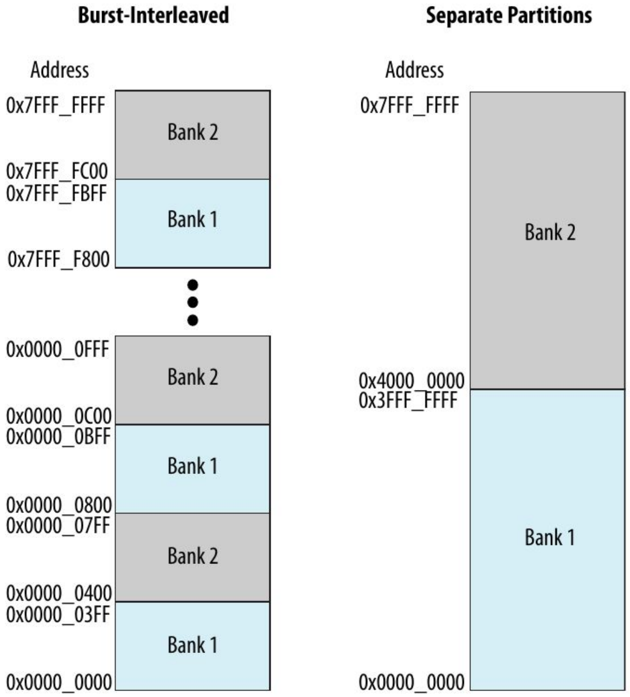


4 memory banks

(source: Bittware 520N FPGA Accelerator Card)

Optimize Global Memory Accesses

- Burst-interleaved configuration:
  - Best load balancing between the memory banks.
  - But non-contiguous memory
- Non-interleaved strategy:
  - Contiguous load and store operations
  - Improve memory access efficiency
  - Increased access speeds
  - Reduced hardware resource needs.
- Non-interleaved ⇔ programmer is in charge of choosing memory banks



(source: Intel documentation)

Using non-interleaving strategy

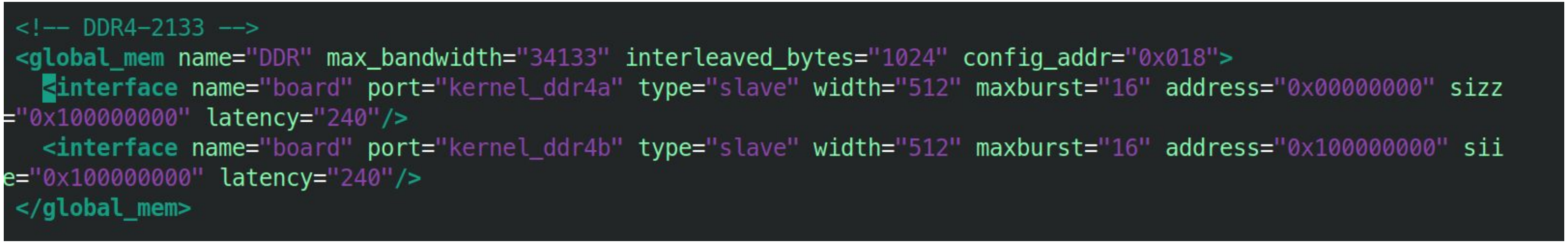
- You need to instruct the offline compiler with the option "**-no-interleaving=<global memory name>**"
- Global memory names are defined in the BSP (see file board_spec.xml) or use "default"
- Don't forget to allocate OpenCL buffer on their respective banks

```
// Example with two banks
int main(){
...
in_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_CHANNEL_1_INTELFPGA ,
sizeof(float)*SIZE, NULL, &status);
out_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_CHANNEL_2_INTELFPGA ,
sizeof(float)*SIZE, NULL, &status);
...}
```

- Non-interleaved memory can be useful if you have read-only and write-only data
- Ex: Vector addition => input vectors are read from BANK 1 and output vector is writing to BANK 2

Choosing Global memory type

- Global memory names and types are defined in the BSP (see file board_spec.xml)



```
<!-- DDR4-2133 -->
<global_mem name="DDR" max_bandwidth="34133" interleaved_bytes="1024" config_addr="0x018">
  <interface name="board" port="kernel_ddr4a" type="slave" width="512" maxburst="16" address="0x00000000" sizz
="0x100000000" latency="240"/>
  <interface name="board" port="kernel_ddr4b" type="slave" width="512" maxburst="16" address="0x100000000" sii
e="0x100000000" latency="240"/>
</global_mem>
```

- Inside the file board_spec.xml, search for “global_mem” (On the figure, we only have DDR)
- To define a different global memory type listed in the board_spec.xml
  - Add to your kernels a different buffer location using attributes

  -
    ```
    __kernel void foo(__global __attribute__((buffer_location("DDR"))) int *x,
                      __global __attribute__((buffer_location("QDR"))) int *y)
    ```
  - Or use:
    ```
    #define DDR __global __attribute__((buffer_location("DDR")))
    #define QDR __global __attribute__((buffer_location("QDR")))
    ```

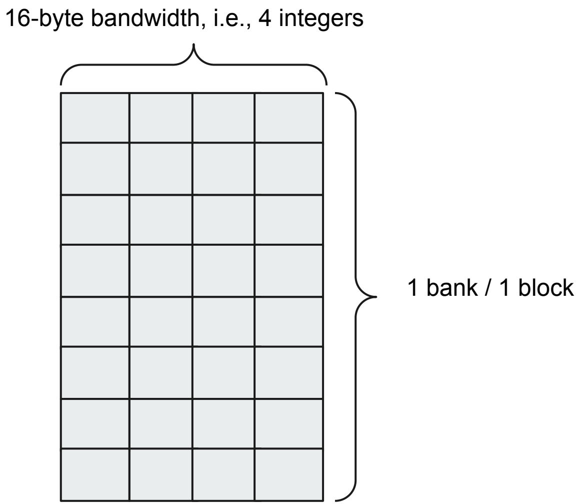
Constant memory

- Constant memory is located in global memory
- But the kernel loads it into an on-chip cache shared by all work-groups at runtime
  - Constant cache is implemented using on-chip RAM blocks
  - However it has large penalties for high cache misses
- By default, constant memory is 16 kB but you can adapt it to your need:
  - With the offline compiler option “**-const-cache-bytes=<N>”**, where <N> is the constant memory size in bytes.
  - Ma use of “__constant” in your kernel to identify constant memory arrays :
  - `__kernel void myKernel ( __constant int * A, ... )`
- You cannot define constant, the kernel argument cannot be assigned to a non-default memory
- More on this in the Intel documentation ...

```
// If the host defines always the same constant values
// Add it directly to the kernel design (A ROM will be created)
// __constant can only created outside the kernel function
__constant int array[4] = {0,1,2,3,4};
__kernel void kernel_with_constant ( ... ) {}
```

Local memory

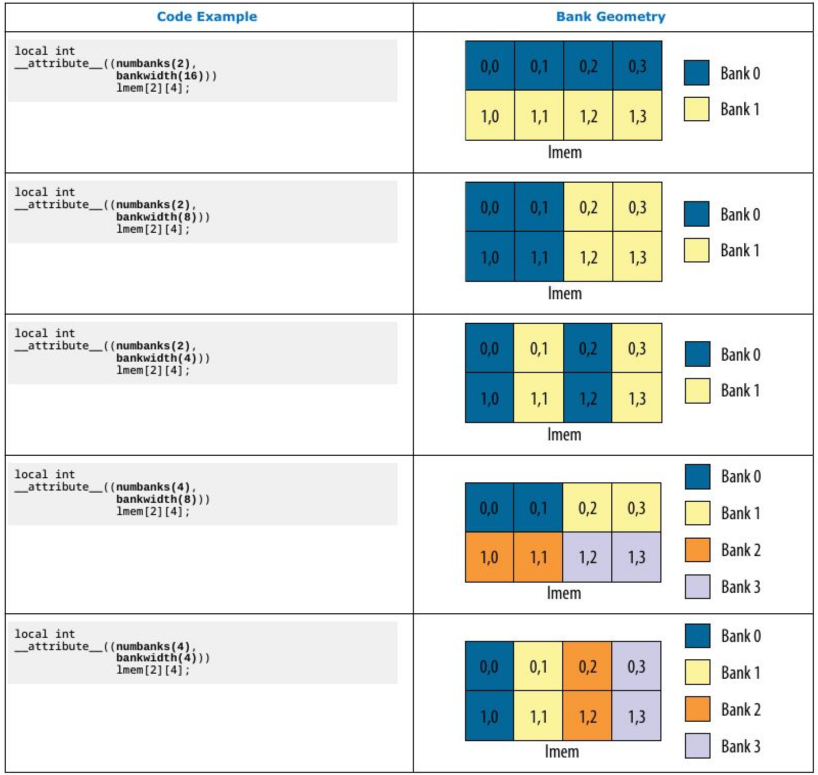
- Local data can be stored in separate local memory banks for parallel memory accesses
- Number of banks of a local memory can be adjusted (e.g., to increase the parallel access)
- Add the following attributes "`__attribute__((numbanks(1),bankwidth(16))` ":
  - numbanks(1) : create a single bank
  - bankwidth(16): assign a 16-byte bank width
- Ex lmem[8][4]
  - No two element can be accessed in parallel in lmem
  - Single bank local memory
- All rows accessible in parallel with `numbanks(8)`
- Different configurations patterns can be adopted



local int __attribute__((numbanks(1),bankwidth(16))) lmem[8][4]

Local memory

(source: Intel documentation)



Code Example	Bank Geometry

`local int`
`__attribute__((numbanks(2),`
`               bankwidth(16)))`
`               lmem[2][4];`
	🟦 0,0 🟦 0,1 🟦 0,2 🟦 0,3
	🟨 1,0 🟨 1,1 🟨 1,2 🟨 1,3
	lmem
	🟦 Bank 0
	🟨 Bank 1

`local int`
`__attribute__((numbanks(2),`
`               bankwidth(8)))`
`               lmem[2][4];`
	🟦 0,0 🟦 0,1 🟨 0,2 🟨 0,3
	🟦 1,0 🟦 1,1 🟨 1,2 🟨 1,3
	lmem
	🟦 Bank 0
	🟨 Bank 1

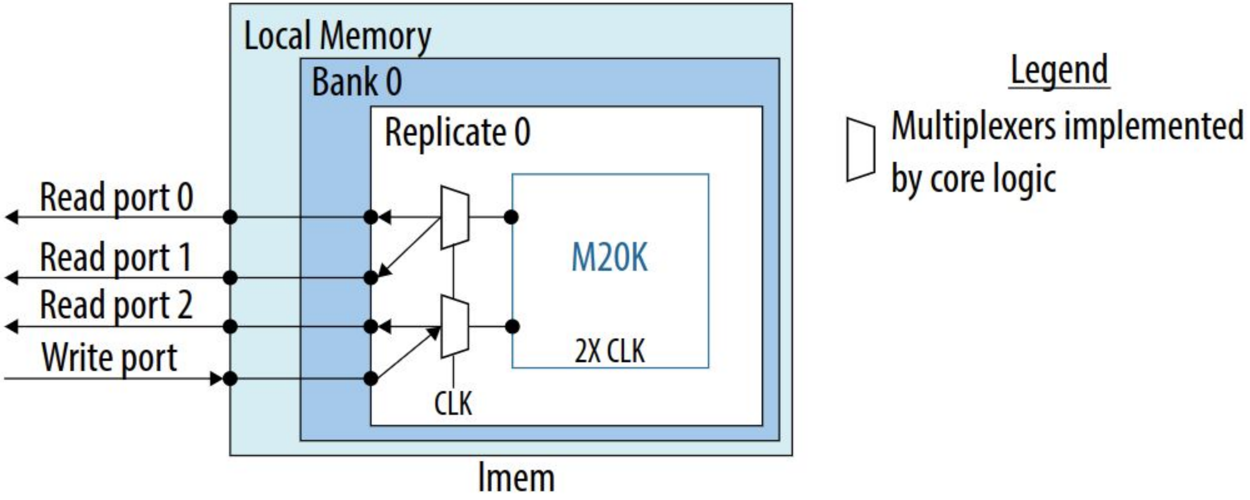
`local int`
`__attribute__((numbanks(2),`
`               bankwidth(4)))`
`               lmem[2][4];`
	🟦 0,0 🟨 0,1 🟦 0,2 🟨 0,3
	🟦 1,0 🟨 1,1 🟦 1,2 🟨 1,3
	lmem
	🟦 Bank 0
	🟨 Bank 1

`local int`
`__attribute__((numbanks(4),`
`               bankwidth(8)))`
`               lmem[2][4];`
	🟦 0,0 🟦 0,1 🟨 0,2 🟨 0,3
	🟧 1,0 🟧 1,1 🟪 1,2 🟪 1,3
	lmem
	🟦 Bank 0
	🟨 Bank 1
	🟧 Bank 2
	🟪 Bank 3

`local int`
`__attribute__((numbanks(4),`
`               bankwidth(4)))`
`               lmem[2][4];`
	🟦 0,0 🟨 0,1 🟧 0,2 🟪 0,3
	🟦 1,0 🟨 1,1 🟧 1,2 🟪 1,3
	lmem
	🟦 Bank 0
	🟨 Bank 1
	🟧 Bank 2
	🟪 Bank 3

Local memory: double pumping

- Double pumping increase virtually the number of ports for local memory
- By doubling the local-memory-clock frequency



(source: Intel documentation)

Local memory: double pumping

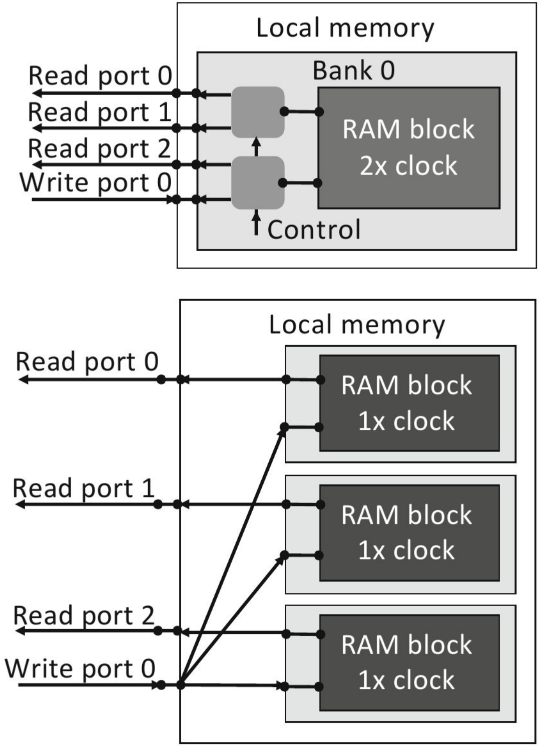
- Avantages:
  - Increases the number of available physical ports
  - May reduce RAM usage by reducing replication
- Disadvantages:
  - Higher logic and latency as compared to single pumped configuration
  - Might reduce kernel clock frequency
- Pumping configuration can be define using “`__attribute__((singlepump))`” and “`__attribute__((doublepump))`”

Local memory replication

- The offline compiler can replicate the local memory
- This allows to create multiple ports
- Behaviour:
  - All read ports will be accessed in parallel
  - All write ports are connected together
  - Data between replicate is identical
- Parallel access to all ports is possible but consumes more resources
- "`__attribute__((max_replicates(3)))`" control the replication factor

```
__kernel void three_parallel_access(int raddr, int waddr) {
int __attribute__((memory,numbanks(1),singlepump,max_replicates(3)))
lmem[16];
lmem[waddr] = lmem[raddr] + lmem[raddr + 1] + lmem[raddr + 2];
}
```

(source: Intel documentation)



(source: Design of FPGA-Based Computing Systems with OpenCL)

Private memory

- Private memory is the on-chip memory dedicated to a single work-item
- Variable without qualifiers declared inside kernel function are defined as private memory
- Private memory can be register or RAM blocks
- To enforce the usage of register, “`__attribute__((register))` ”
  - Ex: `int __attribute__((register)) array[SIZE];`
- You can also apply memory attributes to data members of a struct

```
struct State {
        int array[100] __attribute__((__memory__));
        int reg[4] __attribute__((__register__));
};
```

- `array[100]` is stored in RAM blocks
- `reg[4]` is stored in register