3 General Performance Model and Optimizations for FPGAs

In this chapter, we will discuss our general performance model for FPGAs starting from a single-pipeline model and then extending it for data parallelism. Then, we outline the difference between the two programing models available in Intel FPGA SDK for OpenCL based on this model. In the next step, we discuss multiple HLS-based optimization techniques for FPGAs ranging from basic compiler-assisted optimizations to advanced manual optimizations, and explain how each relates to our model. The contents of this chapter have been partially published in [17].

3.1 General Performance Model

3.1.1 Single-pipeline Model

For a given pipeline with a depth of P, a loop trip count of L (i.e. number of inputs) and an initiation interval of *II*, the total number of clock cycles to finish computation is:

$$T_{cycle} = P + II \times (L - 1) \tag{3-1}$$

Here, P cycles are required until the pipeline is filled and the first output is generated, and after that, a new output is generated every *II* cycles. To convert this value to time, we have:

$$T_{seconds} = \frac{T_{cycle}}{f_{max}} = \frac{P + II \times (L - 1)}{f_{max}} \tag{3-2}$$

In Eq. (3-2), f_{max} is the operating frequency of the FPGA that is determined after placement and routing and is typically between 150 to 350 MHz on Intel Stratix V and Arria 10 devices. Among the parameters in Eq. (3-2), P is controlled by the compiler; however, as a general rule of thumb, simpler code will result in a shorter pipeline and lower P. L is also application-dependent and cannot be directly controlled by the user. f_{max} is also generally a function of circuit complexity and size. Loop-carried dependencies and feedbacks in the design will adversely affect f_{max}. Moreover, the bigger the design is and the closer utilization of each resource is to 100%, the more f_{max} will be lowered due to placement and routing complications. In Section 3.2.4.4, we will show an advanced optimization technique that can significantly improve f_{max} in Single Work-item kernel. The only remaining parameter is *II*. This parameter is the one that can be most directly influenced by the programmer and hence, most of the performance optimization effort will be spent on improving this parameter.

II is influenced by multiple factors: loop-carried dependencies, shared on-chip resources like shared ports to local memory buffers implemented as multi-ported RAM/ROM, and accesses to/from external memory and on-chip channels since they can be stalled. These sources can be split into two groups: sources that affect compile-time initiation interval (II_c), and sources that affect run-time initiation interval (II_r). For Single Work-item kernels, the effect of loop-carried dependencies and shared on-chip resources is determined at compile-time and II_c is adjusted accordingly. In NDRange kernels, loops are *not* pipelined and hence, no such analysis is done and we can assume $II_c = 1$. However, in both kernel types, accesses to external memory and channels will still influence II_r. By default, the compiler inserts enough stages in the pipeline to hide the *minimum* latency of these operations, and accesses that take longer at run-time result in a pipeline *stall*. To estimate compile-time initiation interval (II_c), we consider each kernel type separately (Fig. 3-1):

- **Single Work-item kernels:** II_c in this case depends on the number of stall cycles per iteration (N_d) determined by the compiler and will be equal to $N_d + 1$. Hence, Eq. (3-1) transforms into:

$$T_{cycle} = P + (N_d + 1) \times (L - 1) \tag{3-3}$$

- **NDRange kernels:** In these kernels, even though we can assume $II_c = 1$, we need to take the overhead of barriers into account. Total run time for an NDRange kernel with N_b barriers is:

$$\begin{aligned} T_{cycle} &= \sum_{i=0}^{N_b}(P_i + L_i - 1) = \left(\sum_{i=0}^{N_b} P_i\right) + (N_b + 1) \times (L - 1) \\ &= P + (N_b + 1) \times (L - 1) \end{aligned} \tag{3-4}$$

In Eq. (3-4), P_i and L_i show the pipeline length and number of inputs (work-items) for each pipeline in an NDRange kernel. Since the number of work-items is fixed per kernel, L_i for every pipeline is the same and equal to L. Furthermore, we will call the accumulated length of all the pipelines, P. After simplifying the equation, we reach a statement that is very similar to Eq. (3-3). **In practice, the number of barriers in an NDRange kernel plays a similar role to that of stalls inserted in the pipeline due to dependencies in a Single Work-item kernel, and we can assume II_c is equal to $(N_b + 1)$ instead of one.**

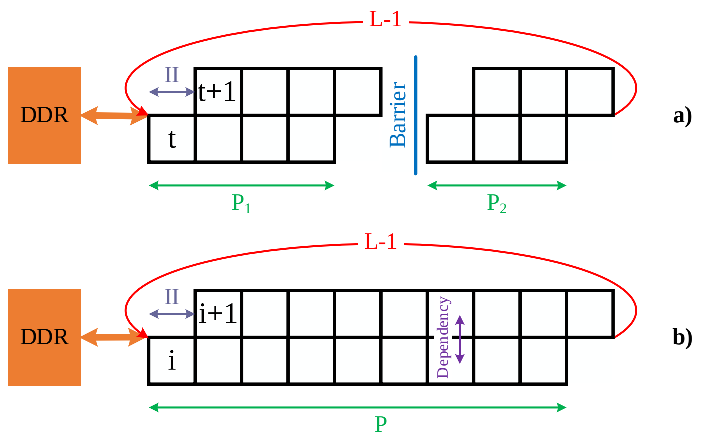

Figure 3-1 NDRange (a) vs. Single Work-item (b) pipeline model

To take the effect of external memory accesses into account and estimate run-time initiation interval (II_r), we use a simple model for external memory. For N_m bytes read from and written to external memory per cycle and an external memory bandwidth per clock cycle of BW, we have:

$$II_r > \frac{N_m}{BW} \tag{3-5}$$

Here, *BW* is determined by the specifications of the external memory on the FPGA board. Furthermore, since our model is simplified and does not take coalescing, alignment and contention from different external memory accesses into account, the right-hand size of (3-5) only shows the *minimum* II_r.

Putting everything together, we have:

$$II > \max(II_c, II_r) \Rightarrow II > \max\left(\begin{cases} N_d + 1 \\ N_b + 1 \end{cases}, \frac{N_m}{BW}\right) \tag{3-6}$$

We ignore the role of stalls caused by on-chip channels here since if the channels are deep enough and the rate of channel reads and writes is similar, channel stalls will be very rare.

3.1.2 Extension for Data Parallelism

To extend our model for cases where data parallelism in form of loop unrolling, SIMD or compute unit replication is employed with a degree of parallelism of N_p (Fig. 3-2), run time can be calculated as:

$$T_{cycle} = P' + II \times \frac{(L - N_p)}{N_p} \tag{3-7}$$

In this case, the pipeline depth generally increases compared to the case where data parallelism is not present. However, for an $L \gg P'$, Eq. (3-7) points to a performance improvement of nearly N_p times since the loop trip count is effectively reduced by a factor of N_p. On the other hand, data parallelism also increases memory pressure by a factor of N_p and hence, *II* will be affected as follow:

$$II > \max\left(\begin{cases} N_d + 1 \\ N_b + 1 \end{cases}, \frac{N_m \times N_p}{BW}\right) \tag{3-8}$$

Based on Eq. (3-7) and (3-8), when data parallelism is present, assuming that sufficient external memory bandwidth is available, performance will improve by a factor close to N_p.

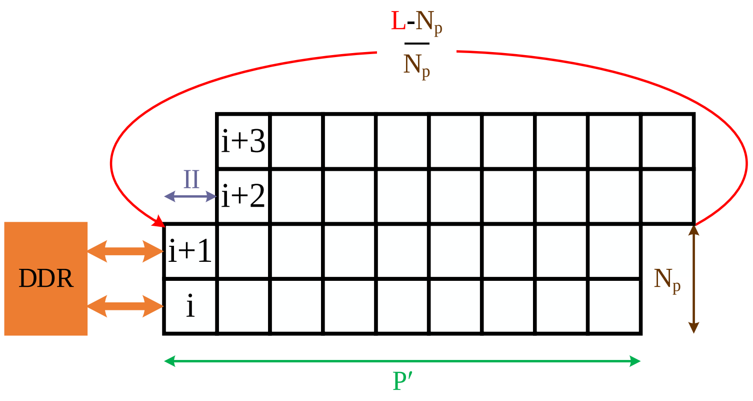

Figure 3-2 Pipeline model for data parallelism

3.1.3 General Optimization Guidelines

Based on our model, we conclude that to improve performance for an HLS-based design on FPGAs, our optimization effort should be focused on:

- Reducing stalls (N_d) in Single Work-item kernels
- Reducing number of barriers (N_b) in NDRange kernels
- Reducing external memory accesses (N_m)
- Increasing data parallelism (N_p)

3.1.4 Single Work-item vs. NDRange Kernels

One important factor in OpenCL-based designs for Intel FPGAs is to decide whether to use Single Work-item or NDRange kernels. Fig. 3-3 shows the most important difference between these two kernel types. Even though we explained earlier that in NDRange kernels, threads are pipelined and no thread-level parallelism exists by default, the programming model itself assumes that threads are running in parallel and hence, the issue distance between the threads neither appears nor can be influenced in the kernel code. Due to this reason, local memory-based optimizations in NDRange kernels require barriers since there is no direct way of transferring data between threads in an NDRange kernel. In contrast, in Single Work-item kernels there is a minimum distance of one clock cycle between loop iterations. It is possible to take advantage of this issue distance to directly transfer data from one loop iteration to another, especially to resolve loop-carried dependencies. This type of communication can be realized using single-cycle reads and writes from and to the plethora of registers that are available in every FPGA. Based on this analysis, we conclude that **in Single Work-item kernels, it might be possible to fully resolve iteration dependencies and reduce N_d to zero; however, in NDRange kernels where local memory-based optimizations are employed, barriers will always be required and N_b will never become zero. Hence, based on Eq. (3-6), a Single Work-item kernel can potentially have a lower effective II_c compared to its NDRange equivalent.** This shows the clear advantage of the Single Work-item programming model compared to NDRange for FPGAs. Moreover, shift registers, which are an efficient storage type on FPGAs, can only be inferred in Single Work-item kernels (Section 3.2.4.1).

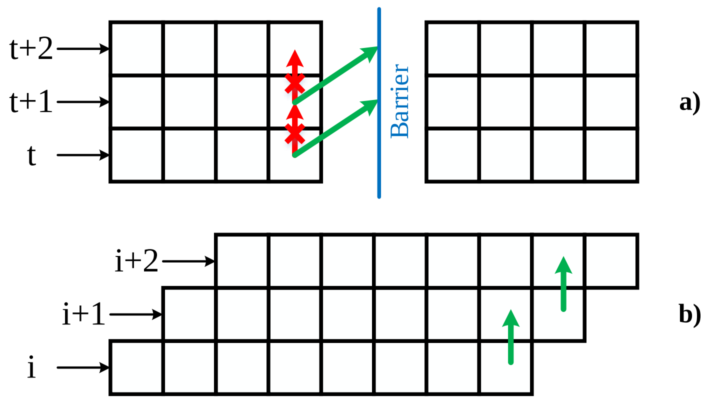

Figure 3-3 Data sharing in (a) NDRange and (b) Single Work-item kernels

On the other hand, NDRange kernels also have their own advantages in certain applications. In cases where an $II_c = 1$ can be achieved, a Single Work-item kernel is preferred. However, for cases where this cannot be achieved, NDRange kernels could potentially achieve better performance. This difference stems from the fact that II_c in Single Work-item kernels is static and is determined based on the worst case loop-carried or load/store dependency at compile-time, while in NDRange kernels, initiation interval is determined at run-time by the thread scheduler. Because of this, in cases where $II_c = 1$ cannot be achieved in the Single Work-item implementation of an application, the thread scheduler in the NDRange equivalent might be able to achieve a lower *average initiation interval* by reordering the threads at run-time, compared to the Single Work-item equivalent with a fixed worst-case initiation interval.

In summary, the following points should be taken into account to decide whether to choose NDRange kernels for a design, or Single Work-item kernels:

- Applications with non-fully-pipelineable loops (e.g. loops with variable exit conditions or complex loop-carried/load/store dependencies) or random external memory accesses can potentially perform better using the NDRange programming model
- Every other application will potentially perform better using the Single Work-item programming model, especially if registers or shift registers can be used to efficiently resolve loop-carried dependencies.

3.2 HLS-based Optimization Techniques for FPGAs

In this section, we will discuss multiple different optimization techniques for HLS-based designs on FPGAs, ranging from basic compiler-assisted optimizations to advanced manual optimizations that require significant code refactoring. The basic optimizations discussed here are techniques that are introduced in Intel's OpenCL documents [18, 19], while most of the advanced ones are not directly discussed in these documents. We only discuss kernel optimization here. Some of the discussed optimizations are only applicable to one kernel type; "NDR" and "SWI" are used to mark optimizations specific to NDRange and Single Work-item kernels, respectively. Optimizations applicable to both have been marked as "BOTH".

3.2.1 Basic Compiler-assisted Optimizations

3.2.1.1 *restrict* keyword (BOTH):

The *restrict* keyword can be added to global pointers in the kernel to prevent the compiler from assuming false pointer aliasing. This optimization usually has little to no effect in NDRange kernels; however, it is a crucial optimization in Single Work-item kernels, which, if not used, can result in very high initiation interval or even full sequential execution. This optimization can improve performance by reducing N_d, and consequently, reducing II_c.

3.2.1.2 *ivdep* pragma (SWI):

The *ivdep* pragma is used to prevent the compiler from assuming false load/store dependencies on global buffers. This pragma in only applicable to Single Work-item kernels and should be used with extreme care since incorrect usage can result in incorrect output. Even though this pragma can also be used for local buffers, it is very rare for the compiler to detect a false dependency on such buffers. This optimization can improve performance by reducing N_d, and consequently, reducing II_c.

3.2.1.3 Removing thread scheduler (SWI):

The run-time thread scheduler is only needed for NDRange kernels and is not required for Single Work-item kernels. However, in cases where a Single Work-item kernel is launched from the host using the *clEnqueueNDRangeKernel()* function, the scheduler is required for correct kernel launch. On the other hand, for cases where the *clEnqueueTask()* function is used to launch such kernel, a compiler-provided attribute can be used to remove the scheduler to save some FPGA resources. The only downside of this optimization is that *clEnqueueTask()* has been deprecated in OpenCL 2.0. This optimization does not directly affect performance and only reduces area utilization by a small amount.

3.2.1.4 Setting work-group size (NDR):

Intel FPGA SDK for OpenCL Offline Compiler provides attributes for NDRange kernels to set the exact or maximum kernel work-group size. This allows the compiler to minimize the size of local memory buffers based on the user-supplied information. Furthermore, the SIMD attribute can only be used if the exact work-group size is set by the programmer. If this information is not supplied, the compiler will assume a default work-group size of 256, potentially wasting valuable Block RAM resources if a smaller work-group size is used at run-time. Using this attribute comes at the cost of kernel execution failure if the run-time-provided work-group size supplied by the host does not align with the information supplied to the compiler in the kernel. This optimization does not have a direct effect on performance; however, the area reduction from this optimization can potentially allow performance improvement by using more data parallelism or larger block size.

3.2.1.5 Data parallelism (BOTH):

For NDRange kernels, data parallelism can be achieved by using the SIMD or *num_compute_units()* attributes. SIMD will provide work-item-level parallelism, while *num_compute_units()* provides work-group-level parallelism. Using SIMD has a lower area overhead since it does not require full compute unit replication, and only the pipeline stages are replicated so that multiple work-items can be issued and processed in parallel. Furthermore, internal and external memory accesses can be coalesced in this case, allowing higher internal and external memory bandwidth with minimum amount of contention and area overhead. However, using this attribute is subject to multiple limitations. First, the SIMD length must be a power of two up to maximum of 16 (which is an artificial compiler limitation since any arbitrary SIMD length should be implementable on an FPGA). Second, no thread-id-dependent branches should exist in the code. Finally, work-group size should be set by the programmer (Section 3.2.1.4) and be divisible by SIMD length. Using *num_compute_units()* has neither of these limitations; however, it comes at the cost of higher area overhead due to complete compute unit replication, and lower memory throughput compared to SIMD due to multiple narrow accesses competing for the external memory bandwidth instead of one wide coalesced access. For Single Work-item kernels, a SIMD-like effect can be achieved using loop unrolling, without any of the limitations that exist for using SIMD. However, area overhead of loop unrolling will be minimized when a loop with a trip count known at compile-time is either fully unrolled, or partially unrolled with a factor that the trip count is divisible by. As explained in Section 3.1.2, these techniques can improve performance by a factor close to the degree of parallelism if sufficient external memory bandwidth is available.

A direct effect of data parallelism in form of SIMD for NDRange kernels and unrolling for Single Work-item kernels is that external memory accesses which are consecutive in the dimension that SIMD or unrolling is applied on will be coalesced by the compiler into wider accesses at compile-time, allowing better utilization of the external memory bandwidth. Compared to having multiple narrow accesses per iteration to external memory, a few wide accesses result in much less contention on the memory bus and much more efficient utilization of the external memory bandwidth. However, using SIMD and unrolling over non-consecutive external memory accesses could instead lead to many narrow access ports and lower performance due to large amount of contention on the memory buss. Using either of these techniques also has a similar effect on local memory buffers. Using SIMD and unrolling over consecutive local memory accesses leads to access coalescing and data interleaving with minimal area overhead, while applying these over non-consecutive accesses will result in high replication factors for local buffers and waste of FPGA area.

3.2.2 Basic Manual Optimizations

3.2.2.1 Shift register for floating-point reduction (SWI):

On most FPGAs, floating-point operations cannot be performed in one clock cycle (unless at the cost of extremely low operating frequency). Because of this, for floating-point reduction operations where the same variable appears on both sides of the assignment (i.e. the reduction variable), data dependency on this variable prevents pipelining with an initiation interval of one and instead, the initiation interval equals the latency of the floating-point operation (e.g. 8 clocks for floating-point addition on Intel Stratix V). As suggested in Intel’s documents [18], this dependency can be eliminated by inferring a shift register with a size equal to the latency of the floating-point operation. In this case, in every iteration data is read form the head of the shift register and written to its tail, with the shift register being shifted afterwards. The use of an array of reduction variables instead of just one such variable effectively eliminates the dependency, reducing N_d to zero and consequently, reducing II_c to one for the reduction loop. To obtain the final output, another reduction is needed on the content of the shift register. It is worth noting that unlike what is suggested in [18], the size of the shift register does not need to be one index bigger than the latency of the reduction operation and in our experience, even if the size is exactly equal to the latency of the operation, the dependency can be eliminated without lowering operating frequency. The transformation from an unoptimized floating-point reduction to optimized version with shift register is shown in Fig. 3-4.

```
float final_sum = 0.0f;

for (int i = 0; i < size; i++)
{
   final_sum += in[i];
}
```

a)

```
#define FADD_LATENCY     8  // latency of floating-point operation

// shift register definition and initialization
float shift_reg[FADD_LATENCY] = {0.0f}, final_sum = 0.0f;

for (int i = 0; i < size; i++)
{
   // add and write to shift register
   shift_reg[FADD_LATENCY - 1] = shift_reg[0] + in[i];

   // shifting
   #pragma unroll
   for (int j = 0; j < FADD_LATENCY - 1; j++)
   {
      shift_reg[j] = shift_reg[j + 1];
   }
}

//final reduction
#pragma unroll
for (int i = 0; i < FADD_LATENCY; i++)
{
   final_sum += shift_reg[i];
}
```

b)

Figure 3-4 Shift register optimization for floating-point reduction

This optimization is usually not enough to provide good performance on its own and it needs to be followed by data parallelism (Section 3.2.1.5). Even though the resulting optimized loop can be partially unrolled by using the unroll pragma to further improve the performance, doing so will break the shift register optimization and requires that the size of the shift register is increased further to accommodate for the unrolling. With large unroll factors, this method can result in large area overhead to implement the shift register. As a much better optimized alternative, it is possible to add data parallelism to the optimized code from 3-4 b) by performing manual unrolling as depicted in Fig. 3-5. In this case, the original loop is split into two loops, with the inner loop being fully unrolled and having a trip count equal to the unroll factor, and the exit condition of the outer loop being adjusted accordingly. In this case, the shift register optimization is not required for the inner loop since full unrolling effectively eliminates the dependency on the reduction variable, and it only needs to be applied to the outer loop. This method makes it possible to achieve efficient data parallelism alongside with the shift register optimization for floating-point reduction, without needing to increase the size of the shift register.

```
#define FADD_LATENCY     8  // latency of floating-point operation
#define UNROLL           16 // unroll factor

// shift register definition and initialization
float shift_reg[FADD_LATENCY] = {0.0f}, final_sum = 0.0f;

// loop exit condition calculation
int exit = (size % UNROLL == 0) ? (size / UNROLL) : (size / UNROLL) + 1;
for (int i = 0; i < exit; i++)
{
   // unrolled addition
   float sum = 0.0f;
   #pragma unroll
   for (int j = 0; j < UNROLL; j++)
   {
      int index = i * UNROLL + j;
      sum += (index < size) ? in[index] : 0.0f;
   }

   // write to shift register
   shift_reg[FADD_LATENCY - 1] = shift_reg[0] + sum;

   // shifting
   #pragma unroll
   for (int j = 0; j < FADD_LATENCY - 1; j++)
   {
      shift_reg[j] = shift_reg[j + 1];
   }
}

//final reduction
#pragma unroll
for (int i = 0; i < FADD_LATENCY; i++)
{
   final_sum += shift_reg[i];
}
```

Figure 3-5 Optimized floating-point reduction with unrolling

As a final note, on the Intel Arria 10 FPGA, it is possible to use single-cycle floating-point accumulation and hence, the shift register optimization is not required on this FPGA. However, due to the requirements for correct inference of single-cycle accumulation by the compiler on Arria 10, it is required that data parallelism is implemented using the aforementioned method rather than applying partial unrolling directly to the reduction loop.

3.2.2.2 Calculating constants on host instead of kernel (BOTH):

For cases where a value is calculated on the kernel and remains constant throughout the kernel execution, calculation of this constant can be moved to the host code to save FPGA area. This optimization is specifically useful in cases where calculation of a constant involves complex mathematical functions (division, remainder, exponentiation, trigonometric functions, etc.) and could use a significant amount of FPGA area. This optimization does not directly lead to performance improvements; however, area savings from this optimization could allow more parallelism or higher f_{max}.

3.2.2.3 Avoiding branches on global memory addresses and accesses (BOTH):

For cases where in a kernel, the external memory address that is accessed could change based on run-time variables, instead of choosing the correct address using branches, it is best if both accesses are performed and results are stored in temporary variables and instead, the correct output is chosen from the temporary variables. This will prevent dynamic addressing and potentially allow the compiler to coalesce accesses when SIMD or unrolling is used. A similar problem exists when a branch involves choosing a value from two different global memory addresses. Also in this case, moving the accesses out of the branch and storing their value in two temporary variables, and instead using the temporary variables in the branch could allow correct coalescing when SIMD or loop unrolling is used. Apart from the area saving due to lower number of ports going to external memory, this optimization could also improve performance by decreasing N_m and consequently, reducing II_r. However, for cases where SIMD or unrolling are not used and compile-time coalescing is not required, the best choice is to choose the correct address and only perform one access to global memory to minimize the number of access ports.

3.2.3 Advanced Compiler-assisted Optimizations

3.2.3.1 Manual external memory banking (BOTH):

By default, Intel FPGA SDK for OpenCL Offline Compiler interleaves all the global buffers between the two (or more) DDR memory banks available on an FPGA board so that the bandwidth of all the banks is efficiently shared between all the buffers. In cases where multiple global buffers exist with different access rates, or a few narrow accesses to global memory exist in the kernel, this automatic interleaving achieves best memory performance. However, in our experience, for cases where only two wide global memory accesses (with or without some accompanying narrow ones) exist in the kernel, each to a different global buffer, this automatic interleaving does not perform optimally and disabling it can improve performance if the buffers are each pinned to a different memory bank. To achieve this, the kernel should be compiled with a specific compiler switch to disable automatic interleaving [19], and the global buffers in the host code should be created with an additional flag that allows the user to manually determine which buffer should reside on which memory bank. In this case, performance is improved by increasing the effective BW from Eq. (3-5), and consequently, reducing II_r. Furthermore, for cases where multiple global memory types exist on the board (DDR, QDR, HBM, etc.), this technique can be used to manually allocate some of the global buffers on the non-default memory type(s).

3.2.3.2 Disabling cache (BOTH):

By default, Intel FPGA SDK for OpenCL Offline Compiler generates a private cache for every global memory access in a kernel if it cannot determine the exact access pattern. This cache is implemented using FPGA Block RAMs and is *not* shared between different accesses to the same global buffer. Despite its simplicity and small size (512 Kbits), this cache can be effective for designs that have good spatial locality that is not exploited by the programmer. However, in two cases this cache not only will not improve performance, but can potentially even reduce it:

- In cases where random accesses exist in the kernel with minimal spatial locality, the cache hit-rate will be very low and hence, disabling it can improve performance by avoiding the overhead of the cache mechanism. The hit-rate of the cache can be determined by using Intel FPGA Dynamic Profiler for OpenCL.
- In cases where data locality is manually exploited by the programmer by using on-chip memory, which will be the case for all well-optimized designs, the cache will not be required anymore and using it will only waste valuable Block RAM resources. In such cases, the cache can be disabled to save area.

To selectively disable the cache for a global buffer, it can be *falsely* marked as *volatile* in the OpenCL kernel. To completely disable the cache for all global buffers in a kernel, "--opt-arg -nocaching" can be added to the kernel compilation parameters. In our experience, this cache is usually not created in NDRange kernels but it is nearly always created in Single Work-item.

3.2.3.3 *Autorun* kernels (SWI):

Intel FPGA SDK for OpenCL provides a specific *autorun* attribute for Single Work-item kernels that do not have an interface to the host or the FPGA external memory but can communicate with other *autorun* or *non-autorun* kernels using on-chip channels [19]. This kernel type does not need to be invoked from the host and automatically launches as soon as the FPGA is programmed. Furthermore, the kernel is automatically restarted whenever it finishes execution. This type of kernel has two main use cases:

- For designs in which data is sent and received directly via the FPGA on-board peripherals, and no interaction from the host is required, this kernel type can be used so that the FPGA can act as a standalone processor. This type of design is specifically useful for network-based processing where data is streamed in and out through the FPGA on-board network ports.
- For streaming designs that require replication of a Single Work-item kernel, this attribute can be used alongside with the multi-dimensional version of the *num_compute_units()* attribute (different from the single-dimensional one used for NDRange kernels). In this case, a *get_compute_id()* function is supplied by the compiler that can be used to obtain the unique ID of each kernel copy at compile-time and then, each kernel copy can be customized using this ID. This attribute is specifically useful for streaming designs in form of multi-dimensional systolic array or single-dimensional ring architectures. Apart from the obvious area reduction duo to lack of interface to host and memory for this kernel type, in our experience, using this kernel type also results in efficient *floor-planning* and good scaling of operating frequency even with tens of kernel copies.

3.2.3.4 Flat compilation (BOTH):

Using Intel FPGA SDK for OpenCL, the FPGA is automatically reprogrammed at run-time with the pre-compiled FPGA bitstream before kernel execution. On the Intel Stratix V device, this reconfiguration is performed using Configuration via Protocol (CvP) [20]. However, CvP update is not supported on the Intel Arria 10 device [21] and hence, the FPGA is reprogrammed at run-time using Partial Reconfiguration (PR) via PCI-E. In this case, the logic related to the OpenCL BSP is the *static* part of the design which resides on a fixed section of the FPGA and is never reconfigured. The rest of the FPGA area can be used by the OpenCL kernel, which acts as the *dynamic* part of the design and is reconfigured at run-time. Using PR for OpenCL on Arria 10 imposes extra placement and routing constraints on the design to ensure correct run-time reconfiguration; this comes at the cost of more placement and routing complications on this device and consequently, worse timing or even an outright unfittable or unrouteable design, especially when FPGA logic or Block RAM utilization is high. Furthermore, in our experience, run-time partial reconfiguration through PCI-E has a high failure rate, resulting in the program or even the OS crashing in many cases. To avoid these issues, the possibility of *flat* compilation on Arria 10 is also provided by the compiler, which disables PR and place and routes the BSP and the OpenCL kernel as one flat design. This eliminates all extra constraints for PR and allows the programmer to use the FPGA area more efficiently and achieve best timing. This compiler optimization can improve performance by both providing better area utilization efficiency for large designs that would have failed to fit or route with the PR flow, and also improving f_{max}. However, using flat compilation comes at the cost of two shortcomings. One is the longer run-time reconfiguration time since the FPGA has to be reconfigured via JTAG instead of PCI-E (15-20 seconds vs. less than 5 seconds). The other is that since all clock constraints except for the kernel clock in the BSP are relaxed for flat compilation, other clocks like PCI-E and DDR might fail to meet timing and hence, the user has to try multiple seeds and manually check the timing report to make sure all timing constraints are met. In practice, we have seen that for large NDRange designs, it might not be possible to meet the timing constraints of the non-constrained clocks regardless of how many different seeds are tried. Hence, this optimization should be limited to highly-optimized Single Work-item designs and it is probably best to use the default PR flow for NDRange.

3.2.3.5 Target f_{max} and seed sweep (BOTH):

By default, Intel FPGA SDK for OpenCL Offline Compiler balances pipeline stages in the design by inserting extra registers and FIFOs towards a target f_{max} of 240 MHz. This target can be increased so that the compiler would insert more registers and FIFOs in the pipeline, potentially allowing a higher operating frequency, at the cost of higher logic and Block RAM utilization. This optimization is not viable for cases where logic and Block RAM utilization is already high, since the extra area usage could instead lead to more routing congestion and worse f_{max}. Furthermore, careful attention is required when changing the target f_{max} for Single Work-item kernels since if the target is set too high, the compiler might have to increase the initiation interval of some loops to achieve the target and this could instead result in performance slow-down despite higher operating frequency. The compiler also provides the possibility to change the placement and routing seed, which can result in better (or worse) f_{max}, at the cost of no extra area utilization. These two optimizations can be used as the *last step* of optimization to maximize f_{max} for a given design by compiling multiple instances of the design, each with a different seed and f_{max} target, and choosing the one with highest operating frequency.

3.2.4 Advanced Manual Optimizations

3.2.4.1 Shift register as local data storage (SWI):

Each FPGA has a plethora of registers available that can be used for temporary data storage. The main advantage of registers over Block RAMs for data storage is that access latency to registers is one clock cycle, allowing efficient data sharing between loop iterations without increasing the loop initiation interval. However, since each ALM has a few registers, and chaining registers requires using FPGA routing resources, implementing large on-chip buffers using registers is not efficient. On the other hand, Block RAMs are suitable for implementing large on-chip buffers, but latency of dynamic access to Block RAMs is not one clock cycle and hence, reading from and writing to on-chip buffers implemented using Block RAMs in a loop can result in read-after-write dependencies and high initiation intervals. With all this, if certain requirements are met, large buffers can be implemented using Block RAMs with an access latency of one clock cycle:

- All accesses to the buffer use static addresses that are known at compile-time
- The content of the buffer is *shifted* once per loop iteration by a fixed amount

This will result in inference of an FPGA-specific on-chip storage called a *shift register* (also called *sliding window* or *line buffer* in literature). Shift registers are suitable for applications that involve the point of computation being shifted over a regular grid, including but not limited to stencil computation, image filtering, and sequence alignment. Due to the above requirements, this on-chip storage type can only be described in Single Work-item kernels.

Fig. 3-6 shows how a Block RAM is used to implement a shift register for 2D stencil computation. In this case, after an initial warm-up period to fill the shift register, all data points involved in the computation of the stencil reside in the shift register buffer at any given clock cycle. By incrementing the starting address of the buffer in the Block RAM, the center of the stencil is effectively shifted forward in the grid, while the relative distance of all neighbors from the starting address remains the same, allowing static addressing in a loop. Since static addressing does not require address decoding, and shifting the buffer forward only involves incrementing the starting address, accesses to shift registers can be done in one clock cycle. Shift registers are one of the most important architectural advantages of FPGAs compared to other hardware.

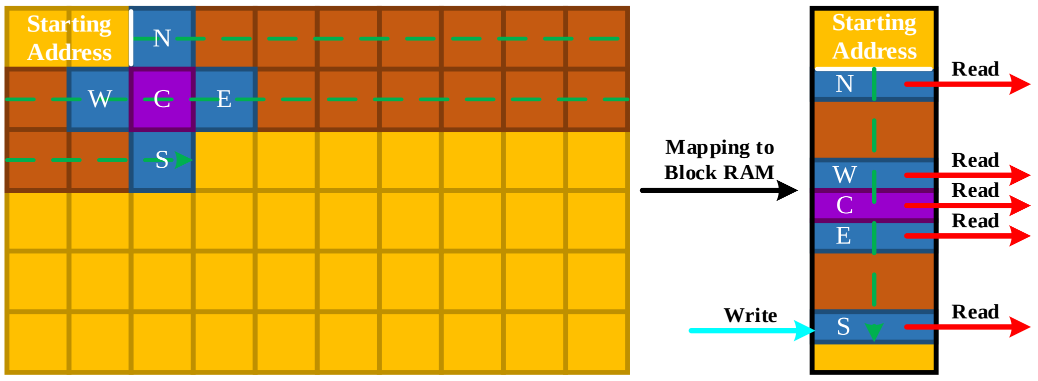

Figure 3-6 Shift register inference

The performance improvement of shift registers is two-fold: compared to using standard Block RAM-based buffers with dynamic access, using shift registers allows the programmer to avoid read-after-write dependencies, reducing N_d to zero and II_c to one. Furthermore, as shown in Fig. 3-6, a shift register can act as an efficient cache for neighboring cells in a grid, reducing redundant off-chip memory accesses (N_m) and improving II_r.

3.2.4.2 Reducing local memory accesses (BOTH):

Local memory-based optimizations are used on every hardware to improve performance by reducing accesses to the slower external memory and instead, storing and accessing frequently-used data in the faster local memory. Due to limited amount of local memory available on each given device, careful attention is required to ensure only widely-used buffers reside in this memory type, and are replaced as soon as they are not required anymore. On CPUs, the cache hierarchy acts as local memory and is mostly hardware-managed; however, programming techniques such as loop tiling can help improve cache hit rate and performance. On modern GPUs, a limited set of registers, some scratchpad memory, and two levels of cache are available. The user-managed resources (registers and scratchpad) make up the smaller chunk of the local memory resources, while the larger chunk consisted of caches is again hardware-managed. On FPGAs, things are different: all local memory resources on FPGAs are user-managed and no explicit cache hierarchy exists. This makes careful utilization and configuration of local memory resources on these devices even more crucial.

Apart from the size of local buffers, the number of accesses to such buffers also plays a crucial role in local memory usage on FPGAs. For small buffers implemented using registers, multiple read and write ports can be connected to the same buffer instance with small area overhead; however, as the number of accesses goes up, the *fan-in* and *fan-out* of the buffer also increase, resulting in routing compilations and lowered f_{max}, or an outright unrouteable design. For larger buffers implemented using Block RAMs, whether in form of shift registers or multi-ported RAM/ROMs, since each Block RAM only has two physical ports, in most cases the buffer needs to be physically replicated on the FPGA to provide the necessary number of access ports for parallel non-stallable accesses to the buffer. For cases with an inefficient access pattern to local memory, this replication can easily result in exhaustion of Block RAM resources on the FPGA. If this happens, the compiler will then restart the compilation and instead shares Block RAM ports between multiple accesses so that replication factor, and consequently, Block RAM utilization, is reduced and the design can fit. Needless to say, port sharing requires arbitration and results in stallable accesses, reducing performance by increasing II_r in pretty much every case.

One compiler-assisted solution to reduce the Block RAM replication factor is to double-pump the Block RAMs. In this case, the number of ports to each Block RAM is effectively doubled since the Block RAM is driven with 2x the clock of the OpenCL kernel. However, correct communication between the OpenCL kernel and the Block RAMs in this case will require extra logic and since there is a physical limit for the operating frequency of Block RAMs (500-600 MHz on current devices), the maximum operating frequency of the OpenCL kernel will be limited to half of that limit (250-300 MHz). The compiler generally employs this technique by default; in fact, for cases where two or more writes to a specific local buffer exist in a kernel, since write ports need to be connected to all replicated instances of the buffer, there is no choice other than double-pumping the Block RAMs to achieve non-stallable accesses. However, the compiler also provides certain attributes for manual double-pumping and port sharing so that the programmer can also influence the compiler's decisions in this case.

A more important method to reduce Block RAM replication factor is to reduce number of accesses to the buffer. This can be done by using temporary registers, or even transposing the local buffer or re-ordering nested loops to allow efficient access coalescing in presence of loop unrolling or SIMD. This optimization, apart from the obvious area reduction, can also allow reducing or completely removing stallable accesses to local memory buffers and improving performance by reducing II_r.

Fig. 3-7 a) shows a code snippet in which a local buffer implemented using Block RAMs is first initialized, and then a reduction operation is performed, with the output being stored in the local buffer. As is evident from the code snippet, one read port and two write ports are required to the *temp* buffer in this part of the kernel. These accesses alone will force the compiler into double-pumping the Block RAMs used to implement this buffer. Furthermore, the buffer will need to be replicated one extra time for every two reads from the buffer (two ports of each double-pumped Block RAM are connected to the write accesses and the remaining two ports can be used for reads). However, by moving the local buffer outside of the reduction loop and replacing it with a temporary register as is shown in Fig. 3-7 b), one read and one write are removed, eliminating the need for buffer replication if up to three reads from the buffer exist in the rest of the kernel, and avoiding the need for double-pumping in case of only one read from the buffer in the rest of the kernel.

```
__global float* a, b;
__local float temp[M];

for(int i = 0; i < M; i++)
{
   temp[i] = 0;
}

for(int i = 0; i < M; i++)
{
   for(int j = 0; j < N; j++)
   {
      temp[i] += a[i] * b[j];
   }
}
```

a)

```
__global float* a, b;
__local float temp[M];

for(int i = 0; i < M; i++)
{
   float reg = 0;

   for(int j = 0; j < N; j++)
   {
      reg += a[i] * b[j];
   }
   temp[i] = reg;
}
```

b)

Figure 3-7 Reducing Block RAM replication by using temporary registers

Another case of reducing Block RAM replication is depicted in Fig. 3-8. In code snippet a), since the inner loop is unrolled on the higher dimension of the local buffer, the accesses to that buffer cannot be coalesced and hence, eight write ports are required to the buffer which results in port sharing. However, by transposing the buffer, i.e. swapping its dimensions as shown in b), this issue can be avoided. In the latter case, only one large coalesced write to the buffer will be required and hence, instead of replicating the buffer, the large write will be made possible by interleaving the buffer across multiple Block RAMs. In this case, for buffers that are small enough to fit in less than eight Block RAMs, eight Block RAMs will still be required to implement the buffer so that enough ports are available; however, for larger buffers, the overhead of interleaving will be minimal. It is worth noting that re-ordering the *i* and *j* loops will also result in correct access coalescing to the local *temp* buffer; however, it will break access coalescing to the global buffer *a*, significantly reducing external memory performance.

```
__global float* a;
__local float temp[N][M];

for(int i = 0; i < M; i++)
{
   #pragma unroll 8
   for(int j = 0; j < N; j++)
   {
      temp[j][i] = a[i * COL + j];
   }
}
```

a)

```
__global float* a;
__local float temp[M][N];

for(int i = 0; i < M; i++)
{
   #pragma unroll 8
   for(int j = 0; j < N; j++)
   {
      temp[i][j] = a[i * COL + j];
   }
}
```

b)

Figure 3-8 Reducing Block RAM replication by transposing the buffer

3.2.4.3 Loop collapse (SWI):

Multiply-nested loops are a recurring pattern in HPC applications. Having such loops incurs extra overhead on FPGAs since, relative to the depths of the loop nest, more registers and Block RAMs will be needed to store the state of the different variables in the loop nest. As an FPGA-specific optimization, loop nests can be *collapsed* into a single loop to avoid this extra area overhead. Even though the same optimization is also regularly performed on OpenMP code running on CPUs, the goal of the optimization on CPUs and FPGAs is completely different. This conversion is shown in Fig. 3-9.

```
for (int y = 0; y < M; y++)
{
   for (int x = 0; x < N; x++)
   {
      compute(x,y);
   }
}
```

a)

```
int x = 0, y = 0;
while (y != M)
{
   compute(x,y);

   x++;
   if (x == N)
   {
      x = 0;
      y++;
   }
}
```

b)

Figure 3-9 Loop collapse optimization

This optimization does not have a direct effect on performance; however, the area reduction resulting from this optimization can open up FPGA resources, providing room for extra parallelism or larger block size and consequently, higher performance. Furthermore, this optimization simplifies the pipeline and can result in lower pipeline latency (P). In the recent versions of Intel FPGA SDK for OpenCL, a new *loop_coalesce* pragma has been introduced that allows this optimization to be performed directly by the compiler. However, using this pragma is subject to certain limitations and does not work for all loop nests. Furthermore, the "exit condition optimization" which will be explained next is only possible after manual loop collapsing.

3.2.4.4 Exit condition optimization (SWI):

Apart from the extra area overhead of multiply-nested loops on FPGAs, having such loops also has another disadvantage: since we want all loops in a loop nest to have an initiation interval of one to achieve maximum performance, the exit condition of all the loops in the loop nest need to be evaluated in one clock cycle. Since the exit conditions depend on each other, a long chain of comparisons and updates on the loop variables is created that will adversely affect the design critical path and reduce f_{max}. In fact, in our experience, **the design critical path of Single Work-item kernels that do not have loop-carried dependencies is nearly always located in the chain of functions required to determine the exit condition of the deepest loop nest in the kernel.** Even though the "loop collapse" optimization introduced earlier reduces the loop nest to one loop, it does not change the loop exit condition. To improve this critical path that gets longer with the depth of the original loop nest, we can replace the exit condition of the collapsed loop with incrementation and comparison on a global index variable that does not depend on the loop variables of the original nested loop. Fig. 3-10 shows the resulting code after applying this optimization to the collapsed loop from Fig. 3-9 b).

```
int x = 0, y = 0, index = 0;
while (index != M * N)
{
   index++;
   compute(x,y);

   x++;
   if (x == N)
   {
      x = 0;
      y++;
   }
}
```

Figure 3-10 Exit condition optimization

For the sake of clarity, we used a basic example in figures 3-9 and 3-10 to show how the loop collapse and exit condition optimizations are applied. However, in real-world scenarios, these optimizations will be applied to much more complex loop nests involving multiple layers of index and block variables. In such cases, the global index variable should be compared with the number of times the original loop nest would have iterated in total. Calculating this number will likely require a complex equation involving the iteration variable and exit condition of all the original loops in the loop nest. This operation can be done in the host code, and the exit condition for the collapsed loop can be passed to the kernel as argument to avoid extra area waste on the FPGA for mathematical computations that will only be performed once per kernel run. **It is worth noting that for deep loop nests, even after applying the exit condition optimization, the design critical path will still consist of the chain of updates and comparisons on the remaining index variables.**

4 Evaluating FPGAs for HPC Applications Using OpenCL

In this chapter, we will port a subset of the Rodinia benchmark suite [13] as a representative of typical HPC applications using the optimization techniques introduced in the previous chapter, and report speed-up compared to baseline, and performance and power efficiency compared to CPUs and GPUs. The contents of this chapter have been partially published in [17] and [22].

4.1 Background

Multiple benchmark suites have been proposed as representatives of HPC applications to evaluate different hardware and compilers. Examples of such suites include Rodinia [13], SHOC [23], OpenDwarfs [24], Parboil [25], PolyBench [26] and many more. Among these suites, Rodinia is regularly used for evaluating performance on different hardware since it includes OpenMP, CUDA and OpenCL versions of multiple benchmarks that can be used to target a wide variety of hardware. Each of the benchmarks in this suite belongs to one of Berkeley's Dwarfs [27]. Each Berkeley Dwarf represents one class of HPC applications that share a similar compute and memory access pattern. We choose Rodinia so that we can take advantage of the existing OpenMP and CUDA implementations for evaluating CPUs and NVIDIA GPUs. Furthermore, we port and optimize the OpenCL versions for FPGAs based on Intel FPGA SDK for OpenCL to be able to perform a meaningful performance comparison between different hardware architectures and show the strengths and weaknesses of them. We evaluate one or two benchmarks from multiple of the Dwarfs, expecting that our FPGA optimization techniques for each application belonging to a Dwarf can be used as guidelines for optimizing other applications belonging to the same Dwarf.

4.2 Methodology

4.2.1 Benchmarks

We use the latest v3.1 of the Rodinia benchmark suite to make sure all the latest updates are applied to the CPU and GPU benchmarks. The benchmarks we evaluate in this study are as follows:

NW: Needleman-Wunsch (NW) is a Dynamic Programming benchmark that represents a sequence alignment algorithm. Two input strings are organized as the top-most row and left-most column of a 2D matrix. Computation starts from the top-left cell and continues row-wise, computing a score for each cell based on its neighbor scores at the top, left, and top-left positions and a reference value, until the bottom-right cell is reached. This computation pattern results in multiple data dependencies. This benchmark only uses integer values.

Hotspot: Hotspot is a Structured Grid benchmark that simulates microprocessor temperature based on a first-order 5-point 2D stencil on a 2D grid and uses single-precision floating-point values. Apart from the center cell and its four immediate neighbors from the *temperature* input, the computation also involves the center cell from a second *power* input. The computation continues iteratively, swapping the input and output buffers after each iteration, until the supplied number of iterations have been processed.

Hotspot 3D: Hotspot is also a Structured Grid benchmark and implements the 3D version of Hotspot using a first-order 3D 7-point stencil on a 3D input grid. Similar to Hotspot, this benchmark also uses the center cell from the *power* input, alongside with the center cell and its six immediate neighbors from the *temperature* input in its computation.

Pathfinder: Pathfinder is a Dynamic Programming benchmark that attempts to find a path with smallest accumulated weight in a 2D grid. Computation starts from the top row and continues row by row to the bottom, finding the minimum value among the top-right, top, and top-left neighbors and accumulating this value with the current cell. Similar to NW, this access pattern results in data dependencies but in different directions. This benchmark also only uses integer values.

SRAD: SRAD is a Structured Grid benchmark used for performing speckle reducing on medical images. Similar to Hotspot, its computation involves stencil computation on a 2D input with single-precision floating-point values. However, SRAD has two stencil passes, is much more compute-intensive, and includes an initial reduction on all of the grid cells.

LUD: LU Decomposition (LUD) is a Dense Linear Algebra benchmark that decomposes an arbitrary-sized square matrix to the product of a lower-triangular and an upper-triangular matrix. This benchmark is compute-intensive with multiple instances of single-precision floating-point multiplication, addition and reduction.

4.2.2 Optimization Levels

For each benchmark on the FPGA platform, we create a set of NDRange and Single Work-item kernels. For each set, we define three optimization levels:

None: The lowest optimization level, i.e. *none*, involves using the original NDRange kernels from Rodinia directly, or a direct Single Work-item port based on either the NDRange OpenCL kernel or the OpenMP implementation of the benchmark. The original NDRange kernel from Rodinia will be used as our FPGA baseline to determine speed-up from our optimizations. To avoid unreasonably slow baselines, we employ the crucial *restrict* (3.2.1.1) and *ivdep* (3.2.1.2) attributes to avoid false dependencies and false loop serialization. This level of optimization shows how much performance can be expected when we only rely on the compiler for optimization.

Basic: For the *basic* optimization level, we only apply basic manual and compiler-assisted optimizations (Sections 3.2.1 and 3.2.2) to the unoptimized kernels of each set. This optimization level acts as a representative of the level of performance that can be achieved using a modest amount of effort and by relying only on optimization techniques defined in Intel's documents for programmers with little knowledge of FPGA hardware.

Advanced: The *advanced* optimization level involves significant code rewrite in most cases alongside with taking full advantage of the advanced manual and compiler-assisted optimizations (Sections 3.2.3 and 3.2.4). This level of optimization shows how much performance can be expected with a large amount of programming effort and moderate knowledge of the underlying FPGA characteristics.

For all optimization levels, all parameters (block size, SIMD size, unroll factor, benchmark-specific input settings, etc.) are tuned to maximize performance and the best case is chosen. It is worth noting that we avoid the --fpc and --fp-relaxed compiler switches which can reduce area usage of floating-point computations at the cost of breaking compliance with the IEEE-754 standard due to introduction of inaccuracies and rounding errors in the computation.

4.2.3 Hardware and Software

We evaluate our benchmarks on two FPGA boards. One contains a Stratix V GX A7 device and the other an Arria 10 GX 1150 device. The newer Arria 10 device has roughly twice the logic, 6% more Block RAMs, and nearly six times more DSPs compared to the Stratix V FPGA. Furthermore, the DSPs in the Arria 10 FPGA have native support for single-precision floating-point operations, giving this FPGA an edge over Stratix V for floating-point computation. Table 4-1 shows the device characteristics of these two FPGAs.

Table 4-1 FPGA Device Characteristics

Board	FPGA	ALM	Register (K)	M20K (Blocks|Mb)	DSP	External Memory
Terasic DE5-Net	Stratix V GX A7	234,720	939	2,560|50	256	2x DDR3-1600
Nallatech 385A	Arria 10 GX 1150	427,200	1,709	2,713|53	1,518	2x DDR4-2133

To keep the comparison fair, we will compare each FPGA device with a CPU and GPU of its age. Table 4-2 shows a list of the hardware used in our evaluation, and a summary of their characteristics. The peak compute performance numbers reported in this table are for single-precision floating-point computation.

To compile our OpenMP kernels on CPUs, we use GCC v6.3.0 with -O3 flag and Intel C++ Compiler v2018.2 with "-fp-model precise -O3" flags and choose the best run time between the two compilers. The extra flag for ICC is used to disable optimizations that might change the accuracy of floating-point computations. All hyperthreads are used on every CPU in this case. For GPUs we use NVIDIA CUDA v9.1 with "-arch sm_35 -O3" flags. Intel FPGA SDK for OpenCL v16.1.2 is also used for the FPGAs. Due to a bug in this version of Quartus that resulted in routing errors with some of our kernels on Arria 10, we disable "Parallel Synthesis" in the BSP of our Arria 10 board. We use CentOS 6 on our FPGA machines and CentOS 7 on the CPU/GPU machines.

Table 4-2 Evaluated Hardware and Their Characteristics

Type	Device	Peak Memory Bandwidth (GB/s)	Peak Compute Performance (GFLOP/s)	Production Node (nm)	TDP (Watt)	Release Year
FPGA	Stratix V	25.6	~200	28	40	2011
	Arria 10	34.1	1,450	20	70	2014
CPU	i7-3930K	42.7	300	32	130	2011
	E5-2650 v3	68.3	640	22	105	2014
GPU	Tesla K20X	249.6	3,935	28	235	2012
	GTX 980 Ti[1]	340.6	6,900	28	275	2015

4.2.4 Timing and Power Measurement

In this study, we only time the kernel execution and disregard initialization and all data transfers between host and device. Even though this puts the CPUs at a disadvantage, doing so allows us to fairly compare the computational performance of the devices without hampering their performance by the link between the host and the device that is independent of the devices. Furthermore, we expect data transfer between host and device for most HPC applications to be small relative to the total run time or else, there would be little reason to accelerate them using a PCI-E-attached accelerator like an FPGA or a GPU. To maximize our timing accuracy, we use the high-precision clock_gettime() function supported by most major Linux distributions with the CLOCK_MONOTONIC_RAW setting. Furthermore, we make sure our input sizes are big enough so that kernel run time is at least a few seconds in every case to further increase the dependability of our timing and power measurement results. However, in a few cases, even with the largest setting that fit in the 4 GB external memory of the Stratix V board (smallest external memory size among all evaluated devices), run time of the fastest cases went below 1 second, for a minimum of a couple milliseconds for Pathfinder on the GPUs.

Our Stratix V board does not have an on-board power sensor; hence, to estimate the power usage of the board, we run quartus_pow on the placed-and-routed OpenCL kernel, and add 2.34 Watts to the resulting number to account for the power consumption of the two memory modules. We assume that each memory module uses a maximum of 1.17 Watts based on the datasheet of a similar memory model [28]. The Arria 10 board, however, includes a power sensor and the manufacturer provides an API to read the power sensor in C programs. We use the values reported by sensor to measure power consumption on this platform. Similarly, we read the power sensor available on the GPU boards and the CPUs using existing APIs, namely NVIDIA NVML [29] and the Linux MSR driver [30]. For the GPUs and the Arria 10 FPGA, the on-board power sensor is queried once every 10 milliseconds during kernel execution and the reported wattage values are averaged. In two cases (NW and Pathfinder), since the benchmark run times were not long enough to get accurate power measurement on the GPUs, the main computation loop of these benchmarks was wrapped in an extra loop to artificially

[1] The GTX 980 Ti GPU used our evaluation is a non-reference model that is shipped with higher core and memory clock compared to the reference model

extend the benchmark run time and allow correct measurement of average power usage. In these cases, run time was measured from the first iteration of the extra loop. Power efficiency in these cases is determined by calculating energy to solution as average power consumption multiplied by the kernel run time. For the CPUs, since the associated MSR register reports energy values, we directly measure energy to solution by subtracting the accumulated energy usage at the beginning of kernel execution from the one at the end. It should be noted that unlike the case for the FPGAs and the GPUs where the power measurement includes the board power, the CPU measurements only include the chip itself and do not include the power consumption of the host memory. This slightly favors the CPUs in power efficiency comparison.

Except a few cases among versions with no or basic optimization on Stratix V where run time was over half an hour, all benchmark configurations on every hardware were repeated five times, and timing and power measurements were averaged.

4.3 Results

4.3.1 Stratix V

We discuss optimization details and performance improvement with different optimization levels only on the Stratix V FPGA. For Arria 10, only the result of the fastest version of each benchmark is measured, which will be reported in the next section. The source code for all the benchmarks reported in this section is available at https://github.com/zohourih/rodinia_fpga.

4.3.1.1 NW

The *original* NDRange kernel from Rodinia implements 2D blocking and takes advantage of diagonal parallelism for this benchmark. For this version we use a block size of 128×128. For the *unoptimized* Single Work-item kernel, we use a straightforward implementation with a doubly-nested loop based on the OpenMP version of the benchmark. Due to load/store dependency caused by the left neighbor being calculated in the previous loop iteration, the compiler fails to pipeline the outer loop, and the inner loop is pipelined with an initiation interval of 328, which is equal to the minimum latency of an external memory write followed by a read.

For the *basic* NDRange version, we set the work-group size (3.2.1.4) and add SIMD and unrolling (3.2.1.5). Setting the work-group size allows the compiler to share the same compute unit between different work-groups to minimize pipeline stalls (*work-group pipelining*), while only one work-group is allowed to occupy a compute unit in the *unoptimized* version due to unknown work-group size. However, since the local buffers need to be further replicated to allow parallel access by the extra work-groups, we are forced to reduce block size to 64×64 in this version. Furthermore, due to the very large number of accesses to the local buffer and numerous barriers in the design, parameter tuning is limited to a SIMD size of two and no unrolling. For the Singe Work-item version of this optimization level, we use one extra register to manually cache the left neighbor and then use this register in the innermost loop (iterating over columns) instead of the external memory access. This allows us to remove dependency to the left neighbor. Since the compiler still detects a false dependency on the external memory buffer, we also add *ivdep* (3.2.1.2) to allow correct pipelining of the inner loop with an initiation interval of one. The outer loop iterating over rows still runs sequentially here due to dependency to top and top-left neighbors updated in the previous row. This dependency is unavoidable in this design. Unrolling cannot be used for the innermost loop since it results in new load/store dependencies.

The characteristics of this benchmark make it clear that a Single Work-item design is more suitable since we could already avoid one of the dependencies in the algorithm by employing an additional register in the Single Work-item version with *basic* optimization. Hence, we choose the Single Work-item model to create the kernel with *advanced* optimization level. In [17], we presented an optimized design for NW that used 1D blocking and took advantage of the shifting pattern of the computation alongside with one block row of extra registers to completely avoid external memory accesses other than to read the initial values on the grid and block boundaries. In this implementation, due to the dependency to the left cell which is computed in the previous iteration, we had to fully unroll the computation over a block row or else the loop-carried dependency prevented pipelining with an initiation interval of one. Because of this, the unroll factor and block size had to be the same, preventing us from using large block sizes to minimize redundant memory accesses on the block boundaries. To avoid this problem, we use a different design here. Fig. 4-1 shows our implementation for this benchmark. For this implementation, instead of computing the cells row-by-row which forces us to unroll the computation over the direction of the loop-carried dependency, we take advantage of *diagonal* parallelism in the algorithm. Our new design uses 1D blocking in the *y* dimension with a block height of *bsize* and a parallelism degree, i.e. number of cells computed per iteration, of *par*. Computation starts from top-left and moves downwards, computing one chunk of columns at a time with a chunk width of *par*. The chunk of columns is processed in a diagonal fashion, with the first diagonal starting from an out-of-bound point and ending on the top-left cell in the grid (yellow color). Then, computation moves downwards, calculating one diagonal with *par* cells (shown with the same color in

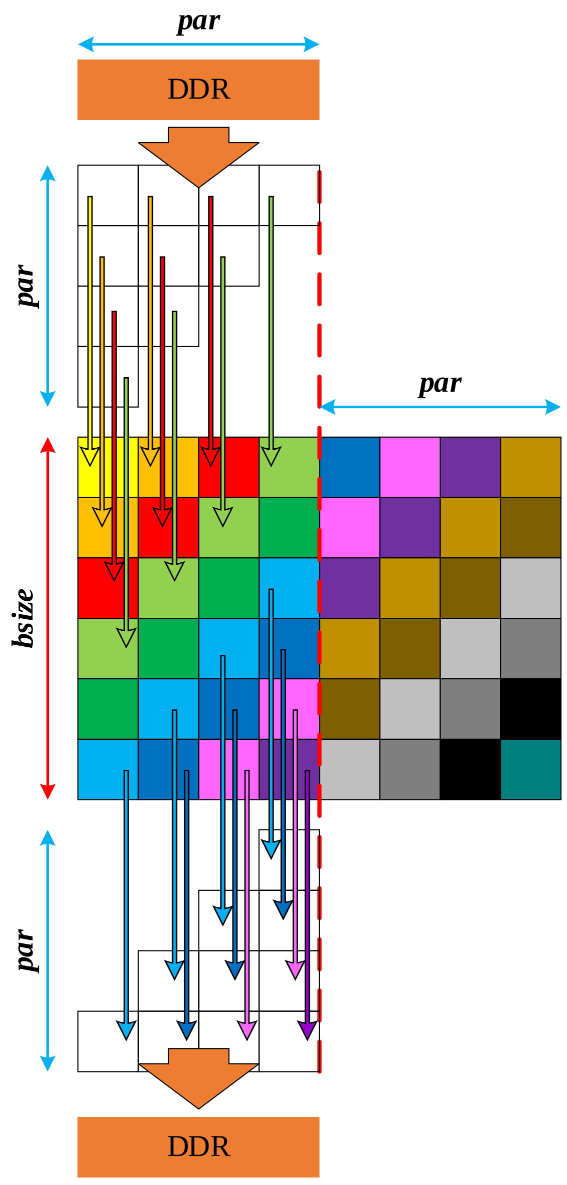

Figure 4-1 NW implementation

Fig. 4-1) per loop iteration until the bottom-cell in the diagonal falls on the bottom-left cell in the block (light blue color). For the next diagonal (dark blue color), the cells that would fall inside the next block instead wrap around and compute cells from the next chunk of columns in the current block. When the first cell in the diagonal falls on the block boundary (violet color), the computation of the first chunk of columns is finished and every cell computed after that will be from the second chunk of columns. When all the columns in a block are computed, computation moves to the next block and repeats in the same fashion. To correctly handle the dependencies, each newly-computed cell is buffered on-chip using shift registers (3.2.4.1) for two iterations to resolve the dependency to the top cell in the next diagonal and top-left cell in the diagonal after that. Furthermore, the cells on the right-most column in the current chunk of columns are buffered in a large shift register with the size of *bsize* so that they can be reused in the next chunk of columns to resolve the dependency to the left neighbor. Finally, the blocks are also overlapped by one row to provide the possibility to re-read the cells on the boundary computed in the previous block and handle the top and top-left dependencies in the first row in the new block. Even though this design allows us to separate block size from degree of parallelism, it breaks external memory access coalescing since accesses are diagonal instead of row-wise, resulting in very poor memory performance. To address this issue, we manually insert a set of shift registers between the memory accesses (both read and write) and computation to delay memory accesses and convert diagonal accesses to consecutive coalesceable ones. These shift registers are shown as white cells in Fig. 4-1. For reading, the shift register for the first column in the chunk has a size of *par* and as we move towards the last column in the chunk, the shift registers get smaller by one cell until the last column where the shift register will turn into a single register. For writes, the set of shift registers instead starts from a single register and ends with a shift register of size *par*. In this case, since writes start *par* iterations after reads, the input width is padded by *par* cells to allow the cells in the rightmost chunk of columns to be written back to external memory correctly. Finally, to improve alignment of external memory accesses, we pass the first column of the input that represents one of the strings and is read-only to the kernel using a separate global buffer so that reads from and writes to the main buffer start from the address 0 instead of 1. For this version, we disable the compiler's cache (3.2.3.2), use loop collapse (3.2.4.3) followed by exit condition optimization (3.2.4.4), and perform seed and target f_{max} sweep (3.2.3.5). Our final implementation has three global buffers, one of which is only accessed once per row (read-only buffer for first column) while the other two (main input and output) are accessed every iteration using large vector accesses. Hence, we also perform manual memory banking (3.2.3.1) and put each of the two frequently-accessed buffers in a different bank to maximize external memory performance. A *bsize* of 4096 and *par* size of 64 are used for this version to maximize its performance.

Table 4-3 shows the performance and area utilization of our kernels on the Stratix V FPGA. For this benchmark, we use an input size of 23040× 23040. As expected, the *original* NDRange kernel performs poorly despite its large block size, due to lack of enough parallelism and large amount of pipeline flushes caused by the plethora of barriers in the kernel. The *unoptimized* Single Work-item kernel performs much worse due to high initiation interval caused by the load/store dependency resulting from the same global buffer being used as both input and output. With *basic* optimizations, the performance of the NDRange kernel slightly improves but still the performance is far from competitive. Furthermore, its operating frequency suffers greatly due to inefficient use of local memory buffers and full utilization of FPGA Block RAMs. On the other hand, the very simple optimization used for the *basic* Single Work-item kernel, even without any explicit parallelism in the kernel (no unrolling), manages to outperform the NDRange kernel and achieve an acceptable level of performance compared to the optimization effort. Finally, the *advanced* kernel manages to achieve nearly 40 times higher performance over the *original* NDRange kernel and brings run time down to a level that can compete with other devices. Performance of this benchmark is now limited by the external memory bandwidth to the point that performance difference between a *par* size of 32 and 64 is less than 5%.

Table 4-3 Performance and Area Utilization of NW on Stratix V

Optimization Level	Kernel Type	Time (s)	Power (W)	Energy (J)	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	DSP	Speed-up
None	NDR	9.937	16.031	159.300	267.52	27%	16%	30%	6%	1.00
	SWI	203.864	12.998	2649.824	304.50	20%	5%	17%	<1%	0.05
Basic	NDR	3.999	16.643	66.555	164.20	38%	68%	100%	8%	2.48
	SWI	2.803	12.137	34.020	191.97	19%	8%	18%	<1%	3.55
Advanced	SWI	0.260	19.308	5.020	218.15	53%	7%	28%	2%	38.22

One point to note here is the relatively low operating frequency of the *basic* and *advanced* Single Work-item kernels compared to the rest of our evaluated benchmarks. Even though we can remove the dependency to the computation of the left neighbor using extra registers or shift registers in these kernels, this optimization requires one write to and one read from registers or shift registers that are updated in the previous cycle, resulting in very tight timing requirements. Because of this, the critical path of design is limited by this read-after-write dependency rather than the loop exit condition, rendering the exit condition optimization performed in the *advanced* version ineffective.

4.3.1.2 Hotspot

The *original* NDRange implementation of Hotspot in Rodinia performs 2D spatial blocking by first moving data from external memory to internal memory and computing all the blocked data before writing them back. This implementation also performs temporal blocking; i.e., it computes each cells for multiple consecutive iterations (time-steps) before writing the final output back to external memory. Without setting the work-group size manually, the compiler assumes a work-group size of 256, which limits the size of the 2D block to 16×16 cells for this version. The *pyramid_height* parameter, which controls the degree of temporal parallelism (i.e. number of fused iterations), is set to one in this case since performance does not scale with higher values due to small block size. We create the *unoptimized* Single Work-item kernel based on the OpenMP implementation of the same benchmark in form of a doubly-nested loop.

For the NDRange kernel with *basic* optimization, we manually set the work-group size (3.2.1.4), add SIMD attribute (3.2.1.5) and move calculation of constants outside of the kernel (3.2.2.2). The compiler's failure in coalescing the accesses to the local buffer prevented area scaling beyond a block size of 64×64 in this case. We also used a SIMD size of 16 and *pyramid_height* of 4 to maximize the performance of this kernel. For the Single Work-item kernel of the same optimization level, we move calculation of constants outside of the kernel (3.2.2.2), remove branches on global memory address calculation (3.2.2.3), and unroll the innermost loop (3.2.1.5). The compiler is successful in achieving an initiation interval of one for the loop nest; however, its performance does not scale beyond an unroll factor of two since the compiler fails to coalesce the accesses to global memory and instantiates multiple ports which compete with each other for the limited external memory bandwidth.

For this particular benchmark, we create both an NDRange and a Single Work-item kernel with *advanced* optimization level. This is due to the fact the *original* NDRange kernel which uses temporal blocking can be further tuned to achieve a reasonably-high performance, while at the same time the stencil-based computation of this benchmark also matches very well with Single Work-item kernels.

Multiple optimizations are performed on the NDRange kernel to minimize its local memory usage. This involves making code changes to correctly coalesce accesses to the on-chip buffers to minimize buffer replication (3.2.4.2) and replacing multiple local buffers with private registers. Specifically, since each index in the buffers used to store *power* and computed *temperature* values are only accessed by one work-item, there is no need to define them as shared local buffers and significant Block RAM saving can be obtained by replacing them with one private register per work-item. Furthermore, branches that are used to make sure out-of-bound neighbors of cells on the border fall back on the border cell itself are optimized so that the compiler can correctly coalesce the accesses to the local buffer for reading the neighbors under the presence of SIMD. This is done by using intermediate registers to read both possible values for each case from the local buffer, and then choosing the correct value from the registers instead of the local buffer. Finally, the two write ports to the remaining local buffer, one from external memory to store values for the first fused iteration, and one to write back the output of the current iteration to be used in the next, are merged into one write port. This is done by conditionally writing to a private register instead, and then writing from the private register to the local buffer. Reducing the number of write ports to the local buffer from two to one halves the buffer replication factor on its own. Combined with the rest of the optimizations, Block RAM replication factor and utilization is significantly reduced, allowing us to use larger block sizes or unroll (3.2.1.5) the iteration loop to improve performance. We also add support for non-square blocks to increase freedom in tuning block size. Compared to the version with *basic* optimization, we increase the block size to 128×64 and add an unroll factor of 2. The kernel becomes limited by logic utilization on Stratix V at this point due to lack of native support for floating-point operations in the DSPs of this FPGA which results in a large amount of logic being used to support such operations. However, even with the larger block size and extra unrolling used in this version, Block RAM utilization is reduced compared to the version with *basic* optimization. Furthermore, we perform seed and target f_{max} sweep (3.2.3.5) for this kernel to maximize its operating frequency and experimentally find that performance now scales up to a *pyramid_height* of 6.

For the Single Work-item kernel with *advanced* optimization level, we adopt 1D spatial blocking (but no temporal blocking). We disable the cache (3.2.3.2) and use loop collapse (3.2.4.3), exit condition optimization (3.2.4.4), shift register-based on-chip storage (3.2.4.1), loop unrolling (3.2.1.5) and seed and target f_{max} sweep (3.2.3.5) for this kernel. These optimizations allow us to achieve a design with a very high operating frequency, fully saturate the memory bandwidth using an unroll factor of 16, and use a very large block size of 4096 to minimize the amount of redundant computation, all with a very modest area utilization. The performance of this kernel cannot be improved any further due to saturation of FPGA external memory bandwidth.

Table 4-4 shows the performance and area utilization of the aforementioned kernels on the Stratix V FPGA. For this benchmark, we use an input size of 8000×8000 and an iteration count of 100. The moderate level of optimization in the *original* Rodinia kernel is still not enough for it to perform well on FPGAs, mainly since the compiler is not able to infer correct run-time parameters like work-group size, and no parallelization is performed by default either. Hence, the relatively straightforward *unoptimized* Single Work-item version manages to outperform this kernel. However, *basic* optimizations significantly improve the performance of the NDRange kernel, while the performance of the Single Work-item version hardly improves by such optimizations. The *advanced* Single Work-item kernel achieves very high speed-up over the *unoptimized* versions and very high operating frequency at a very modest area utilization. However, its performance is limited by the external memory bandwidth. On the other hand, the *advanced* NDRange kernel, since it employs temporal blocking, can break away from the limit imposed by the external memory bandwidth and with our careful optimizations, manages to achieve over twice higher performance compared to the *advanced* Single Work-item kernel.

Table 4-4 Performance and Area Utilization of Hotspot on Stratix V

Optimization Level	Kernel Type	Time (s)	Power (W)	Energy (J)	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	DSP	Speed-up
None	NDR	45.712	13.337	609.661	303.39	22%	5%	17%	12%	1.00
	SWI	21.388	13.353	285.594	303.39	21%	10%	22%	10%	2.14
Basic	NDR	3.276	31.561	103.394	234.96	58%	37%	78%	27%	13.95
	SWI	14.614	13.685	199.993	255.68	24%	12%	23%	4%	3.13
Advanced	NDR	1.875	28.181	52.839	206.01	78%	42%	71%	52%	24.38
	SWI	4.102	16.533	67.818	304.41	47%	5%	19%	26%	11.14

We need to emphasize here that the performance difference between the two *advanced* kernels does not show the advantage of the NDRange programming model, but rather, it shows the advantage of temporal blocking for stencil computation. In fact, due to the shifting pattern of stencil computation and the effectiveness of shift register optimization, which is only applicable to the Single Work-item programming model, it is certain that this model should be preferred over NDRange for stencil computation. The relatively high difference between the operating frequencies of the two *advanced* kernels further affirms which kernel model matches better with the underlying hardware. In Chapter 45, we will revisit this benchmark and show that a highly-optimized Single Work-item design with temporal blocking can achieve even higher performance than we achieved here with the NDRange implementation.

4.3.1.3 Hotspot 3D

Unlike the 2D version of this benchmark, the *original* NDRange implementation of Hotspot 3D in Rodinia neither employs explicit spatial blocking nor temporal blocking. However, it uses multiple private registers to cache consecutive neighbors to be used by the same thread when traversing the z dimension. Similar to Hotspot (2D), we also create the *unoptimized* Single Work-item kernel based on the OpenMP implementation in form of a triply-nested loop.

For the NDRange kernel with *basic* optimization, we set the work-group size (3.2.1.4) and add SIMD (3.2.1.5). The performance of this kernel does not scale beyond a SIMD size of 8 due to unoptimized memory access coalescing. For the Single Work-item kernel with the same optimization level, we remove branches on global memory address calculation (3.2.2.3) and unroll the inner loop (3.2.1.5). Similar to the 2D version, initiation interval of one is achieved for the loop nest, but performance does not scale beyond a SIMD size of four due to memory contention caused by tens of global memory ports competing with each other.

For the *advanced* optimization level, we only create a Single Work-item kernel since it best matches with the shifting pattern of stencil computation. The *original* NDRange implementation of Hotspot 3D in Rodinia is not as optimized the 2D version and hence, creating an *advanced* NDRange implementation based on that will be fruitless. We use the exact same set of optimization as the equivalent kernel for the 2D version in a very similar design. More specifically, we use 2D spatial blocking and collapse a larger loop nest (3.2.4.3) alongside with exit condition optimization (3.2.4.4). This kernel uses a relatively large block size of 512×512 with the cache disabled (3.2.3.2) and an unroll factor of 16 (3.2.1.5). Seed and target f_{max} sweep (3.2.3.5) is also performed to maximize its operating frequency. Again, similar to the 2D case, the performance of this kernel does not scale any further due to saturating the external memory bandwidth.

Table 4-5 shows the performance and area utilization of our Hotspot 3D kernels on the Stratix V FPGA. For this benchmark, we use an input size of 960×960×100 with an iteration count of 100. The *original* OpenCL kernel from Rodinia performs very poorly on FPGAs and probably other hardware due to lack of sufficient optimization. On the particular case of FPGAs, it is outperformed even by the *unoptimized* Single Work-item kernel despite the sheer simplicity of the latter kernel. *Basic* optimizations on the NDRange kernel prove effective but still not enough to overtake even the *unoptimized* Single Work-item kernel. Furthermore, the operating frequency of this kernel drops significantly due to high area utilization. Even though the *unoptimized* Single Work-item kernel achieves relatively good performance, *basic* optimizations on this kernel only yield minor performance improvement due to lack of external memory access coalescing. On the other hand, the kernel with *advanced* optimization achieves a noticeable jump in performance due to careful optimization of external memory accesses and efficient data caching. The operating frequency of the final kernel could easily be improved to over 300 MHz with slightly smaller blocks (512×256 or 256×256), but the improvement in operating frequency proved to be insufficient to make up for the performance gap caused by more redundant computation. The only reason for the sub-300 MHz operating frequency of this kernel is the placement constraints arisen from the large shift register used for spatial blocking that takes up a large portion of the Block RAM resources.

Table 4-5 Performance and Area Utilization of Hotspot 3D on Stratix V

Optimization Level	Kernel Type	Time (s)	Power (W)	Energy (J)	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	DSP	Speed-up
None	NDR	249.164	14.991	3735.2	271.00	28%	11%	26%	13%	1.00
	SWI	32.224	13.656	440.1	303.49	21%	13%	25%	5%	7.73
Basic	NDR	54.834	27.813	1525.1	202.38	80%	31%	78%	78%	4.54
	SWI	24.813	15.689	389.3	255.36	32%	21%	35%	15%	10.04
Advanced	SWI	5.760	19.892	114.6	260.41	48%	37%	60%	52%	43.26

4.3.1.4 Pathfinder

The *original* NDRange version of this benchmark uses 2D blocking, with the block width being defined at compile-time but the block height being configurable at run-time using the input *pyramid_height* parameter. The block height (*pyramid_height*) controls how many block rows are processed (fused) without writing any data to external memory. As many cells as the block width are loaded from off-chip memory into on-chip memory, and computation iterates over *pyramid_height* rows using only on-chip memory, and then data is written back to off-chip memory. When a complete column of blocks is computed, the next block column starts. Due to the triangular/cone-shaped dependency pattern of the algorithm, the blocks are overlapped by 2 × *pyramid_height* columns to ensure correct output. This implementation very much resembles Rodinia's implementation of Hotspot. Since the compiler limits work-group size to 256 unless it is manually set, we use a block size of 256 for the *unoptimized* NDRange kernel and experimentally tune *pyramid_height*, which achieves the best performance if it is equal to 10. For the *unoptimized* Single Work-item kernel, we wrap the computation in a doubly-nested loop like the OpenMP version of the benchmark, but only keep the loop on columns inside the kernel and put the loop on rows in the host code since it is not pipelineable due to data dependency between computation of consecutive rows. The loop left in the kernel achieves an initiation interval of one.

For the *basic* optimization level, we set the work-group size (3.2.1.4) in the NDRange kernel, unroll the loop on the fused rows, and add SIMD and kernel pipeline replication (3.2.1.5). Specifically, we use a block size of 1024, a SIMD size of 16 and a kernel pipeline replication factor of 2. We also experimentally find a *pyramid_height* of 32 to achieve the best performance in this case. Due to inability of the compiler in correctly coalescing the accesses to the local buffer under the presence of SIMD, unrolling the iteration loop proved to be ineffective and was avoided since it increased the number of write ports to the local buffer, significantly increasing Block RAM replication factor. Furthermore, more pipeline (compute unit) replication was not possible in this kernel due to FPGA area limitation. For the Single Work-item kernel with this optimization level, we move external memory accesses outside of branches and replace them with registers (3.2.2.3) so that the accesses can be correctly coalesced with loop unrolling, and then unroll the loop by a factor of 64 which is the highest value that achieves meaningful performance scaling.

Similar to Hotspot, we create both an *advanced* Single Work-item and an *advanced* NDRange version for this benchmark. On one hand, the NDRange implementation of Pathfinder is very similar to Hotspot and with some extra effort, we expect to be able to significantly improve its performance similar to Hotspot. On the other hand, we also expect it to be possible to efficiently implement Pathfinder using a Single Work-item kernel due to the shifting pattern of the computation, and use shift registers as an efficient cache to minimize external memory accesses while also resolving the loop-carried dependency.

For the *advanced* NDRange version, we use a very similar set of optimizations to that of Hotspot. Specifically, we replace the local *result* buffer with a private register since it is only read and written by the same work-item. Furthermore, all accesses to the remaining *prev* buffer from inside of branches are replaced with temporary registers by first moving both possible values from the local buffer to the temporary register, and then choosing the correct value from the register inside the branch (3.2.4.2). Finally, the two writes to the *prev* buffer are merged into one using extra private registers; this optimization halves the Block RAM replication factor for implementing the buffer on its own. In the end, the number of reads from and writes to the buffer are reduced to the minimum value of 3 and 1, regardless of SIMD size. The large reduction in Block RAM usage in this case allows us to, unlike the *basic* NDRange version, successfully unroll the iteration loop (3.2.1.5) and further increase the block size. Specifically, we increase the block size to 8192, which allows the performance to improve up to a *pyramid_height* of 92, and use a SIMD size of 16 and an unroll factor of 2. Kernel pipeline replication is not used in this case since performance benefits more from unrolling and it is preferred to spend the FPGA resources on this form of parallelism. Seed and target f_{max} sweep (3.2.3.5) is also performed to maximize the operating frequency of this kernel.

Even though Pathfinder has spatial dependencies similar to NW, the dependency is to cells from the previous row and hence, no dependency to the output of the previous iteration exists. Hence, we do not need to follow the same optimization strategy as NW and instead, for the *advanced* Single Work-item version of Pathfinder we use the same design strategy as the NDRange kernel but with an implementation similar to Hotspot and shift registers used as on-chip buffer. Doing so significantly reduces Block RAM usage for this kernel and allows us to increase block size to 32768. The block size can still be increased in this case, but performance improvement from lower redundancy will become minimal and instead, operating frequency will decrease due to complications arisen from placing large shift registers on the FPGA. Moreover, this version uses loop collapse (3.2.4.3) and exit condition optimization (3.2.4.4) alongside with an unroll factor of 32 (3.2.1.5). The cache created by the compiler is also disabled (3.2.3.2) since we perform caching manually using shift registers.

Table 4-6 shows the performance of our kernels in this benchmark with different optimization levels. We use an input size of 1,000,000×1,000 for this benchmark. Decreasing the number of columns and instead increasing the number of rows would have resulted in longer run time and more dependable timing results in this case. However, since most optimizations (SIMD, unrolling, etc.) are performed on the loop on columns, doing so would have resulted in low pipeline efficiency on our FPGA design and less improvement over baseline, and also low occupancy on the CPU and GPU versions, resulting in an unfair comparison. Hence, we chose to use a bigger number of columns instead at the cost of short execution time.

Table 4-6 Performance and Area Utilization of Pathfinder on Stratix V

Optimization Level	Kernel Type	Time (s)	Power (W)	Energy (J)	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	DSP	Speed-up
None	NDR	3.918	12.901	50.546	303.39	20%	4%	16%	2%	1.00
	SWI	3.605	12.764	46.014	304.50	20%	5%	16%	<1%	1.09
Basic	NDR	0.310	30.916	9.584	221.68	54%	35%	80%	3%	12.64
	SWI	0.749	14.469	10.837	226.03	40%	20%	32%	<1%	5.23
Advanced	NDR	0.188	20.716	3.895	239.69	44%	32%	55%	2%	20.84
	SWI	0.234	15.314	3.583	278.39	34%	7%	21%	<1%	16.74

The *original* kernel from Rodinia fails to achieve good performance due to small block size and lack of explicit parallelism, which also leads to poor utilization of external memory bandwidth. Similarly, the *unoptimized* Single Work-item kernel achieves low performance due to poor utilization of external memory bandwidth and lack of efficient data caching. With *basic* optimizations, the performance of both kernel variations improves, but the *basic* NDRange kernel achieves higher performance due to better caching of data. With *advanced* optimizations, the NDRange kernel achieves over 20 times higher performance than baseline, thanks to our careful optimizations on the local memory buffers that make it possible to use much bigger block sizes. The *advanced* Single work-item kernel achieves 25% lower performance compared to its NDRange counterpart but with a much higher operating frequency due to better critical path optimization, and much lower Block RAM utilization despite much bigger block size which also results in better power efficiency. The input buffer in Pathfinder is the only global buffer that is accessed every iteration and the output buffer is only written in the last fused row. Since the FPGA board has two banks, even with interleaving, it is not possible to efficiently saturate the external memory bandwidth when only one buffer exists in the kernel that is accessed every iteration. Apart from that, memory access efficiency is low in this benchmark due to unaligned memory accesses caused by block overlapping. The NDRange kernel in this case seems to achieve better performance due to *work-group pipelining* which can potentially allow better utilization of the memory bandwidth, and this is likely the reason for its higher performance. We expect the performance of the Single Work-item to be improved further by optimizing memory access alignment using padding, as we will discuss in Section

5.3.3, but such optimization is outside the scope of this chapter. It is likely that the operating frequency of the Single Work-item kernel could also be improved further if the number of fused rows is converted from a run-time variable to a compile-time constant, since it would simplify the critical path. In the end, we choose the *advanced* NDRange kernel as the best implementation for this benchmark.

4.3.1.5 SRAD

The *original* implementation of SRAD in Rodinia consists of six kernels. Two of these kernels (*compress* and *extract*) perform pre- and post-processing on the input image and only take a very small portion of the run time. Since neither of these kernels are timed on any of the platforms, we move them to the host code in the OpenCL implementation to avoid spending FPGA area on these kernels. In this implementation, the computation starts by reading the input image and calculating the value of each cell multiplied by itself and saving it into another buffer in the *prepare* kernel. In the *reduce* kernel, an additive reduction is performed on the input image and the new buffer from the *prepare* kernel, and two summation outputs are generated. In the *srad* kernel, the first stencil pass, which is a 2D 5-point star-shaped stencil, is performed on the input image and the summation results calculated in the previous kernel are used in the computation. Instead of calculating the addresses of the neighbors inside of the kernel, this implementation calculates the addresses for all cells in the host code and creates four additional buffers with the same size as the input image to store them on the FPGA external memory. Then, in the kernel, the address to access each neighbor in each iteration is read from these buffers, and the neighbor value is then read from the image buffer, resulting in unnecessary indirect memory accesses and extremely poor memory access efficiency. Then, the output of this kernel is stored in five different external memory buffers, one having the same coordinates as the input image and the others each being one column or one row shifted compared to this buffer so that address calculation for accessing neighbors can also be avoided in the next kernel. In the final *srad2* kernel, another stencil pass is performed, this time a 2D 3-point stencil that only uses the *center, east* and *south* cells, and the output is stored in a final buffer. The strange design decisions in this implementation make it an unlikely candidate to achieve high performance on any hardware:

- Separating the *prepare* and *reduce* kernels results in unnecessary external memory loads and stores
- The implementation of the *reduce* kernel is extremely inefficient
- Indirect memory accesses in the *srad* and *srad2* kernels to avoid basic address calculation lead to poor memory bandwidth utilization
- Lack of even basic caching in the stencil passes results in a significant amount of redundant external memory accesses.

For this version, we use a work-group size of 256 to align with the limitation imposed by the compiler when work-group size is not manually set. For the *unoptimized* Single Work-item version, we use the same kernel structure as the *original* NDRange implementation, but implement the first two kernels as basic single-loops and each of the two stencil passes as a doubly-nested loop. The *ivdep* pragma (3.2.1.2) is also used to avoid a false dependency in the *srad2* kernel.

For the NDRange kernel with *basic* optimization, the work-group size is manually set for all kernels (3.2.1.4), and SIMD is employed (3.2.1.5) with the exception of the *reduce* kernel in which SIMD cannot be used due to thread-id-dependent branching. Instead, in the *reduce* kernel, full unrolling is used for a simple loop, and partial unrolling is used for other loops (3.2.1.5). After extensive parameter tuning, best performance is achieved by using a work-group size of 256, a SIMD factor of 8 for the *prepare* kernel and 2 for the *srad* and *srad2* kernels, and an unroll factor of 2 for the *reduce* kernel. For the Single Work-item kernel with the same optimization level, the reduction in the *reduce* kernel is optimized using shift registers (3.2.2.1), and the innermost loops of all the kernels are partially unrolled. The latter forces us to add yet another *ivdep* pragma (3.2.1.2) to avoid a new false dependency in the *srad2* kernel. After parameter tuning, best performance for this version is achieved by using an unrolling factor of 8 for both the *prepare* and *reduce* kernels, and 2 for the remaining kernels.

For *advanced* optimization, we choose the Single Work-item kernel type since it can be used to efficiently implement both the reduction operation and the two stencil passes in this benchmark. Considering the suboptimal implementation of this benchmark in Rodinia, a complete code rewrite was required to achieve reasonable performance on our FPGA platform. Specifically, to minimize external memory accesses and maximize local data sharing, we combine all the original kernels into one kernel. The loops of the original *prepare* and *reduce* kernels are combined into one loop, eliminating two global buffers and all accesses associated with them. Also all indirect memory accesses to read neighboring cells are converted to direct addressing, and the four global buffers holding the address of the neighbors from the original implementation are eliminated. Then, the two stencil passes (*srad* and *srad2)* are merged, eliminating five more global buffers that were originally used to pass data from the first pass to the second pass. Over 10x reduction in global memory traffic and usage is achieved like this. The second stencil pass of the computation (*srad2*) can only start when the center, south and east cells are already computed by the first pass (*srad*). This means that if computation starts from the top-left of the grid as is the default case, it is not possible for the second pass to start right after the first cell is computed by the first pass, and some cells need to be buffered until all neighbors are ready to compute the first cell in the second pass. However, if the starting point is changed to bottom-right, since the necessary neighbors for computation of the first cell in the second pass fall outside of the grid boundary and the boundary conditions will be used instead, the second pass can start right after the first pass without any delay. We take advantage of this technique to minimize resource overhead. Similar to Hotspot 2D, we then employ 1D overlapped blocking with shift registers used as on-chip buffers (3.2.4.1). However, since two stencil passes are involved in this benchmark with a dependency between them, it is required that we increase the width of the halo region from one to two cells to correctly handle the dependency. Moreover, loop collapse (3.2.4.3) and exit condition optimization (3.2.4.4) are employed, all loops are partially unrolled (3.2.1.5), the auto-generated cache is disabled (3.2.3.2), and seed and target f_{max} sweep (3.2.3.5) is performed as is the usual case for all of our *advanced* Single Work-item kernels. We also employ manual memory external banking (3.2.3.1) for this benchmark since the final version of our design reduces the total number of global memory buffers to two, with one being read and the other being written every clock cycle. Surprisingly, optimizing this benchmark did not end here. After place and routing the final design, we encountered lower-than-expected operating frequency on both Stratix V and Arria 10. After extensive troubleshooting, it turned out that the version of Intel FPGA SDK for OpenCL we are using has some problem with balancing pipeline stages when a floating-point variable is multiplied by a constant floating-point value. We worked around this problem by converting such multiplications to division without any loss of accuracy. For example, $0.25 \times x$ is replaced by $^{x}/_{4.0}$. The best-performing configuration for this version uses a block size of 4096 and unroll factors of 4 and 16 for the stencil computation and the reduction operations, respectively. No more unrolling is possible here due to DSP limitations on the Stratix V FPGA.

Table 4-7 shows the performance and area utilization of the aforementioned kernel versions. We use an input size of 8000×8000 and 100 iterations for this benchmark. Due to poor design, the *original* kernel from Rodinia performs very poorly on our FPGA. The *unoptimized* Single Work-item kernel, despite being based on the NDRange kernel, achieves higher performance, mainly due to more efficient implementation of the *reduce* kernel. *Basic* optimizations hardly improve the performance of the NDRange kernel due to poor baseline implementation, but the Single Work-item kernel can achieve a reasonable speed-up since the implementation of the reduction operation in this version is close to optimal. Finally, the kernel with *advanced* optimization not only achieves a notable speed-up over baseline, but also very high operating frequency, which shows that this design matches the underlying FPGA architecture very well.

Table 4-7 Performance and Area Utilization of SRAD on Stratix V

Optimization Level	Kernel Type	Time (s)	Power (W)	Energy (J)	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	DSP	Speed-up
None	NDR	346.796	18.913	6558.953	248.20	47%	22%	42%	26%	1.00
	SWI	276.807	16.558	4583.370	270.56	36%	15%	33%	24%	1.25
Basic	NDR	265.784	24.587	6534.831	248.57	64%	34%	78%	52%	1.30
	SWI	42.346	20.358	862.080	251.69	48%	37%	57%	46%	8.19
Advanced	SWI	9.060	18.904	171.270	304.41	57%	8%	27%	87%	38.28

4.3.1.6 LUD

The *original* implementation of this benchmark uses 2D square blocking with three kernels. First, the *diameter* kernel computes the top left block in the matrix, then, the *perimeter* kernel computes the remaining blocks in the first block column and block row and then, the remaining square of blocks are processed by the *internal* kernel. In the next step, the starting position of the matrix is moved one block forward in both the *x* and the *y* dimension and the same chain of kernel operations is performed on the new submatrix. In the final round of computation, only the bottom right block will be left and in this case, only the *diameter* kernel processes this block to finish the computation. Every kernel takes full advantage of local memory by avoiding all redundant memory accesses in the block that is being processed. By default, the compiler auto-unrolls some of the loops in the kernel, which results in lower performance since the external memory accesses in the auto-unrolled loops are not consecutive and hence, numerous ports to external memory are created, resulting in a significant amount of contention on the memory buss. We prevent this by forcing an unroll factor of one for these loops. A block size of 16×16 is used for this version since larger values are cannot be used unless work-group size is set manually. For the *unoptimized* Single Work-item implementation, we base our kernel design on the OpenMP version of the benchmark which uses a similar computation pattern to that of the NDRange version. We also use *ivdep* (3.2.1.2) to avoid false dependencies detected by the compiler in the middle loops of the *diagonal* kernel and the outermost loop of the *internal* kernel. Dependencies detected in some other loops were real and hence, *ivdep* was not used for those. Initially, we encountered what seemed to be a functional bug in the compiler using this version. We worked around this issue by making minor modifications in the *internal* kernel to swap the order of the two innermost compute loops, which then allowed us to merge the write-back loop into the compute loop and replace the arrays of *sum* variables with a single variable. A block size of 16×16 was also used in this version since larger block sizes resulted in lower performance.

For the NDRange kernel with *basic* optimization, we manually set the work-group size for each kernel (3.2.1.4) and add SIMD and kernel pipeline replication (3.2.1.5) to all kernels. Furthermore, the loop in the *internal* kernel is fully unrolled, while the loops in the other kernels are partially unrolled with compile-time configurable unroll factors (3.2.1.5). In practice, SIMD could not be used in the *diameter* and *perimeter* kernels due to thread-id-dependent branching. Furthermore, using kernel pipeline replication for the *diameter* kernel was avoided since this kernel is only executed by one work-group. Since run time is dominated first by dense matrix multiplication in the *internal* kernel, and then by the computation in the *perimeter* kernel, while the *diameter* kernel accounts for less than 0.1% of the total run time, we configure the parameters in a way that allocates resources to each kernel based on its portion of the total run time. Based on this, block size is increased to 64×64, the *diameter* kernel is left as it is, while an unroll and kernel pipeline replication factor of 2 is used for the *perimeter* kernel. For the *internal* kernel, a pipeline replication factor of 3 is used on top of full unrolling of the loop in this kernel. These parameters nearly maximize the DSP and Block RAM utilization of the device and higher values cannot be used any more. For the Single Work-item kernel with *basic* optimization, we increase block size to 64×64 and use the shift register-based optimization for floating-point reduction (3.2.2.1) which then allows us to also partially unroll the reduction loops in the *diameter* and the *perimeter* kernels. For the *internal* kernel, the innermost loop is fully, and the middle loop is partially unrolled (3.2.1.5). Furthermore, the loop over blocks for the *perimeter* and *internal* kernels, which are not pipelineable, are moved to the host to save area. Partial unrolling of the innermost loops in the *diagonal* and *perimeter* kernels resulted in slow-down and hence, was avoided. Moreover, apart from fully unrolling the innermost loop of the internal kernel, the middle loop is unrolled by a factor of 2. Higher values could not be used since an unroll factor of 3 resulted in load/store dependencies and a very high initiation interval due to the loop trip count (block width) not being divisible by 3, and a factor of 4 resulted in DSP overutilization on the Stratix V FPGA.

To choose the best kernel type for creating the version with *advanced* optimization, we have to consider two important characteristics of this benchmark: first, the outer loops in the *diagonal* and *perimeter* kernels are not pipelineable due to variable exit condition of the middle or innermost loops. As mentioned in Section 3.1.4, NDRange kernels are preferred in such case since the run-time scheduler can allow a lower average initiation interval compared to sequential execution of the non-pipelineable loops in the equivalent Single Work-item implementation. Second, blocking in this benchmark requires that data transfers between off-chip memory and on-chip memory be separated from the computation, with a complete block being loaded into on-chip memory, computed, and then written back. Hence, no overlapping of computation and memory accesses will exist in a Single Work-item implementation, resulting in poor performance. On the other hand, in an NDRange implementation, each compute unit is shared between multiple work-groups and one work-group can occupy the compute part of the pipeline while another work-group is occupying the memory access part, allowing efficient *work-group pipelining* and overlapping of computation and memory accesses. Hence, we choose NDRange as the best kernel type for this optimization level. Compared to the NDRange kernel with *basic* optimization level, multiple optimizations are employed to minimize local memory ports and replication factor (3.2.4.2) for the *advanced* version. A temporary variable is used as reduction variable instead of the local buffers themselves, to reduce number of ports to the local buffers in the *diagonal* and *perimeter* kernels. The local buffer in the *diameter* kernel and the *dia* buffer in the *perimeter* kernel are split into two buffers, one of which is loaded row-wise and the other, column-wise. Replacing the column-wise accesses to the original buffers with row-wise accesses to the new buffers that are filled in column-wise order allows correct access coalescing under partial unrolling of the loops in these two kernels and significant reduction in Block RAM usage. The *peri_row* buffer in the *perimeter* kernel is also transposed for the same reason. Loads from and writes to external memory are modified in the *perimeter* kernel to remove thread-id-dependent branching. Furthermore, writing back the content of the *peri_row* buffer to external memory is merged into the compute loop to remove one extra read port from this buffer. The same can be done with the write-back of the content of the *peri_col* buffer; however, that would result in an external memory access pattern that is not consecutive based on work-item ID and hence, lowers performance. The write-back for this buffer is kept outside of the compute loop after a barrier so that data can then be written back in a way that accesses are consecutive based on work-item ID. Also common subexpression elimination is performed in all kernels and constant common subexpressions are moved to the host code to minimize logic and DSPs used for integer arithmetic. All these optimizations allow us to increase block size to 96×96, with an unroll factor of 4 for the *diameter* kernel, an unroll factor of 8 and compute unit replication factor of 2 for the *perimeter* kernel, and a SIMD size of 2 for the *internal* kernel. However, fitting this configuration on the FPGA required that we manually perform port sharing on the *diameter* kernel to reduce its Block RAM usage so that more Block RAMs are available to the rest of the kernels. Even though doing so slightly reduced the performance of the *diameter* kernel, the extra performance gained by faster execution of the rest of the kernels made up for the difference. Finally, seed and target f_{max} sweep (3.2.3.5) is performed to maximize the performance of this kernel. Unlike every benchmark discussed so far where we use this compiler-assisted optimization to maximize performance by maximizing the operating frequency of the design, in the particular case of this benchmark, performance increases with operating frequency up to a certain point and after that, it starts decreasing. The reason for this is that with the implemented optimizations, the *internal* kernel nearly saturates the FPGA external memory bandwidth and increasing the operating frequency beyond the saturation point will result in more contention on the memory bus and instead, decreases performance. In the end, we timed the final kernel running at different operating frequencies and chose the fastest one.

Table 4-8 shows the performance of LUD with different optimization levels. We use an input matrix size of 11520×11520, which is divisible by all the block sizes used for every version. The *original* NDRange kernel achieves low performance here due to small block size and lack of explicit parallelism. The *unoptimized* Single Work-item kernel achieves even worse performance due to the non-pipelineable loops and lack of compute and memory access overlapping. With *basic* optimization, the NDRange kernel achieves over two orders of magnitude speed-up, mostly due to full unrolling of the loop in the *internal* kernel that is the most compute-intensive kernel in the benchmark. The performance of the Single Work-item kernel, however, improves only slightly with *basic* optimization since the fundamental problems associated with using this kernel type for this benchmark are still not addressed. Finally, our *advanced* optimizations on the NDRange kernel allow us to maximize the performance of this benchmark on our device with near full utilization of DSP and Block RAM resources. The performance of this benchmark on the Stratix V FPGA is limited by DSP and Block RAM resources.

Table 4-8 Performance and Area Utilization of LUD on Stratix V

Optimization Level	Kernel Type	Time (s)	Power (W)	Energy (J)	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	DSP	Speed-up
None	NDR	1944.820	15.580	30300.296	262.60	30%	14%	28%	13%	1.00
	SWI	2451.187	15.885	38937.105	267.73	34%	12%	28%	16%	0.79
Basic	NDR	14.800	29.712	439.738	234.57	69%	42%	95%	99%	131.41
	SWI	1273.347	25.667	32682.997	254.32	65%	24%	61%	65%	1.53
Advanced	NDR	13.159	19.832	260.969	224.40	81%	50%	98%	96%	147.79

It is worth mentioning that one compiler limitation, which applies to all NDRange kernels and still exists even in the latest version of the compiler (v18.0), severely limited our parameter tuning freedom for this benchmark. As mentioned in Section 2.3.2, the compiler automatically pipelines multiple work-groups in the same compute unit for NDRange kernels to maximize pipeline efficiency and minimize the negative performance impact of barriers. However, the number of work-groups that can run simultaneously in a compute unit is automatically decided by the compiler at compile-time, with no way of being influenced by the programmer. In many cases the compiler tries to support tens or even hundreds of work-groups per compute unit, and then replicates all the local buffers in the compute unit by the same number, resulting in significant waste of Block RAMs, while the same performance could probably be achieved using a much smaller number of simultaneous work-groups. In the particular case of the LUD benchmark, the *diameter* kernel does not even require work-group pipelining since it is only executed by one work-group, while the compiler still replicates the local buffers inside of this kernel to support two simultaneous work-groups. Moreover, we had to abandon multiple optimization ideas for the *perimeter* kernel since they resulted in the compiler increasing the degree of work-group pipelining and making the kernel unfittable. If it was possible to directly influence the degree of work-group pipelining, the trade-off between performance and Block RAM utilization could be optimized much more efficiently.

4.3.2 Arria 10

4.3.2.1 Changes Compared to Implementations on Stratix V

For NW and Hotspot 3D we use the exact same settings as Stratix V since the small improvement in external memory bandwidth on Arria 10 compared to Stratix V prevents these benchmarks from benefiting from the more resources available in this FPGA. The same applies to Pathfinder; however, we ended up using a smaller block size of 4096 for this benchmark on Arria 10 since bigger block sizes lowered operating frequency by an amount that cancelled out the effect of the bigger block size.

For Hotspot, we decrease the block size to 64×64 but increase the unroll factor to 3 compared to Stratix V. This allows a small performance improvement over using the same configuration as Stratix V despite significant reduction of operating frequency.

For SRAD, we increase the unroll factor for the stencil passes from 4 to 16 on Arria 10, which is made possible by the significant improvement in the number of DSPs and native support for floating-point operations on this device. We also take advantage of single-cycle floating-point accumulation, which is an Arria 10-specific optimization, to eliminate the need for shift register optimization for floating-point reduction (3.2.2.1). However, to our surprise, our implementation on Arria 10 achieved lower operating frequency compared to Stratix V. Further troubleshooting showed that similar to the issue discussed in Section 4.3.1.5, the source of the problem on Arria 10 also seemed to be from the way pipeline stage balancing was performed by the compiler. Specifically, the compiler seemed to implement floating-point division operations inefficiently since removing all such divisions from the kernel increased the operating frequency of the kernel to nearly 350 MHz. Unfortunately, by the time of writing this thesis, we could not find a work-around for this problem. However, even with the lowered operating frequency, the benchmark is memory-bound on Arria 10 and performance improvement is expected to be minimal with higher operating frequency.

For LUD, even though the higher number of DSPs available in Arria 10 compared to Stratix V eliminated one of the two main area bottlenecks, the other area bottleneck, i.e. Block RAM count, is not improved much in the new FPGA and still limits the performance on this device. Furthermore, as mentioned in Section 3.2.3.4, we could not use *flat* compilation for LUD on Arria 10 since an NDRange kernel was being used and regardless of the number of different seeds we tried, we could not prevent timing failure for the peripheral clocks (DDR, PCI-E, etc.). This forced us to use the default PR flow, which resulted in fitting or routing failure for any configuration that utilized more than 95% of the Block RAM resources on the device, further reducing our ability to efficiently use this important resource. To be able to increase the block size to 128×128, we have to decrease compute unit replication for the *perimeter* kernel to one and instead make up for it by increasing the unroll factor to 32 (unrolling has much less Block RAM overhead than pipeline replication). We also increase the unroll factor for the *diameter* kernel to 8 and use a SIMD size of 4 for the *internal* kernel. With *flat* flow, the compute unit replication factor of two could be kept for the *perimeter* kernel with an unroll factor of 16, resulting in 5% higher performance compared to our final configuration with the PR flow, but this configuration had to be discarded due to timing constraints not being met. This shows that using Partial Reconfiguration on Arria 10, apart from the direct disadvantage of lowering operating frequency, can also indirectly result in even more performance disadvantage when area utilization is high by preventing efficient utilization of FPGA resources.

Table 4-9 shows the best-performing results for each benchmark on both Arria 10 and Stratix V, and the resource that is bottlenecking the performance in each case. "BW" in this table refers the external memory bandwidth, while M20K refers to FPGA on-chip memory blocks. The clear trend here is that performance on Arria 10 is limited by its low external memory bandwidth in nearly every benchmark. Because of this, performance improvement in cases where performance was already bottlenecked by this resource on Stratix V shows minimal improvement. Furthermore, power efficiency is lowered in these benchmarks compared to Stratix V due to higher static power consumption and inefficient use of the FPGA area on Arria 10. The only benchmarks that achieve meaningful performance improvement on Arria 10 are SRAD and LUD, which were bound by FPGA resources on Stratix V. SRAD also becomes memory bound with the higher unroll factor used on Arria 10 despite modest area usage. For LUD, minimal improvement in Block RAM count prevents us from trading off more external memory bandwidth by on-chip memory compared to Stratix V and in the end, we cannot utilize even half of the DSPs of this device. Even if more Block RAMs were available

Table 4-9 Performance and Power Efficiency of All Benchmarks on Stratix V and Arria 10

Benchmark	FPGA	Time (s)	Power (W)	Energy (J)	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	DSP	Bottleneck
NW	Stratix V	0.260	19.308	5.020	218.15	53%	7%	28%	2%	BW
	Arria 10	0.176	32.699	5.755	201.06	28%	8%	25%	<1%	BW
Hotspot	Stratix V	1.875	28.181	52.839	206.01	78%	42%	71%	52%	Logic, BW
	Arria 10	1.616	45.732	73.903	179.89	31%	44%	81%	29%	M20K, BW
Hotspot 3D	Stratix V	5.760	19.892	114.578	260.41	48%	37%	60%	52%	BW
	Arria 10	5.254	35.147	184.662	239.39	14%	36%	53%	10%	BW
Pathfinder	Stratix V	0.188	20.716	3.895	239.69	44%	32%	55%	2%	BW
	Arria 10	0.141	34.397	4.850	258.97	27%	19%	35%	<1%	BW
SRAD	Stratix V	9.060	18.904	171.270	304.41	57%	8%	27%	87%	DSP
	Arria 10	4.721	40.889	193.037	277.33	44%	14%	27%	62%	BW
LUD	Stratix V	13.159	19.832	260.969	224.40	81%	50%	98%	96%	DSP, M20K
	Arria 10	5.279	46.671	246.376	240.74	33%	45%	93%	41%	M20K, BW

on Arria 10, performance will not improve much further since the most time consuming part of this benchmark (the *internal* kernel) is already nearly memory-bound.

4.3.3 CPUs

Table 4-10 shows the performance and power efficiency of all the benchmarks on both of our evaluated CPUs using both GCC v6.3.0 and ICC 2018.2. The best performance for each benchmark on each CPU has been colored in green. None of the benchmarks are modified other than to add timing and power measurement functions. Most of the CPU benchmarks in Rodinia already take advantage of optimization techniques like loop tiling. We expect that the existing code optimizations coupled with using two state-of-the-art compilers should allow us to achieve a reasonable level of performance on the CPUs and allow fair comparison with the rest of the hardware.

Table 4-10 Performance and Power Efficiency Results of All Benchmarks on CPUs

Benchmark	CPU	Compiler	Time (s)	Power (W)	Energy (J)
NW	i7-3930k	GCC	719.651	116.691	83.977
		ICC	744.204	115.767	86.148
	E5-2650 v3	GCC	371.479	81.910	30.428
		ICC	395.222	83.746	33.090
Hotspot	i7-3930k	GCC	4056.987	126.988	515.180
		ICC	3331.503	127.817	425.818
	E5-2650 v3	GCC	3149.191	87.131	274.391
		ICC	2659.946	87.814	233.579
Hotspot 3D	i7-3930k	GCC	7752.818	152.252	1180.363
		ICC	8806.121	151.272	1331.353
	E5-2650 v3	GCC	6881.140	100.302	690.168
		ICC	6794.439	99.955	679.140
Pathfinder	i7-3930k	GCC	306.995	133.308	40.925
		ICC	293.070	140.161	41.074
	E5-2650 v3	GCC	297.511	83.687	24.896
		ICC	309.270	86.892	26.874
SRAD	i7-3930k	GCC	41206.358	113.265	4667.282
		ICC	15008.157	153.048	2296.995
	E5-2650 v3	GCC	46510.895	58.414	2716.417
		ICC	11825.654	100.860	1192.733
LUD	i7-3930k	GCC	22048.880	142.271	3136.958
		ICC	19396.328	133.585	2591.064
	E5-2650 v3	GCC	17896.558	94.115	1684.335
		ICC	14326.216	88.891	1273.477

Based on our results, in most cases ICC outperforms GCC by a large margin. Also other than Pathfinder which does not seem to scale well with multi-threading, the newer CPU is faster than the old one in every benchmark. However, this CPU is at best twice faster than the old one, despite being of a much newer generation and having four (6 vs. 10) more cores.

4.3.4 GPUs

Table 4-11 shows performance and power efficiency of all of our benchmarks on both of our evaluated GPUs. Default block size was increased to 32×32 for Hotspot, and *pyramid_height* was tuned for Hotspot and Pathfinder on each GPU. Changing the default parameters were also attempted in other benchmarks, but were discarded since they did not improve performance. No further modifications were made in any of the benchmarks other than adding timing and power measurement functions. All of the CUDA versions of the benchmarks in Rodinia have already gone through a good degree of optimization and we believe that coupled with NVIDIA's most recent CUDA toolkit and compiler, the performance results we have obtained here are a good representative of the capabilities of the GPUs.

Table 4-11 Performance and Power Efficiency Results of All Benchmarks on GPUs

Benchmark	GPU	Time (s)	Power (W)	Energy (J)
NW	K20X	270.587	102.184	27.649
	980 Ti	133.116	132.465	17.633
Hotspot	K20X	823.476	132.297	108.943
	980 Ti	1161.366	152.340	176.921
Hotspot 3D	K20X	2893.110	118.531	342.922
	980 Ti	1393.586	174.916	243.748
Pathfinder	K20X	50.200	138.755	6.965
	980 Ti	21.503	219.690	4.724
SRAD	K20X	3758.656	145.440	546.660
	980 Ti	2374.360	222.598	528.516
LUD	K20X	4884.329	134.892	658.856
	980 Ti	1292.572	237.113	306.458

Based on our results, the newer 980 Ti GPU outperforms its older counterpart by two times or more in nearly every benchmark; the only exception is Hotspot were 980 Ti actually achieves lower performance. Since Hotspot relies heavily on caching, this performance regression could be caused by differences in the memory and cache hierarchy of these two GPUs which are from different generations.

4.3.5 Comparison

Fig. 4-2 shows the performance and power efficiency of all of our benchmarks on all the evaluated hardware. Our results show that the FPGAs can outperform their same-generation CPUs in every case while achieving up to 16.7 times higher power efficiency. Compared to the GPUs, however, the results are different. Other than the NW benchmark in which the Stratix V FPGA can narrowly overtake its same-generation GPU, in no other case can any of the FPGAs outperform their same-generation GPUs. Furthermore, the newer Arria 10 FPGA is outperformed by even the older K20X GPU in every benchmark other than NW; though the difference tends to be smaller in the more compute-intensive SRAD and LUD benchmarks.

Still, the FPGAs have a clear power efficiency advantage over the GPUs to the point that the aged Stratix V FPGA can achieve better power efficiency than the much newer GTX 980 Ti GPU in every benchmark. The largest power efficiency advantage is observed in the NW benchmark were the Stratix V FPGA is 5.6 times more power efficient than it same-generation GPU. Unfortunately, power efficiency improvements on Arria 10 compared to Stratix V are minimal to none since in none of the benchmarks we can efficiently use the resources of this newer FPGA due the external memory bandwidth bottleneck. LUD is the only benchmark in which Arria 10 achieves better power efficiency than Stratix V.

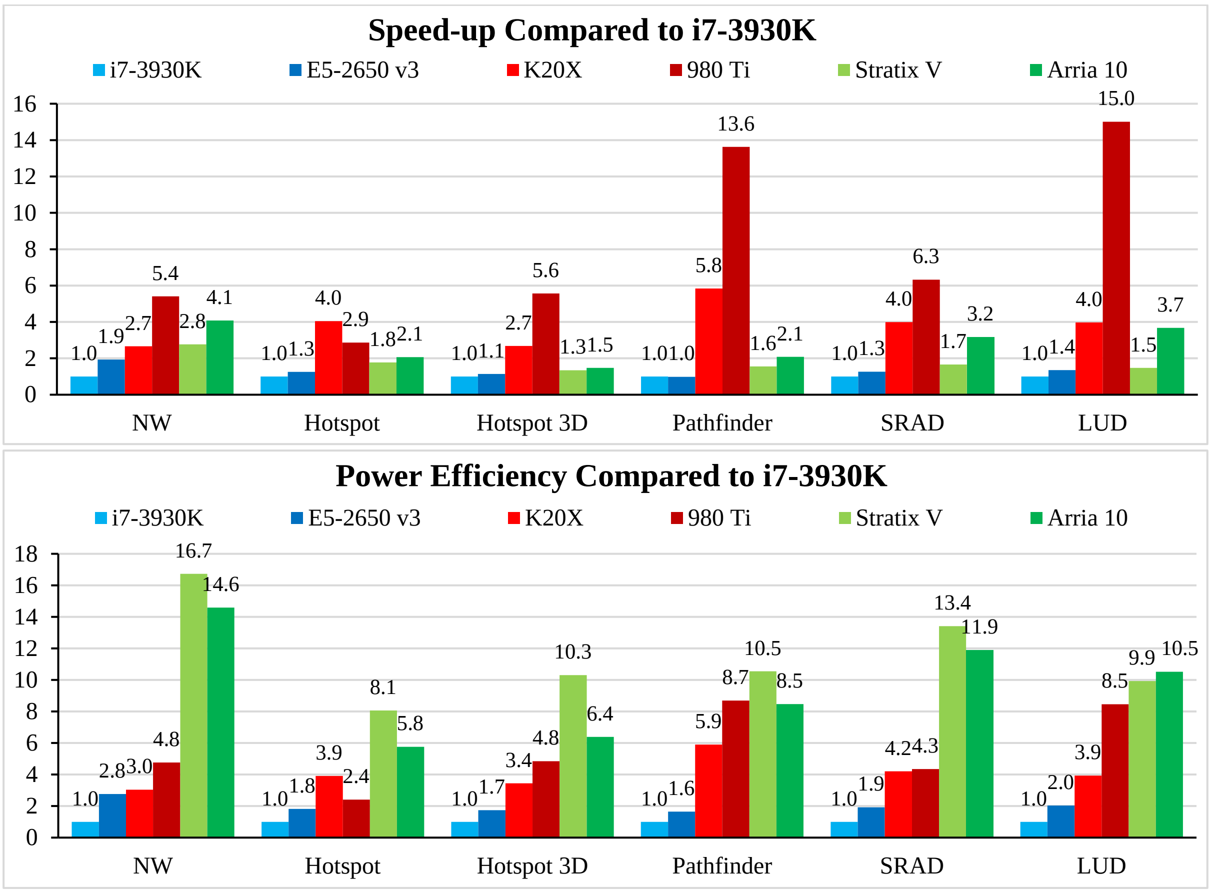

Figure 4-2 Performance and Power Efficiency Comparison Between Different Hardware

The smallest performance difference between the Arria 10 FPGA and the 980 Ti GPU is observed in the NW and Hotspot benchmarks. For the case of NW, the gap between the FPGAs and GPUs is minimized since our FPGA design can efficiently handle the complex loop-carried dependency of this benchmark while fully pipelining the design with an initiation interval of one. Our FPGA implementation of this benchmark is near-optimal and its performance is effectively limited by the external memory bandwidth of the FPGAs. However, the GPU implementation of Rodinia is sub-optimal and a more optimized implementation could likely achieve better performance on the GPUs and increase the gap. On the other hand, our FPGA implementation and Rodinia's GPU implementation for Hotspot are based on the same algorithm, but temporal blocking achieves better scaling on FPGAs compared to GPUs. Hence, the performance gap between these devices could potentially become even smaller if temporal blocking is employed more efficiently on the FPGAs. Performance of this benchmark scales up to a degree of temporal parallelism (*pyramid_height*) of 6 on the FPGAs due to large block size, while on the GPUs the performance only scales up to a degree of 2 or 3 due to relatively smaller block size. Hence, we conclude that stencil computation is likely one of the computation patterns that could be efficiently accelerated on FPGAs, and despite the large external memory bandwidth gap between FPGAs and GPUs, it could be possible to achieve competitive performance on FPGAs compared to GPUs for this computation pattern if temporal blocking is efficiently utilized. Dynamic programming applications like NW and Pathfinder are also good candidates for acceleration on FPGAs. However, as we shown in this chapter, these types of applications quickly reach the limit of the external memory bandwidth on FPGAs and cannot benefit from temporal blocking since they are generally not iterative. Hence, an optimized implementation of such benchmarks on a GPU will likely always be faster than an optimized implementation on an FPGA due to their large gap in external memory bandwidth. For the more compute-bound benchmarks, the performance difference between FPGAs and their same-generation GPUs gets larger since the large gap in the peak compute performance of these devices cannot be easily filled. Even though FPGAs allow us to create custom pipelines for each application to achieve higher computational efficiency compared to GPUs, it is generally not enough to fill the large compute performance gap unless a specific application achieves very low computational efficiency on GPUs (less than 10%).

In the end, we should emphasize that the goal of this chapter was *not* to achieve the best performance for each benchmark on the FPGAs, but rather, to determine effective FPGA-based optimizations for each benchmark and optimize each to a degree that would allow fair comparison with the CPU and GPU platforms. We expect that there could still be room to further optimize some of benchmark on the FPGAs. For example, all of the stencil benchmarks could benefit from temporal blocking (some of which we will revisit in the next chapter), or LUD could benefit from a systolic array-based implementation which could match much more efficiently with the underlying FPGA hardware. Reducing the width of datatypes is also an important FPGA-specific optimization that we did not study here; this optimization could significantly reduce area and memory bandwidth usage on these devices for some benchmarks (especially integer ones) and lead to noticeable performance improvements. However, if such optimizations are performed for the FPGAs, the optimization level of the FPGA kernels might go beyond the CPU and GPU kernels and prevent fair comparison between the devices.

4.4 Related Work

In [24], the authors present OpenDwarfs, a multi-platform benchmark suite written in OpenCL that can target different hardware including FPGAs. They implement and evaluate multiple benchmarks on a range of CPU and GPU devices and one Xilinx Virtex-6 FPGA, and report performance results and detailed comparison. For converting the OpenCL kernels to synthesizable HDL code, they use SOpenCL [31]. Our work differs from theirs in this respect that we use newer hardware and a much more mature OpenCL compiler for our FPGA platform which is made by the FPGA manufacturer. Furthermore, they explore very few optimizations on each hardware and their FPGA optimizations are limited to basic loop unrolling, while we use CPU and GPU implementations that are already optimized to a reasonable degree and also manually apply a range of FPGA-specific optimizations to our FPGA kernels to allow fair comparison with the other hardware. In [32], the same authors present a visual programming framework that can automatically generate and tune code targeting Intel FPGA SDK for OpenCL. Their framework supports generation of both NDRange and Single Work-item kernels and can automatically apply optimizations related to data parallelism (Section 3.2.1.5), shift register optimization for floating-point reduction (Section 3.2.2.1), *restrict* keyword (Section 3.2.1.1), aligned DMA transfers, and general shift register inference (Section 3.2.4.1). However, the framework cannot automatically detect patterns that could benefit from shift register inference in the application and hence, this optimization needs to be guided by the programmer. The authors then implement three benchmarks (Electrostatic Surface Potential Calculation, Gene Sequence Search and Time-domain FIR Filter) on an Intel Stratix V FPGA, and discuss the effect of using the NDRange or Single Work-item kernel type, basic data parallelism optimizations, and shift registers as on-chip memory.

In [33], the authors of the Rodinia benchmark suite present a preliminary study using three benchmarks, namely NW, DES and Gaussian Elimination, and compare performance in terms of number of clock cycles between a CPU, a GPU and an FPGA. This work uses VHDL to implement the benchmarks on the FPGA and does not discuss platform-specific optimizations in detail. The Xilinx Virtex-II FPGA they use is also relatively old compared to the other hardware used in their evaluation. In contrast, we use two modern generations of FPGAs and compare each to a CPU and GPU of its own generation. Furthermore, we use a mature HLS tool to implement the benchmarks on the FPGAs that significantly reduces development effort, and take advantage of multiple FPGA-specific optimizations to allow a fair comparison with the already-optimized CPU and GPU implementations.

In [34], the authors present a framework for converting C/C++ code that is annotated with OpenACC directives to OpenCL targeting Intel FPGA SDK for OpenCL. They add equivalent functions to OpenACC to support basic OpenCL functions like host/device data transfer, setting kernel arguments, kernel invocation, etc. Furthermore, the framework automatically enables aligned DMA transfers, and provides a set of pragmas to support on-chip channels, loop unrolling, SIMD and kernel pipeline replication. All of these functions and pragmas are directly mapped to their equivalent in OpenCL host and kernel code to be used with Intel FPGAs. One shortcoming of this work is that the resulting OpenCL kernel uses the NDRange programming model, which is not the preferred model on FPGAs. Furthermore, all optimizations and parameter tuning are still left to the programmer. They perform basic parameter tuning on FPGAs for multiple benchmarks, and present performance comparison with CPU, GPU and Xeon Phi platforms using the same OpenACC code, which is not necessarily optimized for either of the hardware. We, however, take advantage of many basic and advanced optimization techniques on FPGAs and provide a more dependable performance comparison since our CPU and GPU benchmarks are already optimized to a reasonable degree. Very recently, the authors extended their work in [35] by adding support for multiple new optimizations to their framework that can significantly reduce programming effort. Specifically, they add support for Single Work-item kernels, shift register optimization for floating-point reduction (Section 3.2.2.1), Loop collapsing (Section 3.2.4.3), shift register inference for local data storage (Section 3.2.4.1) and branch-variant code motion optimization. They also provide support for both of the shift register-based optimizations being coupled with unrolling. Some analysis is provided on the effect of each optimization on a set of benchmarks and performance and power efficiency comparison between a Stratix V D5 FPGA, a Xeon CPU and an NVIDIA GPU is reported. One shortcoming of the work is that when unrolling is coupled with optimization of floating-point reduction operations, they directly pass the unroll pragma to the OpenCL kernel and let the OpenCL compiler unroll the loop. As mentioned in Section 3.2.2.1, this method of unrolling would require increasing the size of the shift register and can result in significant area overhead for large unroll factors. In contrast, if the loop is unrolled using the method we proposed in Section 3.2.2.1, a larger shift register will not be required and the extra area overhead can be avoided. In addition, since their compiler does not yet support loop blocking/tiling using OpenACC directives, using the automatic shift register inference will require manual loop blocking by the programmer since this optimization cannot be applied if the loop bounds are not known at compile-time.

In [36], the authors use Xilinx SDAccel to implement a set of benchmarks (K-Nearest Neighbor, Monte Carlo Method and Bitonic Sort) on a Xilinx Virtex-7 FPGA, and compare performance and power efficiency with two GPU platforms. A set of basic FPGA-based optimizations are performed and better power performance in some cases and better power efficiency in most is reported compared to the GPUs. However, multiple shortcomings reduce the dependability of their results. The GPUs used in the study are relatively low-end and not comparable with the FPGA platform. On top of that, the majority of the GPU kernels are not optimized, which gives an unfair advantage to the FPGA. In addition, the input sizes are very small in every case and benchmark run times are below 1 ms or even 1 µs in most cases, and in one case the input is saved on the FPGA on-chip memory instead of external memory, giving the FPGA platform even more unfair advantage over the GPUs. Power consumption comparison also puts the GPU board against the FPGA chip alone, without considering the FPGA external memory. In contrast, we keep the comparison as fair as possible by using same-generation devices, employing kernels on each device that have gone through a reasonable amount of optimization, and using large input sizes that allow dependable run time measurements.

In [37], the authors present an OpenCL-based benchmark suite for FPGAs targeting Intel FPGA SDK for OpenCL. They discuss nine benchmarks extracted from Rodinia [13], OpenDwarfs [24], Intel's OpenCL examples and other sources, and compile each with multiple performance parameters for a total of over 8000 different configurations. They also attempt to mathematically model the relationship between design parameters and performance. This work only uses the NDRange programming model, which is not the preferred programming model on FPGAs, and only takes advantage of basic compiler-assisted optimization techniques. They conclude that the relationship between performance and design parameters is difficult to model mathematically due to complex interactions between such parameters.

In [38], the authors present a framework that uses information from both OpenCL host and kernel code to automate certain optimizations that Intel FPGA SDK for OpenCL Offline Compiler cannot perform on its own due to lack of knowledge of the host code. Specifically, they provide the means of automatically adding the *restrict* keyword to kernel buffers (Section

3.2.1.1) if no pointer aliasing exists in the host code, and converting NDRange kernels to Single Work-item by wrapping each region between two barriers in a multiply-nested loop using local and global work size information extracted from the host code. Moreover, they automate shift register inference for optimizing floating-point reduction (Section 3.2.2.1). Their work is an early step in automating optimizations that we performed manually here.

In [39], the authors perform a study similar to ours but target Xilinx tools and FPGAs. They port multiple benchmarks from the Rodinia benchmark suite [13] and optimize each using a set of general optimizations targeting Xilinx Vivado HLS [5] and SDAccel [7]. Speed-up over baseline, and performance and power efficiency comparison between a Xilinx Virtex-7 FPGA and an NVIDIA Tesla K40c GPU are reported, with the FPGA being faster than the GPU in some kernels. They use sequential implementations that do not even have pipelining enabled as FPGA baseline, resulting in over 4,000 times speed-up after optimization for some benchmarks. We, however, use baselines that are optimized for GPUs rather than such unoptimized codes that do not represent real-world scenarios to avoid such unrealistic speed-up ratios. Furthermore, their timing results seem to suggest that some of the benchmarks were running for a few days (e.g. 50 hours for LUD). Based on the more detailed version of the publication available at [40], it seems the reported timing values are actually in µs, and the timing unit is incorrectly reported as seconds in [39]. Having this in mind, the run time for all of the benchmarks in their case is lower than 200 ms, and go as low as 48 µs in case of NW; such short run times that can be heavily affected by profiling overhead cannot be considered as basis for dependable performance comparison. In fact, the three kernels in which they report better performance on the FPGA compared to the GPU are among the shortest ones. Moreover, as mentioned in Section 0, such short run times will not allow correct power readings on the GPU either as is also evident in their reported power consumption for the GPU. Only two of their evaluated benchmarks use more than 100 Watts on the GPU, and most are under 80 Watts (less than one third of the GPU’s TDP) and go as low as 53.7 Watts in case of NW; such values represent the idle power of the GPU rather than power usage during kernel execution. They also do not report how FPGA power consumption is measured and whether it includes FPGA external memory or not. In contrast, apart from general optimizations, our work also includes benchmark-specific transformations in every case and our results are highly dependable since we make sure to use big input sizes to allow correct time and power measurement. In addition, we achieve good performance and better power efficiency on the FPGAs compared to GPUs in every case, while lack of benchmark-specific optimizations in their case results in very low performance in some benchmarks that we successfully optimize (e.g. LUD and Hotspot) and lower power efficiency compared to the GPU in multiple benchmarks.

In [41], the authors present another study similar to ours that also targets Xilinx FPGAs and tools. This work presents a wide range of HLS-based optimization techniques for FPGAs, some of which we also covered in this chapter, and includes code examples for many of them. They apply the optimization techniques to three applications, namely a Jacobi 2D stencil, General Matrix Multiplication (GEMM) and an N-body code, and report speed-up achieved with each level of optimization. This work is analogues to ours on the Xilinx platform but lacks performance comparison with other devices.

4.5 Publication Errata

The publication in WRC’16 [22] presented a very preliminary version of the results presented here with only four benchmarks and limited advanced optimizations. The current results are updated and largely different from the numbers reported in that publication.

Compared to the publication at SC’16 [17]:

- Performance model has been improved by splitting initiation interval into compile-time and run-time initiation interval and discussing them separately.
- Multiple new optimization techniques have been added and all optimizations have been discussed in a more organized manner.
- All the results presented in this chapter are updated with more optimizations and parameter tuning in nearly every case, resulting in better performance and power efficiency.
- All kernels have been compiled using a newer version of Quartus and Intel FPGA SDK for OpenCL Offline Compiler.
- Hotspot 3D has been added to the benchmarks but CFD has been removed.
- Arria 10 results have been reported for every benchmark and performance and power efficiency comparison between Stratix V and Arria 10 has been added.
- Performance of Hotspot on FPGAs has significantly improved compared to other hardware since the input settings used in the publication were too small, resulting in the input completely fitting in the CPU and GPU caches and giving them an unfair advantage.
- In the publication, it is incorrectly assumed than loop unrolling increases the pipeline depth by the unroll factor; the pipeline depth does increase with unrolling, but not by the unroll factor. This issue is has been corrected in this document.
- Newer versions of ICC and CUDA have been used to maximize the performance of the CPUs and GPUs. Furthermore, “-fp-model precise” is added to ICC compilation parameters to make sure that accuracy of floating-point computations is the same on all hardware.

4.6 Conclusion

In this chapter, we ported a subset of the Rodinia benchmark suite for FPGAs as a representative of typical HPC workloads. We showed that even though the original NDRange kernels designed for GPUs generally perform poorly on FPGAs, and basic guidelines from Intel’s documents hardly improve their performance, advanced FPGA-specific optimizations are effective on both NDRange and Single Work-item kernels, allowing us to achieve at least one order of magnitude performance improvement over the baseline on FPGAs for every benchmark. Furthermore, we showed that in most cases the Single Work-item programming model matches better with the underlying FPGA architecture, allowing us to take better advantage of the unique features of these devices.

Our results showed that FPGAs have a performance and power efficiency advantage over their same-generation CPUs in every case. However, compared to GPUs, it is generally not possible to achieve better performance due to the large gap in external memory bandwidth and compute performance between current-generation FPGAs and GPUs. The main bottleneck of performance for current-generation FPGAs is their low external memory bandwidth, resulting in memory-bound performance for nearly every benchmark on the new Arria 10 FPGA. Despite these limitations, FPGAs can still achieve higher power efficiency compared to not only their same-generation GPUs (up to 5.6 times) but also newer-generation ones.

Among the evaluated benchmarks, stencil-based applications showed better matching with the FPGA architecture than the rest and we expect that by taking full advantage of the FPGA-specific shift register buffers and temporal blocking, we should be able to minimize the performance gap between FPGAs and GPUs for this type of computation. Based on this conclusion, we extend our work by implementing a highly-optimized stencil accelerator on FPGAs, which will be discussed in the next chapter.