Abstract: Using directive techniques for accelerating loops on heterogeneous systems, object-oriented programs can be hundreds of folds faster with a short learning curve required from programmers. Despite much research on various loop parallelization techniques, the opportunity to accelerate a compact construct like range-based loops using directive-based compilers has not yet been investigated. The paper addresses this issue and presents directive-based compiler techniques for accelerating range-based loops using OpenCL. The experimental results based on two workloads show the speedup of 3-135 folds achieved from accelerating range-based loops using the proposed compiler. The literature review depicts an average of 12-135 folds speedup gain by using traditional loop acceleration by existing directive-based compilers. This highlights the potential of exploiting loop-level parallelism by using range-based loop acceleration.

Key words: Loop parallelization, GPU, OpenCL, compiler, C++, acceleration, heterogeneous systems.

1. Introduction

In modern language like C++11, users can loop through each element in collection without having to deal with index values using powerful constructs called range-based loops. In the C++11 language standard [1], iterating through range can be done using the ranged-based for syntax or the std::for_each template from the STL algorithms. Fig. 1 shows an example code using both loops to perform summation of elements of an array stored in an STL vector. The clear and concise syntax of ranged-based for make the source code more understandable and easier to debug. A difference of the C++11 standard from the previous one is that it provides the syntax of lambda functions used for creating a function object. Thus in an STL for_each template, users can define a lambda function to express the loop body. Distinct from the lambda functions of other languages (like .NET languages, Java or Python), lambda functions in C++11 allow users to pass objects and variables in the scope of their declaration into the function body. As shown in line 5, the variable x was passed by reference into the body of lambda function using the capture syntax specified in square brackets.

Range-based loops are commonly used with the STL containers to express operations on elements which can be done in parallel. Thus, these loops have great potential to be accelerated on any heterogeneous systems e.g., multicore CPUs, graphical processing units (GPUs), or coprocessors liked Intel Xeon Phi.

To exploit data parallelism on various heterogeneous systems, OpenCL [2] language layer has been widely accept for its portability and competitive performance. Research has shown that a single version of OpenCL code can be executed on three different platforms with performance losses of less than 12%, if the parameters are well chosen [3]. The performance of some OpenCL code could be fairly competitive with CUDA on Nvidia...
hardware [4]. As OpenCL programs have numerous details and difficult to optimize, many directive-based compiler frameworks have been proposed to address this issue [5]-[10]. Research results have confirmed that directive-based compilers make heterogeneous systems more accessible. Moreover, the compiler optimization offered could help programmers to exploiting the computational power of the underlining platforms as effectively as hand coded (See Section 4).

```
vector<float> v = {1.2, 3.5, 3.8, 7, 3};
float sum = {0};
for (auto x : v) sum += x;
for_each(v.begin(), v.end(), [&sum](float x){sum += x;});
```

Fig. 1. Array summation using the range-based for and for-each.

Although much research on compiler acceleration is underway to achieve portable performance on OpenCL enable devices, techniques of directive-based compilers to accelerate range-based loops has not yet been investigated. Auto-parallelizing C++11 lambda functions was addressed in the LambdaJIT compiler [10]. Using LambdaJIT, the workloads outperformed CUDA by up to 10%. These results confirm that accelerating the for_each template on heterogeneous systems is viable and may lead to portable performance.

This research aims to investigate the performance of using the directive-based techniques to accelerate range-based for_each loops on heterogeneous systems. The research makes three contributions as follows:

1) A source-to-source compiler framework, based on the Clang front end of the LLVM infrastructure, foraccelerating STL for_each templates on heterogeneous systems using OpenCL is proposed (Section 2).
2) The performance of the ForCL generated code in comparison to the manual OpenCL code on two embarrassingly parallel workloads is discussed (Section 3).
3) The performance gain from existing compilers obtained via a systematic review is reported (Section 4).

A simple directive similar to OpenMP is used to specify which for_each templates are to be offloading to heterogeneous systems. The ForCL compiler reads the directives and translates the body of the for_each templates into OpenCL kernels. The ForCL generated code of two embarrassingly parallel workloads had achieved similar performance in comparison to the OpenCL hand-tuned code (less than ±9% difference). Both workloads accelerated for 2.9-135 folds with reference to sequential CPU. To quantify the benefit of accelerating range-based for_each template, meta-analysis method was used to capture the speedup of code generated by existing directive-based compilers. The analysis results depict that the ForCL generated code achieved the similar range of performance from previous work. Thus, this highlights the potential of exploiting loop-level parallelism by using range-based loop acceleration.

2. The ForCL Compiler Framework

![Fig. 2. Overview of the ForCL compilation process.](image)

The ForCL compiler is a directive-based compiler for generating the C++11 for_each template to OpenCL code. Unlike traditional loop parallelization, attempts to parallel the C++11 for_each template have to deal with two issues i.e. (a) the unstructured in memory of STL containers and (b) the list of captured variables specified in lambda function. Thus, the ForCL compiler must know the information about variable scopes and
container size before the body of lambda function can be understood. For this, the compiler has been developed by extending the Clang [11] frontend of the LLVM [12] infrastructure. The ForCL compiler reads a C++ code augmented with the ForCL directives (see Fig. 2). It generates two output files i.e., (a) the C++ code embedded with OpenCL host methods, and (b) a kernel file containing the data parallel version of the for_each body encapsulated as OpenCL kernel functions. The ForCL compiler consists of four modules, as highlighted in Fig. 2. These modules work with the Clang frontend by extracting the required information from the abstract syntax tree (AST) created by Clang.

2.1. ForCL Directives and Pragma Handler

A simple directive, similar to OpenMP, is used in ForCL. As shown in Fig. 3(a) line 3, the #pragma forcl parallel reduction(+:count) is used to tell the ForCL compiler to parallel the for_each loop which is immediately followed. The clause reduction specified the requirement for mutual exclusion on the variables when using with the denoted operator. Variables specified in the reduction clause must be visible in the scope where the for_each loop is declared, and must be captured by reference into the lambda function.

The pragma handler extends the Clang preprocessor class in order to specify the location of ForCL pragmas in the source code. When a pragma construct is found, the handler records its parameter and marks the location in the source code. The handler then passes the source program to the ordinary Clang frontend. The program is parsed and its AST is pulled from Clang to the ForCL lambda extraction module.

2.2. Lambda Extraction

Once the AST is received from the Clang frontend, the context of lambda expression defined in the for_each loop is extracted. Lambda extraction module traverses the AST for collecting information about the STL container used, the variables captured, the parameters and member variables of the lambda function. The mutable property, types, size and other attributes are kept in the ForCL symbol table. The list of the STL container variable and mutable variables, which are captured by reference in lambda function and appeared in the reduction clause (e.g. variable count in Fig. 3(a) line 3), is kept in a separate table for the memory mapping process. Once the information of all variables has been collected, the next module is called.
2.3. Memory Mapping

The memory mapping module adds the code to generate an OpenCL host program. The host program gets platforms and computes devices (e.g., CPU, GPU or other accelerators); creates a Context for communicating with devices; creates command queue for each platform; and creates a program object (See the dash lines in Fig. 4). The host reads a kernel file (.cl) and sends all kernel functions to build using OpenCL compiler at runtime. The binaries of these functions are kept in the program object ready to be executed. Prior to a kernel is sent for execution, all variables used are allocated. The memory mapping module uses the information about the input container and mutable variables for allocating the memory for CPU (host) and the memory on device. As shown in Fig. 4, data has to be transferred between host and device’s memories. At this step, the ForCL compiler will select which memory on device (i.e., global or local memory) is best for each variables.

The number of elements, or iteration space, is extracted from the iterator range defined in the for_each loop. Each iteration is assigned to a work item inside an OpenCL compute device. Fig. 4 depicts architecture of a compute device having eight work items, each of which has its own private memory. The work item is a logical view of a processing element where a kernel is executed. The ForCL memory mapping module gets the total number of work items required from the number of iterations. The information of OpenCL devices, e.g., the number of cores, is used to map these work items into work group. The algorithm of work-group size selection from [13] is used at this step. Once the code of host methods has been added to the adjusted AST representing the program, the module responsible for generating the OpenCL kernel functions is called.

2.4. OpenCL Kernel Generation

The information about memory mapping is used to create the list of corresponding parameters, and generate the body of the OpenCL kernel functions. As shown in Fig. 3, any variable used in the body of the lambda function in Fig. 3(a) is generated as the parameters in the kernel function (in Fig. 3(b)). The keyword __global specified the location of device memory where the variables are kept. In the previous module, the host code which performs memory mapping between host and device were added to the AST. Then the host method to submit a kernel to the command queue is added. When this method is executed, the generated kernel function (see Fig. 3(b)) will be called. The OpenCL runtime would pass these parameters to the kernel function. Each kernel function represents the work of each work item.

The beginning part of the kernel function is to get the identity of each work item on a workgroup, which are assigned to map with elements of STL container. After that, each work item performs the calculation as defined in the lambda function of the for_each loop. Vectorization techniques are applied here to allow each work item performing calculation on many elements at the same time. Once the optimization is done, the codes are written to a kernel file (.cl) as an output. The adjusted AST is used to generate the output C++ file.

3. Experimental Results and Discussion

Table 1. Experimental Platforms

<table>
<thead>
<tr>
<th>Detail</th>
<th>Platform #1</th>
<th>Platform #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Intel Core i7 860 2.8 GHz, 4 cores, 8 MB Cache, 4 GB RAM</td>
<td>Intel Xeon CPU E5-2680 2.70GHz 8 (2 logical cores per physical), 20 MB Cache, 62 GB RAM</td>
</tr>
<tr>
<td>Machine</td>
<td>NVIDIA GeForce GTX 295, 240 cores, 1242 MHz Clock, Memory of 896 MB, OpenCL 1.1</td>
<td>Intel Xeon Phi Coprocessor 7120, 61 cores 1181 MHz Clock, 5772 MB Memory, OpenCL 1.2</td>
</tr>
<tr>
<td>Accelerator</td>
<td>Windows 7 Service Pack 1</td>
<td>Linux CentOS release 6.5</td>
</tr>
</tbody>
</table>

Two workloads which require high precision of floating point results have been chosen for testing the compiler. These are the Sine of Vector’s elements (SV) and the Pi calculation using Monte Carlo method (PI). Each workload has been developing in three versions: sequential, ForCL-generated, and hand-tuned OpenCL.
All code were compiled using the MinGW compiler with –O2 flag to optimize for the fastest speed; and were executed on two experimental platforms as shown in Table 1 for ten replications. For each execution, the numbers of array elements computed per picosecond (items/picosecond) were measured. The average of the results obtained from ten replications is used to represent the performance of each particular execution.

3.1. Baseline Performance of the Workloads

The performance results (in items/ps.) were collected from executing the sequential versions of SV and PI on both platforms. These results were used as baseline performance. Fig. 5 depicts the performance of the SV workload having 1, 2.5, 5, 7.5 and 10 million elements of single-precision floating point. For all data sizes, the workload achieved an average 27.23 and 48.58 items/picosecond on Platform #1 and Platform #2 respectively. Fig. 6 depicts the sequential performance results of the PI workload on Platform #2. The experiments were done using 1, 1.5, 5, 7.5 and 10 million random values. The generated random values and the results of PI were recorded to be used as inputs to verify the accelerated codes.

3.2. Performance of the Accelerated Versions

The hand-tuned OpenCL and ForCL-generated code were executed on both platforms using the recorded inputs obtained from the sequential versions. Their outputs were verified against the recorded output, and their throughputs (in items/picosecond) were recorded. Speedup of the accelerated code relative to the baseline performance was used to represent the performance gain. The speedup of both workloads were collected on two different devices i.e., on CPU and on accelerator (GPU or Xeon Phi). Fig. 7 shows the performance of the hand-tuned OpenCL and the ForCL-generated SV workload on Platform #1, using five sizes of array. On CPU, the ForCL-generated version achieved the same level of speedup with the hand-tuned OpenCL one (-1.5% to +1.1% difference). The SV workload achieved an average of 54.2 speedup. However when offloading the computation to GPUs, the hand-tuned version outperformed the ForCL generated one for 6.1% on the smallest array size; and having similar performance on the larger sizes. In average, the ForCL-generated code accelerated for 127.3 folds similar to the hand-tuned one (126.1 folds).

The performance of the SV workload on Platform #2 was an unexpected surprise. As shown in Fig. 8, when using CPU as the device, both versions of the workload outperformed the Xeon Phi results. The reason can be that the maximum speed of CPU outperforms that of the Intel Xeon Phi. The baseline performance was the sequential version of this workload running on this CPU. Thus, the lower speedups were observed on Platform #2 in comparison to the results of Platform #1. In average, the ForCL-generated workload obtained very competitive speedup to those of the hand-tuned one i.e., having the difference of speedup from -9% to +5% on CPU and ±3% on Xeon Phi. The smaller of performance gap was seen in the larger data sizes of array.

Stochastic nature of the PI workload impacted the level of acceleration obtained from both Platform #1 and Platform #2. As shown in Fig. 9 and Fig. 10, the range of speedup obtained from the accelerated PI versions is 2.9–51.2 folds. On Platform #1, the ForCL-generated codes achieved the same speedup to the hand-tuned one, with -6% differences in the smallest data size of array.

More variety of performance was observed from the results of Platform #2. As shown in Fig. 10, the speedup of accelerating the PI workload on Xeon Phi was rising constantly in responding to the data size. The results of CPU show more aggressive speedup, yet drop to half after the array size reached five million. This is when the working set size goes beyond the cache limit. Thus, constant overhead from cache misses was observed; and the speedup of CPU remains constantly low at 21 folds.

3.3. Discussion

The ForCL generated code on embarrassingly parallel workloads seems to perform similar to the hand-tuned OpenCL code. However, the performance is not portable and still not scalable. The performance of
generated code of PI only gains 3x speedup on CPU of Platform #1. This can be because of all elements of the container have been allocated in the host memory before transferring to the device. Thus, the more suitable memory optimization for offloading STL containers to accelerator should be investigated in the future work.

The variety of performance results obtained from Xeon Phi may due to the software scheduling overhead used in the OpenCL runtime. As mentioned in [3], another technique for maximize the performance of Xeon Phi, the working set size should be divisible by 16. However, this device specific tuning is not covering in the ForCL compiler yet. The more detail study on the cause of performance variation found on Xeon Phi is needed.

Fig. 5. Performance of the sequential SV workload on two platforms.

Fig. 6. Performance of sequential PI workload on two platforms.

Fig. 7. Speedup of the accelerated versions of SV on Platform#1.

Fig. 8. Speedup of the accelerated versions of SV on Platform#2.

Fig. 9. Speedup of the accelerated versions of PI on Platform#1.

Fig. 10. Speedup of the accelerated versions of PI on Platform#2.
4. Related Works

In the past decade, a number of research studies have made GPUs more accessible for general purpose programming. Several directive-based compilers used for generated GPU or OpenCL code have been proposed. These compilers are based on (a) custom directives [14], [15]), (b) the standard OpenCL directives for multicores [7], [9], or (c) the standards directives for heterogeneous systems like OpenACC, HMPP [5], [8], [16]-[19]. Apart from directive-based technique, extending an existing programming languages like Java to express heterogeneous computing is also proposed in Lime [6].

To extract from unrelated literatures how much speedup can be achieved by the compiler-generated code for heterogeneous systems, the related works mentioned were analyzed using the meta-analysis method [20]. Fig. 11 depicts the forest plot showing the speedup of compiler-generated code for GPU and heterogeneous systems obtained from six literatures. The square shows the mean value. The size of the square depicted the weight (the number of workloads used in the literature is used as its weight, ranging from 13 to 5 workloads).

Fig. 11(a) presents the performance of generated code by using normalized speedup in comparison to the hand-tuned CUDA or OpenCL. The latter two plots show the performance of all directive-based compiler generated code mentioned in [7], [8]. The results show that the generated code perform 0.2–3 times faster relative to hand-tuned heterogeneous code. The majority of work obtained similar results to the hand-tuned code (having speedup nearly 1). Fig. 11(b) presents the speedup obtained from the compiler-generated code in comparison to the sequential CPU code. The results of Griffon [14] and LIME are the performance of their generated code. However, the result of [16] in the plot represents the performance of all compiler-generated code mentioned in the paper. Overall, the range of speedup gain from generated code is 12–135 folds with reference to sequential CPU. The LIME generated code show the highest speedup obtained over the Java byte code on nine workloads at the average of 88.97× (with ±46.5 CI). Among the literature reviewed, this work show the highest varieties of performance observed on different workloads.

The results achieved by these works were sufficiently encouraging for further efforts to improve code generation for heterogeneous systems towards single code for any accelerators. The LambdaJIT project [10] has highlighted the opportunity to accelerate lambda functions in C++11 using both online and offline modes.

The ForCL generated code achieved the similar range of performance from previous work (2.9–135×), on two embarrassingly parallel workloads. Thus, this confirms the potential of exploiting loop-level parallelism by using range-based loop acceleration. However, more realistic workloads such as those from the SHOC benchmark [21] should be used to evaluate the performance of the generated code.

5. Conclusion

Using directive techniques for accelerating loops on heterogeneous systems, object-oriented programs can be hundreds of folds faster with a short learning curve required from programmers. Despite strong research findings on compiler support for loop parallelization, the opportunity to accelerate a compact construct like range-based loops has not yet been investigated. The paper has addressed this issue and makes three key
contributions. First, the magnitude of performance gain by using loop acceleration obtained via a systematic review has been reported. Second, a compiler support for accelerating range-based loops using OpenCL has been described. Third, the performance study of the range-based loop acceleration has been summarized. The results show that, accelerating traditional loops using directive techniques has achieved an average of 12-135 folds speedup over sequential CPU. The preliminary results of the proposed compiler show the speedup of 2.9-135 folds achieved from accelerating range-based loops. This highlights that accelerating range-based loops is viable and could give the similar performance to that of traditional loop acceleration. The ForCL compiler will be further evaluated on more realistic workloads. This is to find the opportunity for optimization to achieve the portable performance on heterogeneous systems.

Acknowledgment

The authors wish to thank Assistant Professor Dr. Putchong Uthayopas at Kasetsart University for the access to the Intel Xeon Phi server and his valuable suggestion. We also thank Dr. Prapaporn Rattanatamrong for the support and valuable discussion. We thank the Department of Computer Science for the laboratory facilities.

References

Worawan Marurngsith received the B.Sc. degree in computer science from Thammasat University, Thailand in 1996, and received the M.Sc. degree in computer science and the Ph.D degree from the University of Edinburgh, UK in 1999 and 2006 respectively. Her post graduate studies were sponsored by the Faculty of Science and Technology and Thammasat University. She is currently a full-time lecturer at the Department of Computer Science, Faculty of Science and Technology, Thammasat University in Thailand, and a founding member of the Parallel Computing and Performance Evaluation Laboratory. Her current research interests include discrete-event simulation techniques for performance evaluation of computer systems, application of high-performance computing, large scale agent-based simulations, compiler techniques for heterogeneous systems, and computers for education.

Chaturapat Suwancharoen has received the B.Sc. degree in computer science from Rangsit University, Pathum Thani, Thailand, in 2008. He is currently working towards the M.Sc. degree in computer science from Thammasat University, Pathum Thani, Thailand. He had worked at Strategic Software Solutions Co., Ltd., Nonthaburi, Thailand, as a software developer. His current research interests include parallel processing, GPU computing and web/mobile development for business.