Design Evaluation of OpenCL Compiler Framework for Coarse-Grained Reconfigurable Arrays

Hee-Seok Kim [#1], Minwook Ahn [*2], John A. Stratton [#3] and Wen-mei W. Hwu [#4]

[#] *Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA*
[1] kim868@illinois.edu
[3] stratton@illinois.edu
[4] w-hwu@illinois.edu

[*] *Samsung Advanced Institute of Technology, San 14-1, Nongseo-dong, Giheung-gu, Yongin-si, Geyonggi-do, Korea*
[2] minwook.ahn@samsung.com

***Abstract*—OpenCL is undoubtedly becoming one of the most popular parallel programming languages as it provides a standardized and portable programming model. However, adopting OpenCL for Coarse-Grained Reconfigurable Arrays (CGRA) is challenging due to divergent architecture capability compared to GPUs. In particular, CGRAs are designed to accelerate loop execution by software pipelining on a grid of functional units exploiting instruction-level parallelism. This is vastly different from a GPU in that it executes data parallel kernels using a large number of parallel threads. Therefore, an OpenCL compiler and runtime for CGRAs must map the threaded parallel programming model to a loop-parallel execution model so that the architecture can best utilize its resources.**

In this paper, we propose and evaluate a design for an OpenCL compiler framework for CGRAs. The proposed design is composed of a *serializer* and post optimizer. The serializer transforms parallel execution of work-items to an equivalent loop-based iterative execution in order to avoid expensive multithreading on CGRAs. The resulting code is further optimized by the post optimizer to maximize the coverage of software-pipelinable innermost loops. In order to achieve the goal, various loop-level optimizations can take place in the post optimizer using the loops introduced by the serializer for iterative execution of OpenCL kernels. We provide an analysis of the propose framework from a set of well-studied standard OpenCL kernels by comparing performance of various implementations of benchmarks.

***Index Terms*—OpenCL, GPU, Coarse-Grained Reconfigurable Arrays, CGRA, Samsung Reconfigurable Processor, SRP, RP**

I. Introduction

OpenCL has emerged in an effort to standardize both platform and programming model for heterogeneous systems [1]. It defines a rich set of specifications ranging from platform to programming model. Those specifications are abstract and meant to encapsulate actual implementation details. Many vendors are adopting OpenCL as the primary software platform for their system, as it embraces a dominating design for many systems comprised of a host processor and accelerators.

OpenCL provides portability of application programs. It allows running programs written in OpenCL over different architectures. Traditionally, architectures with specific functionality, denoted as *accelerators*, require programmers to write in a dedicated way to make full use of the functionality. Often times it complicates the programming interface and migrating optimized programs for an accelerator to other architecture becomes a nontrivial task. OpenCL therefore greatly increases the usability of accelerators by providing language standardization and application portability.

However, running an OpenCL program is challenging, because accelerator architectures are very diverse. OpenCL programming model is largely inspired by GPUs and designed to best utilize such architectures. The architectures that lack hardware mechanisms for deeply engrained OpenCL programming patterns would suffer performance degradation from having to pursue costly alternative approach to mimic such features. For example, being able to create and run a thousands of threads simultaneously would be unrealistic on a scalar processor or even commodity multicore CPU systems. Another major problem is distinct memory spaces of OpenCL which makes it hard to natively map the memory hierarchy to other architectures. Therefore, the challenge of implementing an OpenCL compiler and runtime for such architectures is mapping OpenCL language constructs to the capabilities of the hardware for best utilization.

In this paper, we evaluate the design of an OpenCL compiler framework for CGRAs. We adapt the serialization technique [2][3] for kernel execution which transforms parallel execution of OpenCL programs into equivalent loop-based execution. Serialization is followed by a post optimizer to perform loop-level optimizations specific to CGRAs. We verify the design by transforming and running a selected OpenCL benchmarks on Samsung Reconfigurable Processor(SRP) [4], a processor implementing CGRA architecture.

The rest of this paper is organized as follows. Section II summarizes architectural difference between GPU and SRP for further discussion. Section III describes the design of the OpenCL compiler framework for CGRAs. Section IV evaluates the performance of the transformation and discuss performance portability using SRP. Related works follow in

978-1-4673-2845-6/12/$31.00 © 2012 IEEE

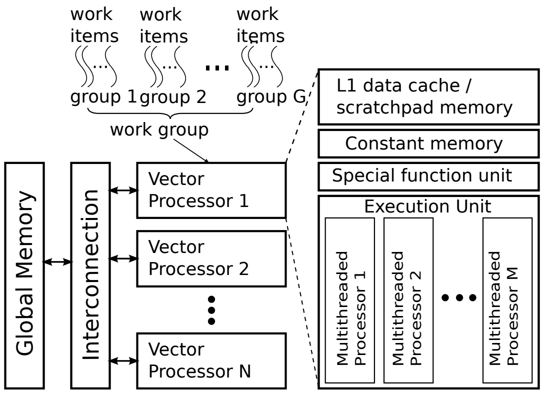

Fig. 1. Block diagram of GPU architecture

Section V. Finally, Section VI summarizes.

II. Comparison of GPU and SRP

This section briefly highlights differences of GPU and SRP in the aspects of architecture and programming model.

A. GPU

Figure 1 illustrates block diagram of GPU architecture. Modern GPU architecture is comprised of vector processors, each of which is capable of running programs independent from each other. The vector processor executes a group of a fixed number of threads in lockstep. The vector processor has hardware contexts for many such lockstep groups for time sharing execution resources between them. In this way, the vector processor can execute up to hundreds of threads in parallel. The GPU architecture has several advantages. First, hardware multithreading allows hiding long latency of some operations such as global memory access. Second, threads within a same logical group can be synchronized efficiently.

A GPU has explicitly differentiated memory spaces to take advantage of special hardware memory resources. A GPU thread has its own private registers. A group of threads under the same logical group share a fast on-chip scratchpad memory bound to a vector processor. Global memory is accessible to all threads. There is constant memory which is small but fast read-only memory. Texture memory is cached read-only global memory combined with a fixed set of computation to support convenient programming model mainly for graphics applications.

OpenCL programming model is designed to fully exploit the hardware resources of GPU. A *kernel* is a function and it is a venue for programmers to write data parallel program. *Work-items* are execution instances of a kernel and are grouped by *work-groups*. Execution of work-items in a work-group can be synchronized at a program location using *barrier*. The memory hierarchy is also exposed to programmers.

The *host program* runs on the host processor and submits commands to execute an OpenCL kernel or manipulate the memory of a GPU. The *command queue* interfaces both host processor and device. On kernel launch, the host processor delivers all code and data to GPU, and receives the result from the global memory.

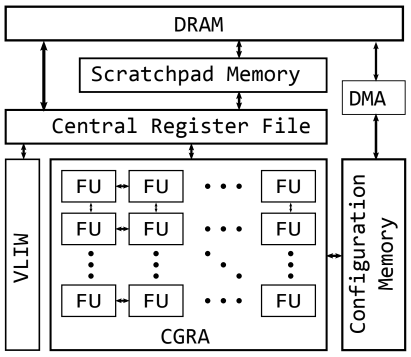

Fig. 2. Block diagram of SRP architecture

B. SRP

SRP is a traditional DSP processor architecture for various multimedia applications without the support of GPU-style multithreading. In order to exploit the instruction-level and loop-level parallelism embedding inside multimedia application, SRP has an accelerator called Coarse Grained Reconfigurable Architecture (CGRA). The CGRA is composed of an array of processing elements (PEs) such as functional units (FUs) and register files (RFs). These PEs are connected to each other by dedicated connection wires. The CGRA also has a dedicated memory for reconfiguring itself called *configuration memory*. By changing the content of the configuration memory, the CGRA can reconfigure itself for different kernels in multimedia applications. The configuration memory can host multiple loops simultaneously as long as their overall size is smaller than the capacity of the configuration memory. The kernels executed on the CGRA must be loops that can be modulo-scheduled [5]. All parts of the SRP code except the accelerated loops are executed on a separate VLIW processor. These code parts contain the instructions for the application control such as branch and jump and prepare the data necessary for the execution of the loops in the CGRA. In order to avoid the data copy for delivering program context from the VLIW to the CGRA or vice versa, the central register file is shared by the VLIW and the CGRA. Due to this, the execution handover from the VLIW to the CGRA or vice versa takes only a few cycles.

SRP has a simple but efficient two level memory hierarchy. Instead of data cache, SRP has a scratchpad memory composed of multiple banks. As the access latency in the scratchpad memory is much shorter than in DRAM, it is usual to preload the necessary data in off-chip DRAM into the scratchpad memory by using direct memory access (DMA).

Programming for the CGRA is implicit. The only thing for the application programmers to do is to select the loops that they want to accelerate on the CGRA. They can do it by adding a pragma right before the loops. Then the compiler for SRP automatically builds a configuration of the CGRA for the loop by modulo-scheduling algorithm [6]. In the modulo-scheduling algorithm for the CGRA, the compiler tries to use not only instruction-level parallelism but also loop-level parallelism by overlapping several loop iterations at the same time. Generally an innermost natural loop is a good candidate for modulo-scheduling. Even though the compiler can find the configuration of good performance in many cases, it is advisable to fine-tune the loop for modulo-scheduling. The performance of the modulo-scheduled loop is mainly influenced by two factors: resource and recurrence constraints [5]. In order to increase the performance of the loop on the CGRA, the application programmers have to make their loop with the minimum recurrence if possible. They also should fit the maximum instantaneous resource usage in the loop to the available resources of the CGRA. Even though if-conversion by the compiler can transform the control flow inside the loop into data dependence, it is preferred to move the control flow out of the loop.

III. OpenCL Compiler Framework for CGRA

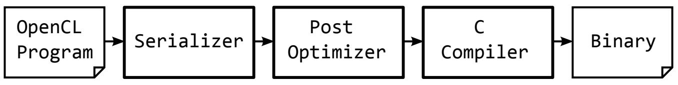

Fig. 3. OpenCL Compiler Framework for CGRA

In this section, the proposed design of OpenCL compiler framework is described for CGRAs, mainly targeting SRP. It is composed of serializer and post-optimizer, in addition to the standard C Compiler. *Serializer* is a OpenCL-to-C translator. It transforms data parallel OpenCL kernel into an equivalent code such that the resultant code does not rely on expensive multithreading on CGRAs. The post optimizer takes over the serialized kernel and performs specific optimizations for CGRAs. Figure 3 shows the block diagram of the framework.

A. *Serializer*

```
for each work group,
    invoke serialized kernel function;
```

(a) OpenCL driver

```
// kernel body is divided into body_N at barrier
void serialized_kernel(...) {
  for each work item in a work group,
    body_1;
  for each work item in a work group,
    body_2;
  ...
}
```

(b) Serialized OpenCL kernel

Fig. 4. OpenCL driver architecture for serialization

When it comes to executing work-items on a CGRA, parallel execution is not practical as managing concurrent threads on a CGRA is expensive. A strategy of mapping a thread per work-item found in GPUs simply takes too much memory and scheduling overhead to a CGRA.

To address the problem, we adapt a technique proposed in [2] and [3], which we call *serialization*, in order to make

```
__kernel void vecAdd(__global float* C,
    __global float* A, __global float* B) {
  int idx = get_global_id(0);
  C[idx] = A[idx] + B[idx];
}
```

(a) OpenCL kernel

```
void serial_vecAdd(float* C, float* A, float* B,
  int global_id_0_begin, int local_sz_0) {
  for (int lx0 = 0; lx0 < local_sz_0; lx0++) {
    int get_global_id_0 = lx0 + global_id_0_begin;
    int idx = get_global_id_0;
    C[idx] = A[idx] + B[idx];
  }
}
```

(b) Serialized kernel

Fig. 5. Serialization of vector addition

```
__kernel void transpose(
    __global float *odata, __global float *idata) {
  __local float block[BLKDIM * BLKDIM];
  int xIndex = get_global_id(0);
  int yIndex = get_global_id(1);

  int index_in = yIndex * width + xIndex;
  block[get_local_id(1)*BLKDIM+get_local_id(0)] =
    idata[index_in];

  barrier(CLK_LOCAL_MEM_FENCE);

  xIndex = get_group_id(1) * BLKDIM + get_local_id(0);
  yIndex = get_group_id(0) * BLKDIM + get_local_id(1);
  int index_out = yIndex * height + xIndex;
  odata[index_out] =
    block[get_local_id(0)*BLKDIM+get_local_id(1)];
}
```

(a) OpenCL kernel

```
void serial_transpose(float *odata, float *idata,
  int gid0_begin, int lsz0, int grid0,
  int gid1_begin, int lsz1, int grid1) {
  /* __local */ float block[BLKDIM * BLKDIM];

  for (int lx1 = 0; lx1 < lsz1; lx1++) {
    int gid1 = lx1 + gid1_begin;
    for (int lx0 = 0; lx0 < lsz0; lx0++) {
      int gid0 = lx0 + gid0_begin;
      int xIndex = gid0;
      int yIndex = gid1;
      int index_in = yIndex * width + xIndex;
      block[lx1*BLKDIM+lx0] = idata[index_in];
  } }
  // Implicit barrier
  for (int lx1 = 0; lx1 < lsz1; lx1++) {
    for (int lx0 = 0; lx0 < lsz0; lx0++) {
      xIndex = grid1 * BLKDIM + lx0;
      yIndex = grid0 * BLKDIM + lx1;
      int index_out = yIndex * height + xIndex;
      odata[index_out] = block[lx0*BLKDIM+lx1];
} } }
```

(b) Serialized kernel

Fig. 6. Serialization of matrix transpose

OpenCL kernel execution suitable for a CGRA. As a result of this process, it produces *serialized kernel*.

The serialization converts parallel execution of work-items into equivalent loop-based iterative execution. The serialization introduces *serialization loop* around a set of statements of kernel code. An iteration of the loop embodies a logical thread for a work-item. The index space of the serialization loop reflects the size of a work-group.

When a kernel code wrapped by the serialization loop contains barriers, it will be split into multiple sets of statements divided by barriers. This process is called *deep fission* according to [2]. As a result, all statements are wrapped in serialization loops such that no serialization loop contains a barrier.

Since the logical threads no longer exist independently, the translated program has to emulate private storage for work-items within a work-group. By creating an array of values of the size of a work-group for each local variable and indexing them by the serialization loop index, it fully emulates the private storage of work-items. Shared variables among work-items per work-group remain unreplicated. The implicit barrier by the serialization loop and the replicated private storage for work-items completely implement the semantic of barrier for work-items.

Memory hierarchy mapping can be done in this stage. Local memory and constant memory can be emulated by preallocating a block of memory in scratchpad memory. The compiler flattens accesses to both memories accordingly. Global memory will be mapped to DRAM. Due to a long latency to access DRAM, proactive data prefetching and overlaying over scratchpad memory is required, as similarly shown in [3]. Unlike [3], CGRAs do not provide a virtual memory layer. Therefore, a new memory management algorithm must be developed. Texture memory is a unique feature of GPUs as it is combined with hardware interpolation. Software-based emulation can be used, however, it is not pragmatic due to high cost. This is why texture memory in OpenCL is optional for non-GPU architectures to implement. We will not be discussing such support for CGRAs here.

Figure 4 shows the architecture of an OpenCL driver with a consideration of serialization. As there is no dependency between execution of work-groups, iterative execution of work-group in turn at OpenCL driver level is desired. Upon a work-group execution, it calls the serialized kernel with proper work-group indices.

Figure 5 shows serialization for vector addition. The OpenCL code does not have barriers and simply casting the serialization finishes the process. Getting global index is replaced with an expression based on the serialization loop index. The resultant code enables software pipelining and can be accelerated on a CGRA.

Matrix transpose shows how the deep fission works in the presence of barrier, as shown in Figure 6. Two code bodies before and after the barrier become units of serialization. Execution of innermost loops of the two bodies can be accelerated on a CGRA.

B. Post Optimizer

We have identified that additional optimizations must be developed in pursuit of maximum performance after serialization. In particular, the strength of CGRAs is the loop acceleration where it exploits wide instruction-level parallelism from an aggressive software pipelining. Since OpenCL kernels are generally regarded as the most performance demanding part of a program, program execution must remain in a CGRA as long as possible when it runs OpenCL kernels. Therefore, optimizations maximizing the coverage of software pipelinable loops should be followed.

The post optimizer takes over the resulting code from the serializer and performs specific optimizations for CGRAs. It is a venue where significant engineering effort of implementing compiler optimizations can take place. While a compiler in general can not identify performance critical region in a program, it is clear that OpenCL programs require intensive optimizations. Thus, the existence of post optimizer clearly contributes to the design of compiler framework for CGRAs.

A closer look at the resultant code reveals that the serialization loop is now fully exposed to the compiler along with the kernel code. The exposed serialization loop is often the innermost loop of the transformed kernel code, making it an excellent target for mapping to a CGRA. For example, the vector addition code is identified as a target for a CGRA after serialization, as depicted in Figure 5 (b).

Serialization loops bring useful properties that the compiler can take advantage of. First, the serialization loops are canonical loops. Second, the serialization loops are natural loops in that they have single entry and single exit. Third, execution of the serialization loops does not carry data dependence over its iterations. Lack of data dependencies is true by the assertion given the nature of the input OpenCL code. Such properties are extremely valuable for the post optimizer to further optimize the code with loop-level transformations. In this section, several useful optimizations utilizing the properties are addressed.

1) Resource Utilization Optimization: High degree of instruction-level parallelism can be achieved from a successful software pipelining of a loop on a CGRA. In software pipelining[7], the total execution cycle of a loop, denoted as T, can be calculated from Eq. 1 as shown below:

$$T = (N - 1 + S) \times II, \tag{1}$$

where N is the trip count of the loop, II is initiate interval and S is stage count. II and S dictate the performance of the loop execution. Both resource and recurrence constraints play a key role for compiler in determining II. Among them, the compiler can ignore recurrence constraint according to a property of serialization loops. Therefore, the performance depends on the resource constraint.

While smaller II implies better performance in general, such low II can be caused by too few operations to schedule, resulting in many unused FUs. Resource utilization, denoted as R, is a metric to measure the efficiency of hardware for a given task as defined in Eq. 2:

TABLE I
II, RESOURCE UTILIZATION AND PERFORMANCE OVER DIFFERENT UNROLLING FACTORS

Unrolling factor	II	Resource utilization	Performance (cycles)
1	5	0.11	5156
2	5	0.17	2608
4	5	0.27	1348
8	8	0.29	1088
16	14	0.30	964

$$R = \frac{M}{II \times W}, \quad (2)$$

where M is the number of operations of the loop, and W is the number of FUs of the reconfigurable grid. The importance of realizing smaller II is stressed here again in pursuit of better resource utilization. It implies a compiler should schedule as many operations as possible under the same II envelop.

Unrolling a loop with low resource utilization is a valuable optimization along with serialization. As previously mentioned, unrolling does not change recurrence constraints due to the inherent properties of OpenCL kernels. Therefore, a compiler can safely unroll a loop until full resource utilization is obtained.

Table I shows trends of II, resource utilization and performance for the serialized vector addition kernel shown in Figure 5 (b) by changing unrolling factor from two to sixteen. The loop is software pipelined over a CGRA of 4x4 FUs. Performance is measured on a cycle-accurate simulator assuming a perfect memory system. The latency of load operation is set to four cycles. The rate of increase of II is far lower than that of the unrolling factor, and it manifests multi-factor speedup. Beyond a point where the performance saturates, eight in this particular case, larger unrolling factor saturates resource utilization and begins to increase II proportionally.

2) Serialization Loop Flattening: Work-items in OpenCL have indices with up to three dimensions. As such, serialization loops are formed as triply nested loops, as illustrated in Figure 7 (a). Therefore, software pipelining can not be done for the outer loops, deferring processing of them to the control processor. As a consequence, branch and arithmetic operations from the outer loops will contribute to the execution cycles. It will also have to tolerate overheads due to switching execution mode between the control processor and the CGRA.

The nested serialization loops can be transformed into a single flattened loop [8]. A single loop reduces branch overhead from a nested loops. In a case where the flattened loop is innermost loop, the resulting code can run on a CGRA for much longer cycles from extended loop trip count. It also removes execution mode switching overhead.

Flattening the serialization loops is straightforward as they are canonical loops and natural loops at the same time. Index calculation from the flattened loop is implemented in a simple arithmetic. Figure 7 (b) shows the transformation example.

SRP's CGRA does not support integer division and modular operations by hardware. The compiler instead replaces

```
void kernel_func_3D(...) {
  for (z = 0; z < nz; z++) {
    for (y = 0; y < ny; y++) {
      for (x = 0; x < nx; x++) {
        ...
} } } }
```

(a)

```
void kernel_func_3D(...) {
  for (zyx = 0; zyx < nz * ny * nx; zyx++) {
    z = zyx / (ny * nx);
    y = (zyx / nx) % ny;
    x = zyx % nx;
    ...
} }
```

(b)

```
// ny = (1 << ly), nx = (1 << lx)
void kernel_func_3D(...) {
  for (zyx = 0; zyx < nz * ny * nx; zyx++) {
    z = zyx >> (ly + lx);
    y = (zyx >> lx) & (ny - 1);
    x = zyx & (nx - 1);
    ...
} }
```

(c)

Fig. 7. Serialization loop and flattening. (a) serialization loops in a form of a triply nested loops (b) flattened serialization loops (c) flattened serialization loops using bit operations for index calculations

them with equivalent software implementations, which in turn disqualifies the flattened loop for software pipelining.

In OpenCL, configuring a power of two number of work-items per work-group is a common practice [9]. This is because GPU hardware is designed to allocate resources in power-of-two units. If the programmer leaves the group size unspecified, then the OpenCL driver can choose a number of work-items per work-group. For the SRP architecture, a heuristic choosing a power-of-two group size would be beneficial.

Under the condition where the loop trip counts are a power of two, flattening becomes available with efficient bit operations, as shown in Figure 7 (c). In this particular case, the compiler can substitute the integer division and modular operations with equivalent bit operations for the index calculation. This method can further be extended to address arbitrary trip counts by making the trip count power of two that is equal to or larger than the actual trip count. The loop body is guarded with a predicate from comparing the loop index to the actual trip count. The conditional statement should be successfully if-converted by the compiler for software pipelining.

3) Serialization Loop Fission: When the original kernel code is imperfectly nested by the serialization loops, it effectively prevents the resulting code from being mapped to a CGRA. This is caused when the kernel code itself contains loops, which we call *kernel loops*, and they have sets of statements to execute either before and after them, such as Figure 8 (a). Because the kernel loop is the innermost loop, it alone will be considered for mapping onto a CGRA, forcing the leading and trailing blocks to execute on the slower control processor. By breaking the kernel code at the boundaries of

```
__kernel void fn(...) {
  ...   // init
  for (i = 0; i < N; i++) {
    ...
  }
  ...   // finish
}
```

(a)

```
void serial_fn(...) {
  for (x = 0; x < nx; x++) {
    ...   // init
    for (i = 0; i < N; i++) {
      ...
    }
    ...   // finish
} }
```

(b)

```
void serial_fn(...) {
  for (x = 0; x < nx; x++) { ... // init }
  for (i = 0; i < N; i++) {
    for (x = 0; x < nx; x++) {
      ...
  } }
  for (x = 0; x < nx; x++) { ... // finish }
}
```

(c)

Fig. 8. An Example of Serialization Loop Fission (a) OpenCL kernel (b) Serialized kernel (c) Serialized kernel after loop fission

kernel loops, the leading or trailing code blocks of a kernel loop become loop bodies of serialization loops, and available for mapping on a CGRA. Breaking the serialization loops is safe because the original OpenCL work-items have no execution dependencies by assertion.

Figure 8 (c) shows an example of loop fission for serialization loops. It contains initialization, an innermost loop and a termination part. After the fission, the initialization and termination can run on a CGRA as they are identified as software pipelinable. Thus, the transformation enlarges the coverage of a CGRA execution of the OpenCL kernel. Note that the post optimizer could further apply additional transformations for kernel loops containing serialization loops. For instance, loop interchange or flattening could be applied to the nested loop in Figure 8 (c), transforming into a perfectly nested loop.

4) SIMDization: SRP's CGRA supports subword parallelism via SIMD intrinsic instructions for a selected set of operations. For SIMD instructions, a 32-bit register can be divided into 2 of 16 bits or 4 of 8 bits. The subword parallelism is especially useful for graphics applications where primitive information is stored in 8 or 16 bits.

SRP's subword parallelism can be used in two situations. First, direct translation of a group of built-in vector data types of OpenCL becomes available. OpenCL supports subword vectors of 8-bit or 16-bit, where supported sizes are 2, 3, 4, 8 and 16. Considering 8-bit subword, char***n*** and uchar***n***, one 4x8bit SIMD operation can replace 4 scalar equivalents when ***n*** is equal to or smaller than 4. For larger ***n***, more than one SIMD operations can jointly be utilized. Second, OpenCL programs using subword scalar data types can be vectorized at the level of serialization loops. The loop-level vectorization requires that data dependency is shorter than the vector length and the loop needs to be innermost. Both can be guaranteed by properties of the serialization loops. The loop is strip-mined by the vector length, two or four in this case, and then each scalar instruction within the loop body is replaced with the corresponding SIMD operation. Figure 9 demonstrates two

```
// OpenCL code
__kernel void fn(__global uchar4* c,
    __global uchar4* a, __global uchar4* b) {
  c[idx] = a[idx] + b[idx];
}

// Serialized kernel code
void serial_fn(srp_uchar4* c,
    srp_uchar4* a, srp_uchar4* b) {
  for (x = 0; x < wgs; x++) {
    c[idx] = _I_intr003_rg_addb(a[idx], b[idx]);
} }
```

(a) Lowering OpenCL vector type

```
// OpenCL code
__kernel void fn(__global uchar* c,
    __global uchar* a, __global uchar* b) {
  c[idx] = a[idx] + b[idx];
}

// Serialized kernel code
void serial_fn(uchar* c, uchar* a, uchar* b) {
  for (x = 0; x < wgs; x+=4) {
    c[idx] = _I_intr003_rg_addb(a[idx], b[idx]);
} }
```

(b) Vectorization at the serialization loop-level

Fig. 9. Two examples of SIMDization for SRP

examples of the transformation. Note that the usage of SIMD intrinsics, as shown in the Figure 9, is adapted from the intrinsic model of the IMPACT compiler [10].

IV. Evaluation

In this section, we evaluate the effectiveness of the proposed compiler framework with a selected OpenCL benchmarks. All experiment results are done using a cycle-accurate simulator for SRP. The simulator assumes all data reside in on-chip scratchpad memory and as such the compiler assigns a uniform latency to all load operations. We also assume the configuration memory preloads all kernels so that no additional costs are added other than a few cycles of the execution handover overhead when reconfiguration happens. The latency of load operation is set to four cycles. The architecture is configured as 2-way VLIW and CGRA with 4x4 FUs.

We used four benchmarks to evaluate the relation of portability and performance. They are vector addition(vecadd), matrix multiplication(mm), matrix transpose(transpose) and reduction. Also, we implemented five versions for each benchmark for comparison. They are unoptimized C code, innermost accelerated of the unoptimized C code, fully hand-optimized C code, OpenCL code with serialization and OpenCL code with serialization and post optimization. For demonstration purpose, we did not use floating point operations as availability of floating point units varies across different configurations of SRP architecture. OpenCL code is assumed to have gone through the compiler pipeline as described in the framework shown in Fig 3. The unoptimized C code in the most simplest form is used as a portable baseline throughout the experiment.

Figure 10 illustrates speedups of various implementations of the benchmarks. For vector addition, the performance of

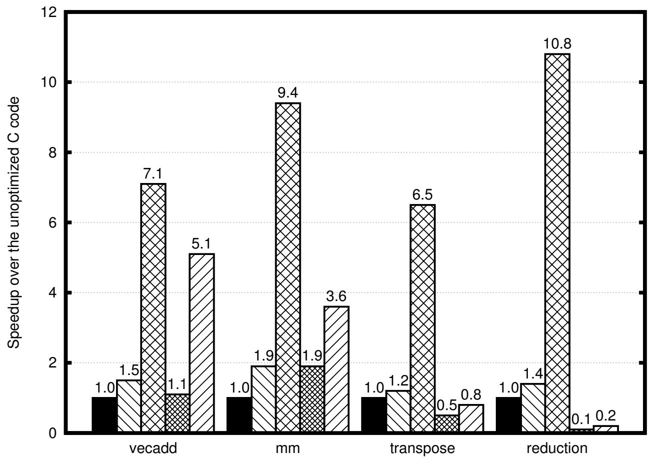

Fig. 10. Experimental results for benchmarks. Five bars for a benchmark represent speedups of 1) the baseline, 2) the baseline with innermost acceleration, 3) fully hand-optimized C, 4) OpenCL kernel with serialization and 5) OpenCL kernel with serialization and post optimization, respectively. The baseline is the unoptimized C code.

OpenCL code is approaching to the hand-optimized version, showing up to 5.1x speedup. The OpenCL code is optimized from unrolling the serialization loop eight times and mapped to the CGRA. In this simple kernel code, overhead compared to the hand-optimized one is attributed to the iterative execution of work-groups where the trip count of innermost loop execution is bounded within dimension of a work-group, leading to a frequent mode switching and execution in the VLIW.

The matrix multiplication demonstrates 3.6x speedup of the baseline, though it is slower than the hand-optimized version by 2.6x. The OpenCL code is optimized from loop flattening of the two nested serialization loops. Then the code is further split into three groups by loop fission, in that the first group sets global ids, the second group performs a dot product and the third group stores the results. The first and third group run in the CGRA. The second group is untouched and only the innermost loop runs in the CGRA. This is not as efficient as the hand-optimized version, where it flattens and unrolls loops to form one innermost loop. We expect the gap would be narrowed as the post optimizer extends its scope of optimizations beyond serialization loops so it can generate code similar to the counterpart.

The OpenCL code of matrix transpose shows 20% slowdown. The OpenCL code is optimized for GPUs where local memory tiling allows significant better memory performance, which is not the case for SRP. From the serialization stage, local memory is lowered to a preallocated memory block in scratchpad memory and it becomes sole burden to the performance. It also introduces a barrier which limits the duration of CGRA execution. On the contrary, the baseline as well as the hand-optimized versions are implemented as a single innermost loop, loading and storing an item of an array using a different index. It exemplifies a challenge using an optimized OpenCL code for portability.

Reduction is a particularly interesting case as the performance gap between the hand-optimized C and OpenCL codes is the most significant. The OpenCL implementation of reduction uses a tree-shaped reduction over local memory, which is an algorithm well suited to a GPU. The code is written assuming that parallel threads execute in lockstep, and that barriers cost no more than any other single instruction. Serializing such code results in a loop for every level of the reduction tree. As the performance results show, the barriers are far from free, and the single-loop implementation used by the baseline implementation is much more effectively software pipelined. This also shows why reduction is usually implemented as a library function by vendors [11].

V. Related Works

The advent of programmable GPUs and its popularity ignited development of several parallel programming languages. CUDA [12] is a data parallel programming model based on SIMT and gained huge attention due to easy programming on NVIDIA GPUs. OpenCL [1] standardizes an open, data-parallel programming model inspired by CUDA. Unlike CUDA, OpenCL rules out any hardware assumptions for portability. OpenACC [13] is another standard for heterogeneous platform programming, mainly driven by supercomputing vendors and applications. It aims to provide an easier programming model via pragmas, inspired by OpenMP [14], while hiding many details such as having to write memory management code. Microsoft proposes C++AMP [15], an open standard programming model which is an extension to C++11 standard [16] in that massive data parallelism can be accelerated by following a dedicated programming method.

The need for optimizing compiler is ever increasing as programming abstractions move further away from specific hardware constructs. Therefore, OpenCL compilers for non-GPU architectures needs to be more capable in order to match what programmers can achieve with CUDA on NVIDIA GPUs. More broadly, the challenge is to optimize architecture-agnostic OpenCL code to reach a level of efficiency equivalent to what programmers could gain with lower-level, architecture-specific programming models.

Researchers have identified the performance portability problem and proposed solutions. Twin peaks is an OpenCL framework for CPU based systems [17]. In this work, each work-item in a work-group is mapped to a lightweight user thread. The user threads belonging to a work-group are scheduled upon a single CPU thread. Barriers for work-items are overhead implementing user threads in a compiler-unaware fashion. This approach, however, suffers from a significant overhead due to exhaustive thread management.

Stratton et al. took a different approach in that the proposed compiler transforms parallel execution of work-items into serial execution [2]. The rationale is that thread management is expensive while loop execution is very cheap on a CPU, considering the granularity of typical OpenCL programs. Serialization is implemented by casting a canonical loop over the index space of the work-group to the OpenCL code.

Lee et al. developed a OpenCL framework for the Cell architecture based on a similar idea [3]. Work item coalescing in his work is essentially serialization of work-item execution. Sophisticated analysis and management of data are one distinctive feature of his work as SPUs have limited-size, incoherent local stores, in contrast to the cache-based flattened memory space of a CPU architecture. For memory data access patterns that are hard to analyze, software-based virtual memory is used as a fallback to cope with memory accesses off the local memory boundary. Another noticeable difference to Stratton et al.'s approach is that the authors also developed an algorithm to minimize variables to be scalar expanded due to harsher limit of available memory to each SPU.

SRP had been conceived from a joint project, called ADRES [18][19]. SRP inherits the concept of ADRES in that it realizes acceleration of loop execution from software pipelining [7] over a grid of functional units. For CGRAs, traditional loop optimizations are crucial for performance. Luckily, CGRAs have simple goals for the loop optimizations, maximizing coverage of innermost loops being the most significant one. Well-known optimizations such as unrolling, flattening, fission, fusion and interchange as well as their combinations [8] could be implemented as a framework for CGRAs. Such a framework would be compulsory for successful adaptation of high level programming languages, OpenCL for example.

VI. Summary

In this paper, we evaluated the design of an OpenCL framework for CGRAs. The framework uses serialization for work-item execution, in order to overcome CGRA's insufficient thread-level parallelism to run work-items in parallel. We have identified that there are significant optimization chances for serialized kernel. Optimizations taking advantage of properties of serialization loops allow applying aggressive loop-level optimizations with greater freedom. It helps excavate more candidate loops in the form of innermost loops that the compiler can target to a CGRA for faster execution. The preferred design for the compiler framework thus is a combined serializer and post optimizer. Evaluation shows promising results for a class of benchmarks provided that the relevant compiler optimizations are followed.

For a class of algorithms, a literal interpretation of the OpenCL codes results in large slow downs compared to what can be achieved from simple hand-written code. We witnessed a need to develop an apparatus for performance portability for them. Also, a treatment of GPU-specific optimizations that makes the ported code slower remains for future work.

We foresee much interesting work to be done in the future. Here we list a few notable ones. First, a fully-developed post optimizer needs to be in place. Unpredictable optimization effects from transformations motivate an autotuning framework. It makes sense to explore a large optimization space provided that CGRAs are used mostly for embedded systems, where the set of applications is fixed. Second, the compiler should cope with large data in realistic settings. The fast on-chip data memory has limited capacity and it does not fit with OpenCL programming model. Fetching data from DRAM takes a very large number of cycles, and the CGRA must stall until the data becomes available. Thus, the compiler must analyze the program and implement proper data memory management algorithm.

Acknowledgments

This work was partially supported by the Gigascale Systems Research Center, funded under the Focus Center Research program, a Semiconductor Research Corporation program.

References

[1] Khronos OpenCL Working Group, "The OpenCL Specification," Nov 2011.

[2] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "Languages and compilers for parallel computing," ch. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pp. 16–30, Berlin, Heidelberg: Springer-Verlag, 2008.

[3] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi, "An opencl framework for heterogeneous multicores with local memory," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 193–204, ACM, 2010.

[4] S. Electronics, "Samsung Exynos 4210 RISC Microprocessor, User's Manual," Aug 2011.

[5] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *In Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 63–74, 1994.

[6] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, (New York, NY, USA), pp. 166–176, ACM, 2008.

[7] M. Lam, "Software pipelining: an effective scheduling technique for vliw machines," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, (New York, NY, USA), pp. 318–328, ACM, 1988.

[8] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[9] NVidia, "Nvidia OpenCL Best Practices Guide," 2009.

[10] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W.-m. W. Hwu, "Impact: an architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th annual international symposium on Computer architecture*, ISCA '91, (New York, NY, USA), pp. 266–275, ACM, 1991.

[11] NVIDIA, "CUDPP: CUDA Data-Parallel Primitives Library," 2009.

[12] NVIDIA, *CUDA Programming Guide*, 2011.

[13] "The OpenACC Application Programming Interface," Nov. 2011.

[14] OpenMP Architecture Review Board, "OpenMP Application Program Interface," specification, 2011.

[15] Microsoft, "C++ AMP: Language and Programming Model," 2011.

[16] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Feb. 2012.

[17] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 205–216, ACM, 2010.

[18] B. Mei, S. Vernalde, D. Verkest, H. D. Man, R. Lauwereins, B. M. , S. V. , D. V. , H. De, M. , and R. L. , "Dresc: A retargetable compiler for coarse-grained reconfigurable architectures," 2002.

[19] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: A case study," in *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, DATE '04, (Washington, DC, USA), IEEE Computer Society, 2004.