# An OpenCL Optimizing Compiler for Reconfigurable Processors

Jeongho Nah*, Jun Lee*, Hongjune Kim*, Jinseok Lee†, Seok Joong Hwang†, Donghoon Yoo† and Jaejin Lee*
* *School of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea*
† *Samsung Electronics, Giheung 446-712, Korea*
{*jeongho, jun, hongjune*}*@aces.snu.ac.kr*
{*jinseok21.lee, sjoong.hwang, say.yoo*}*@samsung.com*
*jlee@cse.snu.ac.kr*
`http://aces.snu.ac.kr`

***Abstract*—This paper presents simple and efficient optimization techniques for an OpenCL compiler that targets reconfigurable processors. The target architecture consists of a general-purpose processor core and an embedded reconfigurable accelerator with vector units. The accelerator is able to switch its architecture between the VLIW mode and the Coarse Grained Reconfigurable Array (CGRA) mode to achieve high performance. One big problem of this architecture is programming difficulty and OpenCL can be a good solution. However, since OpenCL does not guarantee performance portability, hardware dependent optimization is still necessary. Hence, we develop an OpenCL compiler framework that exploits the mode switching capability and vector units. To measure the effectiveness of the techniques, we have implemented the OpenCL framework and evaluate their performance with fourteen OpenCL benchmark applications.**

***Keywords*-Coarse Grained Reconfigurable Arrays, VLIW, SRP, OpenCL, Compilers, Optimizations, Performance Analysis**

## I. Introduction

Conventional application processors in ASICs provide sufficient computational power to process various functions in current mobile devices. However, as the complexity of the applications for the mobile devices grows, the programmability of processors has become a major concern. As a result, reconfigurable processors have been studied intensively[1], [2], [3], [4] as a programmable alternative to the ASICs.

The reconfigurable processor typically consists of multiple basic components, e.g., functional units (FUs) and register files (RFs). Those components comprise processing elements (PEs), which are connected by dedicated wires for fast communication. The PEs and overall topology can be configured flexibly during runtime.

Among those architectures, the Samsung Reconfigurable Processor (SRP)[5] provides C-based design tools that can both exploit high parallelism and deliver a software-like design experience. The SRP tightly couples a VLIW and a reconfigurable matrix, called CGRA. It can execute the data-parallel applications by switching between the VLIW and CGRA modes. The VLIW mode is useful for general-purpose computation such as conditional branch selections and function invocations. On the other hand, the CGRA mode makes full use of the software pipelining[6] via modulo scheduling[7] to accelerate data-parallel loops. The reconfigurable processor mostly operates next to a general-purpose host processor, typically a RISC, which executes a host program.

In such architectures, the major challenge is programming difficulty. Application programmers need to explicitly control the accelerator, manage communications between the host and the accelerator, and guarantee coherence and consistency between memory blocks. OpenCL[8] can help to alleviate this difficulty because it hides bare-metal details of the underlying architecture. Application developers can focus their efforts on the functionality of their applications.

OpenCL is an open standard for heterogeneous, accelerator-based parallel programming. Applications written in OpenCL are portable across the heterogeneous platforms that consist of multicore CPUs, GPGPUs, and other accelerators. It has a strong CUDA heritage[9], but it provides a rich set of common functionality that is supported by various devices and defines its own memory hierarchy and memory consistency model.

In this paper, we propose and evaluate an OpenCL optimizing compiler that exploits mode switching capability and vector units of the target reconfigurable processor, the SRP. The compiler is based on an open-source OpenCL-C-to-C translator of SnuCL[10]. An output of our compiler is C code, which is an input of the proprietary SRP C compiler included in the SRP SDK. We focus on source-level optimizations performed in the OpenCL-C-to-C translator, especially vectorization.

Basically, our optimizing compiler relies on a work-item coalescing technique[11] that is a procedure to serialize work-items to run them on a single accelerator core sequentially. This technique can reduce context switching overhead significantly. The work-item coalescing technique generates triply nested loops of the kernel code. The innermost loop in the triply nested loops can be naturally mapped for the CGRA mode of the SRP. By vectorizing the innermost loop at source level, the vector functional units can be fully exploited for higher performance. To do so, we convert scalar types into built-in vector types for variables in the kernel code. In addition, we apply two source code transformation techniques to some specific code patterns. One is *conditional*

**978-1-4799-2198-0/13/$31.00 ©2013 IEEE**

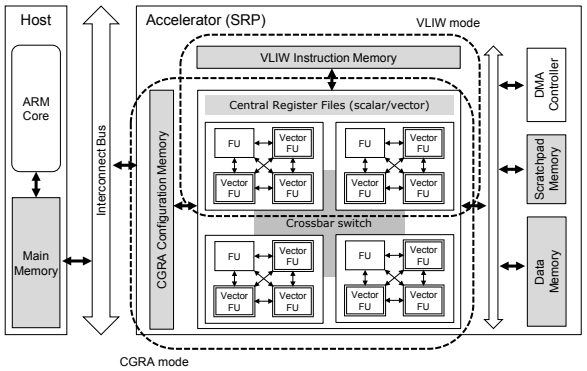


Figure 1: The target architecture

*statement optimization* that converts a conditional statement without side effects to a non-conditional statement to facilitate vectorization. The other is *work-item pruning* that removes an unnecessary conditional branch by pruning the OpenCL work-item index space.

To show the effectiveness of these techniques, we have implemented the techniques in the OpenCL-C-to-C translator included in the SnuCL framework, and evaluate its performance on the SRP using OpenCL benchmark applications.

The main contributions of this paper are:

- To exploit mode switching capability of the target architecture, we map the innermost loop of the triply nested loops to CGRA mode. This can be done easily after work-item coalescing and increases utilization of the SRP core.
- We describe a simple and efficient vectorization of OpenCL kernel programs. It converts scalar types into OpenCL built-in vector types for variables and arrays.
- We introduce two new source code transformation techniques, conditional statement optimization and work-item pruning. The conditional statement optimization enables efficient vectorization of a program that has irregular control-flow. Work-item pruning removes unnecessary conditional branches by pruning the OpenCL work-item index space.
- To show effectiveness of the proposed optimization techniques, we evaluate our full OpenCL implementation with 17 kernels from 14 OpenCL benchmark applications.

## II. Background

In this section, we describe the organization of our target architecture and OpenCL platform and execution models.

### A. Target architecture

The target architecture consists of a general-purpose host CPU and a reconfigurable accelerator. The host CPU performs system management tasks and executes operating system routines. The accelerator is dedicated to compute-intensive workloads. The host and the accelerator are connected by an interconnect bus, such as AMBA 3.0 AXI[12]. Typically a shared address space is defined for the communication between the host and the accelerator.

Our target accelerator is Samsung Reconfigurable Processor (SRP)[5]. Conceptually, it contains two register files for scalar and vector operations and sixteen functional units, as shown in Figure 1. The sizes of scalar and vector registers are 32-bit and 128-bit respectively. The functional units are organized into four sub-arrays, which are functionally identical. Each sub-array has four functional units, one for scalar operations and three for vector operations. The SRP has a scratchpad memory (SPM) instead of the data caches. The application programmer is responsible for explicit data copies between the SPM and the data memory by using a DMA controller.

The SRP has two execution modes: VLIW mode and CGRA mode, which are represented in two dotted boxes in Figure 1. The CGRA mode is used to accelerate the loops in a highly parallel way, whereas the VLIW mode is used for general-purpose computation such as conditional branches and function invocations. For VLIW mode, two operations can be issued per instruction and thus maximum two functional units in two sub-arrays can be utilized in parallel. On the other hand, CGRA mode can utilize the whole sixteen functional units in parallel.

### B. OpenCL Platform and Execution Model

In our OpenCL framework, a single ARM core becomes the host processor and OpenCL runtime thread runs on the host where the OpenCL host program also runs. A single SRP processor becomes the compute device and has a single compute unit in the device. Local memory and private memory are allocated to the SPM and it is handled by our OpenCL framework. In this implementation, optional global/constant memory cache is not supported.

An abstract index space is defined for a kernel program. It is a three dimensional space and an instance of the kernel executes for each point in this index space. The kernel instance is called work-item, and is uniquely identified by its global ID defined by the point in the index space. Each work-item executes the same kernel code but its data accesses and execution pathway can vary.

A work-group is composed of one or more work-items. Each work-group has a unique work-group ID and assigns a unique local ID to each work-item within itself. Thus a work-item is identified by its global ID or by a combination of its local ID and work-group ID. For example, the global ID for dimension 0 is represented as `local_size[0] * work_group_id(0) + local_id(0)`, where `local_size[0]` is the size of a work-group of dimension 0.

### C. OpenCL-C-to-C Translator

Our OpenCL framework assumes that work-items are executed on virtual PEs, since there is no physical PEs in

```
__kernel void vec_add(__global float *A, __global float *B,
                      float P) {
  int x = get_global_id(0);
  int y = get_global_id(1);
  B[x + 256*y + 1024] = A[x + 16] + P;
}
```

(a) OpenCL kernel code

```
void vec_add(float *A, float *B, float P) {
  int x, y;
  for( int k = 0; k < local_size[2]; k++){
    for( int j = 0; j < local_size[1]; j++){
      for( int i = 0; i < local_size[0]; i+=1){
        x = goffset(0) + i;
        y = goffset(1) + j;
        B[x + 256*y + 1024] = A[x + 16] + P;
      }
    }
  }
}
```

(b) Transformed C code

Figure 2: A simplified example of work-item coalescing

a SRP processor. To emulate the virtual PEs, the runtime executes each work-item one by one in a serialized fashion. This can be done by a context switching mechanism[13]. However, executing work-items by context switching introduces a significant overhead.

To avoid this, our OpenCL framework relies on the work-item coalescing technique proposed by Lee et al.[11]. By applying the technique, the kernel code can be represented as a triply nested loop when the kernel and its callee functions do not contain any barrier. This loop iterates on the index space of a work-group. The dimension decreases from the outer loop to the inner loop. Figure 2 shows an example of work-item coalescing.

## III. OpenCL Framework Implementation

This section describes the components of the SnuCL framework we modified to run the framework on the SRP system.

The runtime and built-in libraries are modified for the SRP architecture. Most of our key implementations and optimizations are done in the OpenCL-C-to-C translator that translates the kernel code for the SRP core.

### A. Compilation Process

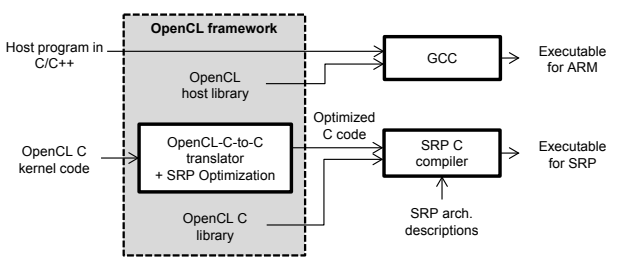


Figure 3: Compilation process

The overall compilation process of an OpenCL application is illustrated in Figure 3. Host application code and host library code that run on the ARM core are compiled with GCC[14]. Kernel code written in OpenCL C and additional device code, such as supporting routines, built-in libraries, need to be compiled to SRP binary code. The kernel code is first compiled to C code by OpenCL-C-to-C translator. The translator is based on LLVM[15] front-end, `clang`. This translator is modified to generate optimized code to exploit parallelism in the SRP core. Generated C code and additional device code are compiled by the proprietary SRP C compiler.

### B. Indicating Code Sections to Run on CGRA Mode

The SRP C compiler generates VLIW mode instructions by default. A Loop intended to be executed in the CGRA mode must be annotated with "#pragma rpcc cga" directive, which indicates that the SRP compiler generates CGRA instructions for the loop section. To exploit maximum parallel computation capacity of the SRP, the parallel portion of the code needs to be identified by the user.

However, our implementation annotates loops for CGRA execution automatically during translation process, so users do not have to explicitly divide the code into sections that have to be run on the VLIW or CGRA mode. In the nested loop generated by work-item coalescing, the innermost loop is annotated for CGRA mode execution.

## IV. Vectorizing the Kernel Code

A number of optimizations are added to SnuCL OpenCL-C-to-C compiler in order to make full use of parallel execution capabilities of the SRP core. In this section, we explain how the coalesced loop is vecotrized and additional optimizations are done.

To exploit the vector capabilities of the SRP, our optimizing compiler performs vectorization on the coalesced loop. In many cases, regular array access throughout the iteration could be vectorized, due to the data independence between OpenCL work-items.

In the generated C code shown in Figure 2b, the body of the innermost loop contains a computation on array elements that are indexed by work-item IDs. When `A` and `B` overlap, there can be cross-iteration dependencies between array accesses. In normal cases this inhibits vectorization of the loop. However, since OpenCL work items could be executed concurrently, the coalesced loops could be safely handled as a *DOALL loop*. A DOALL loop is a loop whose iterations can be correctly run in any order[16]. Thus it is safe to vectorize coalesced loops without loop dependence analysis and code transformation for dependence removal.

Figure 4 is the vectorized code generated from the kernel code in Figure 2. The innermost loop code is transformed to handle multiple data elements at once by a vector instruction. The number of elements packed and handled together is called a *vectorization factor*. From the example we see:

```
1  void vec_add(float *A, float *B, float P) {
2    float4 *vA = (float4*)A;
3    float4 *vB = (float4*)B;
4    float4 vP = (float4)P;
5    int4 vx, vy;
6    for( int k = 0; k < local_size[2]; k++){
7      for( int j = 0; j < local_size[1]; j++){
8        #pragma rpcc cga
9        for( int i = 0; i < local_size[0]; i += 4){
10         vx = (int4)(goffset(0) + i) + (int4)(0, 1, 2, 3);
11         vy = (int4)(goffset(1) + j);
12         vB[vx[0]/4 + 64*vy[0] + 256] = vA[vx[0]/4] + vP;
13       }
14     }
15   }
16 }
```

Figure 4: A vectorization example

- Scalar types are converted to vector types and variables are renamed.
- Input values and constant values are converted to vector types.
- Global index values are handled specially.
- Index variable of the innermost loop is increased by the vectorization factor.
- Array subscript expressions are converted to access the element by the size of vectorized chunk.

### A. Array Access Pattern Analysis

Note that not all the coalesced kernel can be transformed to vectorized code. Thus, before code translation, it is required to discriminate code that is not possible to vectorize. To vectorize code with array accesses, there are two constraints:

- Contiguity: The innermost iteration has to access the array items contiguously. If the access stride is bigger than 1, data elements need to be gathered for vector operation and overhead for copying elements could exceed the benefit of vectorization.
- Alignment: In the SRP, the addresses of operands used in vector operations need to be aligned to the size of the vector width (128 bit).

To satisfy the alignment constraint, our runtime guarantees that the newly created buffer objects are aligned by the vector width. To assure the alignment, we check if all the array indexes are a multiple of the vectorization factor.

By performing symbolic analysis during the code translation process, we check for the constraint on array indices. As most of the interesting applications access multi-dimensional arrays with an affine index of loop induction variables, we only accept such forms. We assume array subscript expressions can be represented in the form of $ax+by+cz+d$, where loop induction variables for dimension 0, 1, and 2 is defined as $x$, $y$, and $z$. Once converted in this form, coefficients could be checked if the access pattern satisfies the constraints.

Our compiler checks if every array subscript expression satisfies the following conditions:

- To guarantee contiguity, $a$ must be 1 to convert the array access to a vector operation. If $a$ is 0, it is treated as a special scalar type. In other cases the coalesced loop may not be vectorized.
- To guarantee alignment, each $b$, $c$ and $d$ must be a multiple of the vectorization factor.

In the alignment test, if all the coefficients are constants, the test could be performed at compile time. But if one or more kernel parameters are included in the coefficients, bail-out code is inserted before the execution of the kernel function. This code checks the alignment of each coefficient and fallbacks to non-vectorized code when conditions do not meet. There also is code to check if the innermost loop size is a multiple of vectorization factor and fallbacks to non-vectorized version if it fails.

There are some additional conditions where our compiler cannot vectorize the code. For example, when OpenCL built-in vector type is used as an element of an array and an element of the vector is accessed explicitly. In this case, it is hard to guarantee the contiguity across the iterations. User defined data types and user defined functions are not supported. But unlike user defined functions, built-in functions are mapped to corresponding vector version if possible.

### B. Vector Code Generation

If a kernel code passes all the tests, it can be translated to vector code. The following is our vecorization process.

- All scalar variables are translated to vector types. Type definitions are translated and variables are renamed to prevent name collision. Kernel parameters, both pointer and scalar types, are converted to vector types by explicit type casting.
- All constant values are expanded to vector values by explicit type conversion. The calculation of ID in dimension 0 is treated specially (line 10 of Figure 4). Since in the original loop, the ID in dimension 0 is incremented every iteration, a vector literal `(0, 1, ..., VF)` has to be added to the vector ID value when translated.
- The innermost loop index is translated to be incremented by the vectorization factor for each iteration. This is shown in line 9 of Figure 4, where the increment of $i$ is changed from 1 to 4.
- Array indices are also translated for vectorization. If the coefficient $a$ from the access pattern analysis is 1, subscript expression is divided by the vectorization factor. This is because when the original scalar array is accessed as a vector array, array index could be calculated as $i_s = \mathrm{VF} \times i_v$, where $i_s$ and $i_v$ are array indices of scalar and vector types, respectively, and $VF$ is the vectorization factor. If $a$ is 0, it is treated as special scalar type. If it is used to read an element from array, it is explicitly type-casted to a vector type. When

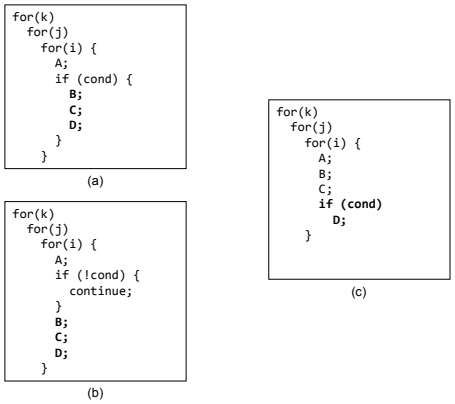


Figure 5: Example patterns for optimizing the conditional statements

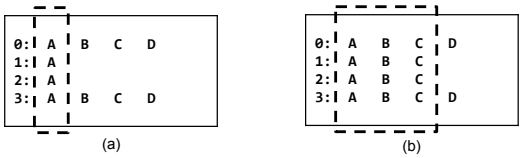


Figure 6: Example executions of code in Figure 5

assigning a value to an array element, the first element of the RHS vector value is assigned to the scalar type array.

- Vectorizing conditional statements requires complicated techniques and hardware support[17]. Our implementation does not vectorize conditional statements; it serializes the statements by unrolling them by the vectorization factor and re-converting the variable accesses and values to scalar types.
- If built-in function is used, its vector version will be used instead.

### C. Optimizing Conditional Statements

We have explained that conditional statements are not vectorized. We introduce an optimization technique for them in this section.

Figure 5a and 5b are some translation examples of an OpenCL kernel. They are semantically the same. Note that code 5b is generated when `return` in the kernel code is converted to `continue` by work-item coalescing. If the condition expression is true on iteration 0 and 3 and false on iteration 1 and 2, the executed statements will be those in Figure 6a. Only statement A can be vectorized and B, C, D may not, because it is not known that these statements would be executed in each iteration. However, when we assume execution of B and C do not have an effect outside the work-item, it is safe to execute the statement unconditionally. Thus the code can be transformed to the code in Figure 5c and its execution will look like Figure 6b. Now statements B and C are also vectorized.

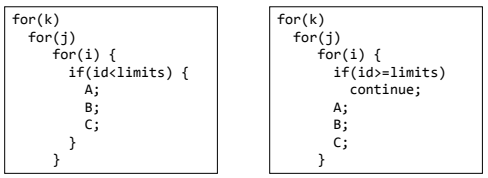


Figure 7: Common patterns appeared in kernel code

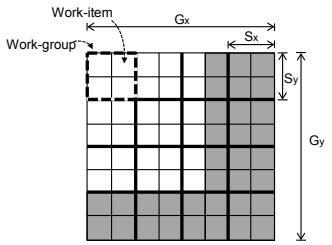


Figure 8: A work-item pruning example

By the following algorithm, kernel code with a certain pattern is translated into the code with decreased number of conditional statements, and more portion of the code can be vectorized.

1) Search for suitable `if` statement. (1) It has to be the last statement of the kernel, or (2) it contains a single `return` statement. In first case, statements contained in the `if` statement, and in second case, statement after the if statements are selected.
2) Original `if` condition and `return` statements are removed.
3) Each target statement selected in step 1) is checked if it includes a write operation to global or local memory. If there are no such writes, it is safe to translate it into a non-conditional statement. If it has such writes, the statement has to be guarded by the condition.

When the vectorization is performed after this translation, only the remaining part of conditional statements will be unrolled and serialized.

## V. Work-item Pruning

In this section, we describe a source code transformation technique to prune inactive range of work-items in a coalesced loop. This technique is more effective when it is combined with kernel vectorization.

### A. Observation

When the size of data items to be accessed does not match the global index size, it is a common programming practice to inactivate the execution of work-items out of the effective space by checking the work-item ID. Figure 7 shows two typical examples of generated code, and Figure 8 shows an example of a 2-dimensional index space. The $G_x$, $G_y$, $S_x$, and $S_y$ are the global work size and local work size for

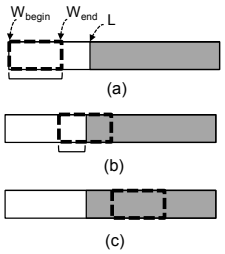


Figure 9: Active work-items and work-groups

```
1  max_k = get_limit(2);
2  for (k = 0; k < max_k; ++k) {
3    max_j = get_limit(1);
4    for (j = 0; j < max_j; ++j) {
5      max_i = get_limit(0);
6      for( (i = 0; i < max_i; ++i) {
7        ...
8      }
9    }
10 }
```

Figure 10: A result of work-item pruning

dimension x and y respectively. The inactive work-items are represented in gray. In a normal OpenCL execution environment, this immediately returned loop will not degrade performance noticeably. But in our case, the conditional statement prevents some of the statements following from vectorization and can hurt the performance seriously.

Our idea is, if the loop range is adjusted to fit to the actual index space, condition clauses can be removed for some popular patterns. Note that this is more powerful than the optimization in Section IV, because the entire condition could be removed. In our implementation, we first apply work-item pruning. Then the conditional statement optimization is applied to the remaining conditional statements.

### B. Pruning Process

First, conditional statements that match our interest are selected. By pattern matching, we select an `if` statement that is (1) a single top-level statement of the kernel or (2) a statement placed at the beginning of the kernel containing a single `return` statement (`continue`). For brevity of explanation, we assume all matched code is in form (1) because (1) and (2) are semantically identical and interchangeable.

For a structurally matched `if` statement, the condition expression is analyzed to check if it is appropriates for work-item pruning. We define *work-limiting-condition* as a single predicate that indicates ID in a single dimension is smaller than a loop-invariant value (e.g., $x < C$ or $y \leq C$, where $x$ and $y$ are global IDs for dimensions 0 and 1, and $C$ is a constant). We can first hoist the loop invariant conditions (predicates not containing variables related to the global IDs) to the outside of the outermost coalesced loop. For the remaining condition, we check if the condition is a `&&` concatenated expression of work-limiting-conditions.

Table I: Characteristics of the OpenCL applications

| Kernel | Source | Input | A | B | C | D |
|---|---|---|---|---|---|---|
| streamcluster-memset | Rodinia | 512 points | (1, 1, 256) | O | O | |
| gaussian-Fan1 | Rodinia | matrix16.txt | (1, 4, 4) | O | | O |
| gaussian-Fan2 | Rodinia | matrix16.txt | (1, 4, 4) | O | | O |
| backprop-layerforward | Rodinia | default | (1, 16, 16) | O | | |
| backprop-adjust_weights | Rodinia | default | (1, 16, 16) | O | | |
| kmeans-kernel | Rodinia | 100 (file) | (1, 4, 4) | O | | O |
| kmeans-swap | Rodinia | 100 (file) | (1, 4, 4) | O | | |
| lavaMD | Rodinia | default | (1, 1, 128) | O | | |
| nn | Rodinia | cane4_0.db | (1, 1, 128) | | | O |
| AlphaBlending | in-house | 64x64 | (1, 16, 16) | O | O | |
| Template | AMD | 256 | (1, 1, 256) | O | O | |
| BinarySearch | AMD | default | (1, 1, 256) | O | | |
| FastWalshTransform | AMD | default | (1, 1, 256) | | O | |
| FloydWarshall | AMD | default | (1, 1, 256) | O | | |
| MatrixTranspose | AMD | default | (1, 16, 16) | O | O | |
| DotProduct | NVIDIA | 4096 | (1, 1, 256) | O | | O |
| VectorAdd | NVIDIA | 4095 | (1, 1, 256) | O | O | O |

A: local work-group size, B: CGRA mapping, C: vectorization, D: work-item pruning

As shown in Figure 10, we insert code before each loop to calculate the limit of iteration bounds. Figure 9 shows many cases where work-group (indicated as dotted box) is launched regarding a single dimension. $W_{\mathrm{begin}}$ and $W_{\mathrm{end}}$ presents the beginning and end of the work-group index range, and $L$ is the limiting condition value from the code. In each case, the loop range can be as follows:

- Case (a) shows when the limiting value $L$ is not inside the work-group range. In this case, the original work-group range $[W_{\mathrm{begin}}, W_{\mathrm{end}})$ can be used as the iteration range.
- Case (b) shows when the limiting value $L$ is included in the work-group range. Iteration range is $[W_{\mathrm{begin}}, L)$
- Case (c) shows when all work-group item is above limited range. No iteration is needed. The range is $[W_{\mathrm{begin}}, W_{\mathrm{begin}})$.

The actual maximum loop bound can be calculated using the size of the range: $W_{\mathrm{end}} - W_{\mathrm{begin}}$, $L - W_{\mathrm{begin}}$, and 0 in each case.

## VI. Evaluation

In this section, we describe our evaluation methodology and results for our optimizing compiler framework. To show the effectiveness of our techniques, we compare the speedup of each technique.

### A. Methodology

We use the SRP architecture described in Section II. To evaluate our implementation we have used a cycle-accurate simulator for the SRP included in the SDK. The simulator provides a functional simulation and performance cycles of a binary.

We run 17 OpenCL kernels from 14 OpenCL applications. They are from various sources: Rodinia[18], NVIDIA[19], and AMD[20]. We also implement an OpenCL version of

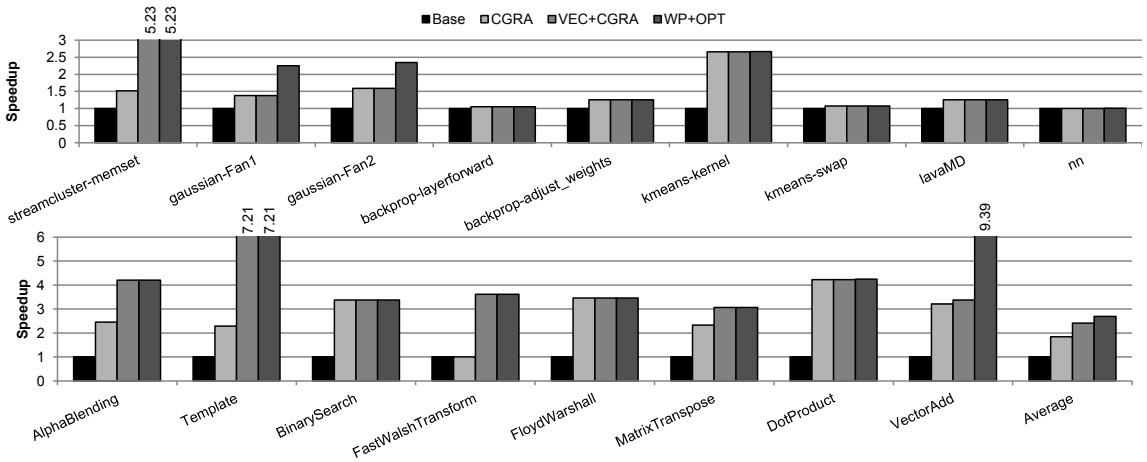


Figure 11: OpenCL kernel speedup

an alpha blending application from in-house C code. The characteristics of the applications are summarized in Table I.

### *B. Results*

Figure 11 shows speedups of the various kernels. The baseline is C code obtained by base SnuCL translator, which is represented by the bar labeled `Base`. No optimization is done on the coalesced code that is executed in the VLIW mode. The bar labeled `CGRA` denotes the speedup of applying CGRA mapping to the baseline implementation. The bar labeled `VEC+CGRA` shows the speedup by vectorization including conditional statement optimization with CGRA mapping. The bar labeled `WP+OPT` shows the result including the work-item pruning in addition to optimizations in `VEC+CGRA`.

As shown in the column B in Table I, most of the kernels are mapped to CGRA successfully and average speedup is 1.84. Two kernels, `FastWalshTransform` and `nn`, are not mapped to CGRA. This is because these kernels include function calls that inhibit the CGRA mapping. `BinarySearch`, `FloydWarshall`, `DotProduct`, and `VectorAdd` show more than 3x speedup compared to the baseline. These applications include compute-intensive code with simple control-flow, which has beneficial for CGRA.

As shown in column C of Table I, six kernels of the benchmark applications are vectorized, which is represented in the bar labeled `VEC+CGRA` in Figure 11. Average speedup of vectorization is 2.4 and the maximum speedup is 7.21.

To increase efficiency of vectorization, we applied the conditional statement optimization, as presented in Section IV. It is applied to `AlphaBlending` and it shows 4.21 speedup.

In Figure 11, the bar labeled `WP+OPT` shows the speedup over the baseline after applying work-item pruning and other optimizations. Average speedup of `WP+OPT` is 2.69 and the maximum speedup is 9.39. In case of `Gaussian-Fan1` and `Gaussian-Fan2`, about half of the iteration range is pruned by work-item pruning. For `VectorAdd`, we can efficiently vectorize the code because control flow divergence is removed by work-item pruning. However, for the other kernels, performance improvements are limited to 1% - 3% because they are not vectorized and pruned iteration ranges are small.

## VII. Related Work

Optimizations for the CGRA architecture have been studied extensively. Mei et al.[2], [3], [4] presented an architecture exploration for a CGRA template and a modulo scheduling algorithm[7] to exploit loop-level parallelism. Miniskar et al.[21] have studied the impact of function inlining and loop unrolling optimizations for the SRP. However their work is focused on the programs written in C.

Li et al.[22] proposed an OpenCL compiler framework for embedded multicore DSPs. They adopt a work-item coalescing technique[11] for work-item serialization. Their vectorization scheme exploits SIMD intrinsics available on the target DSPs, which is different from our approach.

Kim et al.[23] proposed an OpenCL compiler framework for the SRP. However the target architecture template and optimization techniques are quite different from our work. Our target architecture template has large vector units and we propose new code transformation techniques to utilize the vector units efficiently.

Karrenberg et al.[24] proposed a language and platform independent code transformation called Whole Function Vectorization (WFV). It performs vectorization of a function given by an arbitrary control flow graph in SSA form. While their approach is similar to our approach in the sense that both exploit data parallel kernels to vectorize, optimization techniques are different.

## VIII. Conclusions

In this paper, we have presented OpenCL implementation targeted on platforms using the SRP processor as an OpenCL compute device. To exploit the parallel processing power of the processor, our compiler maps the parallel loop into the CGRA mode and translates the code to a vectorized version. We have introduced a vectorization technique to translate a coalesced loop into a vectorized version. We also have introduced two techniques to remove conditional statements: by converting conditional statement with no side-effects to unconditional statement, and removing conditions that limit index space in a work-group.

We have shown the effectiveness of our OpenCL optimizing compiler and evaluated its performance using 17 kernel programs. Evaluation results show that our approach is effective and promising for the SRP.

## Acknowledgment

This work was supported by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) from the National Research Foundation of Korea funded by the Korean government (Ministry of Education, Science and Technology). This work was also supported in part by Brain Korea 21 Project. ICT at Seoul National University provided research facilities for this study.

## References

[1] G. Lu, H. Singh, M.-h. Lee, N. Bagherzadeh, F. Kurdahi, and E. Filho, "The morphosys parallel reconfigurable system," in *Euro-Par99 Parallel Processing*, ser. Lecture Notes in Computer Science, P. Amestoy, P. Berger, M. Dayd, D. Ruiz, I. Duff, V. Frayss, and L. Giraud, Eds. Springer Berlin Heidelberg, 1999, vol. 1685, pp. 727–734. [Online]. Available: http://dx.doi.org/10.1007/3-540-48311-X_102

[2] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: a case study," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 2, feb. 2004, pp. 1224 – 1229 Vol.2.

[3] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," *Design Test of Computers, IEEE*, vol. 22, no. 2, pp. 90 – 101, march-april 2005.

[4] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 10 296–. [Online]. Available: http://dl.acm.org/citation.cfm?id=789083.1022741

[5] Samsung Electronics, Ltd., "Samsung Exynos 5250 RISC Microprocessor," Oct. 2012.

[6] 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*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 318–328. [Online]. Available: http://doi.acm.org/10.1145/53990.54022

[7] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: ACM, 1994, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/192724.192731

[8] Khronos OpenCL Working Group, "The opencl specification," Nov. 2012.

[9] NVIDIA, "CUDA C Programming Guide," Oct. 2012.

[10] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: an opencl framework for heterogeneous cpu/gpu clusters," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 341–352. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304623

[11] 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*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854301

[12] ARM, "Amba open specifications," Mar. 2004.

[13] R. S. Engelschall, "Portable multithreading: the signal stack trick for user-space thread creation," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '00. Berkeley, CA, USA: USENIX Association, 2000, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267724.1267744

[14] R. M. Stallman, "Using the GNU Compiler Collection," 2008.

[15] LLVM Team, "The llvm compiler infrastructure," Dec. 2012.

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

[17] J. C. H. Park and M. Schlansker, "On predicated execution," 1991.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, oct. 2009, pp. 44 –54.

[19] NVIDIA, "NVIDIA GPU Computing Software Development Kit," Feb. 2010.

[20] AMD, "Amd accelerated parallel processing sdk v2.4," Apr. 2011.

[21] N. R. Miniskar, P. S. Gode, S. Kohli, and D. Yoo, "Function inlining and loop unrolling for loop acceleration in reconfigurable processors," in *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, ser. CASES '12. New York, NY, USA: ACM, 2012, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/2380403.2380426

[22] J.-J. Li, C.-B. Kuan, T.-Y. Wu, and J. K. Lee, "Enabling an opencl compiler for embedded multicore dsp systems," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, sept. 2012, pp. 545 –552.

[23] H.-S. Kim, M. Ahn, J. A. Stratton, and W.-M. W. Hwu, "Design evaluation of opencl compiler framework for coarse-grained reconfigurable arrays," in *Field-Programmable Technology (FPT), 2012 International Conference on*, dec. 2012, pp. 313 –320.

[24] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, april 2011, pp. 141 –150.