Chapter 14

HPC Programming Framework Research Team

14.1 Members

Naoya Maruyama (Team Leader)

Motohiko Matsuda (Research Scientist)

Shinichiro Takizawa (Research Scientist)

Mohamed Wahib (Postdoctoral Researcher)

Satoshi Matsuoka (Senior Visiting Scientist)

Tomoko Nakashima (Assistant)

14.2 Research Activities

We have developed high performance, highly productive software stacks that aim to simplify development of highly optimized, fault-tolerant computational science applications on current and future supercomputers, notably K computer. In this report, we present an overview of one of our high productivity frameworks.

14.3 Research Results and Achievements

14.3.1 A Programming Framework for Hierarchical N-Body Algorithms

Hierarchical tree-based algorithms are often used in simulations such as molecular-dynamics and particle simulations, and especially, algorithms such as Barnes-Hut and Fast Multipole Method (FMM) are the most popular ones in N-body simulations. To assist experiments of such algorithms, a prototype framework called *Tapas* is being developed for distributed memory CPU/GPU computers. Direct interaction calculations, which compute pairwise forces among nearby subsets of the bodies, are still important even in sophisticated tree-based algorithms. Direct interaction calculations are algorithmically simple *all-pairs* operations, but they contribute to a non-negligible part of the total execution time in practice. In some modest-scale experiments, roughly a half of the execution time is consumed in all-pairs operations. It is compute-intensive, and thus, requires state-of-the-art adaptations to particular architectures of varying platforms of CPUs and GPUs.

An all-pairs operation mainly consists of nested loops working on two vectors. There is numerous work on all-pairs implementations on manycore CPUs and GPUs. But, existing implementations have flaws in its interface design, possibly because it is too simple and not seriously designed. Using an example of the force calculation in an N-body simulation, decomposition of an all-pairs operation reveals three loops: one for a pair-wise force calculation, one for a summation of forces to each body, and one for updating vectors of positions and velocities. These are carried out in sequence by a mapping-of-mappings for a force calculation, by a mapping-of-reductions for a summation of forces, and a mapping for updating. Existing implementations ignore the updating part,

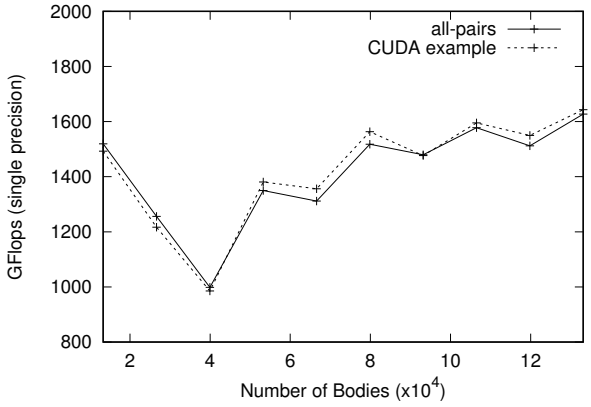

Figure 14.1: Comparison of N-body simulations between the all-pairs code and the CUDA example code on a GPU (K80).

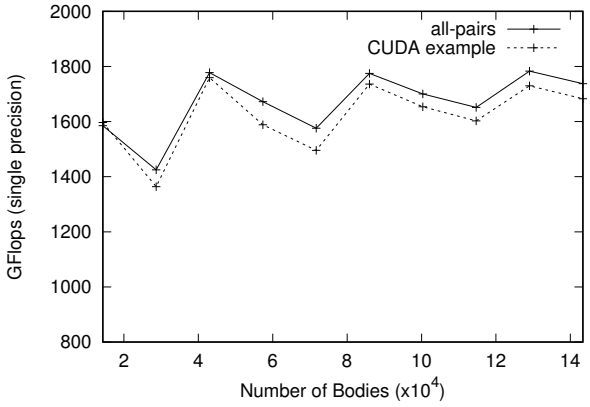

Figure 14.2: Comparison of N-body simulations between the all-pairs code and the CUDA example code on a GPU (K20Xm).

and it is implicitly performed in the summation. The updating part is a non-nested single loop, and it was thought to be performed simply. It is not that simple. It is known that the loops for a mapping-of-mappings and a mapping-of-reductions can be fused. However, it is known that the loops for a mapping for updating and a mapping-of-reductions cannot be fused. Thus, the design of the all-pairs operation should consider the updating part explicitly. Adding an explicit mapping for updating increases flexibility in adopting the all-pairs operation in the application code. It makes, for example, the code simple in absorbing the mismatch of the representation of vectors of positions and velocities, such as array-of-structures (AoS) or array-of-structures (SoA), which are specific to the applications.

It is important to keep the performance critical code to be abstract and compact. Although an all-pairs could be performed by a composition of mappings and reductions, the CUDA language for GPUs does not well admit such nested compositions of mappings and reductions due to its host-device distinction and difficulty in using the fast scratchpad memory. Thus, the all-pairs operation should be defined as a single fused-form, which is important for performance. Making a properly designed interface also facilitates direct comparison with existing fast reference implementations such as given in the CUDA examples.

In the evaluation, the performance of a direct N-body simulation are compared between the reference code in the CUDA examples and the code using the all-pairs. In addition, the performance of an FMM N-body simulation are compared between the hand-written code and the code using the all-pairs. These performance indications support the suitability of the all-pairs interface. The evaluation is done with using two GPU platforms K80 and K20Xm, both with the Kepler micro-architecture. K80 has 13 SMs (streaming multiprocessors) with micro-architecture revision 3.7, and K20Xm has 14 SMs with micro-architecture revision 3.5. K80 runs at 823 MHz and its memory at 2505 MHz, while K20Xm runs at 732 MHz and its memory at 2600 MHz. K80's memory is a bit slower. The host for K80 is Intel Xeon E5-2698 2.3 GHz, and the host for K20Xm is Intel Xeon E5-2670 2.6 GHz. The compilers are CUDA 8.0 and Intel ICC 16.0.4 both for K80 and K20Xm.

The first evaluation uses the direct N-body simulation code in the CUDA examples. The purpose of this evaluation is to check the overhead of abstraction is small enough, because the all-pairs implementation has a very similar structure to the code in the CUDA examples. Figure 14.1 shows the performance on K80, and Figure 14.2 shows the performance on K20Xm, which compare the performance of Tapas's all-pairs and the code in the CUDA examples. The measurements were run using a single GPU. The floating-point numbers are single-precision in this benchmark. GFlops is calculated as 20 operations per interaction in the same way as the CUDA examples. Note that the different steps for the number of bodies (X-axis) are used to make them multiples of the number of SM's. The step is 13×1024 for K80, and 14×1024 for K20Xm. The appearances of the code are very different, because the main parts of the calculations are extracted as function objects in the all-pairs case but they are laid out in the code in the CUDA examples. However, the both code has very similar structures in the GPU kernels, and the difference in performance is very small. K80 is generally slower than K20Xm in our runs, but we did not investigate the reasons.

The second evaluation compares an FMM N-body simulation between Tapas's original hand-written code and the code using the all-pairs. Figure 14.3 and Figure 14.4 show the GPU performance, and Figure 14.5 shows the CPU performance. Two GPUs on a single host are used in this benchmark, because Tapas runs

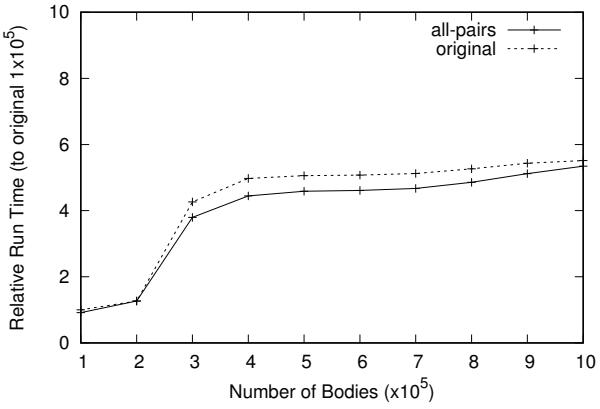

Figure 14.3: Comparison of Tapas runs between the all-pairs code and the hand-written code (original) on GPUs (using two K80).

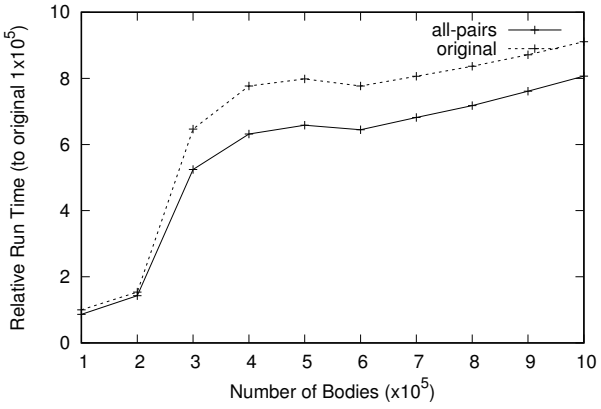

Figure 14.4: Comparison of Tapas runs between the all-pairs code and the hand-written code (original) on GPUs (using two K20Xm).

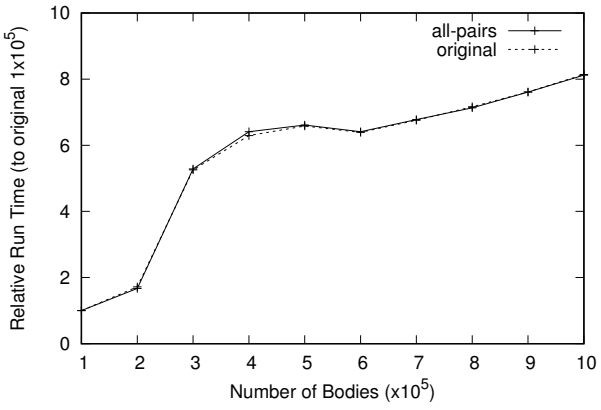

Figure 14.5: Comparison of Tapas runs between the all-pairs code and the hand-written code (original) on CPUs (Xeon E5-2670).

with at least two processes. The measurements in the benchmarks are on the first round of interactions, where the bodies are uniformly distributed initially. Both of the all-pairs code and the hand-written code (labeled as original) show similar performance, obviously because the all-pairs is the redesign of the interface and the difference in the code fragments is very small. However, unexpectedly, there is some observable speed-up in performance. It is unintended, but there are actually some differences in the register usage. The report of *ptxas info* (the assembler of GPU) from the CUDA compiler tells "Used 46 registers, 436 bytes cmem[0], 44 bytes cmem[2]" for the all-pairs code, while "Used 40 registers, 400 bytes cmem[0], 44 bytes cmem[2]" for the original code. The reports are the same for the both architectures of K80 and K20Xm.

We introduced a fused-form of the all-pairs operation which takes an explicit update function. The restricted execution model of CUDA prevents expressing nested calls as a composition of a mapping of mappings or a mapping of reductions. Thus, an all-pairs operation needs to be defined as a single operation. The explicit update function is suggested by considerations of algorithmic skeletons, where it in general cannot be fused the compositions of a function application to the result of a reduction. Recent programming in C++ is approaching to functional programming, where the important aspect is composability. However, building libraries with composable functions is still difficult. In introducing the all-pairs operation, we make it generalized to compensate the loss of composability of mappings and reductions in CUDA. Systematic development with the help of algorithmic skeletons has been successful in this regard. It provides a way to an appropriate design of data-parallel libraries in C++.

14.4 Schedule and Future Plan

Over the last five years, our team has developed tools and frameworks for simplifying developing high performance applications on large-scale computing systems. While our team is now closed, we hope that our software as well as the research results presented as peer-reviewed papers will continue to contribute to productive high performance computing.

14.5 Publications

14.5.1 Conference Papers

[1] Motohiko Matsuda, Keisuke Fukuda, Naoya Maruyama. "A Portability Layer of an All-pairs Operation for Hierarchical N-Body Algorithm Framework Tapas", In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018 (2018).

[2] Shinichiro Takizawa, Motohiko Matsuda, Naoya Maruyama, Yoshifumi Nakamura. "A Scalable Multi-Granular Data Model for Data Parallel Workflows", In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018 (2018).

14.5.2 Software

[3] KMR version 1.8.1 (April 2016). URL: http://mt.r-ccs.riken.jp/kmr.