Parallel Computing 120 (2024) 103081

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Analyzing the impact of CUDA versions on GPU applications

Kohei Yoshida [*], Shinobu Miwa, Hayato Yamaki, Hiroki Honda

The University of Electro-Communications, 1-5-1, Chofugaoka, Chofu, Tokyo, Japan

ARTICLE INFO

Keywords:
CUDA toolkit versions
Evaluation
Performance
Power consumption
Energy consumption

ABSTRACT

CUDA toolkits are widely used to develop applications running on NVIDIA GPUs. They include compilers and are frequently updated to integrate state-of-the-art compilation techniques. Hence, many HPC users believe that the latest CUDA toolkit will improve application performance; however, considering results from CPU compilers, there are cases where this is not true. In this paper, we thoroughly evaluate the impact of CUDA toolkit version on the performance, power consumption, and energy consumption of GPU applications with four GPU architectures. Our results show that though the latest CUDA toolkit obtains the best performance, power consumption, and energy consumption for many applications in most cases, but we found a few exceptions. For such applications, we conducted an in-depth analysis using the SASS to identify why older CUDA toolkit achieve performance improvement. Our analysis showed that the factors that caused them are by three phenomena: aggressive loop unrolling, inefficient instruction scheduling, and the impact of host compilers.

1. Introduction

In the field of high-performance computing (HPC), CUDA toolkits [1] are widely used to develop applications running on NVIDIA GPUs. They include compilers (e.g., NVCC) and runtime systems, and are frequently updated to both integrate state-of-the-art compilation techniques and handle state-of-the-art GPU architectures. For this reason, many HPC users believe that the latest CUDA toolkit should be used to maximize application performance.

However, there are cases where this is not true. Fig. 1 exemplifies this. The box plots show the distribution of the slowdown of the 32 kernels executed using the eight older CUDA toolkit versions with respect to the latest version (i.e., CUDA12M) on the Tesla P100. The details of our experimental methodology are described in Section 3. Because many compilation techniques have both advantages and disadvantages, there are cases in which an advanced compilation technique is unsuitable for some classes of GPU applications. In such cases, the object code generated from an application code with an older CUDA toolkit performs (up to 20%) better than one generated with the latest toolkit. Therefore, it is important to understand the impact of the optimizations applied by each CUDA toolkit version (hereafter referred to as CUDA version) of the compiler and to select the CUDA version that maximizes the application performance. There are many studies that investigate the impact of compiler versions on the performance, power consumption, and energy consumption of CPU applications (e.g., [2]), but for GPUs, to the best of our knowledge, no previous work has focused on this issue.

In this study, we thoroughly evaluated the impact of CUDA versions on the performance, power consumption, and energy consumption of GPU applications with various GPU architectures, since application performance is affected not only by the compiler but also by the hardware. More specifically, we compiled 32 kernels selected from 19 CUDA applications for four different GPU architectures using nine CUDA versions, and then measured the performance, power consumption, and energy consumption of each object code. We evaluated the impact of CUDA versions by comparing the performance, power consumption, and energy consumption of GPU applications for each object code. Furthermore, for some cases where the older versions were superior, we conducted an in-depth analysis of each object code at the SASS [3] level to identify the optimizations that affected the improvement in performance. As the performance improvement of their applications is of significant interest to many HPC users, our study emphasizes performance improvements.

The main contributions of this paper are summarized as follows.

- The use of newer CUDA versions is preferable in many cases. For example, the oldest CUDA version yields a slowdown of 1.16x on the Tesla P100 GPU compared to the latest CUDA version, and this slowdown gradually decreases as the CUDA versions are updated.
- The differences in the metrics obtained by the CUDA versions decrease from 12% to 1% as the GPU architecture advances.
- The difference in power consumption caused by the CUDA version is very small on all GPUs (only ±2%).

* Corresponding author.
E-mail address: yoshida@hpc.is.uec.ac.jp (K. Yoshida).

https://doi.org/10.1016/j.parco.2024.103081
Received 29 August 2023; Received in revised form 18 December 2023; Accepted 24 February 2024
Available online 29 February 2024

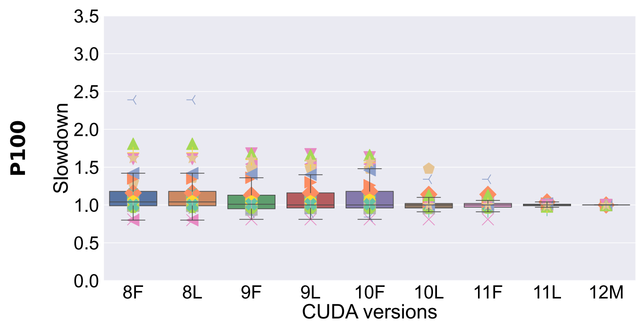

Fig. 1. Slowdown with respect to CUDA12M on the Tesla P100.

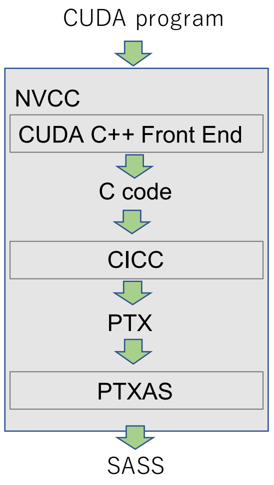

Fig. 2. Compilation flow of CUDA programs.

- We demonstrated that it is possible to optimize compiler options to reduce the performance gap between object codes generated by the latest and older CUDA versions, even when execution based on the latest CUDA version is slower.
- We demonstrated that the performance of object code was affected by not only the version of CUDA but also the version of the C++ compiler in some cases.

The remainder of this paper is organized as follows. First, we describe NVCC, a CUDA compiler, in detail in Section 2. In Section 3, we describe the experimental setup to measure the performance and power and energy consumption of a GPU. In Section 4, we present our experimental results and provide a detailed analysis of CUDA kernels for cases where execution with an older CUDA version is faster. We summarize related work in Section 5 and present the conclusions of this study in Section 6.

2. NVCC

NVCC, which is available in a CUDA toolkit, is a compiler for CUDA programs. It consists of three (front-end, intermediate, and back-end) compilers, as shown in Fig. 2. The NVCC compile process for a given device code is made up of the following three steps.

1. First, the CUDA C++ front-end compiler generates C code from the device code.
2. Second, the intermediate compiler, called the CICC, generates PTX (Parallel Thread eXecution) code, which includes a low-level parallel thread execution virtual machine and instruction set architecture, from the C code.
3. Finally, the back-end compiler, called PTXAS, generates SASS code, which is CUDA assembly code, from the PTX code.

Each of CICC and PTXAS has its own optimizer implemented differently depending on CUDA versions. Thus, NVCC generates different PTX and SASS codes from an identical device code depending on the CUDA versions. As a result, the object codes generated with different CUDA versions often show differences in performance and power and energy consumption.

3. Experimental methodology

3.1. GPU architectures

The experimental systems considered in this study are summarized in Table 1. We evaluated four generations of GPUs (i.e., Tesla P100, NVIDIA V100, NVIDIA A100, and NVIDIA H100), each with a different architecture.

3.1.1. Tesla P100

The Tesla P100 GPU [4], based on the Pascal architecture, was announced in April 2016. It contains 56 streaming multiprocessors (SMs), each with 64 CUDA cores that can perform both FP32 and INT32 operations. Thus, the P100 chip has a total of 3,584 CUDA cores. Unlike the NVIDIA V100 and A100, the Tesla P100 has no tensor core. The Tesla P100 GPU we use has a peak FP32 (FP64) performance of 10.6 (5.3) TFLOPS and 16 GB HBM2 with a bandwidth of 732 GB/s. Thermal design power (TDP) is 300 W. We used a Tesla P100 GPU in the TSUBAME3.0 supercomputer [5] for our experiment.

3.1.2. NVIDIA V100

The NVIDIA V100 GPU [6] adopts the Volta architecture and was announced in May 2017. The GPU has 84 SMs and 5,376 FP32 cores in total (i.e., each SM has 64 FP32 cores). In NVIDIA V100 GPUs, one CUDA core is split into separate FP32 and INT32 cores, allowing FP32 and INT32 operations to be performed concurrently. In addition, tensor cores have been added to the SMs in these GPUs. The NVIDIA V100 GPU we use has a 32 GB HBM2 with 900 GB/s and a peak FP32 (FP64) performance of 15.7 (7.8) TFLOPS. The TDP of the NVIDIA V100 GPUs is 300 W. We used an NVIDIA V100 GPU in the Flow Type-II supercomputer [7] for our experiment.

3.1.3. NVIDIA A100

The NVIDIA A100 GPU [8] is based on the Ampere architecture and was announced in May 2020. This GPU has 108 SMs and 6,912 FP32 cores in total. Separate FP32/INT32 and tensor cores are implemented on NVIDIA A100 GPUs, just as they are on NVIDIA V100 GPUs. The NVIDIA A100 GPU we use has a 40 GB HBM2 with 1,555 GB/s. The peak FP32 and FP64 performance is 19.5 and 9.7 TFLOPS, respectively and TDP is 400 W. An NVIDIA A100 GPU in the Wisteria-A supercomputer [9] was used.

3.1.4. NVIDIA H100

The NVIDIA H100 GPU [10] is based on the Hopper architecture and was announced in March 2022. This GPU has 114 SMs and 14,592 FP32 cores in total. Separate FP32/INT32 and fourth generation tensor cores are implemented on NVIDIA H100 GPUs, just as they are on both NVIDIA V100 and NVIDIA A100 GPUs. The NVIDIA H100 GPU we use has a 80 GB HBM2 with 2039 GB/s. The peak FP32 and FP64 performance is 51.2 and 25.6 TFLOPS, respectively. TDP is 350 W. An NVIDIA H100 GPU in the Pegasus supercomputer [11] was used.

3.2. Benchmarks

We used a variety of single-GPU programs selected from three benchmark suites for our experiments. The 19 benchmark programs

Table 1
GPU architectures under consideration.

Product name	# of SMs	# of Cores	TFLOPS (FP32)	TFLOPS (FP64)	Memory (GB)	TDP (W)
Tesla P100	56	3,584	10.6	5.3	16	300
NVIDIA V100	84	5,376	15.7	7.8	32	300
NVIDIA A100	108	6,912	19.5	9.7	40	400
NVIDIA H100	114	14,592	51.2	25.6	80	350

Table 2
Benchmark programs.

Suites	Program name	Code description	Kernels
CUDA samples	BlackScholes	Computation of the BlackScholes equation	BlackScholesGPU
	FDTD3d	Stencil computation for 3D surface	FIniteDifferencesKernel
	matrixMul	Matrix multiplication	MatrixMulCUDA
	matrixMulCUBLAS	Matrix multiplication with cuBLAS	cublasSgemm
	reduction	Perform reduction operations for an array	reduce0, reduce1, reduce2, reduce3, reduce4, reduce5, reduce6
	simpleMultiCopy	Copy array data	incKernel
Rodinia	cfd	Solver for 3D Euler equations	cuda_compute_flux, cuda_compute_step_factor, cuda_time_step
	dwt2d	2D wavelet transform	c_CopySrcToComponents, fdwt53Kernel
	heartwall	Simulate the movement of a mouse heart	kernel
	hotspot	Compute 2D chip temperature distribution	calculate_temp
	hotspot3D	Compute 3D chip temperature distribution	hotspotOpt1
	lavaMD	Compute particle potential and relocation due to mutual forces between particles in 3D space	kernel_gpu_cuda
	particlefilter_float	Object position estimation executed totally on GPU	find_index_kernel, likelihood_kernel, normalize_weights_kernel, sum_kernel
	particlefilter_naive	Object position estimation executed partially on GPU	kernel
	streamcluster	Clustering for an input stream	kernel_compute_cost
Parboil	cutcp	Compute the short-range component of Coulombic potential at each grid point over a 3D grid	cuda_cutoff_potential_lattice6overlap
	lbm	Fluid dynamics simulation using the Lattice Boltzmann method	performStreamCollide_kernel
	mri-q	Compute a matrix Q, representing the scanner configuration for calibration	ComputePhiMag_GPU, ComputeQ_GPU
	tpacf	Statistical analysis of the spatial distribution of astronomical bodies	gen_hists

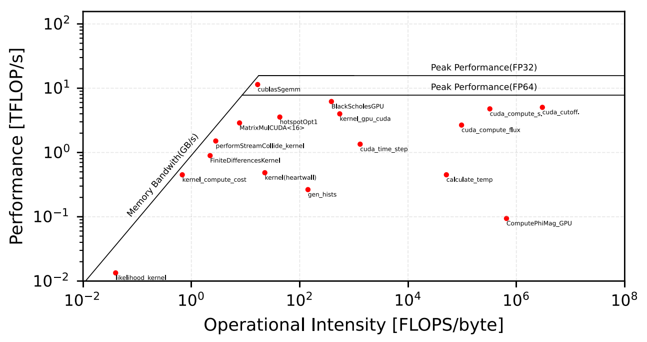

Fig. 3. Roofline analysis of our benchmarks with CUDA12F on V100 GPU.

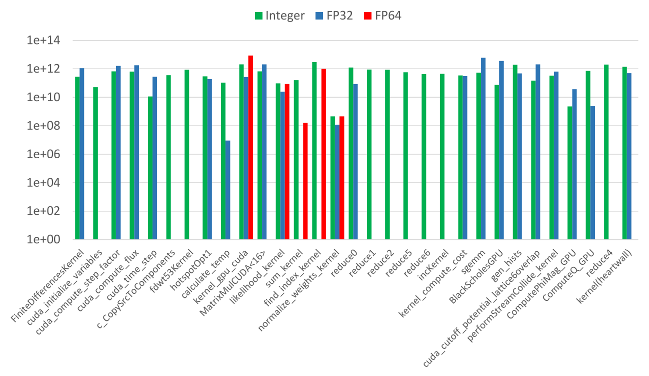

Fig. 4. The numbers of Integer, FP32, and FP64 instructions included in the benchmarks complied with CUDA12F for V100 GPU.

used in this study are listed in Table 2. We select the six programs from the CUDA samples [12] to test primitive operations on the GPUs. The nine programs from the Rodinia 3.1 benchmark suite [13] were used to evaluate compute-intensive workloads. The four programs from the Parboil benchmark suite [14] were used to test more diverse workloads.

Fig. 3 shows the results of the roofline analysis of these kernels. We also show the number of Integer, FP32 and FP64 instructions for each kernel when using CUDA12F with V100 in Fig. 4. The kernel included in particlefilter_naive is not shown in these figures because we were unable to obtain the number of instructions for this kernel due to an error of NVIDIA Nsight Compute [15]. About half of the kernels do not include floating-point instructions, as shown in Fig. 4. These kernels are also removed from Fig. 3. As shown in these figures, our benchmark programs consist of a variety of applications from memory-bound to compute-bound. In addition, because these benchmarks are widely used to assess the performance of a single GPU [16], we believe that our selected benchmark programs are sufficiently comprehensive.

Because individual kernel functions within a program have their own performance and power characteristics, we perform the analysis per kernel in Sections 4 and 5. To do this, we extracted 34 kernel functions from the 20 programs and executed them on each GPU. We executed each kernel function 20 times to reduce the impact of run-to-run variations.

Table 3
CUDA Toolkit versions.

CUDA versions	Description	Release date
CUDA8F	CUDA Toolkit 8.0 GA1	Sep. 2016
CUDA8L	CUDA Toolkit 8.0 GA2	Feb. 2017
CUDA9F	CUDA Toolkit 9.0	Sep. 2017
CUDA9L	CUDA Toolkit 9.2	May 2018
CUDA10F	CUDA Toolkit 10.0	Sep. 2018
CUDA10L	CUDA Toolkit 10.2	Nov. 2019
CUDA11F	CUDA Toolkit 11.0.0	Mar. 2020
CUDA11L	CUDA Toolkit 11.6.0	Jan. 2022
CUDA12F	CUDA Toolkit 12.0.0	Dec. 2022
CUDA12M	CUDA Toolkit 12.1.0	Jan. 2023

3.3. CUDA versions

The CUDA toolkits used in this study are shown in Table 3. We use nine CUDA versions composed of the first and latest minor versions in each major version from CUDA8 to CUDA11 and the first minor version of CUDA12. We note that some CUDA versions were not available for the systems we used. The details are as follows.

- CUDA12F was not available for TSUBAME3.0, so we used CUDA12M on P100 instead.
- CUDA8F, CUDA8L, and CUDA9F were not available for Wisteria-A, so the other 6 CUDA versions were used on A100.
- Only CUDA11F, CUDA11L, and CUDA12F were available for Pegasus, so we used these three CUDA versions on H100.

Using the first and latest minor versions in each major version enables us to experimentally identify the impact of major updates to CUDA toolkits on the performance, power consumption, and energy consumption of a GPU application.

3.4. Power and performance measurement techniques

We used the NVIDIA System Management Interface [17] to measure power. This interface, included in the CUDA toolkit, is a command-line tool for analyzing CUDA programs running on GPUs. Executing the command `nvidia-smi` on a host CPU enables us to monitor various states within the GPU (e.g., GPU clock frequency) during the execution of the program. The command `nvidia-smi` has many options to collect GPU states. We used the following two options in our experiments.

timestamp The current system timestamp at the time when `nvidia-smi` was invoked

power.draw The last measured power draw for the entire GPU board. It may contain a calibration error of ±5W according to the man page of `nvidia-smi`.

Repeating `nvidia-smi` with the above options produces time-series data for the power consumption of a GPU. Using this data, we computed the average power consumption of the GPU when running a kernel. We repeatedly called `nvidia-smi` at 1 ms intervals.

We measured the CUDA kernel performance from the host code. To measure the per-kernel power consumption of a GPU, we must execute kernels individually. To do this, we inserted the `cudaDeviceSynchronize()` function after kernel launch to wait for the kernel execution to complete. We also used manual timers to map timestamps reported by `nvidia-smi` to intervals during kernel execution. More specifically, our timer function reports two OS times: the time before kernel launch and the time after synchronization. We consider a kernel to be running on a GPU when the timestamps reported by `nvidia-smi` are included in the duration between these two times.

4. CUDA version analysis

4.1. Overview

Fig. 5 shows the performance, and power and energy consumption of the kernels compiled with the nine CUDA versions on four different GPUs. The horizontal axes represent CUDA versions, while the vertical axes represent slowdown, power, or energy. The numbers of each CUDA version are normalized to those of the latest CUDA version (i.e. CUDA12M (P100), and CUDA12F (the other GPUs)). The top, middle, and bottom lines of each box represent the 75th percentile, median, and 25th percentile values in the set of 32 data points that correspond to 32 kernels. Each kernel is illustrated with the same marker across these figures.

From these figures, we obtain the following findings regarding the CUDA versions.

- The use of newer CUDA versions is preferable in many cases. For example, CUDA8F produces a 1.16× slowdown on the Tesla P100 GPU compared to CUDA12M, and this slowdown decreases gradually as CUDA versions are updated. In particular, the latest CUDA version shows the best in two metrics (performance and energy consumption) on the Tesla P100, NVIDIA V100, and NVIDIA H100 GPU.
- The differences in the metrics obtained by the CUDA versions decrease as the GPU architecture advances. For example, the performance difference when going from CUDA9L to CUDA11L on the Tesla P100, NVIDIA V100 and NVIDIA A100 GPUs is within 12%, 4%, and 1%, respectively.
- The difference in power consumption caused by the CUDA version is relatively small on all GPUs (only ±2%) compared to those in performance and energy consumption.

We note that we removed some benchmark programs from the figure for the following reasons.

- The particlefilter_float program was removed because we encountered an error when compiling it with CUDA12M on the P100 GPU.
- The matrixMulCUBLAS program was excluded due to an error during the execution with CUDA9L and CUDA10F on the A100 GPU.

As shown in Fig. 5, CUDA12F and CUDA12M achieve the highest performance in many cases, but there are a few exceptions. Hereafter, we present three sources of performance degradation caused by CUDA12F and CUDA12M, which were found through our in-depth analysis of the code of various kernels.

4.2. Case 1: Aggressive loop unrolling

NVCC uses loop unrolling to optimize the performance of GPU applications. Unrolling loops has various benefits such as a reduction in instruction count and increased opportunity for compilers to perform better instruction scheduling, though it increases both code size and register usage. The increase in code size and register usage, respectively, reduces the numbers of instruction cache hits and threads that are executed concurrently; therefore, applications perform worse when newer CUDA versions unroll loops to an excessive degree.

We found that the above situation occurred when executing `FiniteDifferencesKernel` on the Tesla P100. This kernel is included in FDTD3d and computes stencils for a 3D surface. The experimental results of this kernel on the Tesla P100 are shown in the upper left of Fig. 6. The vertical axis represents the execution time, whereas the horizontal axis represents the CUDA versions. For this kernel, CUDA8F and CUDA8L obtain the shortest execution time of 1.84 s, which is 80% of the execution time of CUDA12M (2.31 s).

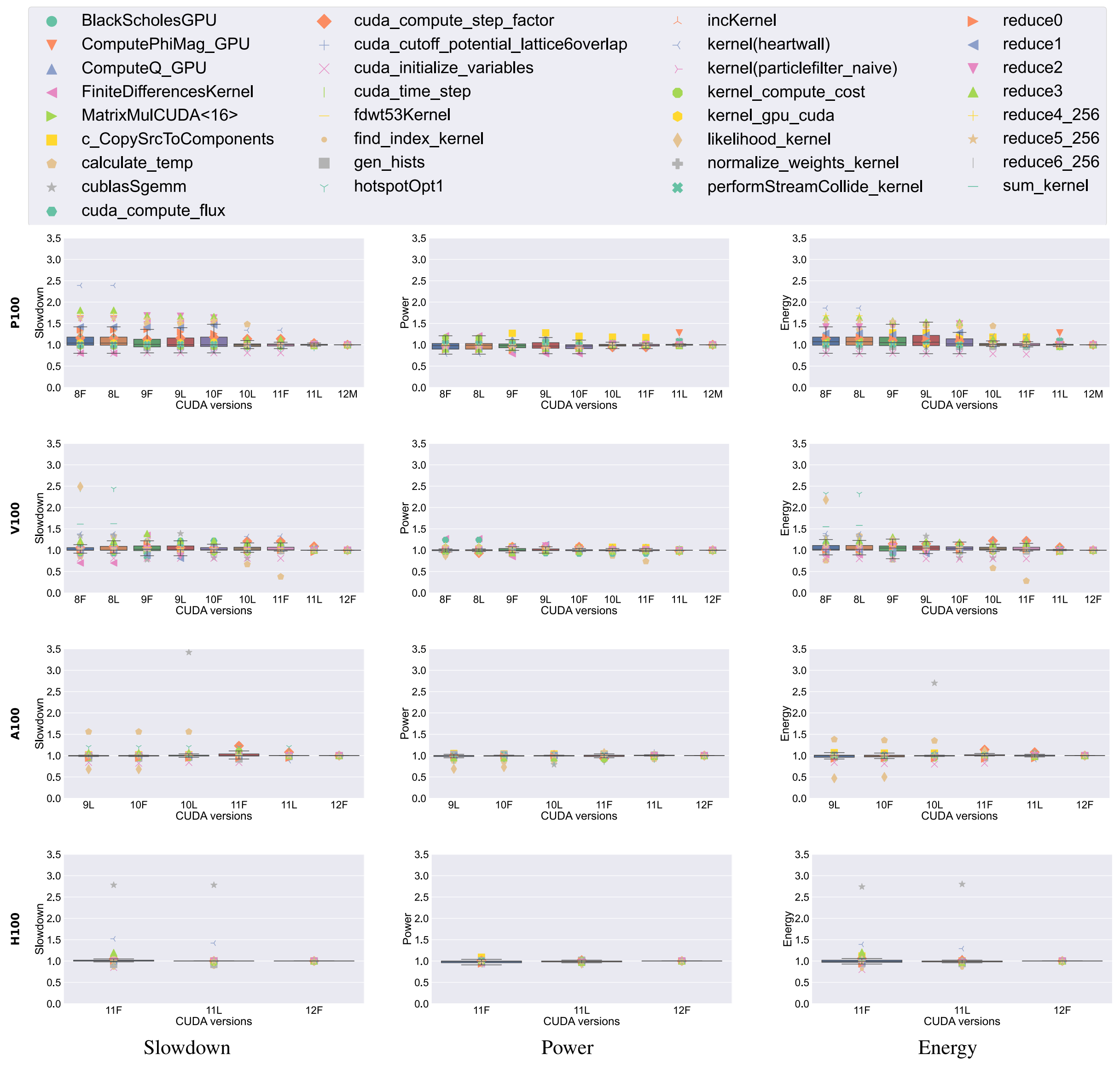

Fig. 5. Summary of our experimental results. The numbers of P100 are normalized to those of CUDA12M and the numbers of the others (V100, A100, H100) are normalized to those of CUDA12F (lower values are better).

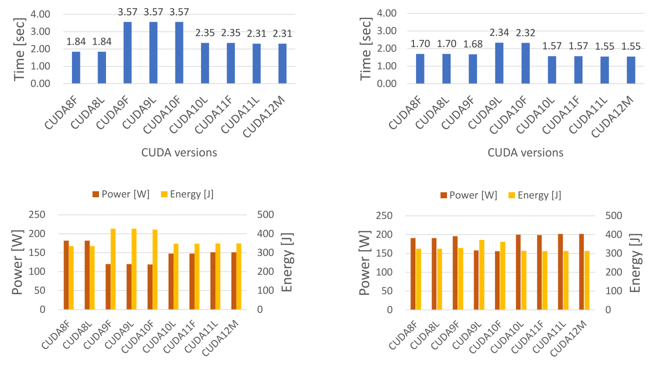

Fig. 6. Performance, power consumption, and energy consumption of `FiniteDifferencesKernel` on the Tesla P100 (left: w/o `maxrregcount`, right: w/ `maxrregcount=40`).

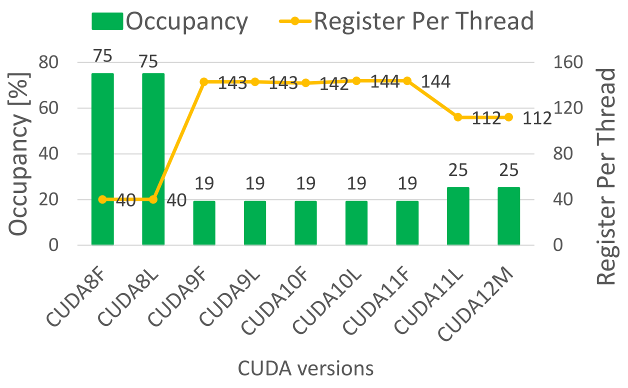

Fig. 7. Number of registers per thread and occupancy of `FiniteDifferencesKernel` on the Tesla P100.

Fig. 7 shows the number of registers per thread and the occupancy of the eight object codes executed on the Tesla P100. Occupancy, which is the ratio of the maximum number of warps executed to the number of active warps, is an indicator of how well the arithmetic units in the

SMs are utilized. We calculated occupancy using the CUDA occupancy calculator [18] provided by NVIDIA.

This figure shows that CUDA8F and CUDA8L have a high occupancy (75%), whereas the other CUDA versions have a low occupancy (19%–25%). In addition, the figure shows that CUDA8F and CUDA8L use a small number of registers per thread (40), whereas the others use a large number of registers per thread (112–144). This is clearly influenced by the degree of loop unrolling. In fact, we read the PTX codes generated by multiple CUDA versions and found that the degree of loop unrolling differed according to the CUDA version (a greater number of iterations are unrolled for a loop in the object code generated using the newer CUDA versions). This suggests that CUDA versions after CUDA9F adopt a more aggressive loop unrolling strategy.

One approach to throttling loop unrolling is to set the maximum number of registers that can be used by GPU functions. We can do this using the `maxrregcount` option of NVCC. We compiled the application code with each CUDA version setting `maxrregcount` to 40, and then executed the generated object code on the Tesla P100.

The experimental results are shown in the upper right side of Fig. 6. The figure shows that restricting the degree of loop unrolling improves the performance of `FiniteDifferenceKernel` for all CUDA versions because of increased occupancy. In particular, CUDA12M obtains the shortest execution time of 1.55 s, which is less than that of CUDA8F and CUDA8L without `maxrregcount` (on the left side of Fig. 6). We note that use of maxrregcount=40 enables the occupancy to reach 75% for all CUDA versions. The figure also shows that use of the appropriate compiler option can reduce a performance gap between the object codes generated by different CUDA versions.

The bottom of Fig. 6 shows the power and energy consumption of `FiniteDifferenceKernel` in the cases of not using `maxrreg-count` and using `maxrregcount=` 40. The horizontal axes represent CUDA versions, the first vertical axes represent power consumption, and the second vertical axes represent energy consumption. Fig. 6 shows that as the execution time becomes shorter, the application consumes more power but less energy.

The primary concerns of supercomputing users are the performance of their application, whereas one of the most important concerns of system operators is the energy efficiency of targeted systems [19–23]. Our experimental results suggest that some optimizations that reduce the execution time of targeted applications have benefit to both supercomputing users (improvement in the performance of their applications) and system operators (reduction in the energy consumption of targeted systems).

4.3. *Case 2: Inefficient instruction scheduling*

Because instructions within a warp are executed on an SM in an in-order manner, the order of instructions, which is determined by NVCC, can affect the application performance. In particular, when a newer CUDA version inserts only a few instructions between the global loads and their consumers, applications suffer from a number of memory-dependent stalls because the long load latency cannot be hidden by the execution of other instructions. We found this situation when running `Kernel` on the Tesla P100 and `kernel_compute_cost` on the NVIDIA V100.

4.3.1. *Case 2a:* `Kernel`

`Kernel` is included in heartwall and simulates the movement of a mouse heart. The experimental results of this kernel on the Tesla P100 are shown in Fig. 8. The vertical and horizontal axes are the same as those of Fig. 6. The upper left side of figure shows that for this kernel, the object code generated by CUDA10F obtains the shortest execution time of 2.50 s, which is 84% of the execution time of the object code generated by CUDA12M (2.97 s).

Fig. 9 shows a code snippet of `Kernel`. At the third line in this code, an FMA instruction is executed with three variables, and the last variable needs to be loaded from global memory. We note that the other two variables are stored in shared memory or a register. Because global memory accesses consume more time than shared memory accesses, the timing of the FMA instruction execution is highly influenced by the timing when the global memory load is completed.

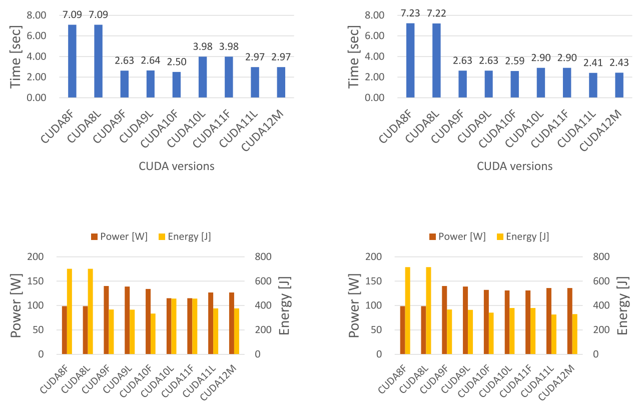

Fig. 8. Performance, power consumption, and energy consumption of `Kernel` on the Tesla P100 (left: w/o `maxrregcount`, right: w/ `maxrregcount=128`).

```
1  for(ia=ia1; ia<=ia2; ia++) {
2     ib = ip1 - ia;
3     s = s + d_in_mod_temp[d_common.in_rows*(ja-1)+ia-1] *
d_unique[bx].d_in2[d_common.in2_rows*(jb-1)+ib-1];
4  }
```

Fig. 9. Code snippet of `Kernel`.

```
1   L_x_24:
2   IADD3 R70, R87, R60, -R73 ;
3   SHR R66, R70.reuse, 0x1e ;
4   ISCADD R70.CC, R70, R12, 0x2 ;
5   IADD.X R71, R13, R66 ;
6   LDG.E R74, [R70] ;
7   LDG.E R76, [R70+-0x4] ;
8   LDG.E R79, [R70+-0x8] ;
9   LDG.E R81, [R70+-0xc] ;
10  IADD R66, R84, R73 ;
11  SHL R78, R66, 0x2 ;
12  LDS.U.32 R75, [R78+0x5b8] ;
13  LDS.U.32 R77, [R78+0x5bc] ;
14  LDS.U.32 R80, [R78+0x5c0] ;
15  LDS.U.32 R83, [R78+0x5c4] ;
16  IADD32I R66, R73, 0x3 ;
17  ISETP.GE.AND P1, PT, R66, R63, PT ;
18  IADD32I R73, R73, 0x4 ;
19  DEPBAR.LE SB5, 0x2 ;
20  FFMA R74, R74, R75, R72 ;
21  FFMA R74, R76, R77, R74 ;
22  FFMA R74, R79, R80, R74 ;
23  FFMA R66, R81, R83, R74 ;
24  {       MOV R72, R66 ;
25  @!P1 BRA `(.L_x_24)      }
```

```
1   IADD R70, R62, -R35
2   LDC.64 R32, c[0x3][R14+0x30]
3   IADD32I R69, R35, -0x1
4   IADD R36, R64, -R39
5   XMAD.MRG R37, R70.reuse, c[0x3] [0xec].H1, RZ
6   XMAD R36, R70.reuse, c[0x3] [0xec], R36
7   XMAD.PSL.CBCC R37, R70.H1, R37.H1, R36
8   SHR R57, R37.reuse, 0x1e
9   ISCADD R36.CC, R37, R32, 0x2
10  IADD.X R37, R33, R57
11  LDG.E R57, [R36]
12  XMAD R56, R69.reuse, c[0x3] [0xdc], R39
13  LDG.E R59, [R36+-0xc]
14  XMAD.MRG R58, R69.reuse, c[0x3] [0xdc].H1, RZ
15  XMAD.PSL.CBCC R56, R69.H1, R58.H1, R56
16  LDG.E R58, [R36+-0x4]
17  SHL R61, R56, 0x2
18  LDS.U.32 R56, [R61+0x5b8]
19  LDS.U.32 R60, [R61+0x5c4]
20  DEPBAR.LE SB5, 0x1
21  FFMA R56, R57, R56, R38
22  LDG.E R38, [R36+-0x8]
23  LDS.U.32 R57, [R61+0x5bc]
24  DEPBAR.LE SB5, 0x1
25  FFMA R56, R58, R57, R56
26  LDS.U.32 R57, [R61+0x5c0]
27  FFMA R38, R38, R57, R56
28  IADD32I R56, R39.reuse, 0x3
29  ISETP.GE.AND P1, PT, R56, R65, PT
30  IADD32I R39, R39, 0x4
31  FFMA R38, R59, R60, R38
32  @!P1 BRA 0x37a0
```

Fig. 10. SASS code of the code shown in Fig. 9. The left code is generated by CUDA10F, whereas the right code is generated by CUDA12M.

The left and right of Fig. 10 show the SASS code generated by CUDA10F and CUDA12M, respectively. Loop unrolling is applied in both instances, and each code therefore has four FFMA (FP32 Fused Multiply–Add) instructions in the loop. We can see that the distance between the FFMA instructions and the corresponding global loads (LDG instructions) differs substantially. The right code includes 9.5 instructions between the FFMA and LDG on average (red: 9, blue: 17, green: 8, purple: 4), whereas the left code has 13 instructions on

```
1  IADD32I R57, R31, -0x1 ;
2  LDC.64 R24, c[0x3][R7+0x30] ;
3  IADD3 R41, R56, R27, -R42.reuse ;
4  SHR R43, R41.reuse, 0x1e ;
5  ISCADD R40.CC, R41, R24, 0x2 ;
6  IADD.X R41, R25, R43 ;
7  LDG.E R44, [R40] ;
8  LDG.E R45, [R40+-0x4] ;
9  LDG.E R48, [R40+-0x8] ;
10 LDG.E R50, [R40+-0xc] ;
11 XMAD R43, R57, c[0x3] [0xdc], R42 ;
12 XMAD.MRG R46, R57.reuse, c[0x3] [0xdc].H1, RZ ;
13 XMAD.PSL.CBCC R43, R57.H1, R46.H1, R43 ;
14 SHL R47, R43, 0x2 ;
15 LDS.U.32 R43, [R47+0x5b8] ;
16 LDS.U.32 R46, [R47+0x5bc] ;
17 LDS.U.32 R49, [R47+0x5c0] ;
18 LDS.U.32 R52, [R47+0x5c4] ;
19 DEPBAR.LE SB5, 0x2 ;
20 FFMA R43, R44, R43, R29 ;
21 IADD32I R29, R42.reuse, 0x3 ;
22 IADD32I R42, R42, 0x4 ;
23 FFMA R43, R45, R46, R43 ;
24 ISETP.GE.AND P1, PT, R29, R30, PT ;
25 FFMA R43, R48, R49, R43 ;
26 FFMA R29, R50, R52, R43 ;
27 @!P1 BRA 0x3690 ;
```

Fig. 11. SASS code of `Kernel` generated by CUDA12M with `maxrregcount`=128.

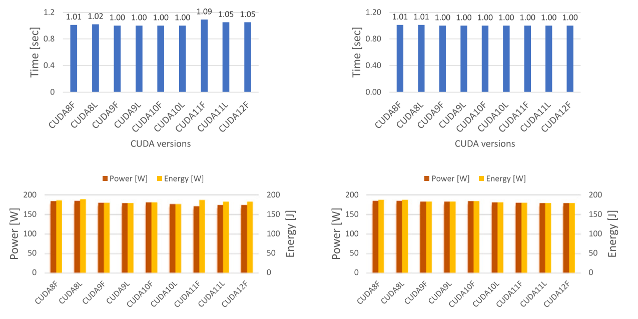

Fig. 12. Performance, power consumption, and energy consumption of `kernel_compute_cost` on the NVIDIA V100 (left: w/o `maxrregcount`, right: w/ `maxrregcount`=64).

average. The left code has 3.5 more instructions between FFMA and LDG, resulting in a smaller number of memory-dependent stalls.

Increasing the number of registers per thread makes loop unrolling more aggressive and somewhat helps extend the distance between global loads and consumers. As a result, the execution time of the object code generated by different CUDA versions will become more similar. To verify this, we compiled `kernel` with each CUDA version while setting `maxrregcount` to 128, and then executed the generated object code on the Tesla P100.

The experimental results are shown in the upper right side of Fig. 8. The execution time of CUDA12M is 2.43 seconds in the case of `maxrregcount`=128, which is 81% of the execution time (2.97 s) when `maxrregcount` is not used. Fig. 11 shows the SASS code generated by CUDA12M with `maxrregcount`=128. This code includes 14 instructions between FFMA and LDG on average (red: 12, blue: 14, green: 15, purple: 15), i.e., it has 4.5 more instructions between FFMA and LDG compared to the code on the right shown in Fig. 10. This reduces memory-dependent stalls, improving the performance of the application.

```
1   for(int i = 0; i < dim; i++){
2       float tmp = coord_d[(i*num)+p1] - coord_d[(i*num)+p2];
3       retval += tmp * tmp;
4   }
```

Fig. 13. Code snippet of `kernel_compute_cost`.

```
1   L_x_6:
2   MOV R7, 0x4 ;
3   IMAD.WIDE R28, R26, R7, c[0x0][0x180] ;
4   IMAD.WIDE R22, R6, R7, c[0x0][0x180] ;
5   LDG.E.SYS R9, [R28] ;
6   LDG.E.SYS R24, [R22] ;
7   IMAD.WIDE R18, R7, c[0x0][0x160], R28 ;
8   IMAD.WIDE R28, R7, c[0x0][0x160], R22 ;
9   LDG.E.SYS R27, [R18] ;
10  IMAD.WIDE R22, R7, c[0x0][0x160], R28 ;
11  IMAD.WIDE R18, R7, c[0x0][0x160], R18 ;
12  LDG.E.SYS R25, [R18] ;
13  FADD R9, -R9, R24 ;
14  LDG.E.SYS R24, [R28] ;
15  LDG.E.SYS R28, [R22] ;
16  FFMA R9, R9, R9, R20 ;
17  IMAD.WIDE R20, R7, c[0x0][0x160], R18 ;
18  IMAD.WIDE R22, R7, c[0x0][0x160], R22 ;
19  LDG.E.SYS R20, [R20] ;
20  IMAD R6, R7, c[0x0][0x160], R6 ;
21  IMAD.WIDE R18, R6, R7, c[0x0][0x180] ;
22  FADD R21, -R27, R24 ;
23  IMAD R24, R7, c[0x0][0x160], R26 ;
24  LDG.E.SYS R27, [R22] ;
25  FFMA R9, R21, R21, R9 ;
26  FADD R26, -R25, R28 ;
27  IMAD.WIDE R28, R24, R7, c[0x0][0x180] ;
28  FFMA R9, R26, R26, R9 ;
29  LDG.E.SYS R26, [R18] ;
30  LDG.E.SYS R25, [R28] ;
31  FADD R27, -R20, R27 ;
            ⋮
109 @!P2 BRA `(.L_x_6) ;
```

```
1   0x2aafc925c290 :
2   MOV R19, 0x4
3   LDG.E.SYS R29, [R22]
4   IMAD.WIDE R20, R19, c[0x0][0x160], R24
5   LDG.E.SYS R26, [R24]
6   IMAD.WIDE R22, R19, c[0x0][0x160], R22
7   LDG.E.SYS R25, [R20]
8   LDG.E.SYS R28, [R22]
9   IMAD.WIDE R20, R19, c[0x0][0x160], R20
10  FADD R29, -R26, R29
11  FFMA R18, R29, R29, R18
12  FADD R26, -R25, R28
13  IMAD.WIDE R28, R19, c[0x0][0x160], R22
14  LDG.E.SYS R25, [R20]
15  LDG.E.SYS R24, [R28]
16  IMAD.WIDE R20, R19, c[0x0][0x160], R20
17  IMAD.WIDE R28, R19, c[0x0][0x160], R28
18  LDG.E.SYS R23, [R20]
19  LDG.E.SYS R22, [R28]
20  FFMA R18, R26, R26, R18
21  FADD R25, -R25, R24
22  FFMA R18, R25, R25, R18
23  IMAD.WIDE R24, R19, c[0x0][0x160], R28
24  FADD R26, -R23, R22
            ⋮
98  @!P0 BRA 0x2aafc925c290
```

Fig. 14. SASS code of the code shown in Fig. 13. The left code is generated by CUDA10F, whereas the right code is generated by CUDA12F.

The bottom of Fig. 8 shows the power and energy consumption of this kernel. The figure shows that, as in `FiniteDifferenceKernel`, optimizing the execution time increases (reduces) the power (energy) consumption of this kernel.

4.3.2. Case 2b: `kernel_compute_cost`

`kernel_compute_cost` is included in streamcluster, which performs clustering on a given input stream. The experimental results of this kernel on the NVIDIA V100 are shown in Fig. 12. The upper left side of figure shows that for this kernel, the object code generated by CUDA9F, CUDA9L, CUDA10F, and CUDA10L has the shortest execution time of 1.00 s, which is 95% of the execution time of the object code generated by CUDA12F (1.05 s).

Fig. 13 shows a code snippet of `kernel_compute_cost`. In the second line of this code, a subtraction is performed with two variables, and both variables need to be loaded from global memory.

The left and right in Fig. 14 respectively show the SASS code generated by CUDA10F and CUDA12F. Loop unrolling is applied in both instances, and each code therefore has 16 FADD (FP32 ADD) instructions in the loop (we only show the first four instructions in the figure). As in Fig. 14, we can see that the distance between FADD and LDG differs in the two codes. The right code includes 4 instructions between FADD and LDG on average (red: 4, blue: 3, green: 5, purple: 4) whereas the left code has 7.25 instructions on average (red: 6, blue: 7, green: 10, purple: 6). The left code has one more instruction between FADD and LDG, resulting in a smaller number of memory-dependent stalls. We note that because one FADD instruction uses two variables stored in global memory, the distance between FADD and LDG is defined using the LDG instruction closest to the FADD instruction.

We also compiled this kernel with `maxrregcount`=64 and then executed the generated code on the NVIDIA V100. The experimental results are shown in the upper right side of Fig. 12. The figure shows that

```
0x2b5b2d25b4d0 :
MOV R13, 0x4
LDG.E.SYS R10, [R18]
IMAD.WIDE R48, R13, c[0x0][0x160], R18
IMAD.WIDE R44, R13, c[0x0][0x160], R14
LDG.E.SYS R15, [R14]
IMAD.WIDE R46, R13, c[0x0][0x160], R48
LDG.E.SYS R61, [R48]
IMAD.WIDE R38, R13, c[0x0][0x160], R44
LDG.E.SYS R58, [R44]
IMAD.WIDE R30, R13, c[0x0][0x160], R46
LDG.E.SYS R59, [R46]
IMAD.WIDE R50, R13, c[0x0][0x160], R38
LDG.E.SYS R56, [R38]
IMAD.WIDE R20, R13, c[0x0][0x160], R30
LDG.E.SYS R57, [R30]
IMAD.WIDE R32, R13, c[0x0][0x160], R50
LDG.E.SYS R54, [R50]
⋮
FADD R15, -R10, R15
FFMA R12, R15, R15, R12
FADD R61, -R61, R58
FFMA R12, R61, R61, R12
FADD R59, -R59, R56
FFMA R12, R59, R59, R12
FADD R57, -R57, R54
⋮
@P3 BRA 0x2b5b2d25b4d0
```

Fig. 15. SASS code of `kernel_compute_cost` generated by CUDA12F with `maxrregcount`=64.

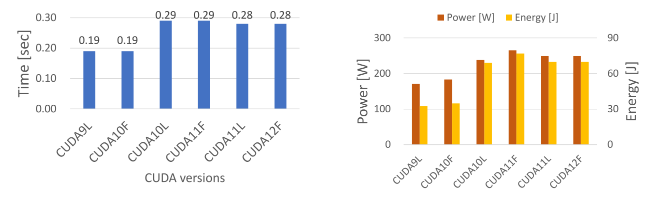

Fig. 16. Performance, power consumption, and energy consumption of `likelihood_kernel` on the NVIDIA A100.

the execution time of CUDA12F with `maxrregcount` is 1.00 s, which is 95% of the execution time of CUDA12F without `maxrregcount`. The use of `maxrregcount` reduces the performance gap between the best (CUDA10F without `maxrregcount`) and CUDA12F from 4% to 0%.

Fig. 15 shows the SASS code generated by CUDA12F with `maxrregcount`=64. As shown in the figure, a sufficient number of registers allow 32 LDG instructions to converge in the first half of the loop and allow the FADD and FFMA instructions to converge in the second half of the loop. The distance between these instructions is enough to hide the latency in global memory accesses, so setting `maxrregcount` to 64 helps improve application performance.

The bottom of Fig. 12 shows the power and energy consumption of `kernel_compute_cost`. Similar to `FIniteDifferencesKernel` and `kernel`, the energy consumption of this kernel is reduced by optimizing the execution time.

4.4. Case 3: Impact of host compilers

Host compilers associated with the CUDA versions often affect the performance of applications running on a GPU. This is because the implementations of some arithmetic functions such as `exp()` and `pow()` in CUDA are borrowed from those in the standard C math libraries included in the host compilers. If an arithmetic function implemented on an old host compiler is more suitable to execution on the GPU, an old CUDA version associated with this host compiler often yields a higher performance than the latest CUDA version.

This situation occurs when `likelihood_kernel` is executed on the NVIDIA A100. It is included in particlefilter_float, which statistically estimates the location of a target object. The experimental results of this kernel on the NVIDIA A100 are shown in the left side of Fig. 16. For this kernel, CUDA9L and CUDA10F obtain the shortest execution time of 0.19 s, which is 67% of the execution time of CUDA12F (0.28 s). We note that GCC-4.8.5 is used to compile the kernel with CUDA9L and CUDA10F because they do not support GCC after version 7. The default is GCC-8.3.1 in Wisteria-A.

```
for (x = 0; x < numOnes; x++)
    likelihoodSum += (pow((double) (I[ind[index * numOnes + x]] - 100), 2) - pow((double) (I[ind[index * numOnes + x]] - 228) 2)) / 50.0;
```

Fig. 17. Code snippet of `likelihood_kernel`.

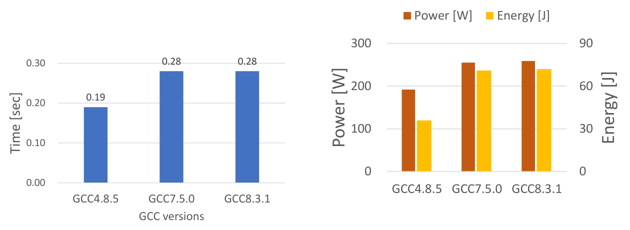

Fig. 18. Performance, power consumption, and energy consumption of `likelihood_kernel` with different GCC versions on the NVIDIA A100 (with CUDA12F).

The code snippet of `likelihood_kernel` is shown in Fig. 17. As shown in the figure, this kernel calls `pow()` with double precision twice, which is not supported as a built-in function in CUDA. In this case, NVCC uses the code of this function included in GCC's math library and then generates the corresponding PTX code using CICC. As a result, the implementation of `pow()` on GCC influences the performance of this kernel.

To confirm that the difference between the GCC versions affects the performance of GPU applications, we compiled this kernel with CUDA12F associated with three different GCC versions and then executed the generated object code on the NVIDIA A100. The experimental results are shown in the left side of Fig. 18. From this result, we can see that GCC-4.8.5 obtains the best performance and the execution time is almost equal to that of CUDA9L and CUDA10F in the left side of Fig. 16. We also confirm that CUDA12F with GCC-7.5.0 and –8.3.1 converts `pow()` on the device code to `pow()` defined in GCC-7.5.0 and –8.3.1, respectively, while CUDA12F with GCC-4.8.5 converts `pow()` on the device code to `__builtin_powi()` defined in GCC-4.8.5. Because the object code using `__builtin_powi()` has 6× smaller number of instructions, CUDA12F with GCC-4.8.5 yields higher performance.

The right sides of Figs. 16 and 18 show the power and energy consumption of `likelihood_kernel`. Like the kernels shown in Sections 4.2 and 4.3, the energy consumption of this kernel is also reduced by optimizing the execution time.

5. Related work

There are many studies that evaluate the impact of differences among compilers on the performance of applications running on CPUs. In the papers [24–26], the authors compare the performance of applications when using different compiler products while selecting one version from each product. In the paper [2], the authors investigated the impact of the GCC version on the performance, power consumption, and energy consumption of CPU applications using six GCC versions (from 5 to 10) and 75 C/C++ benchmark programs. Their results show that GCC-10 obtains the best results for only 25% of the programs tested, and there are many cases in which an older version obtains better performance and/or power consumption and/or energy efficiency. In the paper [27], the authors measured the performance of various applications using multiple versions of GCC but did not mention the performance differences in the GCC versions.

As for GPUs, many researchers have evaluated the impact of differences in architecture on the performance or power consumption of applications. The papers [28–30] report the impact of up to six NVIDIA GPU architectures on application performance. In the papers [31–33], the authors evaluated variations in performance and power efficiency between GPUs. However, only one CUDA version was used in these studies. Comparison among multiple CUDA versions was not made.

The impact of compiler options for an NVCC on the energy efficiency of applications has been investigated. Arafa et al. evaluated the energy consumption of a micro benchmark implemented with more than 40 PTX instructions on four different generations of NVIDIA GPUs (Maxwell, Pascal, Volta, and Turing) and showed that the optimization options of NVCC affect the energy consumption of each instruction [34]. In contrast to our work, they used only one CUDA version. Moreover, they did not evaluate the performance and power consumption of the application, and therefore, the relationship between the performance and energy-efficiency of the application remains unknown.

6. Conclusions

In this paper, we demonstrated that object codes generated with the latest CUDA version are not always optimal. We first evaluated the impact of nine different CUDA versions on the performance, power consumption, and energy consumption of GPU applications. Our experimental results with 32 CUDA kernels revealed that the latest CUDA version obtained the best performance, power consumption, and energy efficiency in many cases, but there were a few exceptions, as expected. In addition, we revealed three sources of performance degradation caused by the latest CUDA version (i.e., aggressive loop unrolling, inefficient instruction scheduling, and the impact of host compilers), which were found through an in-depth analysis of several instances of SASS code. We also showed that in such cases, the latest CUDA version was able to achieve better performance when the appropriate compiler options were used.

The CUDA toolkits will continue to be updated while integrating more advanced compilation techniques, which will have different impacts on the performance, power consumption, and energy consumption of GPU applications. Therefore, we will need to continuously investigate the impact of up-to-date CUDA versions in the future. Furthermore, we will attempt to develop some models that can estimate the impact of CUDA versions on the performance, power consumption, and energy consumption of GPU applications.

CRediT authorship contribution statement

Kohei Yoshida: Data curation, Formal analysis, Investigation, Methodology, Visualization, Writing – original draft, Conceptualization. **Shinobu Miwa:** Conceptualization, Formal analysis, Funding acquisition, Methodology, Supervision, Validation, Writing – review & editing. **Hayato Yamaki:** Supervision, Writing – review & editing, Validation. **Hiroki Honda:** Supervision, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Shinobu Miwa reports financial support was provided by KDDI Foundation and JST, PRESTO.

Data availability

Data will be made available on request.

Acknowledgments

This study was carried out using the Pegasus supercomputer operated by University of Tsukuba, the Wisteria/BDEC–01 supercomputer operated by the University of Tokyo, the "Flow" supercomputer operated by Nagoya University, and the TSUBAME3.0 supercomputer operated by the Tokyo Institute of Technology. Part of this research is supported by the KDDI Foundation Research Grant and JST, PRESTO Grant Number JPMJPR22P9. The authors thank Kimberly Moravec, PhD, from Edanz for editing a draft of this manuscript.

References

[1] NVIDIA, CUDA toolkit archive, 2022, URL https://developer.nvidia.com/cuda-toolkit-archive.
[2] M. Larabel, GCC 5 through GCC 10 compiler benchmarks - five years worth of C/C++ compiler performance, 2019, Internet:www.phoronix.com/review/gcc5-gcc10-benchmarks.
[3] NVIDIA, CUDA Binary Utilities, 2023, Internet:docs.nvidia.com/cuda/cuda-binary-utilities/index.html.
[4] NVIDIA, NVIDIA tesla P100, 2022, Internet:www.nvidia.com/en-us/data-center/tesla-p100/.
[5] Tokyo Institute of Technology, TSUBAME computing services, 2022, Internet:www.t3.gsic.titech.ac.jp/en.
[6] NVIDIA, NVIDIA V100 tensor core GPU, 2022, Internet:www.nvidia.com/en-us/data-center/v100/.
[7] Nagoya University, Supercomputer, 2022, Internet:icts.nagoya-u.ac.jp/en/sc/.
[8] NVIDIA, NVIDIA A100 tensor core GPU, 2022, Internet:www.nvidia.com/en-us/data-center/a100/.
[9] The University of Tokyo, Wisteria/BDEC-01 supercomputer system, 2022, Internet:https://www.cc.u-tokyo.ac.jp/en/supercomputer/wisteria/service/.
[10] NVIDIA, NVIDIA H100, 2023, Internet:www.nvidia.com/en-us/data-center/h100/.
[11] University of Tsukuba, Pegasus, 2023, Internet:www.ccs.tsukuba.ac.jp/eng/supercomputers/#Pegasus.
[12] NVIDIA, CUDA toolkit, 2020, Internet:developer.nvidia.com/cuda-toolkit.
[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 44–54.
[14] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, W. Hwu, Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing, Center for Reliable and High-Performance Computing, 2012.
[15] NVIDIA, NVIDIA nsight compute, 2023, Internet:www.developer.nvidia.com/nsight-compute.
[16] Y. Abe, H. Sasaki, S. Kato, K. Inoue, M. Edahiro, M. Peres, Power and performance characterization and modeling of GPU-accelerated systems, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 113–122.
[17] NVIDIA, NVIDIA system management interface, 2022, Internet:developer.nvidia.com/nvidia-system-management-interface.
[18] NVIDIA, CUDA Occupancy Calculator, 2023, Internet:docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.
[19] Flagship 2020 Project, 2017 Annual report, 2019, Internet:www.r-ccs.riken.jp/wp-content/uploads/2019/08/Flagship_2020_Project_2017.pdf.
[20] J. Gao, F. Zheng, F. Qi, Y. Ding, H. Li, H. Lu, W. He, H. Wei, L. Jin, X. Liu, et al., Sunway supercomputer architecture towards exascale computing: analysis and practice, Sci. China Inf. Sci. 64 (4) (2021) 1–21.
[21] The Cambridge Open Zettascale Lab, Energy-efficient computing, 2023, Internet:www.zettascale.hpc.cam.ac.uk/themes/energy-efficient-computing/.
[22] P. Messina, S. Lee, The U.S. exascale computing project, in: The 2016 International Conference for High Performance Computing, Networking, Storage and Analysis (Birds of a Feather), 2016.

[23] Z. Wang, Y. Tang, J. Chen, J. Xue, Y. Zhou, Y. Dong, Energy wall for exascale supercomputing, Comput. Inform. 35 (2016) 941–962.
[24] J. Domke, A64FX – Your compiler you must decide!, in: 2021 IEEE International Conference on Cluster Computing, CLUSTER, 2021, pp. 736–740.
[25] A. Burford, A. Calder, D. Carlson, B. Chapman, F. Coskun, T. Curtis, C. Feldman, R. Harrison, Y. Kang, B. Michalowicz, E. Raut, E. Siegmann, D. Wood, R. DeLeon, M. Jones, N. Simakov, J. White, D. Oryspayev, Ookami: Deployment and initial experiences, in: Practice and Experience in Advanced Research Computing, 2021.
[26] A. Poenaru, T. Deakin, S. McIntosh-Smith, S.D. Hammond, A.J. Younge, An evaluation of the fujitsu A64fx for HPC applications, in: Presentation in AHUG ISC 21 Workshop, 2021.
[27] B. Michalowicz, E. Raut, Y. Kang, T. Curtis, B. Chapman, D. Oryspayev, Comparing the behavior of openmp implementations with various applications on two different fujitsu A64fx platforms, in: Practice and Experience in Advanced Research Computing, PEARC '21, 2021.
[28] M. Svedin, S.W. Chien, G. Chikafa, N. Jansson, A. Podobas, Benchmarking the nvidia GPU lineage: From early K80 to modern A100 with asynchronous memory transfers, in: Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, 2021, pp. 1–6.
[29] Y.M. Tsai, T. Cojean, H. Anzt, Evaluating the Performance of NVIDIA's A100 Ampere GPU for Sparse Linear Algebra Computations, 2020.
[30] H. Anzt, Y.M. Tsai, A. Abdelfattah, T. Cojean, J. Dongarra, Evaluating the performance of nvidia's A100 ampere GPU for sparse and batched computations, 2020, pp. 26–38.
[31] P. Sinha, A. Guliani, R. Jain, B. Tran, M.D. Sinclair, S. Venkataraman, Not all GPUs are created equal: characterizing variability in large-scale, accelerator-rich systems, 2022, arXiv preprint arXiv:2208.11035.
[32] T. Patki, Z. Frye, H. Bhatia, F. Di Natale, J. Glosli, H. Ingolfsson, B. Rountree, Comparing GPU power and frequency capping: A case study with the MuMMI workflow, in: 2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), 2019, pp. 31–39.
[33] Y. Kohei, S. Rio, M. Shinobu, Y. Hayato, H. Hiroki, Analyzing Performance and Power-Efficiency Variations among NVIDIA GPUs, in: 51th International Conference on Parallel Processing - ICPP, ICPP '22, 2022.
[34] Y. Arafa, A. ElWazir, A. ElKanishy, Y. Aly, A. Elsayed, A.-H. Badawy, G. Chennupati, S. Eidenbenz, N. Santhi, Verified Instruction-Level Energy Consumption Measurement for NVIDIA GPUs, CF '20, 2020, pp. 60–70.