Automatically Parallelizing Compiler Cooperative OSCAR Vector Multicore

Keiji Kimura, Kazuki Fujita, Kazuki Yamamoto, Tomoya Kashimata,
Toshiaki Kitamura and Hironori Kasahara
Department of Computer Science and Engineering, Waseda University
27 Waseda-cho, Shinjuku-ku, Tokyo, Japan
Email: keiji@waseda.jp, {kazuki_fujita,kyamamoto,kashi,kitamura}@kasahara.cs.waseda.ac.jp, kasahara@waseda.jp

I. Introduction

The importance of vector computation has been still increasing in embedded area as well as HPC area because of the widely spread machine- and deep-learning applications. Particularly, autonomous driving cars require rich vector computation ability to realize highly accurate surrounding environment recognition with low power consumption.

To implement a learning application in an embedded system, only providing rich vector computation power is not enough. The computation result is usually used in other information processing part in the system such as decision and operation parts. These parts are not always vector intensive. In other words, scalar processors and vector processors must be integrated, and they must cooperatively work in a system. However, utilizing such a heterogeneous system has introduced program difficulty.

To overcome this problem, we have developed the OSCAR compiler cooperative vector multicore processor, named OSCAR Vector Multicore. Each core in this multicore has its own vector accelerator core. The compiler automatically exploits hierarchical parallelism from a source program. Then, it assigns coarse grain tasks to CPU cores and inner loop parallelism in a task to a vector processor attached to a CPU core.

In the later part of this paper, we will describe the overview of the OSCAR vector multicore and its compilation flow. Then, the evaluation result will be shown.

II. OSCAR Multigrain Parallelizing Compiler

This section gives an overview of the OSCAR multigrain parallelizing compiler [1], [2]. Here, multigrain means three kinds of parallelism: (1) loop iteration level parallelism also used by conventional parallelizing compilers, (2) coarse grain task parallelism among loops or loops and subroutine calls, and (3) statement level near fine grain parallelism.

In hierarchical coarse grain task parallel processing on the OSCAR compiler, a source C or Fortran sequential program is decomposed into three kinds of coarse grain tasks, or Macro Tasks (MTs), such as basic block (BPA), loop (RB), and subroutine call (SB). After generating Macro Tasks, the compiler analyzes both control flow and data dependencies among Macro Tasks, and represents them as a Macro Flow Graph (MFG). Next, the compiler applies the earliest executable condition analysis, which can exploit parallelism among Macro Tasks considering with both control dependencies and data dependencies. The analysis result is represented as a Macro Task Graph (MTG).

If a Macro Task is a subroutine call or a loop that has coarse grain task parallelism, the compiler generates inner Macro Tasks inside that Macro Task hierarchically. When an MT is a parallelizable loop (doall or forall loop), the compiler decomposes it into multiple MTs to utilize its parallelism as coarse grain task parallelism, or execute it in loop iteration level parallel processing. When an nested loop has parallelism in both of the outer-most nest level and the inner-most nest level, the outer level parallelism can be processed by multiple cores and the inner level parallelism can be processed by a vector accelerator in each core.

After exploitation of the parallelism, when an MTG in a source program has no conditional branch, MTs in it are statically scheduled, otherwise dynamic scheduling is employed. Then the compiler generates the parallelized C or Fortran program as a result. It contains OSCAR API directives for parallel processing [3]. OSCAR API is defined based on OpenMP [4], therefore an OpenMP compiler can generate a parallelized executable binary. If the local memory optimization and the power control optimization by the OSCAR compiler are required, an API translator is employed in front of an back-end compiler like gcc and llvm to translate the directives into runtime library calls and target specific directives .

III. OSCAR Vector Multicore[5]

Fig. 1 depicts an overview of the OSCAR vector multicore architecture. Each core in a chip contains a CPU, an instruction cache (I-CACHE), a data cache (D-CACHE), a local data memory (LDM), a vector accelerator (Vector), and a data transfer unit (DTU).

An LDM is shared by a CPU and a Vector in a core. An LDM can be accessed by a CPU in a different core. A CPU can access an I-CACHE and a D-CACHE in addition to an LDM as well as an on-chip centralized shared memory (CSM) and an off-chip CSM, or a main memory. On the other hand, a Vector can access only an LDM in a same core. This limitation can prevent a Vector from exposing its required

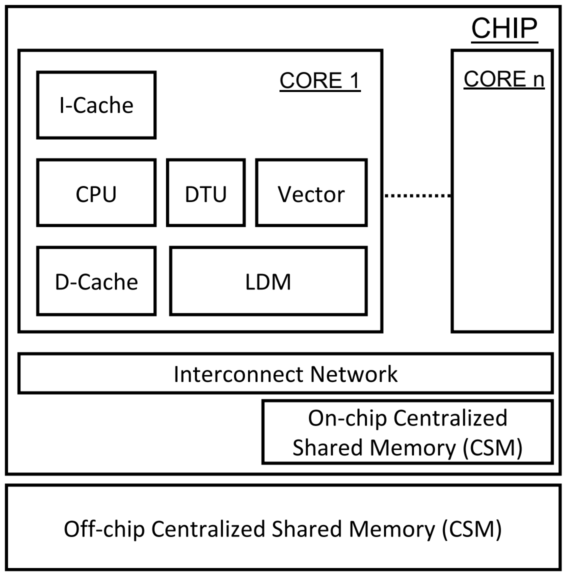
CHIP
CORE 1
I-Cache
CPU
DTU
Vector
D-Cache
LDM
CORE n
Interconnect Network
On-chip Centralized Shared Memory (CSM)
Off-chip Centralized Shared Memory (CSM)

Fig. 1. Overview of OSCAR Vector Multicore

memory bandwidth outside a core, and results in saving a cost for the memory modules.

A DTU is a kind of a direct memory access controller (DMAC). It performs data transfer among an LDM in a core, LDMs in other cores, an on-chip and an off-chip CSMs. A CPU, a Vector and a DTU can be processed simultaneously to improve their utilization. The OSCAR compiler schedules tasks on them to overlap their execution as much as possible.

Fig. 2 depicts an overview of a Vector in a core. It contains a vector pipeline and a scalar pipeline. “vadd/vsub”, “vmul”, “vdiv” are a vector function units (FUs) for additions and subtractions, multiplications, and division, respectively. Similarly, “add/sub”, “mul”, and “div” are a scalar FUs for them. The Vector fetches its instructions from the LDM in a same core. Scalar and vector memory accesses are also performed on the LDM.

The scalar pipeline supports scalar integer and floating point operations and, has 32 integer registers and 32 floating-point registers. It can work as a simple in-order scalar pipeline.

The vector pipeline has a vector register file and a mask register file. The pipeline consists of 8 lanes for double precision elements, which means 8 double precision operations can be performed at a time.

The capacity of a vector register file is 32 registers $\times$ 256 bytes. A vector register can have different number of elements according to the element size: For instance, when an element size is 32 bit (4 byte) floating point, the number of elements becomes 64. Similarly, when an element size is 8 bit (1 byte) integer, the number of elements becomes 256. The vector length for vector operations can be changed dynamically. The pipeline also supports a chaining execution to reduce the vector instruction execution overhead when a set of two successive instructions has a read-after-write (RAW) dependency.

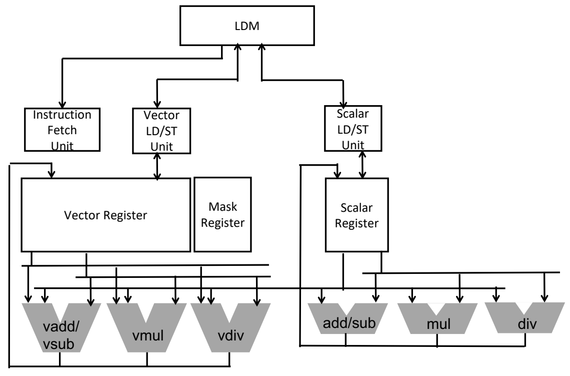

Fig. 2. Overview of Vector Accelerator Module

IV. Compilation Flow of OSCAR Vector Multicore

Fig.3 depicts an overview of the compilation flow of the OSCAR vector multicore processor. It consists of the OSCAR compiler, the host CPU compiler, the vector processor compiler based on Clang/LLVM compiler [6], and a linker to generate parallel executable binary file from the object files for the host CPU compiler and the vector processor compiler.

Firstly, the OSCAR compiler takes a sequential C program as a source file. It exploits multigrain parallelism from that program. Simultaneously, the compiler detects vector parallelizable loops from the program and separates them into newly prepared functions. The compiler generates a multigrain parallelized C program for the host CPU. In addition, the compiler also generates a vector parallelized C program for the vector accelerator. Note that the vector operations are represented as C intrinsic functions defined for the vector accelerator.

Then, the host compiler takes a multigrain parallelized C program and generates an object file for the host CPU. The generated multigrain parallelized C program is parallelized by OSCAR API as explained in Section II.

Similarly, the vector compiler takes a vectorized C program and generates an object file for the vector processor. We extended the Clang/LLVM compiler for this purpose. It can recognize vector intrinsic functions, which covers the vector instruction set of the OSCAR vector multicore. It also employs basic compiler optimizations, such as loop unrolling and redundant instruction removing, as well as instruction scheduling.

Finally, the linker generates a parallelized executable binary file from the object files for the host CPU and the vector processor. To keep the portability for different kinds of host CPU processors, the object file for the vector processor is formed as array variable objects for the host CPU. In other words, the vector instructions and initialized data are embedded as array objects for the host CPU. Thus, we can easily replace the host CPU architecture without modifying the vector processor compiler, the assembler and the final linker.

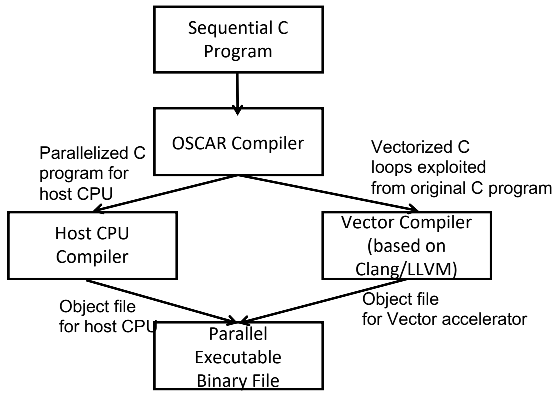
Sequential C Program
OSCAR Compiler
Parallelized C program for host CPU
Vectorized C loops exploited from original C program
Host CPU Compiler
Vector Compiler (based on Clang/LLVM)
Object file for host CPU
Object file for Vector accelerator
Parallel Executable Binary File

Fig. 3. Overview of Compilation Flow

V. Preliminary Evaluations

We implemented the OSCAR vector multicore on an Intel Arria10 SoC FPGA chip. We used NIOS II with FPU as a CPU core in the evaluated systems. The clock frequency for the CPU used in this evaluation is 50MHz. About the vector processor, 16 single precision operations can be performed at a cycle. The clock frequency for the vector processor is 40MHz. The bandwidth of LDM to the vector processor is 32 byte/clock cycle.

In this preliminary evaluation, the LDM size is 1MB to show the performance of the vector processor module. Therefore, all data is located on the LDM instead of off-chip DDR memory module. Similarly, only one CPU and one vector processor are used.

We used a matrix-multiply (MM), a 2D-convoluion (2D-Conv), 1D-FFT (FFT), and Cholesky decomposition (Cholesky) in this evaluation. The data size of each program is as follows: 256×256 for MM, 256×256 for 2D-convolution, 2048 for FFT, and 64×64 for Cholesky. All of them used single precision (32bit) data elements.

Fig. 4 shows the evaluation result. Each bar shows the speedup obtained by the vector processor compared with the CPU. The vector processor achieves $25.01\times$ speedup at maximum for 2D-conv, $9.57\times$ speedup at minimum for FFT, and $16.17\times$ speedup on average. The performance improvement depends on the available vector length exploited from the target kernel loops. More performance improvement can be expected when multiple cores are available.

VI. Conclusion

This paper introduced the OSCAR Vector Multicore. This multicore consists of multiple cores, each of which has a vector accelerator beside a CPU core and a data transfer unit (DTU). The OSCAR compiler controls them to to provide sufficient data to the vector accelerator, as well as the exploitation of the parallelism hierarchically from a source program. A preliminary evaluation was conducted on the OSCAR Vector Multicore implemented on an FPGA evaluation board. We used four kernel loop programs to evaluation the performance improvement by the vector accelerator module compared with

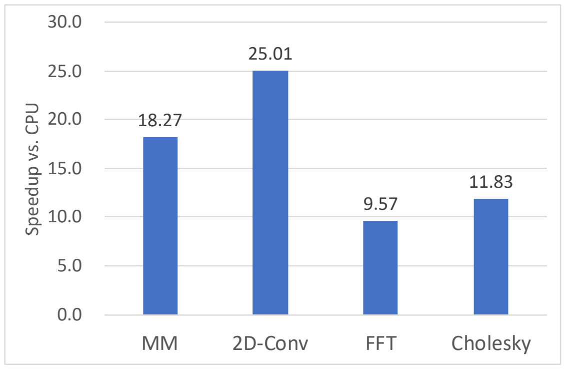

Fig. 4. Performance Improvement by Vector Processor

an CPU core. The evaluation shows the vector accelerator achieves $25.01\times$ speedup at maximum and $16.17\times$ speedup on average.

Acknowledgement

Part of this paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita, "A multi-grain parallelizing compilation scheme for oscar (optimzally scheduled advanced multiprocessor)," in *Proc. 4th Intl. Workshop on LCPC*, August 1991, pp. 283–297.

[2] M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka, and H. Kasahara, "Hierarchical parallelism control for multigrain parallel processing," in *Proc. 15th Intl. Workshop on LCPC*, August 2002.

[3] K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako, and H. Kasahara, "Oscar api for real-time low-power multicores and its performance on multicores and smp servers," *Lecture Notes in Computer Science*, vol. 5898, pp. 188–202, 2010.

[4] The openmp api specification for parallel programming. [Online]. Available: https://www.openmp.org/

[5] T. Kashimata, T. Kitamura, K. Kimura, and H. Kasahara, "Cascaded dma controller for speedup of indirect memory access in irregular applications," in *9th Workshop on Irregular Applications: Architectures and Algorithms*, November 2019.

[6] The llvm compiler infrastructure. [Online]. Available: http://www.llvm.org/