PreScaler: An Efficient System-Aware Precision Scaling Framework on Heterogeneous Systems

Seokwon Kang
Department of Computer Science
Hanyang University
Seoul, Republic of Korea
kswon0202@gmail.com

Kyunghwan Choi
Department of Computer Science
Hanyang University
Seoul, Republic of Korea
alpha930@naver.com

Yongjun Park
Department of Computer Science
Hanyang University
Seoul, Republic of Korea
yongjunpark@hanyang.ac.kr

Abstract

Graphics processing units (GPUs) have been commonly utilized to accelerate multiple emerging applications, such as big data processing and machine learning. While GPUs are proven to be effective, approximate computing, to trade off performance with accuracy, is one of the most common solutions for further performance improvement. Precision scaling of originally high-precision values into lower-precision values has recently been the most widely used GPU-side approximation technique, including hardware-level half-precision support. Although several approaches to find optimal mixed-precision configuration of GPU-side kernels have been introduced, total program performance gain is often low because total execution time is the combination of data transfer, type conversion, and kernel execution. As a result, kernel-level scaling may incur high type-conversion overhead of the kernel input/output data.

To address this problem, this paper proposes an automatic precision scaling framework called *PreScaler* that maximizes the program performance at the memory object level by considering whole OpenCL program flows. The main difficulty is that the best configuration cannot be easily predicted due to various application- and system-specific characteristics. PreScaler solves this problem using search space minimization and decision-tree-based search processes. First, it minimizes the number of test configurations based on the information from system inspection and dynamic profiling. Then, it finds the best memory-object level mixed-precision configuration using a decision-tree-based search. PreScaler achieves an average performance gain of 1.33x over the baseline while maintaining the target output quality level.

CCS Concepts • **Software and its engineering** → *Just-in-time compilers*; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

Keywords HSA, Precision Scaling, Profile-guided, Compiler, Runtime

ACM Reference Format:
Seokwon Kang, Kyunghwan Choi, and Yongjun Park. 2020. PreScaler: An Efficient System-Aware Precision Scaling Framework on Heterogeneous Systems. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368826.3377917

1 Introduction

Massively data-parallel architectures such as graphic processing units (GPUs), have become the most prominent accelerators to efficiently support many emerging applications such as social network service (SNS) analysis, machine learning (ML), and image processing. To provide high data processing throughput within allowable power consumption, GPUs utilize a massive number of execution units concurrently using a single instruction, multiple thread (SIMT) programming models such as CUDA [21] and OpenCL [12]. However, GPU scaling by improving each core performance and increasing the number of cores [22, 24, 25] cannot meet the increasing performance demands of the applications [5].

To achieve further performance improvement, approximate computing, which is trading-off target output quality (TOQ) for performance, has been widely adopted because the emerging applications often do not require perfect accuracy and increased performance with tolerable accuracy loss [18] is preferred. For example, many ML applications decide to improve the system performance by applying various approximation techniques [6, 8, 19, 40, 41]. In general-purpose GPUs (GPGPUs), various techniques [2, 11, 16, 35, 36] have been proposed to approximate target programs based on pre-defined code patterns.

Among these various techniques, precision scaling, which tunes the floating-point precision of target programs, is a well-known, powerful, and simple approximation technique. Recent GPUs have supported half-precision arithmetic operations with half-precision variables [22, 24, 25], and thus precision scaling has become more promising. However, few

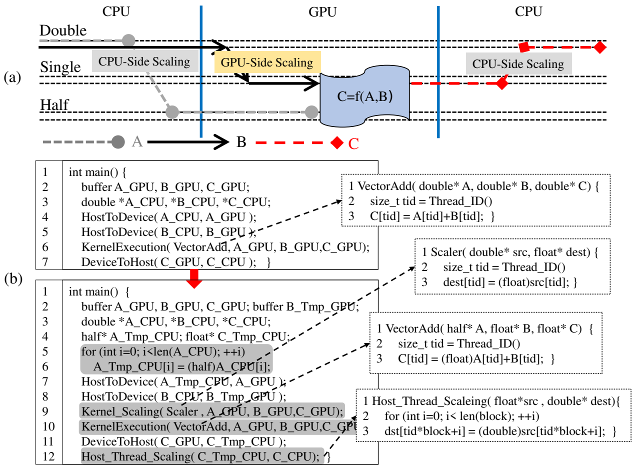
CGO '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7047-9/20/02...$15.00
https://doi.org/10.1145/3368826.3377917

GPU-based precision scaling frameworks [13] have been introduced, whereas precision scaling is one of the most widely used approximation techniques in CPU-based programming [14, 30, 33, 37–39]. This is because GPGPU kernel codes are known to have simple code structures and contain only small number of floating-point variables compared to legacy codes. As a result, a simple exploration approach to maximize the number of variables with low precision has been thought to be enough for fair performance gain.

However, precision scaling of GPGPU programs is not such a simple mission in reality but rather a highly challenging problem owing to the following problems:

[1. Whole Program-level Consideration] GPGPU programs are not always dominated by simple kernel codes. Typical GPGPU programs consist of host programs on CPUs and kernel programs on GPUs. The main part of host programs is data transfer between the host and the device memory, and the total execution time of data transfer is often much larger than that of kernel execution for data-intensive applications, whereas kernel execution time is still dominant for computation-intensive applications. Therefore, a kernel-level approach is not effective enough when executing data-intensive applications. The total performance can be further improved if the input data of target kernels are translated to a lower precision before data transfer from the host and the kernel executes without type conversion overhead. Thus, an improved approach that considers whole program-wide scaling beyond simple kernel-level precision scaling techniques is necessary to maximize the effectiveness.

The critical challenge with the whole program consideration is how to find the best precision configuration among a massive number of possible configurations because program-wide precision scaling offers additional type-converting methods outside kernels. For example, when a kernel decides to change an input double-precision data array to a half-precision data array, the data can be converted using one of three options: host-side scaling (before Host-to-Device (HtoD) data transfer), device-side scaling (between HtoD and kernel execution), and in-kernel scaling. Figure 1 shows an example precision scaling scenario for an OpenCL program to update a double-precision data array (C) from two input double-precision data arrays (A, B). In this program, kernel execution is assumed to show maximum performance when the (A, B, C) array data are processed with (half, single, single) precision, respectively. For the situation, input A and B array type can be translated using host-side and device-side scaling, and the output C array type can be translated into double precision using host-side scaling for maximum performance. Therefore, program-level precision scaling should further consider *"translation method"* in addition to the *"target precision"* of each variable, and the increased search space makes manual configuration exploration almost impossible.



```
int main() {
    buffer A_GPU, B_GPU, C_GPU;
    double *A_CPU, *B_CPU, *C_CPU;
    HostToDevice( A_CPU, A_GPU );
    HostToDevice( B_CPU, B_GPU );
    KernelExecution( VectorAdd, A_GPU, B_GPU,C_GPU);
    DeviceToHost( C_GPU, C_CPU );   }
```

```
VectorAdd( double* A, double* B, double* C) {
    size_t tid = Thread_ID()
    C[tid] = A[tid]+B[tid];  }
```

```
int main()  {
    buffer A_GPU, B_GPU, C_GPU; buffer B_Tmp_GPU;
    double *A_CPU, *B_CPU, *C_CPU;
    half* A_Tmp_CPU; float* C_Tmp_CPU;
    for (int i=0; i<len(A_CPU); ++i)
      A_Tmp_CPU[i] = (half)A_CPU[i];
    HostToDevice( A_Tmp_CPU, A_GPU );
    HostToDevice( B_CPU, B_Tmp_GPU );
    Kernel_Scaling( Scaler , A_GPU, B_GPU,C_GPU);
    KernelExecution( VectorAdd, A_GPU, B_GPU,C_GP
    DeviceToHost( C_GPU, C_Tmp_CPU );
    Host_Thread_Scaling( C_Tmp_CPU, C_CPU); }
```

```
Scaler( double* src, float* dest) {
    size_t tid = Thread_ID()
    dest[tid] = (float)src[tid]; }
```

```
VectorAdd( half* A, float* B, float* C)  {
    size_t tid = Thread_ID()
    C[tid] = (float)A[tid]+B[tid];  }
```

```
Host_Thread_Scaleing( float*src , double* dest){
    for (int i=0; i< len(block); ++i)
    dst[tid*block+i] = (double)src[tid*block+i];  }
```

Figure 1. An example precision scaling scenario of an OpenCL program on a target CPU/GPU server system.

[2.Unpredictable Scaling Effectiveness] Precision scaling typically tries to maximize the number of precision-lowering opportunities while avoiding TOQ errors because it assumes that a lower-precision operation generally results in better performance. However, the configuration with maximum precision-lowering opportunities does not always show the best performance when considering whole program procedures due to various target system characteristics such as GPU-specific compute capability [22, 24, 25], data transfer bandwidth, and type-conversion overhead. For instance, a single-precision-based operation may be better than half-precision on some target GPUs [25], even when all the configurations can meet TOQ requirements, and the best scaling methods may be different depending on target array sizes. Owing to the unpredictability, an efficient precision scaling framework on GPGPU programs is necessary.

[3.Half-precision Support on GPUs] The last issue is the half-precision support on GPGPU programming. This newly introduced feature makes two aforementioned issues more serious because it significantly increases both the number of possible scaling configurations and performance unpredictability depending on system configuration including different GPU/CPU generations.

To address these difficulties, we first introduce a *System Inspector*, which is an application-independent one-time process that extensively analyzes the target system information relative to precision scaling. First, it finds all the static system information such as double-/single-/half- precision operation availability and the latency data of target GPUs, the GPU global memory sizes, and the number of CPU cores. Then, it gathers dynamic system information such as average data transfer bandwidth between the host and device memory, host-scaling and device-scaling performances for various type-conversion and array-size combinations, by executing several pre-defined test programs. As the static/dynamic system information is independent of the application characteristics, the system inspector is executed once. The resulting information can be used to minimize the optimal configuration searching space by efficiently excluding massive possible configurations that show low performance. The information is further valuable as it can provide several critical hints to the optimizer. For example, it can force the optimizer check a hybrid option where a kernel uses a data with single precision but the data transfers are performed in half precision, in case that it is expected to show better performance even though type conversion occurs twice.

This paper proposes a *PreScaler*, which is an automatic precision scaling framework that maximizes program performance by considering whole data-parallel program procedures using a light-weight decision-tree-based precision tuning process (*Decision Maker*). PreScaler first collects system- and application- specific information using the system inspector and dynamic profiling. Based on the information, PreScaler tries to find a whole program-wide best mixed-precision configuration of memory objects, including both kernel execution and data transfer between host and device memories. To do this, PreScaler first minimizes the search space by pruning low-performance features. Then, it constructs a simplified decision tree where each node is used to test performance when changing the precision of a target memory object, and the order of test objects is sorted by their effective execution time. Finally, PreScaler finds the best mixed-precision configuration by setting the target precisions and translation methods of all target memory objects iteratively. Moreover, to avoid finding local optimum configuration and achieve further performance gain, PreScaler performs a pre-full-precision scaling to set an initial type configuration of a decision-tree-based search, and adds a supplemental test configuration referred to as the *Wildcards* for each decision-making process. The wildcard test configurations are carefully added based on target system information in order to not increase complexity. When PreScaler finds the best configuration, it automatically generates a precision-scaled program using an LLVM compilation framework [1, 15]. This paper offers the following contributions:

- An in-depth analysis of critical challenges on the precision scaling of OpenCL programs due to system- and application-specific factors.
- PreScaler: an automatic precision-scaling framework considering both data transfer and kernel execution.
  1. System inspector: collects system information to help the decision maker be more light-weighted.
  2. Decision maker: finds the best precision configuration using a simple decision-tree search with wildcard insertion.
- Extensive evaluation of the PreScaler framework across multiple OpenCL programs on various GPU versions.

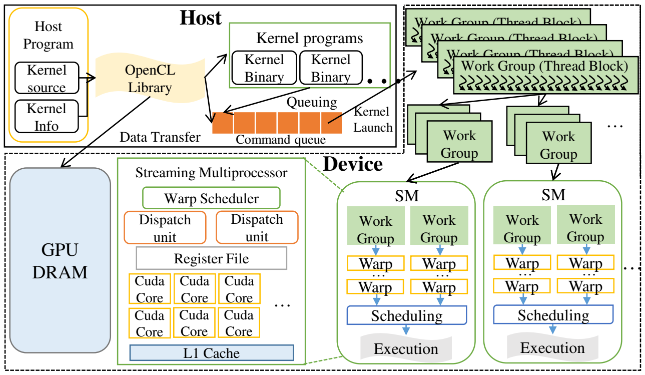


Figure 2. A GPU execution model overview

2 Background

2.1 Data Parallelism on Heterogeneous Systems

Figure 2 illustrates the execution of an example OpenCL program on a target GPU. Each GPU has multiple Streaming Multiprocessors (SMs) each of which is composed of a cluster of multiple computation cores. Each SM concurrently processes a large number of threads using SIMT programming model. Multiple threads are grouped into several thread blocks and are allocated to available SMs. Thread blocks are divided into multiple warps, each of which is a group of 32 threads, and instructions on each warp are executed in lockstep. When a current warp stalls, the warp scheduler replaces it with another warp using fast context switching to hide latency.

In general, OpenCL program performs three steps during execution. First, the program allocates device memory and transfers input data from the host to the device. After all input data have been transferred, the program launches kernels on the device to process the input data. When kernel execution is completed, the program transfers the output data back from the device to the host. These steps are performed by calling OpenCL APIs in the host program.

2.2 Type-conversion Methods on OpenCL Memory Objects

Precision scaling is usually performed at the level of a variable [7, 33, 38]. When a variable must be scaled to a different precision, its input data should be remapped to the new variable type, which is typically done using type-casting instructions just before a mismatch with considerable overhead [20, 33]. When data are identified for use in a different type later in program, type conversion can be performed more efficiently before an actual request. In OpenCL, memory objects are allocated on GPU memory and the data are transferred from the CPU memory before kernel execution, and therefore, the type differences can be easily identified. Consequently, various efficient type conversion methods with data transfer can be considered.

Figure 3 illustrates various techniques to convert data between CPU and GPU memories. The simplest technique is a single-loop type conversion on the host (Fig 3 (a)). On decent CPUs containing multiple cores, this can be accelerated via multithreading (Fig 3 (b)). A similar approach, parallelized type conversion can be performed on GPU side (Fig 3 (c)). Here, the lower overhead of host-side type conversion significantly reduces the size of transferred data and can improve total performance. Armed with this insight, an intermediate type can be introduced to maximize the transfer gain (Fig 3 (d)), called as "transient conversion". Despite this performance gain, transient conversion can incur more errors than direct conversion when an intermediate type has a lower precision than both source and destination types. Finally, data transfer and computation on the host can be further accelerated by overlapping type conversion and transfer as shown in Figure 3 (e) (pipelining).

When memory objects are scaled using the conversion methods shown in Figure 3, several parameters should be considered. First, host-side conversion performance can be improved by utilizing SIMD units. As well, when using multithreading, the number of threads is one of the most important factors impacting performance. Further, due to the launching latency of OpenCL APIs [17], too small atom transfer size can cause severe performance degradation in pipelining. Therefore, the optimal conversion methods varies depending on data sizes, conversion types, and system configurations, as discussed in Section 3.2.2.

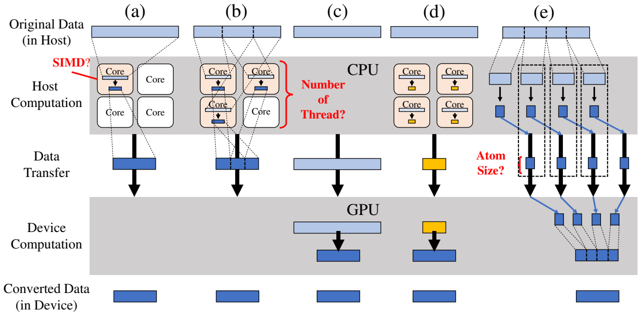


Figure 3. Various type conversion methods and their parameters for an array data transfer from host to device. (a) host-side conversion using a single loop, (b) host-side conversion using a multithreading scheme, (c) device-side conversion, (d) transient conversion, and (e) pipelining of conversion and transfer processes.

3 Motivation

Most precision scaling techniques on GPUs have mainly focused on improving kernel performance with tolerable accuracy loss. In this section, data-parallel programs are shown to require more than simple kernel execution, and the difficulties of whole program-level precision scaling due to increased search space and performance unpredictability are discussed. Experimental data for this section is generated by executing several GPGPU benchmarks on an NVIDIA Titan XP GPU [25] as explained in Tables 4 and 3.

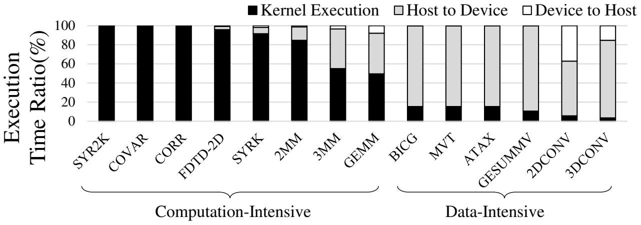


Figure 4. OpenCL program categorization: the components of the bar indicate the portion of execution time in HtoD, kernel execution, and DtoH.

3.1 Limitations on Kernel-based Precision Scaling

3.1.1 Limited Effectiveness

Figure 4 shows the relative fraction of kernel execution and data transfer between memories. When extending precision scaling from the kernel-level to the whole program-level, performance can be greatly improved in two ways. First, if type-casting can be efficiently performed before kernel execution, the total performance gain of precision scaling for computation-intensive applications can be maximized by minimizing in-kernel casting overheads. Second, host-side scaling can greatly enhance the performance of data-intensive applications by minimizing data transfer durations.

3.1.2 Increased Opportunities on Whole Program Consideration

Extending precision scaling to the whole program-level significantly increases the total number of possible precision configurations of memory objects because type-conversion method also should be additionally considered to target precision. In kernel-level precision scaling, the total number of possible type-conversion scenarios of a typical GPGPU program is $(1 + \#_{Conv\_Types})^N$, where $\#_{Conv\_Types}$ and N represent the number of possible precision changes and the number of variables, respectively. In Figure 1, $\#_{Conv\_Types}$ is 2 because two possible type changes (double -> single and double -> half) for input variables are allowed. Therefore, the total number of possible configurations for this example is 27 (= $(1 + 2)^3$), with three input/output variables.

However, when type conversion can be performed in various ways before or after kernel execution, the number of test cases increases because the relation changes to $(1 + \#_{Conv\_Methods} \times \#_{Conv\_Types})^N$, where $\#_{Conv\_Methods}$ refers the number of type conversion methods. Therefore, if a target system offers multiple type-conversion techniques, such as CPU/GPU-side type-conversion with varying thread numbers or pipelining with multiple stages, the number of possible configurations increases significantly even with small number of variables. In Figure 1, when the system provides five scaling techniques as we discussed in Section 2.2, total 1,331 (= $(1 + 5 \times 2)^3$) possible cases exist. This number can be further increased with considering the parameters of each scaling techniques. Therefore, search space minimization, by efficiently excluding suboptimally performing candidates, becomes a critical challenge.

3.2 Performance Unpredictability

Finding the best performance candidate configurations on program-level precision scaling is not easy, as it highly depends on various system- and application- specific features. In this section, several salient features are described that make the prediction of scaling performance difficult.

3.2.1 Half Precision Support Capability Variance

The central objective of traditional precision scaling is to use the lowest precision with a tolerable error rate [7, 14, 33, 37, 38]. This is based on the key assumption that the latencies of lower-precision operations are smaller than those of higher-precision operations. While this assumption is generally true, relatively high type-casting overheads often prevent scaling to lower precision when the expected performance improvement is insufficient. As the performance differences between precisions vary significantly depending on target GPUs, a simple policy to choose the lowest precision operation may not find the best configuration. Table 1 shows the throughput of FP16/FP32/FP64 operations on multiple CUDA compute capabilities from version 3.0 to 7.x, where "compute capability" identifies the hardware features of a specific NVIDIA GPU. The performance improvements when using FP32 over FP64 varies from 2 (7.x) to 24 (3.0), and therefore, while using FP32 is always faster than FP64, careful consideration is still required to lower the precision when the scaling effectiveness is small. Furthermore, the FP16 performance is even lower than that of FP32 and FP64 on GPUs with compute capability 6.1. Thus, in this architecture, FP16 should not be utilized, which is counter to the general assumption.

Table 1. Throughput of native arithmetic operations on various NVIDIA GPU generations [29]. (N: "Not Supported")

Arithmetic	Compute Capability (# of Results per Cycle per SM)							
Operation	3.0, 3.2	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x
FP16(Half)	N	N	N	256	128	2	256	128
FP32(Single)	192	192	128	128	64	128	128	64
FP64(Double)	8	64	4	4	32	4	4	32

3.2.2 Variance in Best Type-conversion Method

Application specific feature also plays a critical role in performance unpredictability. As discussed in Section 2.2, precision scaling during data transfer can improve performance and various options can exist. An interesting question is which scaling technique can maximize performance when multiple options exist. For this question, the best technique can be chosen based on the size of the input arrays of the target program, because each technique performs differently with varying array sizes. Figure 5 illustrates the performance changes for different input sizes when using multiple scaling techniques. The x-axis denotes the array size and the y-axis is the relative data-transfer and type-casting durations of the given techniques normalized to single loop execution time. As shown, different techniques offer optimal performance depending on the target array size. Therefore, the choice of the best type-conversion should be carefully considered with the target application. Note that transient scaling should be carefully applied because it can cause additional error.

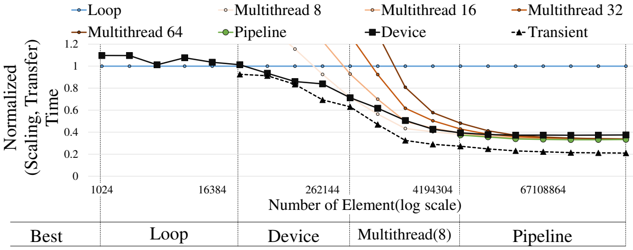
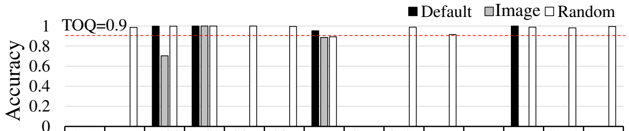


Figure 5. Total execution times of (data transfer (HtoD), scaling (Double -> Single)) processes on various scaling methods with varying data sizes. The best method per data size is also shown (except transient conversion).

3.2.3 TOQ Variance at Different Input Data Ranges

Although half precision operations have been widely adopted, they often incur TOQ failure due to the limited range. Thus, half precision should not be used in these cases, but it also cannot be easily predicted because the errors occur based on the target application characteristics such as input data range, which cannot be easily predicted. Figure 6 shows the output qualities for benchmarks of different input data sets when all floating operations were converted to half precision (target TOQ: 0.9): default [31], image [34] (0.0-256.0), and random (0.0-1.0). Clearly, the errors are hardly predictable before execution.



	ATAX	BICG	2DCONV	3DCONV	CORR	COVAR	FDTD-2D	GEMM	GESUMMV	2MM	3MM	MVT	SYR2K	SYRK
■ Default	X	X	O	O	X	X	O	X	X	X	X	O	X	X
▨ Image	X	X	X	O	X	X	X	X	X	X	X	X	X	X
□ Random	X	O	O	O	O	O	X	X	O	O	X	O	O	O

Figure 6. Output quality for each input set when all the memory objects are set to half precision.

3.3 High Level Insights

In this section, the primary difficulties of scaling the precision of data-parallel programs were described. Kernel-level scaling cannot address all the data-parallel applications because it is mainly effective for computation-intensive programs. Using program-level scaling complicates the situation because it increases the number of possible configurations. Finally, optimal scaling configurations cannot be easily predicted because it depends on system and application characteristics. Therefore, a main objective is to precisely find minimal number of best-performance candidate configurations to test performance and accuracy.

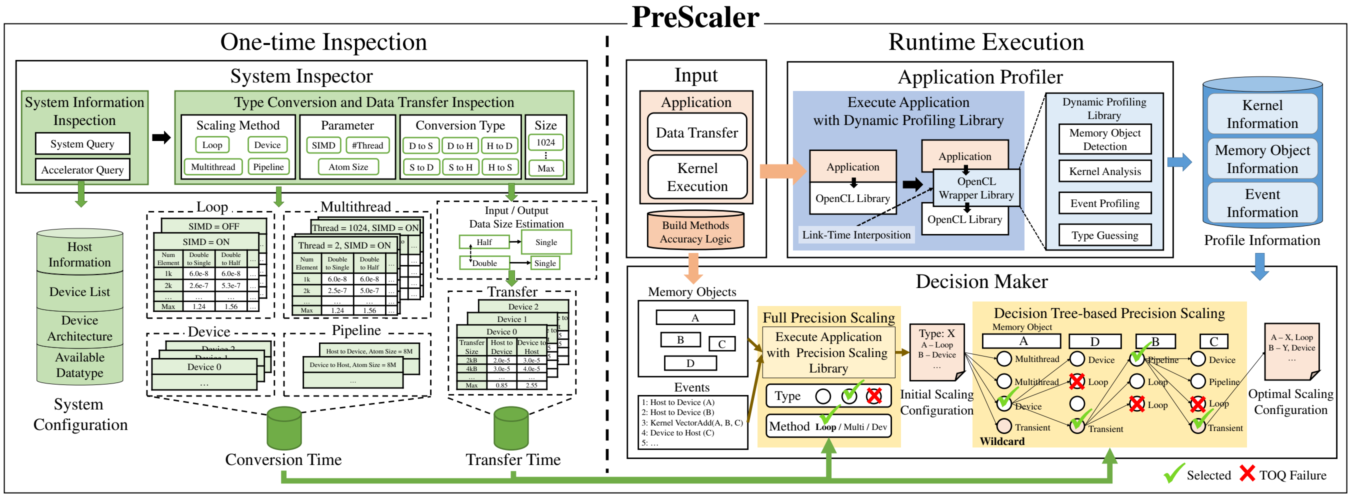


Figure 7. An overview of a PreScaler framework.

4 PreScaler Framework

4.1 Framework Overview

In this section, the PreScaler compilation framework is described that automatically generates an optimally precision-scaled OpenCL program by considering whole- program-level kernel and data transfer flows. As shown in Figure 7, PreScaler consists of three main processes: a system inspector, an application profiler, and a decision maker. The system inspector collects all available static and dynamic information for precision scaling on target heterogeneous systems. When a specific target application is identified, application profiler performs dynamic profiling to gather application-specific information, and detects memory objects with floating point elements targeted for precision scaling. The decision maker first performs a full precision scaling by testing basic scaling configurations. It then constructs a decision tree based on the priority of all memory objects, which is simplified through a suboptimal configuration pruning approach using system and application information. Based on the decision tree, decision maker sequentially determines the best target precision and type conversion method for each memory object by testing the performance and accuracy of the least number of cases. Finally, PreScaler generates the execution binary of the target program with the optimal scaling configuration.

4.2 System Inspector

As discussed in Section 3.2.2, the choice of an optimal type conversion method depends on various system features such as data size, data transfer direction, and conversion type. In this situation, checking all methods whenever a new application is introduced is not recommended due to its too large search space. Therefore, the system inspector extensively examines the application-independent system characteristics related to type conversion, which may be helpful in predicting the optimal scaling method. As this process is not related to any application, one-time system inspection is performed when a target system is fixed.

The system inspector first collects the static characteristics of the target system: number of host CPU cores, number of GPU devices and their detailed architecture versions, and PCI-Express interface information. Based on this, it then collects dynamic information related to type-casting by performing a pre-defined in-depth test. It tests the throughput performance on a real system of multiple {type-conversion then transfer} methods across multiple data sizes and records the results to the inspector result database. A substantial number of type-conversion methods can be tested depending on various options such source/destination precision type, different degrees of single-loop/multi-threading, pipelining, and host/device-side options, as shown in Section 2.2.

4.3 Application Profiler

The application profiler collects three types of application information: kernel, memory object, and event information. Kernel information comes from the device and includes a kernel name and argument types. Memory object information is about memory objects, which are allocated by OpenCL programs. It includes the allocated data size, the original data type, and the number of elements. The original data type and number of elements of a memory object can be inferred by checking the mapping between the memory object and the corresponding kernel argument. The profiler also collects information related to kernel execution (execution time and argument mapping) and data transfer(target memory object, effective transfer size, and offset information). The application profiler is executed automatically through a light-weight application profiler library when a new application is launched.

4.4 Decision Maker

When system and application information is ready, PreScaler searches for the optimal scaling configuration by testing a minimum number of candidate configurations. The total number of possible configurations can be roughly calculated as shown in Equation 1, where $MObj$, $\#_{Conv\_Type}$,

Algorithm 1 Optimal Target Scaling Type Search

```
Require: AvailableDatatypeSet:Set of Types, Original:Type, TOQ:Number, RelatedEvents:List of Data Transfers Related to Current Memory Object
Ensure: OptimalScalingConfig
 1: initialize NormalBestConfig, AcceptedTargetSet, FailedTarget
 2: for Target in AvailableDatatypeSet do
 3:     if Original == Target then continue end if
 4:     initialize NormalConfig, KernelTimeMap
 5:     for Event in RelatedEvents do
 6:         NormalConfig += getBestScalingMethod( Event, Original, Target, Set{ Original, Target })
 7:     end for
 8:     KernelTime, TransferTime, Accuracy := RunApp(NormalConfig)
 9:     KernelTimeMap[Target] = KernelTime
10:     if Accuracy < TOQ then FailedTarget := Target; break end if
11:     AccpetableTargetSet += Target
12:     if NormalConfig.ExecutionTime < NormalBestConfig.ExecutionTime then NormalBestConfig := NormalConfig end if
13: end for
14: initialize WildcardBestConfig
15: for Target in AcceptedTargetSet do
16:     initialize WildcardConfig
17:     for Event in RelatedEvents do
18:         WildcardConfig += getBestScalingMethod(Event, Original, Target, Set{ AcceptedTargetSet, FailedTarget } )
19:     end for
20:     WildcardConfig.ExpectedTime = getExpectedTransferTime( WildcardConfig ) + KernelTimeMap[Target]
21:     if WilecardConfig.ExpectedTime < WildcardBestConfig.ExpectedTime then WildcardBestConfig := WildcardConfig end if
22: end for
23: if WildcardBestConfig.ExpectedTime < NormalBestConfig.ExecutionTime then
24:     if FailedTarget is used as intermediate type in WildcardBestConfig then
25:         if RunApp(WildcardBestConfig).Accuracy < TOQ then
26:             return NormalBestConfig
27:         end if
28:     end if
29:     return WildcardBestConfig
30: else
31:     return NormalBestConfig
32: end if
```

$\#_{Conv\_Method}$, and $\#_{Event(m)}$, refer to the set of memory objects, the number of possible precision changes, conversion methods, and data transfer events, respectively. The number can be more increased when further considering parameters of conversion methods.

$$\#_{Tot} = \prod_{m \in MObj} (1 + \#_{Conv\_Type} \times \#_{Conv\_Method}{}^{\#_{Event}(m)}) \quad (1)$$

To minimize complexity, PreScaler adopts a search based on decision trees (via the decision maker) with a minimal number of program execution trials. It first sorts all memory objects in descending order of execution time, where the execution time of each memory object is defined as the sum of the execution times of related events. Then, possible target precision types and the scaling methods of each memory object are searched one-by-one to find the best configuration of the object per the minimum execution time. This process is performed by constructing a simple decision tree where each node corresponds to a memory object and edges are configurations with varying features related only to the input node memory object. Based on construction of the decision tree, the number of test configurations when using this scheme is reduced as shown in Equation 2.

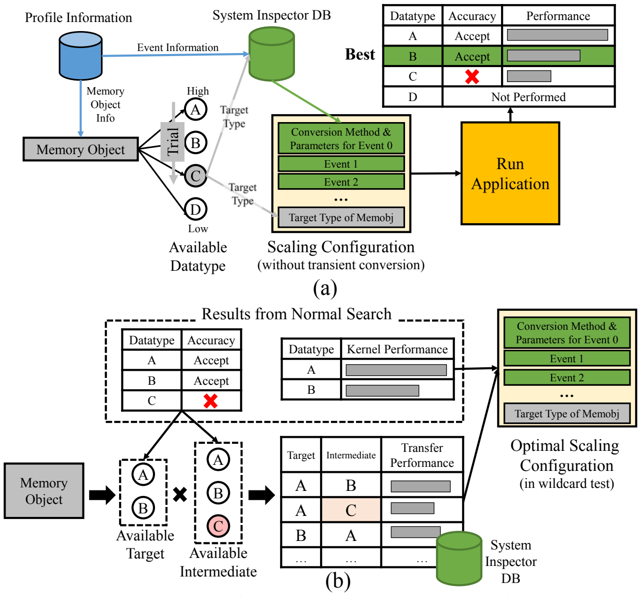


Figure 8. An optimal target scaling type and conversion method search process for a memory object: (a) a normal search and (b) a wildcard test.

$$\#_{Tot} = \sum_{m \in MObj} (1 + \#_{Conv\_Type} \times \#_{Conv\_Method}{}^{\#_{Event}(m)}) \quad (2)$$

The decision maker further reduces the number of execution trials by predicting the best conversion method and its parameters for given target scaling type from the information resulting of the system inspector. As discussed, the optimal conversion method can be system-dependent when the target scaling type and the size of the memory object are fixed. Based on this, the decision maker predicts the optimal conversion methods for all of the scaling points of a memory object with a specific target scaling type by relying on the system inspector information. The resulting decision tree is thus simpler as each node is only connected to only a small number of edges, which are the target scaling types. Therefore, the total number of scaling configurations to be tested by execution decreases significantly as shown in Equation 3.

$$\#_{Tot} = \#_{MObj} \times (1 + \#_{Conv\_Type}) \quad (3)$$

Figure 8 and Algorithm 1 illustrate how the decision maker obtains the optimal target scaling type and conversion method for a memory object. As shown in Figure 8, the process consists of two steps, a normal search (lines 1-13) and a wildcard test (lines 14-32). In the normal search, decision maker tries all of the types available for each memory object in descending order of precision (lines 2-13). In each trial, PreScaler determines the optimal conversion method for each event by calling a *getBestScalingMethod*() function to access the system inspector information database without actual execution (lines 5-7). (In the normal search, transient conversion is not allowed, and only direct type conversion is considered to minimize complexity.)

Table 2. PreScaler profiling and scaling wrapper libraries and their corresponding OpenCL API calls.

OpenCL API	Dynamic Profilig Library	Precision Scaling Library
clCreateProgramWithSource clCreateKernel	Record kernel information.	Compile OpenCL program with cusomized compiler. Compiler generates precision scaled kernel in all possible case
clCreateBuffer	Record memory object information.	Allocate device memory for memory object with scaled size Perform type conversion and data transfer
clEnqueueReadBuffer	Record host to device data transfer event.	
clEnqueueWriteBuffer	Record device to host data transfer event.	according to the predefined scaling configuration
clEnqueueNDRangeKernel clSetKernelArgs	Record kernel execution event with current kernel argument-memory object mapping snapshot. Guess type of mapped memory object and update memobj info.	Execute precision-scaled kernel.

The decision maker then checks the output accuracy and total execution time including kernel execution by performing actual target application execution (line 8). It does not continue to search for lower precisions if output accuracy of a tried precision does not meet the TOQ (line 10), i.e., type D is not tried due to the accuracy failure for the higher precision type C (Figure 8 (a)). After finding the best scaling configuration by the normal search, an additional *WildCard* test is performed to find a candidate configuration offering the best performance when also considering an use of the transient conversion with an intermediate type between original and target types (lines 15-22). Note that an actual application execution trial is not required because kernel execution durations for possible target types were already obtained from the normal search.

Algorithm 2 describes how the decision maker obtains the optimal conversion method for a specific target scaling type. Transient conversion, which performs conversions twice on both host and device using an intermediate type, can be considered, but this is checked only during wildcard test due to the high complexity. This is controlled by setting the intermediate type same to the target or original type in the normal search (Alg 1, line 6), and marking all the reasonable types as intermediate type in wildcard test (Alg 1, line 18).

Algorithm 2 Optimal Conversion Method Search

```
Require: Event, Original:Type, Target:Type, IntermediateCandidateSet:Set of Types
Ensure: HostSideMethod, IntermediateType, DeviceSideMethod
initialize BestHostSideMethod, BestIntermediate, BestDeviceSideMethod
for Intermediate in IntermediateCandidateSet do
    Host := getBestHostConversionMethod( Event, Original, Intermediate )
    Device := getBestDeviceConversionMethod( Event, Intermediate, Target )
    if isBest( Event, Host, Device, Intermediate ) then
        BestHostSideMethod := Host
        BestIntermediate := Intermediate
        BestDeviceSideMethod := Device
    end if
end for
return BestHostSideMethod, BestIntermediate, BestDeivceSideMethod
```

Though the actual execution trial is not required to find the best wildcard configuration because kernel execution time have already been measured in the normal search step, an additional execution trial for the chosen wildcard configuration is needed because new accuracy failures may occur when data are transferred using an intermediate type, as shown in Figure 8 (b) type C. (Alg 1, lines 24-28). The decision maker finally selects the best wildcard configuration as the optimal scaling configuration for a current memory object, when it expects better performance and does not incur accuracy failure. Otherwise, it selects the best configuration obtained through the normal search process. By determining the program-level mixed-precision configuration of all the memory objects through the decision tree, PreScaler can achieve high performance gain.

4.4.1 Initial Type Setting via Full-Precision Scaling

By using a decision tree based search, PreScaler can search for the optimal scaling configuration with minimum execution trials. However, PreScaler considers only one memory object per test, and may converge to a configuration that is the local minimum. To reduce this possibility, it first performs a pre-full-precision scaling, which finding the best configuration when all memory objects are of the same type, and then sets the resulting types to the initial types of the decision tree based search.

4.5 Implementation

To automate dynamic profiling and precision scaling without changing of original application codes, PreScaler is implemented using link-time interposition [4]. PreScaler wraps the original OpenCL APIs into a customized version and links the library to the application properly. Table 2 lists the OpenCL API calls modified by the PreScalar framework.

5 Experiment

5.1 Experimental Setup

Implementation The PreScaler framework consists of a system inspector and a PreScaler executable binary with custom runtime libraries. The system inspector is a single process to be executed once for a target system to construct a system information matrix. PreScaler receives a target OpenCL source code and a system information matrix as inputs, and performs program-level precision scaling of the target application based on the information collected by system inspection and application profiling. When the best scaling configuration is found, PreScaler generates an executable binary of the target OpenCL program. PreScaler is implemented on an LLVM [15] compiler infrastructure. The host code is compiled using Clang [1] 6.0 with O3 option, and the precision scaling of kernel code is performed on LLVM [15] 6.0.

Performance Comparison The PreScaler performance was compared with an original program and two different

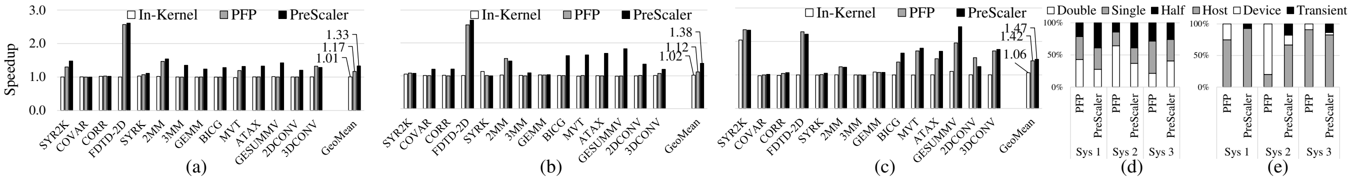


Figure 9. Performance comparison of In-Kernel, PFP, and PreScaler on three different systems (default input set): (a) system 1, (b) system 2, and (c) system 3. Result type and data conversion method distribution on different target systems: (d) type and (e) conversion. All the data are normalized to that of a baseline.

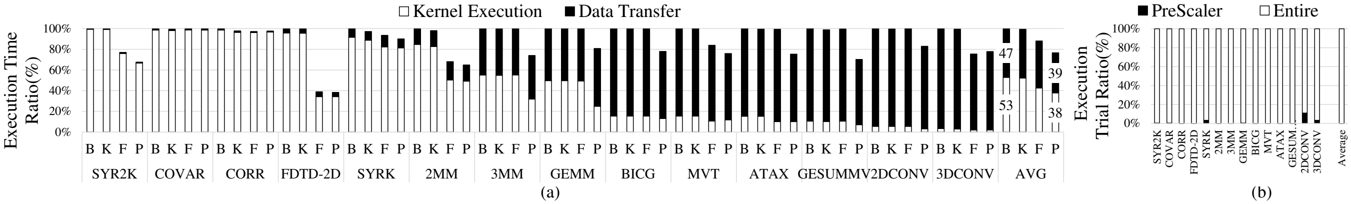


Figure 10. A detailed analysis of each scaling technique on system 1. (a) normalized execution time and (b) number of execution trials. "B", "K", "F", and "P" indicates Baseline, In-Kernel, PFP, PreScaler, respectively.

scaling techniques: Baseline, In-Kernel scaling, and Program-level Full Precision (PFP). In-Kernel represents the kernel-level mixed-precision scaling technique. In this method, we insert type conversion instructions inside the target kernel to scale input data. To ensure fair performance gain, we test all possible configurations, and therefore, a performance gain similar to the Precimonious [33] framework is expected. PFP represents the program-wide full precision scaling technique, where all memory objects used in the kernel have the same precisions. Therefore, we select the best performance configuration after testing all available type configurations. For type conversion, both a host-side multithreaded method and device-side method are considered, where the number of threads is set to the number of logical CPU cores. PFP is considered as a manual program-level precision scaling optimization by experienced programmers.

Table 3. Target system configurations

	System 1	System 2 [23]	System 3
OpenCL Driver	OpenCL 1.2		
CPU	Xeon E5-2640v4 [9]	Xeon E5-2698v4 [9]	Xeon Gold 5115 [10]
Number of Core/Threads	10 / 20	20 / 40	10 / 20
MAX CPU Clock	3.40GHz	3.60GHz	3.40GHz
Instruction Set Extension	AVX2	AVX2	SSE 4.2, AVX AVX2, AVX-512
GPU	Titan Xp [25]	Tesla V100 [24]	2080Ti [27]
Number of SMs	30	80	68
MAX GPU Clock	1582MHz	1380MHz	1545MHz
Bandwidth	PCIe 3.0 x16 (x8)	PICe 3.0 x16	PCIe 3.0 x 16
GPU Compute Capability	6.1(Pascal)	7.0(Volta)	7.5(Turing)

Table 4. Benchmark specification

		Input Range			Input Range		
Benchmark	Input Size	Default	Benchmark	Input Size	Default	Image	Random
2DCONV	16MB	0.0-1.0	COVAR	4MB	0.0-2048.0	0.0-255.0	0.0-1.0
2MM	16MB	0.0-2051	FDTD-2D	4MB	-9.01-2041		
3DCONV	16MB	0.0-59.0	GEMM	0.25MB	0.0-513.0		
3MM	1MB	0.0-515.0	GESUMMV	16MB	0.0-4096		
ATAX	16MB	0.0-4094.0	MVT	16MB	0.0-2.0		
BICG	16MB	0.0-4096×π	SYR2K	4MB	0.0-2050.0		
CORR	4MB	0.0-2047.0	SYRK	1MB	0.0-1026.0		

Benchmarks and System Configuration We evaluated the performance of the PreScaler framework on various environments running the Ubuntu Linux OS, as shown in Table 3: (1) a Xeon E5 and NVIDIA Titan Xp, (2) a Xeon E5 and NVIDIA Tesla V100 (DGX Station), and (3) a Xeon Gold and NVIDIA RTX 2080 Ti. All the CPUs/GPUs support double-, single-, and half-precision operations. For host-side type conversion, we use an open-source half-precision math library [32] to support half-precision data types, and also extensively utilize SIMD instructions such as SSE and AVX for faster type conversion.

We select 14 applications in the Polybench benchmark suite [31]. We used mean relative error as the error metric and set default TOQ to 90%. We use three input sets with different data value ranges: Default, Image, and Random. For the image input set, we use ILSVRC 2012 [34] datasets. Detailed information on the applications and their input sets is provided in Table 4.

5.2 Performance Evaluation on Various Systems

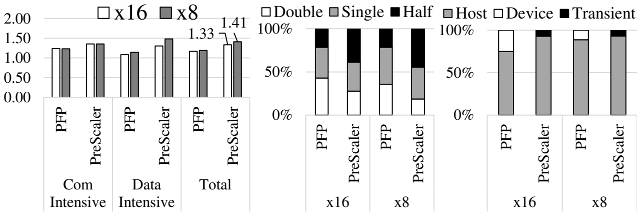
Figure 9 shows the performance improvements of all the benchmarks on three different systems, which are listed in Table 3. Each bar indicates the relative performance improvements of In-Kernel, PFP, and PreScaler, normalized to the baseline (original precision). As shown in the figure, PreScaler achieves a fair speedup of 1.33x, 1.38x, and 1.47x, and all the gains are substantially higher than those of In-Kernel and PFP. This is mainly because PreScaler can effectively minimize the data transfer overhead as well as determine the optimal mixed precision configuration.

In-Kernel methods cannot improve the performance of most programs significantly. This is because (1) kernel execution does not dominate the program performance in data-intensive applications and (2) additional type conversion instructions incur large overheads in kernels. Unlike In-Kernel, memory object-level scaling such as PFP and PreScaler improve kernel performance without inserting type conversion instructions in kernels, and therefore, they can achieve a better performance in both kernels and the entire program. Figure 10 (a) shows that PFP and PreScaler not only improve data transfer but also enhance kernel execution, and their kernel performances are often better than that of In-kernel scaling.

These results also indicate that mixed-precision is important in memory object-level precision scaling as the performance of PreScaler is higher than that of PFP. Figure 9 (d) and (e) show the distribution of the resulting memory object types and the conversion methods for all the applications. As shown in the figure, PreScaler achieves a better performance by utilizing more lower precision types than PFP. This explains why fine-grained mixed-precision is an important factor in performance gain. Figure 9 (e) also shows the effectiveness of the wildcard approach by showing considerable fractions for transient conversion.

5.3 Number of Execution Trials

Figure 10 (b) shows the total number of execution trials required to determine the optimal scaling configuration among whole search space. As discussed in Section 4.4, the number of all possible scaling configurations can be calculated using Equation 1. PreScaler performs actual execution trials of only a few configurations by adopting a decision tree based search using a system-information-based performance estimation. As a result, PreScaler tests only 0.00001% of all possible cases. Note that the "Entire" number of configurations in Figure 10 (b) is calculated by considering only four conversion methods (loop/multithread/pipeline/device) without consideration of more detailed information such as the number of threads.



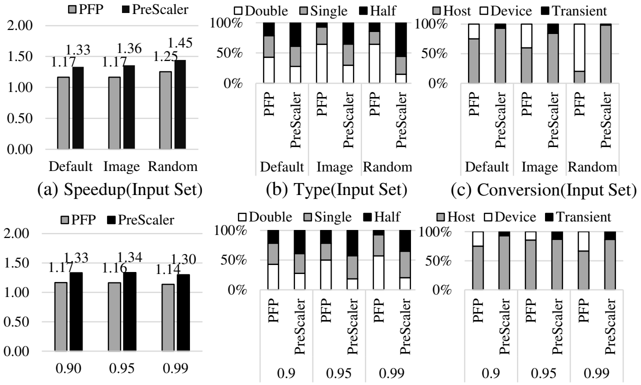
(a) Speedup (b) Type (c) Conversion

Figure 11. System adaptivity with different PCIe bandwidths (x8, x16): (a) speedup, (b) type distribution, and (c) data conversion method distribution.

5.4 System Adaptivity: PCIe Bandwidth

To evaluate the adaptivity on systems, we measured the speedup of all the applications with different transfer bandwidths. To control the transfer bandwidth, we evaluated PreScaler on a system which is identical to system 1 in all aspects except the PCIe bandwidth. When the bandwidth is reduced from x16 to x8, the transfer time increases while the kernel time still maintained at a constant value. By the result, the execution time of data-intensive applications increases while the execution time of computation-intensive ones are similar.

Figure 11 (a) shows the performance improvements of PreScaler and PFP on both the original system (x16) and the limited bandwidth system (x8). As shown in Figure 11 (a), PreScaler shows a speedup of 1.41x on x8, which is greater than that of 1.33x on 16x. This is because the portion of transfer time in precision scaling increases. This portion of increased transfer time indicates that the transfer gain potential also increases, and therefore, we can employ more lower precision scaling opportunities that were not chosen due to insufficient gain. Figure 11 (b) shows that a greater number of memory objects are scaled into lower precisions.



(a) Speedup(Input Set) (b) Type(Input Set) (c) Conversion(Input Set)

(d) Speedup(TOQ) (e) Type(TOQ) (f) Conversion(TOQ)

Figure 12. Application adaptivity with different input sets (Default, Image, Random) and different TOQs (0.90, 0.95, 0.99)

5.5 Application Adaptivity: Input Set and TOQ

As discussed in Section 3.2, the best scaling configuration varies highly depending on application specific features. The optimal configuration of these applications can also be different based on the TOQ. Therefore, we evaluate the effectiveness of the PreScaler framework when changing application related features: input set, and TOQ.

Figure 12 (a) shows the speedup of PreScalar for three different input sets as shown in Table 4, and PreScalar shows the fair speedup of 1.33x, 1.36x, and 1.45x with Default, Image, and Random input sets, respectively. The performance result with random inputs shows the highest speedup because the ratio of half precision memory objects is higher than other input sets as it does not have many TOQ errors due to the small value range (0.0 1.0) (Figure 12 (b), (c)). Figure 12 (d) shows the speedup of PreScalar framework when changing the TOQ from 0.9 to 0.99 for a default input set, and it proves the effectiveness by retaining fair performance gain even with higher TOQ.

6 Related works

Approximation Framework As approximate computing is effective, several approximation frameworks have been introduced [3, 11, 16, 35]. Green [3] is a well-known approximation framework that automatically approximates target code region with specific annotations. GATE [11] is another approximation tuning framework that uses a subgraph-level approximation technique. While these frameworks are similar to PreScaler, the target approximation techniques are different from precision scaling, and thus, PreScaler framework is an orthogonal approach to them.

Precision Scaling on CPUs There are several precision scaling frameworks [20, 33, 38] to scale precisions of CPU-target programs to achieve high performance gains. EnerJ [38] suggests a Java-based precision scaling model with special annotations provided by programmers. For the variables for precision scaling, EnerJ approximates based on energy saving and performance estimation. Precimonious [33] is another well-known precision scaling framework, which is also implemented on an LLVM compilation framework. It creates a unique structure for each target variable and determines the best configuration by evaluating two versions with different precisions. While these precision scaling frameworks look highly related to PreScaler frameworks, there is a substantial performance difference between PreScaler and other frameworks when used in to GPGPU programs: PreScaler focuses mainly on memory objects, and therefore, it can achieve a higher performance gain by reducing the data transfer cost, whereas other frameworks cannot minimize the data transfer cost as they consider only variables.

Precision Scaling on GPUs While several CPU-based precision scaling frameworks have been introduced as discussed above, few studies has been conducted with regard to the GPU target. Among the frameworks reported in these studies, GPUMixer [13] is the mostly closest related to PreScaler, however this framework cannot improve performance of data-intensive programs because it improves the kernel performance by maximizing the ratio of low-precision arithmetic operations inside the target kernels. Therefore, PreScaler can be a better precision scaling approach as it can improve both kernel and data transfer performance.

7 Conclusion

In this work, an automatic program-level precision scaling framework called *PreScaler* is proposed to efficiently accelerate OpenCL applications. It first proposes a memory object-level scaling to cover both data transfers and kernel executions. It further provides a simple but effective decision-tree-based precision configuration search mechanism to minimize the searching space. The best data transfer with type conversion method per each configuration can be easily decided without execution trials, using system information. *PreScaler* shows 1.33x average speedup without accuracy problems for polybench workloads, on a baseline system.

A Artifact Appendix

A.1 Abstract

Our artifact provides all the binaries, libraries, benchmarks, and scripts, for evaluating our PreScaler implementations. These require target heterogeneous systems that contain Intel CPUs and NVIDIA GPUs. We especially recommend testing on the DGX Station(System 2) [23].

We provide our artifact in two forms: Docker images and native binaries for several different system environments. These include installation scripts for software dependencies except NVIDIA Graphic Driver and CUDA toolkit. When you use a Docker image, all the software dependencies are preinstalled.

A.2 Artifact Check-list (Meta-information)

- **Algorithm:** Automated precision scaling considering both kernel execution and data transfer of GPGPU programs.
- **Program:** PreScaler binaries, libraries, and benchmark source codes.
- **Binary:** Included for Linux (Ubuntu 16.04 recommended) for x86-64.
- **Hardware:** We recommend heterogeneous systems with Intel CPUs and NVIDIA GPUs.
- **Output:** Performance improvement data files (CSV format) are output.
- **Experiments:** Manual Linux shell scripts.
- **How much disk space is required (approximately)?:** About 10GBs.
- **How much time is needed to complete experiments (approximately)?:** For system inspection (one time overhead), it requires about several hours to three days (particularly long for multi-GPU systems). For the searching process of optimal scaling configurations of all the benchmarks after system inspection, it requires about three hours (10 to 30 minutes per benchmark).
- **Publicly available?: Yes**

A.3 Description

A.3.1 How Delivered

Our binaries, libraries, benchmarks, and scripts are available on Zenodo: https://doi.org/10.5281/zenodo.3605359 [1]

A.3.2 Hardware Dependencies

PreScaler works on heterogeneous systems. Our artifact requires a target system that includes Intel CPUs which can execute SSE3 and AVX intrinsic instructions and NVIDIA GPUs whose architectures are Pascal, Volta, or Turing. We perform experiments on three different systems, as shown in Table 3. We recommend evaluating PreScaler performance on the DGX Station (System 2) to reproduce the results shown in this paper. Other similar systems will give comparable results.

A.3.3 Software Dependencies

Our binaries and libraries assume that they are executed using Ubuntu 16.04 on x86-64 systems. These also require the NVIDIA Graphic Driver [26](4xx.x recommended) and CUDA [21] toolkit (9.0 or higher. 10.0 recommended). These should be installed according to NVIDIA's instructions.

Our binaries also require several packages and libraries, including an LLVM [15] framework and a half precision math library [32], as software dependencies. These dependencies can be easily installed through our provided scripts. If you use a given Docker

[1] The artifact is also available on Github: https://github.com/HyuCASS/CGO20PreScalerArtifact

image for evaluation, the Nvidia-docker [28] is required, and all the software dependencies are preinstalled on the image.

A.4 Installation

After downloading and extracting the compressed repository file, *PreScaler_Artifact.tar*, You can use either Docker images or native binaries to evaluate.

Docker You can load the Docker image as follows:

```
$ docker load -i PreScaler.Docker.tar
$ nvidia-docker run -d -it --name PreScaler
  prescaler/release /bin/bash
$ docker exec -it PreScaler /bin/bash
$ cd
```

All software dependencies are preinstalled in the images.

Manual You can install PreScaler's software dependencies as follows:

```
$ tar -xvf PreScaler.Binary.tar
$ bash install.sh
```

This can take up to five hours.

A.5 Experiment Workflow

PreScaler consists of two processes: (1) collecting system information through the system inspector and (2) searching optimal scaling configurations of target applications through the application profiler and decision maker. The system inspector process is performed only once per system, and the application profiler and decision maker processes are performed for each application. You can skip the system inspector process by using precollected informations when you evaluate on our evaluation systems (including the DGX station [23]).

A.5.1 System Inspector

Navigate to home directory. Run the following commands:

```
$ source setup_environment.sh
$ PreScaler/bin/system_inspector/insepct_all
```

This process can take between several hours to three days. Please note that you have to perform this process once per target system. For more accurate inspection, we recommend that you do not perform any other host/device jobs during this process.

A.5.2 Application Profiler and Decision Maker

To run benchmarks continuously, navigate to home directory and run the following commands:

```
$ source setup_environment.sh
$ bash Benchmark/Polybench-1.0/run_all.sh
```

This takes about three hours.

To run each benchmark, navigate to home directory and run the following commands:

```
$ source setup_environment.sh
$ cd Benchmark/Polybench-1.0/<BenchmarkDirectory>
$ make framework_execution
```

This takes about 10 to 30 minutes for each benchmark.

A.6 Evaluation and Expected Result

Navigate to *Benchmark/log*. A CSV file will be output for every benchmark. This includes kernel execution time, data transfer time, accuracy, and the number of trials for each application. You can compare these results with the corresponding results in this paper on similar systems. Depending on the CPUs, GPUs, and their connections to the evaluation systems, you can observe comparable speedups and accuracy losses.

A.7 Experiment Customization

All our scripts are customizable. You can change the input data or TOQ by modifying *Benchmark/Polybench-1.0/run_all.sh.* You can also observe the performance improvements on different systems by changing the system configuration. In order to apply our technique to other benchmarks, our framework requires several application information such as build scripts, execution scripts, data transfer time, kernel execution time, and accuracy for each execution trials. When system inspection is finished, you can apply PreScaler to other OpenCL applications by running PreScaler/bin/-precision_scaler/framework.

Acknowledgments

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-03. Yongjun Park is the corresponding author.

References

[1] Clang. a C language family frontend for LLVM, 2007. http://clang.llvm.org.

[2] A. Anant, M. C. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. 2009.

[3] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *In ACM SIGPLAN Conference on Programming language design and implementation*, PLDI '10. Citeseer, 2010.

[4] R. E. Bryant, O. David Richard, and O. David Richard. *Computer systems: a programmer's perspective*, volume 281. Prentice Hall Upper Saddle River, 2003.

[5] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[6] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[7] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière. Autotuning for floating-point precision with discrete stochastic arithmetic. *Journal of Computational Science*, 2019.

[8] S. Hong, I. Lee, and Y. Park. Nn compactor: Minimizing memory and logic resources for small neural networks. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 581–584. IEEE, 2018.

[9] INTEL. Intel xeon e5-2600 model specification, 2016. https://www.intel.com/content/www/us/en/processors/xeon xeon-e5-brief.html.

[10] INTEL. Intel gold 5115 model specification, 2017. https://ark.intel.com/content/www/kr/ko/ark/products/120484 intel-xeon-gold-5115-processor-13-75m-cache-2-40-ghz.html.

[11] S. Kang, Y. Yu, J. Kim, and Y. Park. Gate: A generalized dataflow-level approximation tuning engine for data parallel architectures. In

Proceedings of the 56th Annual Design Automation Conference 2019, page 24. ACM, 2019.

[12] KHRONOS Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2010. http://www.khronos.org.

[13] I. Laguna, P. C. Wood, R. Singh, and S. Bagchi. Gpumixer: Performance-driven floating-point tuning for gpu scientific applications. In *International Conference on High Performance Computing*, pages 227–246. Springer, 2019.

[14] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 369–378. ACM, 2013.

[15] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[16] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang. Input responsiveness: Using canary inputs to dynamically steer approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 161–176, New York, NY, USA, 2016. ACM.

[17] D. Lustig and M. Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 354–365. IEEE, 2013.

[18] S. Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, Mar. 2016.

[19] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. Snnap: Approximate computing on programmable socs via neural acceleration. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 603–614. IEEE, 2015.

[20] R. Nathan, H. Naeimi, D. J. Sorin, and X. Sun. Profile-driven automated mixed precision. *arXiv preprint arXiv:1606.00251*, 2016.

[21] J. Nickolls et al. NVIDIA CUDA software and GPU parallel computing architecture. In *Microprocessor Forum*, May 2007.

[22] NVIDIA. NVIDIA Tesla P100, 2016. https://https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[23] NVIDIA. NVIDIA DGX Station, 2017. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-station/nvidia-dgx-station-datasheet.pdf.

[24] NVIDIA. NVIDIA Tesla V100, 2017. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[25] NVIDIA. NVIDIA Titan Xp Graphics Cards, 2017. https://www.nvidia.com/en-us/titan/titan-xp/.

[26] NVIDIA. Nvidia Graphic Driver, 2018. Available at https://www.nvidia.com/Download/index.aspx.

[27] NVIDIA. NVIDIA RTX 2080 Ti Graphics Cards, 2018. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/.

[28] NVIDIA. Nvidia Container Toolkit. build and run docker containers leveraging nvidia gpus, 2019. Available at https://github.com/NVIDIA/nvidia-docker.

[29] NVIDIA. Throughput of native arithmetic instructions, 2019. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[30] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris. Flexjava: Language support for safe and modular approximate programming. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 745–757. ACM, 2015.

[31] Polybench. the polyhedral benchmark suite, 2011. http://www.cse.ohio-state.edu/ pouchet/software/polybench.

[32] C. Rau. Ieee 754-based half-precision floating-point library, 2017. http://half.sourceforge.net.

[33] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[34] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[35] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 35–50, New York, NY, USA, 2014. ACM.

[36] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. Sage: Self-tuning approximation for graphics engines. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 13–24, Dec 2013.

[37] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01*, 1(2), 2015.

[38] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[39] Y. Tian, Q. Zhang, T. Wang, F. Yuan, and Q. Xu. Approxma: Approximate memory access for dynamic precision scaling. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 337–342. ACM, 2015.

[40] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan. Axnn: energy-efficient neuromorphic systems using approximate computing. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 27–32. ACM, 2014.

[41] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu. Approxann: An approximate computing framework for artificial neural network. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 701–706. EDA Consortium, 2015.