Tuning Stencil Codes in OpenCL for FPGAs

Qi Jia
Dept. of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina
qjia2@ncsu.edu

Huiyang Zhou
Dept. of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina
hzhou@ncsu.edu

***Abstract*—OpenCL is designed as a parallel programming framework to support heterogeneous computing platforms. The implicit or explicit parallelism in OpenCL kernel code enables efficient FPGA implementation from a high-level programming abstraction. However, FPGA architecture is completely different from GPU architecture, for which OpenCL is widely used. Tuning OpenCL codes to achieve high performance on FPGAs is an open problem and the existing OpenCL tools and optimizations proposed for CPUs/GPUs may not be directly applicable to FPGAs. In this paper, we explore OpenCL code optimizations for stencil computations on FPGAs. We propose tuning processes for stencil kernels in both the Single-Task and NDRange modes. Our optimized 1D convolution, 2D convolution and 2D Jacobi iteration kernels can achieve up to two orders of magnitude performance improvement over the naïve kernels. Also, compared to Altera design examples our optimized kernels achieve 7.1× and 3.5× speedups for the Sobel and Time-Domain FIR Filter, respectively. This study also includes benchmarking of the FPGA memory system, revealing how code patterns affect the performance of different types of memory on FPGAs.[1]**

Keywords—Stencil codes; OpenCL; Performance Optimization; FPGAs.

1. INTRODUCTION

FPGAs are an attractive choice to accelerate a wide range of applications, such as high performance computing [7], databases [12], neural networks [9], etc., due to their high energy efficiency and throughput. However, the programmability of FPGAs is a major hurdle for them to be widely adopted. Many previous works have aimed to automating the FPGA design process through high-level synthesis. Recently, OpenCL, a parallel programming framework designed for heterogeneous architectures, is being supported to enable programmers to develop high-level code for FPGAs. Although using OpenCL significantly improves the programmability of FPGAs, how to write high performance OpenCL code for FPGAs remains a critical challenge. The reason is that the architecture of FPGAs generated from OpenCL code is fundamentally different from that of CPUs/GPUs, on which the existing OpenCL tools and optimizations mainly focus. Therefore, it is crucial to understand the FPGA architectures and the characteristics of FPGA on-chip resources in order to optimize OpenCL code for FPGAs.

Stencil (nearest-neighbor) computations are an important class of algorithms at the heart of many calculations involving structured (i.e., rectangular) grids, including implicit and explicit partial differential equation (PDE) solvers.

Given stencil computations' importance and wide usage, there has been much effort on optimizing them on different computing platforms ranging from supercomputers to smartphones. Most of the existing works [4][8][11][13] are primarily designed for CPUs/GPUs. Unfortunately, many of these schemes may not be directly applied to FPGAs. Although there are also prior works [6][10][14] focusing on customizing stencil computations on FPGAs, they mainly utilize hardware description languages while our focus is the high-level OpenCL framework. To the best of our knowledge, this is the first work on tuning stencil codes in OpenCL for FPGAs.

In this work, we tune the 1D/2D stencil codes using different optimizations in the two modes supported in the Altera OpenCL for FPGA framework, NDRange Kernels and Single-Task Kernels (aka Single-Work-Item Kernels). We also reveal how different types of memory will be affected by OpenCL code patterns on FPGAs.

As shown in a previous work [16], tuning OpenCL program code has a remarkable performance impact on the synthesized FPGAs. Our study confirms the case for OpenCL stencil codes. Using the 2D convolution (2DConv) as an example, we optimize the OpenCL code with different optimizations. The normalized performance speedups from these optimizations (details of the optimizations are in Section 3) are shown in Fig. 1. From the figure we can see that the optimized version can be over two orders of magnitude faster than the baseline (i.e., un-optimized) code. This highlights the importance of OpenCL code optimizations for FPGAs.

In summary, this paper makes the following two contributions. First, we propose performance tuning processes for 1D/2D stencil OpenCL code for FPGAs and the resulting optimized code significantly outperforms the design examples provided by Altera. Second, we reveal details about how different types of memory on FPGAs are generated and their corresponding performance characteristics.

[1] This work is supported in part by NSF grants CCF-1216569 and CCF-1618509.

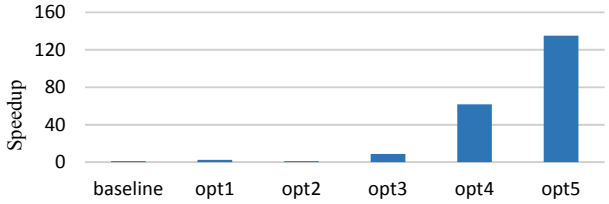

Fig. 1. 2DConv kernel performance improvements over the baseline from different optimizations.

The remainder of the paper is organized as follows. In Section 2 we introduce the OpenCL architectures on FPGAs. We propose the performance tuning processes for stencil codes in Section 3. We characterize different types of memory resources on FPGAs in Section 4. Section 5 analyzes our experiments results. We compare our work with previous related work in Section 6. Finally, Section 7 concludes.

2. OpenCL Architectures for FPGAs

Recently developed OpenCL frameworks for FPGA, such as the Altera SDK for OpenCL [1], enable a user to replace/augment the traditional hardware FPGA design flowwith a much faster and higher-level software development flow. Instead of treating FPGAs as pure hardware resources such as look-up tables (LUTs), registers, DSP blocks and memory blocks, the OpenCL for FPGA architecture is a massive parallel computing engine, as shown in Fig. 2. The generic OpenCL for FPGA architecture consists of one or more compute units (CUs), different types of memory and the interconnects among them.

Each instruction in an OpenCL kernel is translated into a functional unit, e.g., an adder or a load-store unit (LSU), and a multi-instruction sequence is synthesized into a pipeline of functional units. The key to achieve high throughput is to convert thread-level parallelism in NDRange kernels or loop-level parallelism in Single-Task kernels into pipeline parallelism. The pipeline synthesized from a kernel is referred to as a Compute Unit (CU). Each CU provides pipelined execution for massive threads/loop iterations of an OpenCL kernel.

Based on OpenCL, there are four types of memory in OpenCL for FPGA systems. First, global memory resides in the off-chip DRAM on an FPGA board. The accesses to global memory have long latencies, and the bandwidth is shared by all the CUs on the FPGA. The data in global memory can be buffered in on-chip caches embedded within LSUs. Second, local memory is low-latency and high-bandwidth software-managed scratchpad memory. Third, constant memory also resides in off-chip DRAM on the FPGA board while its data can be buffered in on-chip constant caches. Fourth, private memory, storing private variables or small arrays for each work-item, is implemented using on-chip registers and has the fastest speed.

There are typical OpenCL optimizations [2][3] presented in the Altera OpenCL for FPGA programming/optimization guide, including the usage of local memory (LM), loop unrolling (LU), kernel vectorization (SIMD) and compute unit replication (CUR). In this work, we propose the tuning process

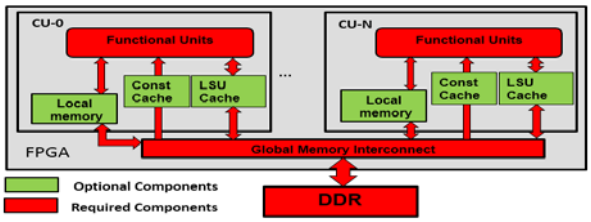

Fig. 2. The Altera OpenCL for FPGA Architecture.

to streamline such optimizations and highlight the importance of additional optimizations such as thread/task coarsening.

3. Tuning OpenCL Stencil Codes for FPGAs

The OpenCL kernels for Stencil codes on FPGAs can be implemented in either the Single-Task mode or the NDRange mode [2]. We propose somewhat different optimization processes and present them separately.

3.1 Optimizing Single-Task Kernels

In the Single-Task mode, a kernel is implemented as a sequential program. The compiler, e.g., the Altera Offline Compiler (AOC), extracts the loop-level parallelism to enable pipelined execution. Our proposed optimization process for stencil kernels in the Single-Task mode is shown in Fig. 3. It consists of both memory architecture tuning and computation pipeline tuning. As the stencil kernels' performance on FPGAs is eventually bound by memory, memory architecture is tuned before computation pipelines.

Tuning memory architecture for stencil kernels in the Single-Task mode mainly includes constant memory (CM) and shift register patterns (SRP).

Putting data in constant memory can leverage constant caches, whose performance is dependent on the cache hit rates. In stencil codes, there are some small-size and frequently-accessed data. Putting these small matrices into constant memory is a natural optimization on GPUs. But for FPGAs, the decision between putting them into either global memory or constant memory is more delicate and we need to consider both the matrix sizes and access patterns. We will discuss the impact from different types of memory in Section 4.

Applying SRP on Single-Task kernels can improve the memory access efficiency significantly for the following two reasons. First, shift registers help to make use of the data temporal locality in stencil codes by buffering the data in registers, which can reduce the global memory accesses significantly. Take 1D convolution (1DConv) as an example. Fig. 4 shows a snippet of the 1DConv Single-Task kernel after applying SRP. From line 3 to line 5, we shift one element and insert a new element from the input, *srcMatrix*. In line 8, the shift registers are used to perform the computation. All the elements in the *srcMatrix* are loaded from global memory only once and fully reused before they are shifted out. Assume the *outputMatrix* and *srcMatrix* have *N* elements while the size of *filterMatrix* is *M*. Then after applying SRP, the number of global memory accesses is decreased from *N*M* to *N*. Second, as stated in [2], the shift register accesses are more efficient than on-chip block RAMs accesses. It is worth indicating that

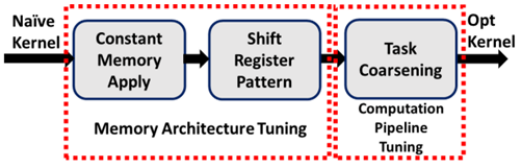

Fig. 3. The tuning/optimization process for single-task kernels.

```
L0:  int shiftReg[N];
L1:  for(int i=0;i<N;i++)        shiftReg[i]=0;
L2:  for(int j=0;j<output_size;j++) {
L3:     #pragma unroll
L4:     for(int i=0;i<N-1;i++)     shiftReg[i]=shiftReg[i+1];
L5:     shiftReg[N-1]=srcMatrix[j];
L6:     sum = 0;
L7:     #pragma unroll
L8:     for(int i=0;i<N;i++)
L9:        sum += shiftReg[i]*filterMatrix[N+j-i];
L10:    outputMatrix[j] = sum;
L11: }
```

Fig. 4. A code snippet of a 1DConv Single-Task kernel using SRP.

in line 3 and line 7, '#pragma unroll' is necessary for AOC to infer the shift registers.

The computation pipeline tuning for stencil kernels in the Single-Task mode is mainly task coarsening (TAC) as the SRP already enforces full unrolling of inner loops. Unrolling inner loops deepens the synthesized pipeline and enables more outer-loop iterations to be in the pipeline simultaneously, thereby improving the throughput.

Besides deepening the pipeline, the throughput can be improved with widening the pipeline. For example, rather than producing one output, *outputMatrix*[*j*], per iteration in the outer loop (line 2 in Fig. 4), we can generate multiple outputs per iteration. We refer this optimization as task coarsening. Task coarsening is apparently similar to unrolling and jamming the outer loop in Fig. 4. But there is a subtle but important difference. Considering task coarsening with a factor of two, i.e., we want each outer loop iteration computes two outputs, i.e., *outputMatrix*[*j*] and *outputMatrix*[*j*+1]. Although this can be achieved by unrolling and jamming the outer loop enclosed between line 2 and line 11, a more efficient way is to change the size of the shift register at line 0, '*int shiftReg[N+1]*', and shift the registers by two at a time, i.e., '*shiftReg[i]=shiftReg[i+2];*' at line 4. Our experiments confirm that for 1DConv, such manual TAC outperforms unrolling the outer loop with '#pragma unroll' by 25.3% when the coarsening (or unroll) factor is 8.

3.2 Optimizing NDRange Kernels

In the NDRange mode, the algorithms need to be parallelized into separate work-items (or threads). In other words, the thread-level parallelism needs to be expressed explicitly by programmers. Fig. 5 shows our optimization process for stencil kernels in the NDRange mode, which is also composed of memory architecture tuning and computation pipeline tuning, similar to the Single-Task mode.

Memory architecture tuning for NDRange kernels includes applying constant memory (CM), local memory (LM), and vectorization (VEC) to improve memory access bandwidth. Utilizing CM needs to consider both the data size and access patterns similar to what we discuss for Single-Task kernels. VEC is applied in the final stage in memory architecture tuning since the memory access type needs to be known before the accesses are vectorized.

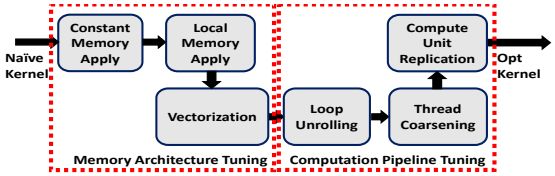

Fig. 5. The tuning/optimization process for NDRange kernels

```
L0:  __local int Slocal[BLK_SIZE+KERNEL_SIZE-1];
L1:  int block_x=get_group_id(0);
L2:  int local_x=get_local_id(0);
L3:  int start_x=BLK_SIZE*block_x;
L4:  Slocal[4*local_x]=Smatrix[start_x+4*local_x];
L5:  Slocal[4*local_x+1]=Smatrix[start_x+4*local_x+1];
L6:  Slocal[4*local_x+2]=Smatrix[start_x+4*local_x+2];
L7:  Slocal[4*local_x+3]=Smatrix[start_x+4*local_x+3];
```

Fig.6. A code snippet of a 1DConv NDRange kernel using LM.

Local memory can be used as software managed caches to reduce the number of global memory accesses. It is appropriate for stencil codes due to their strong data reuses. Fig.6 shows the LM optimization for 1DConv NDRange kernel. On GPUs, a programmer typically aims to make use of all available shared memory resources as long as shared memory is not the limiting factor for thread-level parallelism. But it is a different story on FPGAs. The kernel performance on FPGA does not always favor large local memory sizes. Take 2DConv as an example. We set the filter matrix size as 17x17 and the work group size as *N*x*N*. Then the local memory size needs to be (*N*+16)x(*N*+16). The larger the local memory size is, the more data reuses will be captured by local memory, which helps to improve the system performance. But if *N* is larger than 16, some of the work-items will be idle when loading elements in the halo from global memory into local memory. As the total execution time of the kernel on FPGAs is determined by the total number of work-items instead of active work-items, the inactive work-items still cause overhead, hurting the system performance. Fig. 7 shows the 2DConv kernel execution time using various local memory sizes (the kernel only makes use of local memory and does not use other optimizations) with the filter size 17x17. From Fig. 7, it can be seen that when the local memory size is 32x32 the performance is the best. So, when applying local memory optimization on OpenCL kernels on FPGAs, the local memory size should be determined by the filter size and workgroup size.

Vectorization helps to coalesce memory accesses to utilize the global memory bandwidth efficiently. Originally as stated in [3], memory coalescing refers to combining multiple consecutive global memory accesses from in a single work-item into a single aligned memory access. In this paper we extend vectorization and coalescing to local memory accesses and constant memory accesses, which will help to improve the system performance further. AOC can coalesce global memory accesses. But for local and constant memory accesses it is not

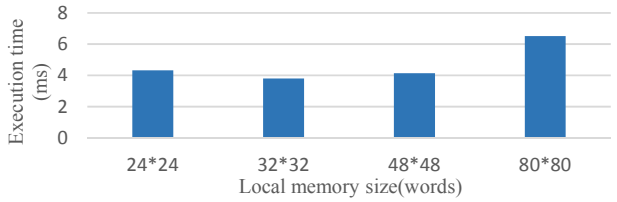

Fig. 7. 2DConv kernel execution time with variable local memory sizes (the lower, the better).

```
L0:  __local int4 Slocal[BLK_SIZE+KERNEL_SIZE-1];
L1:  int block_x=get_group_id(0);
L2:  int local_x=get_local_id(0);
L3:  int start_x=BLK_SIZE*block_x;
L4:  Slocal[local_x]=(int4)(Smatrix[start_x+4*local_x],
L5:                         Smatrix[start_x+4*local_x+1],
L6:                         Smatrix[start_x+4*local_x+2],
L7:                         Smatrix[start_x+4*local_x+3]);
```

Fig. 8. A code snippet of a 1DConv NDRange kernel using LM and VEC.

guaranteed. To implement vectorization for local and constant memory, we directly use vector data types (e.g. float2, int2) to access memory instead of scalar types such as float and int.

Fig. 8 shows the local memory access vectorization for the 1DConv NDRange Kernel. In line 0, the vector data type int4 is used instead of int. Then from line 4 to line 7, four consecutive local memory accesses are combined into a single vector (int4) access. The impact of vectorizing local and constant memory accesses will be discussed further in Section 4.

Tuning the computation pipeline is composed of loop unrolling (LU), thread coarsening (TC) and compute unit replication (CUR). Although both LU and TC introduce more hardware resources to improve pipeline throughput, full LU also eliminates loop-control overhead. Thus, LU is applied before TC such that hardware resources are prioritized for full loop unrolling.

Thread coarsening merges the workload of multiple work-items to one work-item, similar to task coarsening discussed earlier. The memory accesses from multiple work-items previously are now in one work-item. This provides AOC with more opportunities to coalesce memory accesses. Also thread coarsening may reduce the local memory accesses by reusing the data among multiple work-items. In this paper we implement thread coarsening by using the attribute "num_simd_work_items" provided by Altera OpenCL SDK. It has the same effect when we perform coarsening along the *X* direction. In general, coarsening is more generic than the SIMD attribute as we can merge multiple work-items in arbitrary ways, e.g., along the *Y* direction.

The kernel pipeline can be replicated to generate multiple CUs to achieve higher throughput if there are sufficient hardware resources on the FPGA. Since CUs consume more hardware resources, the operating frequency of FPGA tends to be lower than that of a single CU. Thus two CUs cannot always double the performance. Another issue with multiple CUs is that the global memory load/store operations from multiple CUs cannot be coalesced and they compete for the global memory bandwidth which may degrade the global memory performance. So, we apply this optimization as the final step in our tuning process only if there are still abundant hardware resources available.

```
L0:  __local int4 Slocal[BLK_SIZE+KERNEL_SIZE-1];
L1:  int block_x=get_group_id(0);
L2:  int local_x=get_local_id(0);
L3:  int start_x=BLK_SIZE*block_x;
L4:  Slocal[local_x]=(int4)(Smatrix[start_x+4*local_x],
L5:                         Smatrix[start_x+4*local_x+1],
L6:                         Smatrix[start_x+4*local_x+2],
L7:                         Smatrix[start_x+4*local_x+3]);
L8:  barrier(CLK_LOCAL_MEM_FENCE);
L9:  #pragma unroll
L10: for(int a=0;a<KERNEL_SIZE;a++) {
L11:    // Multiplication between Slocal and Fmatrix, then
        // accumulate the result to store into sum_0
L12:    // Multiplication between Slocal and Fmatrix, then
        // accumulate the result to store into sum_1
L13: }
L14: Omatrix[get_global_id(0)]=sum_0;
L15: Omatrix[get_global_id(0)+1]=sum_1;
```

Fig. 9. The optimized 1DConv NDRange kernel.

Table 1. Latency of different types of memory on an Altera Stratix V FPGA.

Memory type	Latency/cycles	Frequency/MHZ
Global memory	82	304
Constant cache	10	302
Local memory	13	301
LSU embedded cache	11	305

Fig. 9 shows the optimized 1DConv kernel after applying the tuning process. In line 0, we utilize local memory. From line 4 to line 7 we apply the vectorization on both global and local memory accesses. Loop unrolling is indicated in line 9. In line 11 and line 12, the thread coarsening optimization is applied with a factor of 2 by merging the workload of two consecutive work-items along the *X* direction. Compute unit replication can be applied using attribute *num_compute_unit* (*N*) directly.

3.3 Comparison with Optimizations for GPUs

In the NDRange mode, our proposed optimizations are not the same as those on GPUs although the intuitions are quite similar. Due to the differences between the two architectures, tuning the local memory size on FPGAs is different from that on GPUs. Also, the choice for appropriate memory types (e.g. constant memory vs. local memory) is also different between FPGAs and GPUs. The details will be shown in the following section. In the SingleTask mode, one important optimization is SRP, which is not applicable on GPUs.

4 Characterizing FPGA Memory Systems

As mentioned in Section 2, there are four types of memory in OpenCL for FPGA architecture. But currently no details on the latencies and structures of these types of memory have been released.

In this section, we first use a pointer-chasing benchmark similar to that used in [15] to reveal the latency and structure of the memory system. The benchmark traverses an integer array

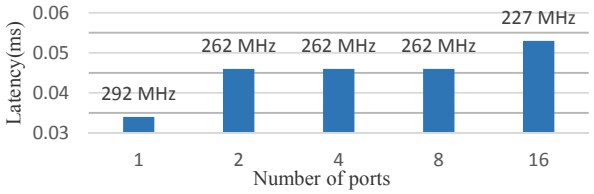

Fig. 10. Constant cache latency and frequency with different number of ports.

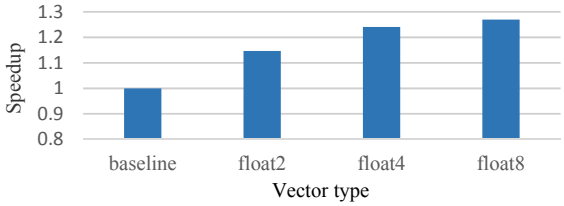

Fig. 11. System speedup over baseline for 1Dconv NDRange kernels with different vector data types.

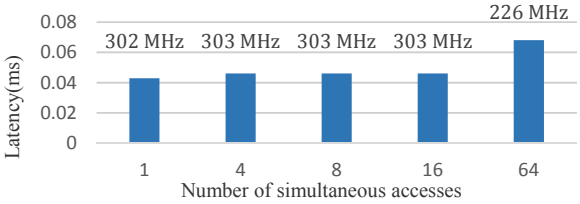

Fig. 12. Local memory latency and frequency with different numbers of simultaneous accesses.

A by running *k = A[k]* in a long loop. The kernel execution time is dominated by the latency of memory accesses. The kernel is launched as a single work-item, which will eliminate the work-item scheduling overhead. Table 1 presents the latencies of global memory, local memory, the constant cache and the LSU embedded cache for a single access.

4.1 Global Memory and LSU embedded caches

As shown in Table 1, global memory access latency is long compared to on-chip memory. But AOC can instantiate on-chip LSU caches for global memory accesses once it detects the global memory data are used repeatedly.

The LSU embedded cache is constructed as a direct-mapped cache with a block size of 64 bytes. The cache capacity is decided by AOC. Once the global memory footprint is too large AOC may choose not to use the cache.

When there are multiple global memory access instructions in the loop, AOC will generate separate LSUs with private embedded caches for each memory access if the data are used repeatedly.

The LSU embedded cache is not tuned in this paper since it is controlled by the AOC compiler and we cannot change its parameters flexibly.

4.2 Constant Caches

Constant caches have the shortest latency among all the on-chip memory as shown in Table 1.

Due to abundant RAM resources on FPGAs, the size of constant cache can be set very large. According to [3], the default constant cache size is 16KB and the programmer can set the size using the flag *–const-cache-size* when compiling the code. If there are multiple constant cache accesses per loop iteration, AOC will choose to increase the number of ports in the constant cache to improve the bandwidth. The constant cache port width will remain the same as 64 bytes.

We perform experiments to test how the number of ports and the cache size would affect the constant cache performance. Fig. 10 shows the latency and frequency of constant caches with different number of ports while maintaining the cache size as 16KB. The bars in the figure indicate the latency and the numbers on top of the bars show the corresponding frequency. From the figure, we can see that when more and more memory requests need to access the constant cache simultaneously, the constant cache performance drops significantly due to the increased number of ports. Our experiments on different cache sizes from 16KB to 1MB also show that the constant cache performance is not apparently affected by the cache size in this range.

4.2.1 Constant Cache Access Vectorization

As shown above, the constant cache performance drops significantly with the increase in the number of ports. The number of ports can be reduced by vectorizing the constant cache accesses, which will help to improve the performance finally. But some control overheads may be introduced when utilizing vectorization. We perform experiments to analyze the effect of vectorization on the performance of 1DConv. We store the filterMatrix in a constant cache and disable all other optimizations. This configuration is considered as baseline.

Then we vectorize the constant cache accesses using different vector data types (float2, float4 and float8). Fig. 11 shows the speedups with different vector data types over the baseline.

From the figure, it can be seen that vectorization can help to improve the performance by up to 26%. For the constant cache, we can coalesce 32 bytes into a single transaction at most. When the memory access width is larger than 32 bytes (e.g., float16), the constant cache is disabled by AOC.

In summary, it is important to consider the port pressure and to coalesce the accesses if possible when utilizing constant caches on FPGAs.

4.3 Local Memory

Local memory has slightly longer latency compared to LSU embedded caches and constant caches as shown in Table 1.

When there are multiple memory requests accessing local memory simultaneously, AOC will either replicate multiple copies of local memory or increase the number of local memory banks to improve the bandwidth. Thus, the actual local memory size would be larger than the size which programmers set in the kernel. Similar to constant caches, we also test the impact of simultaneous local memory accesses and local memory sizes on the performance. We present their performance impacts in Fig. 12 and Fig. 13 separately. From these two figures, we can see that the local memory performance is also significantly degraded when many memory requests access local memory simultaneously due to

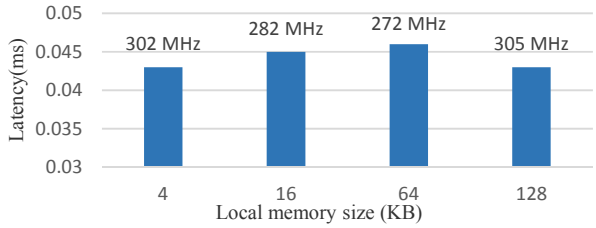

Fig. 13. Local memory latency and frequency with different memory sizes.

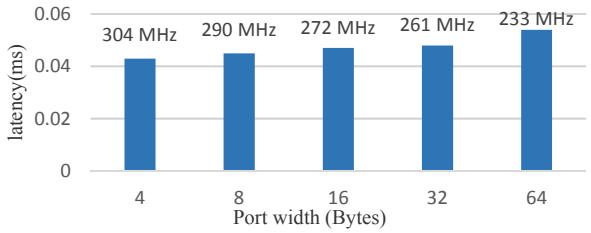

Fig. 14. Local memory latency and frequency with different port widths.

the increased interconnects complexity between the compute unit and local memory. And the local memory size does not affect the memory performance to a large degree.

4.3.1 Local Memory Access Vectorization

As shown above, when many memory requests access local memory simultaneously, the performance is degraded. Vectorizing local memory accesses will help to reduce the number of simultaneous local memory accesses. But when the local memory accesses are vectorized, the local memory port width will be increased, which will degrade the system frequency.

Fig. 14 shows the system frequency and local memory access latency with different port widths. From the figure we can see with the increase in local memory port width, the local memory access latency is increased and the system frequency drops significantly. Also, local memory access vectorization incurs additional control overhead sometimes. Take Fig. 9 as an example, control logic is needed to determine which fields of *int4 Sloacl* are required for computation when *KERNEL_SIZE* is 17. Thus for stencil kernels, we disable the local memory access vectorization.

4.4 Constant Cache vs. LSU Embedded Cache

In Section 3, we raise the question on whether the small-size and frequently-reused data in stencil codes should be put into constant caches or LSU caches. According to the properties of both memory types we can see that if the simultaneous memory access pressure is not high or the compiler cannot detect data reuse for the data, constant caches are the choice. In the other scenarios, LSU caches are more appropriate.

Take the code in Fig. 9 as an example, after applying the loop unrolling optimization, the memory access pressure for *Fmatrix* is high even with access vectorization. In this scenario, *Fmatrix* should be put into LSU caches to maintain high bandwidth and short latency. We conduct experiments to validate our analysis. Our experiment shows that for 1DConv kernel in the NDRange mode the performance of putting *Fmatrix* in LSU caches outperforms that of putting *Fmatrix* in constant caches with access vectorization by 9.5% under the configuration, *srcMatrix* size: 1024x1024 and *Fmatrix* size: 17x17. For single-task kernels, it is the same case since applying shift register pattern requires full loop unrolling, which favors the utilization of LSU caches over constant caches.

Table 2. Benchmarks and input data.

Application	Data Input
1D Convolution	1048576*1 Source Matrix 17*1 Filter Matrix
2D Convolution	1024*1024 Source Matrix 9*9 Filter Matrix
2D Jacobi Iteration	1024*1024 Matrix

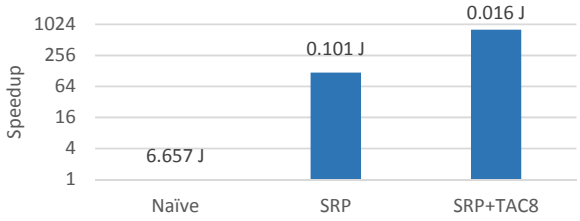

Fig. 15. Speedups and energy consumption of 1DConv kernels after applying optimizations in the Single-Task mode.

5 Experiments

5.1 Experiment Setup

All of our experiments were conducted on a Terasic's DE5-NET board, which includes 2-bank 4GB DDR3 device memory and an Altera Stratix V GX FPGA. The Altera OpenCL SDK v16.0 is used to compile the OpenCL code for FPGAs.

We choose three OpenCL programs of stencil codes for optimization to evaluate our tuning process. They are 1D convolution, 2D convolution and 2D Jacobi iteration algorithms, as shown in Table 2.

We implement both Single-Task and NDRange kernels for these benchmarks and then apply our proposed tuning processes[2]. We collect the kernels' energy statistics using a KILL A WATT EZ power meter [5].

5.2 1D Convolution

As discussed in Section 4.4, for stencil codes, LSU caches are better than constant caches for performance due to the large number of simultaneous memory accesses. Thus for 1D convolution we disable the constant cache utilization. As there are not enough hardware resources for CUR in the final stage, we do not include it in our results. LU in NDRange kernel in our experiments indicates full loop unrolling. It is the same case in 2D convolution.

For the single-task 1DConv kernel, Fig. 15 shows the speedup over the naïve baseline kernel and energy consumption after applying our Single-Task kernel tuning process. The bars in the figure indicate the speedup compared to the naïve baseline kernel while the numbers on top of the ba-

[2] Our source code and the experimental results are released as open source [17].

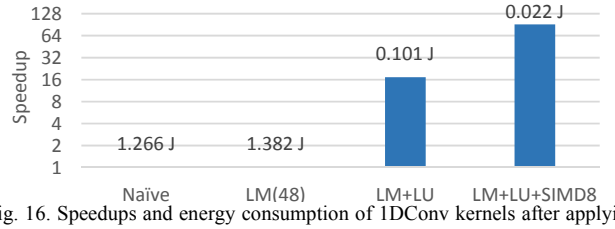

Fig. 16. Speedups and energy consumption of 1DConv kernels after applying optimizations in the NDRange mode.

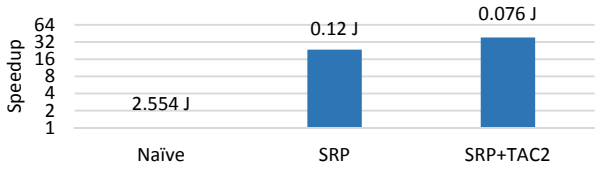

Fig. 17. Speedups and energy consumption of 2DConv kernels after applying optimizations in the Single-Task mode.

rs represent the energy consumption. The same convention is used for other figures in this section. From the figure it can be seen that the optimized kernel can achieve around 803.1× speedup and reduce the energy consumption significantly.

For the NDRange 1DConv kernel, Fig. 16 shows the speedup over the naïve baseline kernel and energy consumption for different optimizations in our tuning process. The optimized local memory size is 48*1 after careful tuning.

The optimized SIMD degree is 8 due to the hardware resource constraints. The local memory optimization alone does not improve the system performance because the 1DConv baseline kernel is computation bound. But when it is coupled with LU and SIMD, the optimized kernel can achieve 91.2× speedup and the energy consumption is reduced significantly.

For 1DConv, compared to the optimized NDRange kernel, the optimized Single-Task kernel has better performance, 0.518 ms vs 0.661 ms, and more energy efficient due to the high performance and energy-efficient shift registers.

5.3 2D Convolution

Fig. 17 and Fig. 18 present the speedup and energy consumption after applying the tuning process in the Single-Task and NDRange modes for 2DConv.

For the single-task 2DConv kernel, our optimized kernel achieve 38.4× speedup and reduces the energy consumption from 2.544 J to 0.076 J.

For the NDRange 2DConv kernel, the optimized local memory size is 32*32. The optimized SIMD degree is 2 due to the hardware resource limit. Similar to 1DConv, the local memory optimization alone does not improve the performance since the naïve baseline kernel is computation bound. Our optimized kernel can achieve 144× speedup and reduce the energy consumption drastically.

For 2DConv, the optimized NDRange kernel outperforms the optimized Single-Task kernel (1.994 ms vs 2.363 ms). The reason is that for 2DConv after applying SRP for the Single-Task kernel there are still nested loops, which make it hard for AOC to extract the parallelism effectively. Also, NDRange ke-

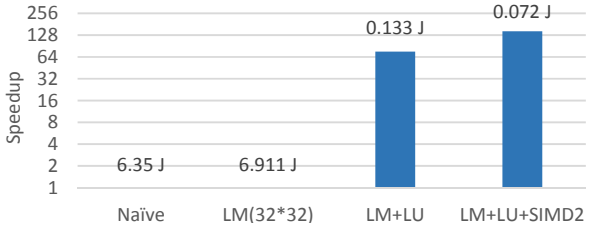

Fig. 18. Speedups and energy consumption of 2DConv kernels after applying optimizations in the NDRange mode.

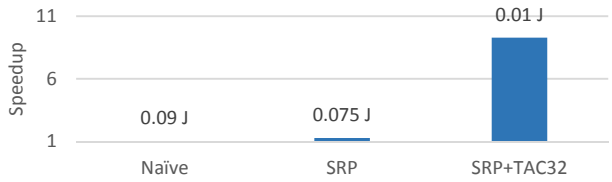

Fig. 19. Speedup and energy consumption of 2D Jacobi Iteration kernels after applying optimizations in the Single-Task mode.

rnels consume less energy than Single-Task ones due to their shorter execution time while Single-Task kernels have lower power due to the more power-efficient shift registers.

5.4 2D Jacobi Iteration

As there are no loops in the NDRange kernel for 2D Jacobi iteration algorithm, we disable the LU optimization in the NDRange mode. Fig. 19 and Fig. 20 present the speedup and energy consumption after applying the tuning process in the Single-Task and NDRange modes.

For the single-task 2D Jacobi kernel, compared to the naïve baseline kernel our optimized kernel can achieve 9.3× speedup and reduce the energy consumption from 0.09 J to 0.01 J.

For the NDRange 2D Jacobi kernel, the optimized local memory size is 18x18. After applying the tuning process, the optimized kernel can achieve 5.2× speedup and reduce the energy consumption from 0.077 J to 0.023 J. It is worth noting that for 2D Jacobi iteration, the optimized SIMD factor is 4. When we increase the SIMD factor to 16, the performance degrades significantly due to the significant drop in the system frequency. The reason is that when the SIMD factor is large, the simultaneous access pressure on local memory is very high which will lead to a significant drop in frequency, as discussed in Section 4. The CU degree is 8 due to the hardware resources constraints.

For 2D Jacobi Iteration, compared with the optimized NDRange kernel, the optimized Single-Task kernel has better performance (0.509 ms vs 0.818 ms) and is more energy efficient.

5.5 Comparision with Altera Design Examples

In this section, we compare our optimized kernels with the two OpenCL kernel design examples, Sobel Filter (SOBEL) and Time-Domain FIR Filter (TDFIR), provided by Altera.

In both design examples, the kernels are implemented in the Single-Task mode and optimized by applying SRP. We further optimize the kernels using our proposed tuning processes. In the Single-Task mode, the optimized kernel for

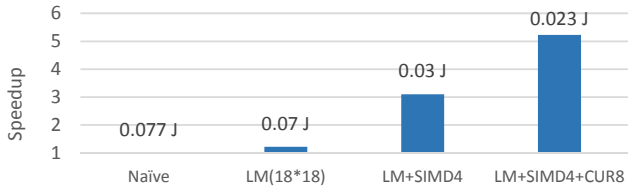

Fig. 20. Speedups and energy consumption of 2D Jacobi Iteration kernel after applying optimizations in the NDRange mode.

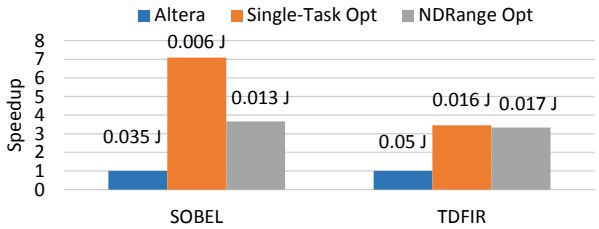

Fig. 21. Speedups and energy consumption of SOBEL and TDFIR compared with Altera design examples.

SOBEL is SRP+TAC64 (i.e., shift register optimization and task coarsening with a factor of 64), and the optimized kernel for TDFIR is SRP+TAC4. In the NDRange mode, the optimized kernel for SOBEL is LM(34*34)+LU+SIMD32 (i.e., local memory size of 34x34, loop unrolling and thread coarsening with a factor of 32), and the optimized kernel for TDFIR is LM(160)+LU+SIMD4. Fig. 21 shows the speedup and energy consumption of our optimized kernels compared with the kernels provided by Altera. From the figure we can see our optimized Singe-Task kernels can achieve a speedup 7.08× and 3.45× for SOBEL and TDFIR, respectively. And the optimized NDRange kernels can achieve a speedup 3.66× and 3.33× for SOBEL and TDFIR, respectively.

6 Related Work

As OpenCL for FPGAs is recently supported, there are few prior works on optimizing the OpenCL code for FGPAs. Wang et al. proposed a performance analysis framework to help to optimize OpenCL codes on FPGAs [16]. Their framework gives a performance estimation of the kernels and provides feedbacks on the current bottlenecks in kernels. There are two aspects that differentiate our work from this performance analysis framework. First, they did not consider constant caches and LSU embedded caches. The memory architecture optimization considered in the framework is only the local memory utilization. Second, this performance framework targets only at NDRange kernel optimization while our work includes optimization processes for both Single-Task kernels and NDRange kernels.

7 Conclusions and Future Work

This paper proposes tuning processes for OpenCL stencil codes on FPGAs in both the single-task mode and NDRange mode. We also characterize different types of memory resources on FPGAs. The experiment results for our benchmarks show the output of our tuning processes can achieve up to two orders of speedup compared to the naïve baseline kernels. Also, our optimized code achieves much better performance compared to the related design examples provided by Altera.

As our future work, we plan to generalize our performance tuning processes for a wide range of applications and aim to automate the optimization flow.

References

[1] Altera. Implementing fpga design with the opencl standard. Altera Whitepaper, 2011.

[2] Altera. Altera SDK Programmer guide, 2014..

[3] Altera. Altera SDK for Optimization guide, 2014.

[4] Datta, Kaushik, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures." In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, p. 4. IEEE Press, 2008.

[5] KILL A WATT EZ. P4460 KILL A WATT EZ power meter operation manual. http://www.p3international.com/manuals/p4460_manual.pdf.

[6] R. Fu, Haohuan, Robert G. Clapp, Oskar Mencer, and Oliver Pell. "Accelerating 3D convolution using streaming architectures on FPGAs." In *2009 SEG Annual Meeting*. Society of Exploration Geophysicists, 2009.

[7] Herbordt, Martin C., Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug DiSabello. "Achieving high performance with FPGA-based computing." *Computer* 40, no. 3 (2007): 50.

[8] Holewinski, Justin, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. "High-performance code generation for stencil computations on GPU architectures." In *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 311-320. ACM, 2012.

[9] Kim, Lok-Won, Sameh Asaad, and Ralph Linsker. "A fully pipelined fpga architecture of a factored restricted boltzmann machine artificial neural network." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7, no. 1 (2014): 5.

[10] Li, Yanhua, Youhui Zhang, Jianfeng Yang, Wayne Luk, Guangwen Yang, and Weimin Zheng. "An approach of processor core customization for stencil computation." In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pp. 182-183. IEEE, 2014.

[11] Maruyama, Naoya, and Takayuki Aoki. "Optimizing stencil computations for NVIDIA Kepler GPUs." In *Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna*, pp. 89-95. 2014.

[12] Mueller, Rene, and Jens Teubner. "FPGA: what's in it for a database?." In*Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 999-1004. ACM, 2009.

[13] Schäfer, Andreas, and Dietmar Fey. "High performance stencil code algorithms for GPGPUs." *Procedia Computer Science* 4 (2011): 2027-2036.

[14] Shafiq, Muhammad, Miquel Pericas, Raul De la Cruz, Mauricio Araya-Polo, Nacho Navarro, and Eduard Ayguadé. "Exploiting memory customization in FPGA for 3D stencil computations." In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 38-45. IEEE, 2009.

[15] Volkov, Vasily, and James W. Demmel. "Benchmarking GPUs to tune dense linear algebra." In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1-11. IEEE, 2008.

[16] Wang, Zeke, Bingsheng He, Wei Zhang, and Shunning Jiang. "A performance analysis framework for optimizing OpenCL applications on FPGAs." In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 114-125. IEEE, 2016.

[17] https://github.com/chrisjia6412/stencil-codes-tuning.