Predicate-Based Filtering for Multi-GPU Utilization in Directive-Based Programming

Kazuaki Matsumura, Simon Garcia de Gonzalo, Antonio J. Peña
Barcelona Supercomputing Center, Barcelona, Spain
E-mail: {kazuaki.matsumura, simon.garcia, antonio.pena}@bsc.es

Keywords—Multi-GPU, OpenACC, Compiler, Code Generation.

I. Extended Abstract

Designing and building supercomputers is a complex task in the field of high-performance computing (HPC). The hardware, middleware and algorithms need to effectively collaborate to achieve ideal results for massive and practical problems. To facilitate the easy usage of supercomputers, compiler technologies have been developed with highly automated program optimizations that use domain-specific knowledge and understandings of target architectures [1].

Directive-based programming has been employed for enabling accelerator use, while replacing vendor-specific coding with directive insertion. Keeping software portability with minimum engineering efforts upon sequential code, OpenACC and OpenMP are now widely used for accelerator programming [2], [3]. However, pursuing ideal performance is often challenging. The bare insertion of directives by the programmers exposes less program characteristics for the compilation; thus, programmers aiming at better efficiency are forced to reshape their code merely for adjusting to the environment such as compilers, software stacks and heterogeneous architecture.

While keeping the productivity, our research extends OpenACC to exploit further optimization opportunities. In a portable fashion that relies on other compilers, our approach provides an environment which enables dynamic analysis of computation and perform on-the-fly kernel specialization. Considering the high memory latency of GPUs, we add a novel code-translation technique named *predicated-based filtering* to automate multi-device utilization. We never split loop ranges nor introduce fine dependency analysis, but divide data ranges to be updated on each device. This idea allows to distribute highly-tuned code without changing code structure nor parallelism.

A. JACC: Runtime-Extended OpenACC

We build JACC, a just-in-time compilation system for OpenACC, in which input directives are replaced with runtime routines. JACC hides every OpenACC feature behind a provided runtime library to cushion dependency to specific compilers. Once a kernel is compiled at first execution, its device code is cached to be reused for subsequent launches. Even though JACC is developed upon existing compilers, it allows calling of CUDA routines and kernels through its library. Fig. 2 shows the converted code of Fig. 1 to call runtime routines. First, combined directives (e.g. `parallel loop` of Line 2 of Fig. 1) are decomposed into three basic directives of `parallel`, `loop` and `data`. Then, for each directive, JACC inserts corresponding routines that are implemented in its library, shown in Fig. 2 (Lines 2, 5 and 12).

```
1  #pragma acc data copyout(x[0:N]) present(y)
2  #pragma acc parallel loop
3  for(int i=0; i<N; i++) x[i] = y[i] * y[i];
```

Fig. 1. Accelerator programming in OpenACC

```
1  /* Entry of #pragma acc data */
2  jacc_create(x, N * sizeof(float));
3
4  /* #pragma acc parallel loop */
5  jacc_kernel_push(
6    "#pragma acc parallel present(x, y)\n"
7    "#pragma acc loop\n"
8    "for(int i=0; i<N ; i ++) /* ... */",
9     /* args */, /* flags */);
10
11 /* Exit of #pragma acc data */
12 jacc_copyout(x, N * sizeof(float));
```

Fig. 2. Converted code by JACC (arguments omitted)

During the execution, JACC data-related routines that wrap OpenACC routines (Lines 2 and 12 of Fig. 2) assume the roles of the original directives. The routine `jacc_kernel_push` launches kernel execution while accepting source code in a string with arguments that hold runtime information (Lines 5-9 of the same figure). It should be noted that the `loop` directive is used for marking parallelism; therefore, the directive is kept in kernel strings. When the routine finds no compiled kernel for given source code or needs to update existing kernels, function code is generated to emit device code by a specified compiler and to have additional arguments for code extension. After linked dynamically, this function is called through a foreign function interface (FFI). JACC's library for each routine is extended to collect runtime information and support dynamic features.

B. Multi-GPU Utilization with Predicates

Towards further utilization of intra-kernel parallelism, we combine multi-GPU execution with JACC. Whereas previous studies have persistently focused on loop splitting over plural GPUs [4], [5], this work divides data regions that each GPU updates to support real applications that usually entangle memory accesses among loop iterations.

Our technique, named *predicate-based filtering*, limits memory accesses depending on data regions that the GPU writes to, assuming that redundant computational code and parameters do not degrade performance due to low computational latency and high memory latency on GPUs. First, we introduce

```
1  a[i]=x; b[i]=a[i]; x=c[j]; a[k]=x; b[k]=a[k];
```

```
1  /* a[i]=x */
2  ((a_lb<=i && a_ub>=i)||
3    (b_lb<=i && b_ub>=i)) ? a[i]=x:a[i];
4  /* b[i]=a[i] */
5  ((b_lb<=i && b_ub>=i)) ? b[i]=a[i]:b[i];
6  /* x=c[j] */
7  x=((a_lb<=k && a_ub>=k)||
8      (b_lb<=k && b_ub>=k)) ? c[j]:0;
9  /* a[k]=x */
10 ((a_lb<=k && a_ub>=k)||
11   (b_lb<=k && b_ub>=k)) ? a[k]=x:a[k];
12 /* b[k]=a[k] */
13 ((b_lb<=k && b_ub>=k)) ? b[k]=a[k]:b[k];
```

Fig. 3. Example of predicate-based filtering in C code. Original (up) and filtered code (down). References to array `a` have predicates for updating array `b` and itself (Lines 2-3 and 10-11), the references to array `b` have for itself (Lines 5 and 13), and the reference to array `c` has for array `a` and `b` (Lines 7-8).

data ranges for each updated array so that array writes can be filtered based on the assigned range. For instance, in C code, array write `a[i]*=2` is rewritten to `(a_lb <= i && a_ub >= i) ? a[i]*=2 : a[i]`, where `a_ub` and `a_lb` indicate the upper and lower bound of array `a`, that are specified depending on the GPU. In Fortran, since there is no nested assignment, we use IF statement for filtering, with subsequent ELSE statement which contains an assignment of the same expression (`a(i)=a(i)`) that is later optimized away but facilitates compiler analysis. Additionally, we develop data-flow analysis for the innermost parallel region in each kernel to detect data dependencies between arrays. Then, we filter them to restrict accesses while solving dependencies as shown in Fig. 3. This analysis converts both array and variable references into the static single assignment (SSA) form, and iteratively finds dependencies among array accesses.

Data ranges linked to given pointers are tracked through JACC's runtime routines, and managed in a red-black tree as OpenACC compilers do [6]. Data transfers are invoked to send updated data across GPUs after each kernel execution. Device-memory allocations and host-to-GPU communications are replicated on all the GPUs and the primary GPU is used for GPU-to-host transfers. To guarantee the result of our analysis, we check kernel arguments so as to duplicate computation and disable communications on data that are referred through more than two pointers which at least one of them is read and one is written. When several pointers share the same array to update, we merge their access ranges to follow the widest. The necessary computation for array-write indexing is always duplicated. Regarding reduction or variable writes that are explicitly exported to host, we filter the computation based on the range of the outermost parallel iterator.

While being applicable to all OpenACC kernels as far as array writes are concerned, our filtering technique needs to duplicate execution on each GPU when references between split ranges are found inside the kernel. We alleviate this restriction by leveraging dimensional information.

In order to avoid lower performance due to data distribution overheads, we enable multi-GPU execution for each kernel in an adaptive way, while otherwise duplicating computation on all GPUs and performing no GPU-to-GPU communication.

C. Results

We integrate predicated-based filtering into JACC, which translator is implemented as a XcodeML [7] converter. We measure the performance changes of our proposed techniques using the NVIDIA Tesla V100 SXM2 GPUs (16GB Memory) on NVIDIA DGX-1. Fig. 4 shows the partial results of our experiments.

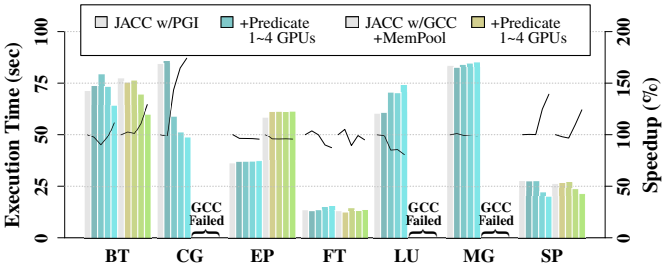

Fig. 4. Performance scaling of predicate-based filtering using NPB with PGI and GCC (top). The blue bars are results of predicate-based filtering with PGI and green bars are with GCC. The execution time is shown by the bar and the speedup by the line.

II. Acknowledgment

We acknowledge the mentorship by a former BSC researcher Leonel Toledo.

References

[1] M. J. Wolfe, C. Shanklin, and L. Ortega, *High Performance Compilers for Parallel Computing*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] T. O. Organization, "Openacc," 2011. [Online]. Available: https://www.openacc.org/

[3] T. O. ARB, "Openmp," 1997. [Online]. Available: https://www.openmp.org/

[4] K. Matsumura, M. Sato, T. Boku, A. Podobas, and S. Matsuoka, "Macc: An openacc transpiler for automatic multi-gpu use," in *Supercomputing Frontiers*, R. Yokota and W. Wu, Eds. Cham: Springer International Publishing, 2018, pp. 109–127.

[5] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama, "Integrating multi-gpu execution in an openacc compiler," in *2013 42nd International Conference on Parallel Processing (ICPP)*. IEEE, 2013, pp. 260–269.

[6] M. Wolfe, S. Lee, J. Kim, X. Tian, R. Xu, B. Chapman, and S. Chandrasekaran, "The openacc data model: Preliminary study on its major challenges and implementations," *Parallel Computing*, vol. 78, pp. 15–27, 2018.

[7] O. C. P. R. CCS), "Xcodeml," 2009. [Online]. Available: https://omni-compiler.org/xcodeml.html

Kazuaki Matsumura Kazuaki Matsumura received the BE degree from University of Tsukuba in 2017 and the MSc degree from Tokyo Institute of Technology in 2019. He is currently working at Barcelona Supercomputing Center while being affiliated with its doctoral program. His research interests include program optimization and compilers for high-performance computing.