Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3316781.3317833

RESEARCH-ARTICLE

GATE: A Generalized Dataflow-level Approximation Tuning Engine For Data Parallel Architectures

SEOKWON KANG, Hanyang University, Seoul, South Korea

YONGSEUNG YU, Hanyang University, Seoul, South Korea

JIHO KIM, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

YONGJUN PARK, Hanyang University, Seoul, South Korea

Open Access Support provided by:

Hanyang University

Korea Advanced Institute of Science and Technology

PDF Download
3316781.3317833.pdf
03 March 2026
Total Citations: 1
Total Downloads: 241

Published: 02 June 2019

Citation in BibTeX format

DAC '19: The 56th Annual Design Automation Conference 2019
June 2 - 6, 2019
NV, Las Vegas, USA

Conference Sponsors:
SIGDA

GATE: A Generalized Dataflow-level Approximation Tuning Engine For Data Parallel Architectures

Seokwon Kang[1], Yongseung Yu[1], Jiho Kim[2], and Yongjun Park[1]
[1]Deptartment of Computer Science, Hanyang University, Seoul, Republic of Korea
[2]School of Electrical Engineering, Korea Advanced Institute of Science and Techinology, Daejon, Republic of Korea
{kswon0202,yongseung91}@gmail.com,jihokim@kaist.ac.kr,yongjunpark@hanyang.ac.kr

ABSTRACT

Although approximate computing is widely used, it requires substantial programming effort to find appropriate approximation patterns among multiple pre-defined patterns to achieve a high performance. Therefore, we propose an automatic approximation framework called *GATE* to uncover hidden opportunities from any data-parallel program regardless of the code pattern or application characteristics using two compiler techniques, namely subgraph-level approximation (SGLA) and approximate thread merge (ATM). GATE also features conservative/aggressive tuning and dynamic calibration to maximize the performance while maintaining the TOQ level during runtime. Our framework achieves an average performance gain of 2.54x over the baseline with minimum accuracy loss.

1 INTRODUCTION

Despite the considerable evolution of emerging applications such as machine learning and image processing, they have encountered difficulties with regard to data throughput. Massively data-parallel architectures such as graphic processing units (GPUs) are the most successful accelerators for speeding up these computations, as they show high throughput with sustainable power budgets and are well supported by data-parallel programming models such as CUDA [9] and OpenCL [5]. However, recent trends demonstrate the need for improvements in performance to satisfy market demands.

Approximate computing, or trading-off output accuracy for performance, is becoming an attractive technique for realizing further performance gains, because 100% accuracy is not always mandatory and some quality degradation is tolerable for a wide range of applications [8]. For example, it may not be necessary to process numerous pixel data with perfect accuracy if the quality loss is not perceivable by human visual systems.

Thus, effective framework support for approximate computing is essential for achieving both high performance and a tolerable error rate. The main challenges in this regard are 1) how to approximate and 2) how to satisfy the target output quality (TOQ) during runtime. Many techniques such as loop perforation [1], SAGE [15], Paraprox [14], and IRA [7] have been proposed to solve these problems using similar core concepts: 1) multiple possible code patterns are proposed to detect the maximum number of approximation opportunities; 2) multiple approximation version codes are generated statically; and 3) an appropriate version is chosen during runtime to achieve high performance with acceptable output quality. While these techniques achieve yield considerable improvements in terms of performance and energy, several critical limitations still exist due to the aforementioned core structure.

[1. Limited Coverage and High Programming Effort]: Approximate computing typically improves performance by removing several operations in the target applications, and the candidate operations are detected by using various predefined patterns. As each pattern cannot cover all the opportunities, multiple predefined specific patterns are required to cover a wide range of applications, and additional patterns is required to achieve further performance gain. While these pattern-based techniques usually support automatic code transformation, they still require substantial programming effort because predefined patterns are often not detected even when the pattern exists algorithmically. Therefore, programmers must understand all the approximation patterns and rewrite the program to achieve better coverage and performance. Therefore, a more general a approach than pattern-based approaches is required for the approximation to be widely used.

[2. No Minimum Performance Gain Estimation]: As the output quality of approximate programs changes dynamically depending on the input values, it is crucial to track the change in output quality during runtime and re-tune the approximation level when the quality is not within the acceptable range. Most prior approaches support a dynamic adjustment mechanism using changes in the running kernels with varying degrees of accuracy without any estimation of worst-case performance gain. In this situation, further optimization can be applied if the minimum performance gain can be measured. For example, if we can expect a performance improvement of at least 20% performance improvement from approximation with any inputs, the operating frequency can be lowered while maintaining the minimum performance requirement.

To solve these limitations, we introduce two dataflow-based low-level approximation techniques—*subgraph-level approximation (SGLA)* and *approximate thread merge (ATM)*—which can achieve broader applicability with minimum programming overhead. SGLA refers to approximation based on groups of instructions (subgraphs) in a basic block. In SGLA, the result value of the last consumer or an input value of the first producer node of a subgraph can be replaced by the value of a different subgraph if the output quality is higher than the TOQ. SGLA can offer a considerable number of approximation opportunities by covering a wide range of patterns as most patterns usually attempt to find 1) load instructions or 2) computations with similar input/output values. To extend the coverage of SGLA from one thread to multiple threads, we also propose

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317833

ATM to merge multiple threads into a large thread. Through ATM, SGLA can perform inter-thread approximation similar to thread fusion [15] as well as intra-thread approximation.

This paper proposes a *Generalized Approximation Tuning Engine (GATE)* framework, which is a novel automatic runtime approximation management system for data-parallel architectures using the above two compiler techniques. The GATE framework uses the information of an original target program in OpenCL [5], TOQ, and error metrics from programmers. Based on the information, GATE performs a three-phase process consisting of conservative tuning, aggressive tuning, and runtime calibration. In the conservative tuning process, GATE first enumerates the compiler level approximation opportunities and evaluates them by using random inputs. Accordingly, GATE generates a conservatively approximate kernel by applying input-insensitive optimizations to achieve a minimum performance gain. GATE subsequently performs aggressive tuning to gather further approximation opportunities using real inputs and generates an aggressively approximate kernel by applying all the newly accepted optimizations. Finally, GATE performs runtime calibration to adjust the approximation level dynamically using just-in-time re-compilation to avoid violating TOQ during runtime without substantial performance loss.

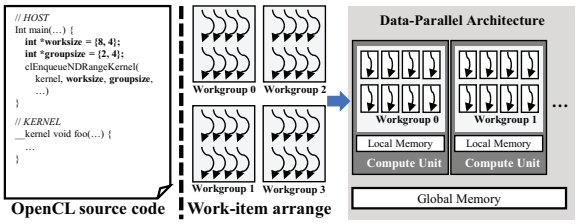


Figure 1: An OpenCL execution flow on data parallel devices.

2 BACKGROUND AND MOTIVATION

2.1 Data Parallel Program Execution

Figure 1 shows the process used to execute an OpenCL program on a data-parallel device. OpenCL programs consist of a host program and several kernels. The kernel functions are executed on the target device by a call of the OpenCL host API with the number of work groups and the number of threads in each work group. Each workgroup is assigned to a compute unit and threads are executed concurrently on multiple compute elements in each compute unit.

2.2 Beyond Conventional Approaches

Several existing approximation techniques [1, 7, 14, 15] rely on removing some of the multiple memory/compute operations that are expected to have similar values or loop iterations. The approximation processes are usually detected using predefined special patterns such as stencil, partition, and reduction [14]. As the output quality can vary depending on input values, most approximation techniques also provide runtime quality checks and approximation level adjustment mechanisms. In this analysis, two limitations of these previous approaches are addressed.

2.2.1 *Dataflow-level Generalization.*

As discussed, most approximation approaches propose their own special patterns to determine approximation opportunities. This indicates that the effectiveness of the approximation can vary considerably depending on the target application property or coding

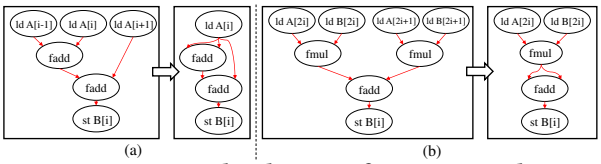


Figure 2: Instruction-level view of conventional pattern-based approximation techniques. (a) Stencil, (b) Reduction

style. For example, loop perforation [1] is an approximation technique that regularly skips some subsets of iterations in a target loop, and the performance gain is reduced if the application does not have any loops inside or the loops are unrolled. Stencil/partition [14] is an approximation technique for detecting adjacent elements in the input array, but the pattern may not be detected if the dimension of the input array of the target application does not exactly match that of the predefined patterns. Therefore, pattern-based approaches cannot guarantee effectiveness for every program.

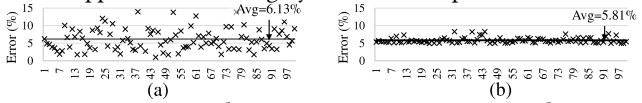
For this problem, the use of an instruction-level dataflow-based approach can be an attractive solution. Among the various proposed patterns, most patterns can be detected at the instruction level and the proposed solutions can also be applied at the instruction level. Figure 2 shows instruction-level approaches of several pattern-based approximation techniques. Figure 2 (a) shows that stencil [14] can be identified at the instruction level by detecting the adjacent load instructions, and the approximation process of the patterns can also be applied at the instruction level by replacing the load instruction result values with the result value of the subsequent instruction and eliminating the load instructions because they then become dead codes having no consumers. Figure 2 (b) is an example of reduction [14] involving identification of pairs of compute instructions and removal of several instructions by not using the result value of a compute instruction. Examples in Figure 2 thus indicate that most pattern-based approximation techniques can be generalized to an instruction-level approach.

As shown in Figure 2, most pattern-based approximation techniques can be generalized by replacing similar instructions. More specifically, in the conventional pattern-based approach, three types of instructions can be replaced with each other: (1) adjacent load instructions; (2) instructions that represent the result of a loop iteration; and (3) compute instructions stored in similar locations. In cases (1) and (3), the instruction pairs can be detected easily by finding the load and store instructions with the same base addresses. However, in case (2), the instruction pairs cannot be as easily determined as in cases (1) and (3). Furthermore, to avoid limited coverage and the requirement of high programming effort, a key concern for the instruction-level approach is to determine the instruction pairs systemically. To address this issue, a subgraph-level approach can be an appropriate solution for higher coverage.

2.2.2 *Two Phase Tuning.*

Another limitation of the previous approaches is that they do not consider the worst-case performance gain. As most runtime quality adjustment systems assume that the output quality always depends on input value, they cannot evaluate the minimum performance gain. However, not all approximation opportunities are strongly dependent on input value variance. Figure 3 shows the errors when executing an approximate version of 2D convolution and backpropagation benchmarks using an image input set [13].

Stencil [14], which shares memory accesses in between, is applied to these two benchmarks. As shown in figures 3 (a) and (b), the result of 2D convolution has an average error of 6.13% with a high input variance, whereas the result of backpropagation has an average error of 5.81%, which is similar to that of 2D convolution with a small input variance. These results indicate that not all approximation opportunities are highly sensitive to input values.



(a) (b)

Figure 3: Error rate changes in approximate stencil pattern applications over 100 test images. (a) 2D-Convolution, (b) Backpropagation

Therefore, we can assume that, if only the input-insensitive approximation opportunities are applied to a program, the output quality of the program will not change considerably with any input. This indicates that input-insensitive approximation opportunities can be applied before runtime and can ensure a minimum level of performance improvement while ensuring better quality than TOQ. Further performance gain can be achieved by applying input-sensitive opportunities at runtime.

3 DATAFLOW-LEVEL APPROXIMATION

In this section, the new approximation techniques proposed in the GATE framework are presented. We first introduce the SGLA technique including forward and backward processes to maximize the approximation effectiveness inside the dataflow of an original target kernel. We then introduce the ATM technique to achieve further gain by merging multiple threads.

3.1 Subgraph-level Approximation

GATE generalizes prior pattern-based approximation techniques using dataflow analysis. To address the challenge, GATE uses SGLA, which is a novel approximation approach to obtain approximate subgraphs that 1) have an isomorphic form of dataflow with the same operations and 2) have approximation candidate instructions that are replaced with the corresponding instructions for other subgraphs, and to remove the candidate instructions and connect all the consumers of the removed instructions with the outputs of the corresponding instructions of the other subgraphs. The approximation candidate instructions are usually memory accesses or floating-point instructions having high execution latency, which are expected to have similar outputs.

Algorithm 1 shows how GATE detect all subgraphs in SGLA. We defined two types of subgraphs: Producer-Based Detected Subgraph (PBDS) and Consumer-Based Detected Subgraph(CBDS). The PBDSs are based on a high-level idea that adjacent elements of an array possess similar values and one of these values can be replaced by other elements. In contrast, CBDSs are based on the idea that adjacent multiple arithmetic results (to be stored in memory) can be replaced with a single arithmetic result when the target store addresses are similar. Hence, GATE searches for PBDS based on the producer-to-consumer relationship and CBDS based on the consumer-to-producer relationship. We call the forward process to detect the PBDS and the backward process to detect the CBDS.

PBDSs are defined as follows: (1) Each load instruction itself is a PBDS; (2) If all the producers of an instruction are the last consumer instruction of other PBDSs or are constant, a new PBDS includes the instruction and all the elements of the PBDSs whose last consumer is one of the producers. CBDSs are defined as follows: (1) A store instruction and its direct producer constitute a CBDS; (2) If one of the direct consumers of an instruction is the first producer of a CBDS, a new CBDS includes the instruction and all the elements of the CBDS.

Algorithm 1 Subgraph Detection

```
Require: Kernel, TargetOutputQuality, IsForwardProcess
Ensure: SetOfSubgraphs
 1: if IsForwardProcess then SetOfSubgraphs := GetLoadInsts(Kernel)
 2: else SetOfSubgraphs := GetDirectProducersOfStoreInst(Kernel)
 3: end if
 4: SubgraphExist := true
 5: while SubgraphExist do
 6:     SubgraphExist := false
 7:     PreviousSet := SetOfSubgraphs
 8:     SetOfSubgraphs := GetSubgraphs(Kernel, SetOfSubgraphs, IsForwardProcess)
 9:     if PreviousSet != SetOfSubgraphs then SubgraphExist := true end if
10: end while
11: if IsForwardProcess then SortDescendingOrder( SetOfSubgraphs )
12: else SortAscendingOrder( SetOfSubgraphs )
13: end if
```

Each subgraph has a unique approximate candidate instruction that is the replacement target with other subgraphs through the SGLA. The last consumer instruction of the PBDS and the first producer instruction of the CBDS are the candidate instructions.

Through the definition of subgraphs, GATE starts detecting PBDSs from loads in the forward process, and CBDSs from the direct producers of the stores in the backward process (line 1-3). GATE then iteratively detects subgraphs until there are no more subgraphs to be detected, according to the definition of PBDS and CBDS (line 4-9). After all subgraphs are detected, GATE sorts the subgraphs in descending (PBDS) or ascending (CBDS) order of the height of subgraphs to maximize the approximation gain(line 11-13). The height of a subgraph is defined as the longest distance between its producers and consumers. This is because a large number of instructions can be eliminated when a larger PBDS or smaller CBDS is replaced.

Algorithm 2 Approximation Kernel Generation

```
Require: SetOfSubgraphs, Kernel, TOQ, IsForwardProcess
Ensure: ApproximateKernel
 1: ApproximateKernel := Kernel
 2: for Target in SetOfSubgraphs do
 3:     MaxAffCost := 0; BestSource := null
 4:     for Source in SetOfSubgraphs do
 5:         if not CheckCondition(Target, Source, IsForwardProcess) then continue end if
 6:         AffCost := GetAffinityCost( Target.CandInst, Source.CandInst )
 7:         if MaxAffCost < AffCost then MaxAffCost := AffCost; BestSource := Source end if
 8:     end for
 9:     if BestSource == null then continue end if
10:     ApproximateKernel := ReplaceInst( Target.CandInst, BestSource.CandInst )
11:     Accuracy := Execute(ApproximateKernel)
12:     if Accuracy < TOQ then RollBack(ApproximateKernel) end if
13: end for
```

Algorithm 2 shows the approximation process by iteratively picking a target subgraph based on the sorted detected subgraphs. GATE then detects subgraphs similar to the target subgraph using a simple condition-checking function (line 5). In this function, GATE first determine two subgraphs have an isomorphic form of dataflow with the same operation. For the PBDS pair, GATE then checks the load instructions in PBDS having the same base addresses as the corresponding load instructions in the other PBDS. In contrast, for each CBDS pair, GATE checks if the store instructions have the same base addresses.

GATE uses an *affinity cost* [11] to detect the closest subgraph pairs among multiple source subgraphs for maximizing the approximation gain (line 6). The key idea of the affinity cost is that closer

instructions in a dataflow graph generally produce more similar results than other instructions. The affinity cost is calculated based on the common consumer and common producer relationship between two instructions for obtaining the most related pair as shown in Equation 1. By calculating affinity cost of the candidate instructions of subgraph pairs iteratively, GATE can obtain the best subgraph pair with the highest affinity cost (line 4-8).

$$when\ d(a,\ b)\ is\ distance\ the\ between\ a\ and\ b,\ affinity_cost = \sum_{a \in ComCons} \frac{1}{d(a, Inst1) * d(a, Inst2)} + \sum_{a \in ComProd} \frac{1}{d(a, Inst1) * d(a, Inst2)} \quad (1)$$

After the best source is determined, GATE replaces the approximation candidate instruction of the target subgraph with the corresponding instruction of the source subgraph (line 10). GATE then executes the approximated kernel to evaluate the replacement (line 11). If the result is not within TOQ, the GATE rolls back the replacement (line 12).

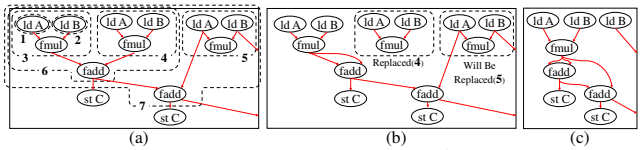


Figure 4: An example of forward process. (a) PBDS detection, (b) instruction replacement, and (c) approximate dataflow

Figure 4 illustrates an example forward process. When there is a dataflow graph as shown in Figure 4 (a), all the load instructions are first identified as PBDSs (1, 2). All fmul instructions having these load instructions as producers are then identified as the last consumers of new PBDSs with the corresponding loads (3, 4, 5). Using the same process, an fadd instruction with two producers PBDSs 3 and 4 is identified as a last consumer of another PBDS (6).

Figures 4 (b) and (c) present an example of PBDS replacement. The highest PBDS is the PBDS 7, and the next is PBDS 6 but each of these has no other PBDSs with the same height. Therefore, PBDSs 3, 4, and 5 can be the next candidate PBDSs to be replaced because of their isomorphic dataflow structures and similar load base addresses. In this case, the process selects the best replacement pair of PBDSs 3 and 4 based on the calculation of affinity cost. Figure 4 (b) presents the dataflow graph after this approximation is applied (PBDS 4 is replaced). Similarly, PBDS 5 can also be replaced. After all the approximations are applied, the final approximate data flow graph is as shown in Figure 4 (c).

3.2 Approximate Thread Merge

SGLA performs approximation within a thread but GATE also includes ATM to support inter-thread approximation. The purpose of ATM is to extend the scope of the SGLA to instructions not only in each thread, but also across similar adjacent threads.

ATM is based on the assumption that control flows between adjacent threads are similar. According to this assumption, all the instructions in each basic block are copied one by one except for the instructions related to control flow, such as branches. Therefore, a target program can produce an error when the execution flows of two merged threads are different. Figure 5 (a) shows the application of ATM. If threads A and B are merged using ATM, the thread B will follow the control flow of thread A. As the total number of instructions in a merged thread is greater than that in the original thread, the effectiveness of SGLA can increase significantly.

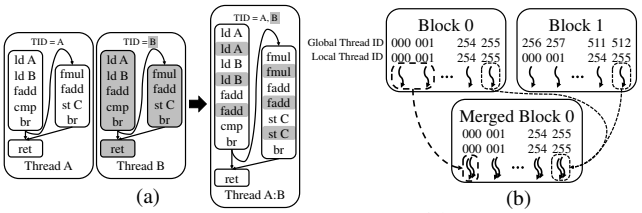


Figure 5: Approximate thread merge. (a) Instruction-level view of ATM, and (b) thread-block-level view of ATM

4 GATE FRAMEWORK

As discussed earlier, approximation opportunities can be roughly categorized into input-sensitive and input-insensitive opportunities. Excluding for some exceptional cases, input-insensitive opportunities can be applied to measure the minimum performance gain and input-sensitive opportunities should be carefully applied to achieve further gain based on runtime TOQ-monitoring. To use this concept, we propose a GATE workflow that consists of three phases: conservative tuning, aggressive tuning, and runtime calibration. An overview of the GATE framework is shown in Figure 6.

[1. Conservative Tuning]: The conservative tuning phase detects as many approximation opportunities as possible and evaluates them to satisfy the TOQ using a random input set. To detect approximation opportunities systemically, we use the two compiler techniques: SGLA and ATM.

GATE first applies SGLA to all the input data-parallel kernels and detects instructions to replace per approximation opportunity. After detecting all the possible cases, GATE checks the output quality variation of the updated kernels with a random input set by applying each opportunity and marks opportunities that pass the TOQ with small quality variation as input-insensitive. As input-insensitive opportunities indicate that the TOQ can be satisfied with any input case, they can be applied without runtime testing.

After all the approximation opportunities from SGLA are obtained and evaluated, GATE attempts to detect more approximation opportunities through ATM. While ATM is an quite effective approximation technique by itself as it can cover inter-thread approximation, it is more valuable because it shows synergy by creating more SGLA opportunities through an increase in the dataflow size of each kernel code.

To maximize the approximation coverage, GATE iteratively detects approximation opportunities using ATM/SGLA and evaluates them until no more approximation opportunities exist. Finally, the compiler generates the final version of the conservatively approximate kernel based on input-insensitive opportunities. During this process, GATE also records a conservatively approximate kernel without using the ATM for runtime calibration.

[2. Aggressive Tuning]: In the aggressive tuning phase, GATE re-evaluates the input-sensitive opportunities obtained in the conservative tuning phase using runtime inputs. Based on the new evaluation, GATE generates an aggressively approximate kernel by adding all the newly accepted opportunities into the conservatively approximate kernel. As the aggressively approximate kernel is input-sensitive, it should be carefully monitored in the runtime calibration phase. The history of approximate opportunities from the conservative and aggressive tuning phases is recorded as approximation levels for future use in the runtime calibration phase.

[3. Runtime Calibration]: After the completion of the tuning phases, GATE obtains an aggressively approximate kernel. As

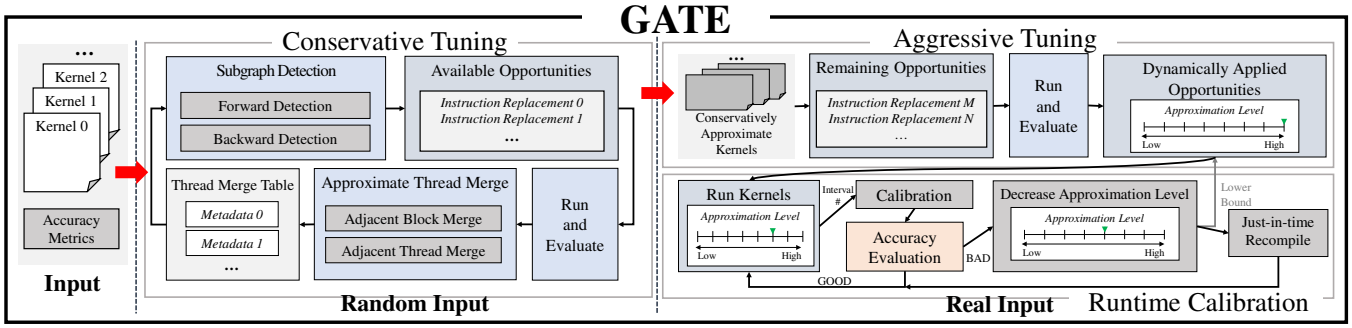


Figure 6: An overview of the GATE framework.

the behavior of target programs can change considerably depending on the inputs at runtime, the approximate level of the kernel should be adjusted in runtime to satisfy quality requirements. To minimize overheads of runtime accuracy measurement, GATE uses a SAGE [15]-style runtime calibration system. GATE periodically checks the accuracy of every calibration interval, increases the calibration interval if the test output quality is better than the TOQ, or executes the more accurate version of the kernel by lowering the approximation level if the test result is worse than the TOQ.

For this, GATE records the approximation opportunities applied to the kernel obtained during the two tuning phases for every approximation level. During conservative tuning, GATE records two set of opportunities: approximation opportunities applied to the conservatively approximate kernel without and with ATM. These records are used for the lowest approximation level. In contrast, during aggressive tuning, GATE records all the applied approximation opportunities from the conservatively approximate kernel to the aggressively approximate kernel. These interim approximation opportunities are used for the other approximation levels.

Instead of creating multiple versions of the kernel, GATE recompiles the new approximate kernel code in the background work on CPUs. When the approximation level falls to the lower bound through this calibration, GATE performs aggressive tuning again and obtains new approximation levels adapted to recent input sets. In the worst case with high input data variation, the running kernel will be the conservatively approximate kernel showing the minimum performance gain while satisfying the TOQ.

Approximation	Error Metric	Application	Error Metric	Application	Error Metric
2DCONV [12]	NED	2MM [12]	NED	ATAX [12]	NED
BICG [12]	NED	3MM [12]	NED	CORR [12]	NED
GESUMMV [12]	NED	FDTD-2D [12]	NED	GEMM [12]	NED
COVAR [12]	NED	MVT [12]	NED	SYRK [12]	NED
MEAN	MRE	GAUSSIAN	MRE	Backprop [3]	MRE
Hotspot [3]	MRE	Hotspot3D [3]	L2-norm	Kmeans [3]	MRE
Blackscholes [10]		L1-norm	ConvolutionSeparable [10]		L2-norm

Table 1: Applications used in the evaluation
NED=Number of Element Difference, MRE=Mean Relative Error

5 EXPERIMENTAL RESULT

5.1 Experimental Setup

Implementation GATE is implemented as an executable binary and custom runtime library. GATE accepts OpenCL kernel source as input and approximates the input kernel through conservative tuning and aggressive tuning in LLVM IR-level. Finally, GATE outputs an approximate version of the input kernel in the form of NVIDIA GPU binary(NVPTX) or SPIR-V [4], a binary intermediate language for OpenCL kernel. GATE uses Clang and LLVM 6.0 [6].

Baseline We evaluated our framework on Intel Core i7-8700k CPU and NVIDIA GeForce 1080Ti with 11 GB global memory. We used OpenCL events to measure the time of kernel execution. Note that our results show only the kernel execution time because other times such as data transfer time are the same as those in the exact version and in the approximate version. All our results are normalized to the execution time of the exact version.

Benchmarks Nineteen applications from Polybench [12], Rodinia [3], and NVIDIA SDK [10] were tested. The applications are presented in detail in Table 1. We used ILSVRC 2012 [13] datasets as the image inputs. In most cases, we used the number of element difference(NED) and the mean relative error(MRE) as error metrics. In NED, the total error of the result is the ratio of the number of elements having more than 10% errors. We also used 90% for TOQ, as was commonly used in previous works [14, 15].

5.2 Performance

5.2.1 Performance Improvements on GPUs. Figure 7 (a) shows the individual performance improvements of all the applications and the geometric mean of the gains for each application at TOQ=90%. The results are normalized to the baseline exact computation performance. The following data are shown in the form of a bar graph: speedup after conservative tuning with SGLA only, speedup after conservative tuning by applying both SGLA and ATM, and speedup after 100 times execution with runtime calibration and aggressive tuning. The best speedup with aggressive tuning is also shown in short horizontal form. As shown in Figure 7 (a), GATE achieved a speed-up of 1.95x, 2.33x, and 2.54x for each bar, with a maximum aggressive tuning gain of 2.65x.

These results prove the effectiveness of each technique of the GATE framework by showing that the three techniques, namely SGLA, ATM, and aggressive tuning, individually produce a substantial amount of speedup. In addition, high performance gain (2.33x) through conservative tuning, which is achieved by obtaining all input-insensitive approximation opportunities, indicates that we can ensure a substantial performance gain through the approximation process without any real input data. The GATE framework can also be used as a general approximation framework because all the benchmarks show meaningful performance gains.

5.2.2 Comparison with a State-of-the-Art Approach. Figure 7 (b) shows the performance comparison between GATE and the state-of-the-art dataflow-based approach, Paraprox [14]. We manually implemented Paraprox's approximation techniques for the stencil/partition and reduction patterns. For a fair comparison, we manually unrolled loops in benchmarks up to four times when applying Paraprox's technique for stencil/partition patterns. The follow-

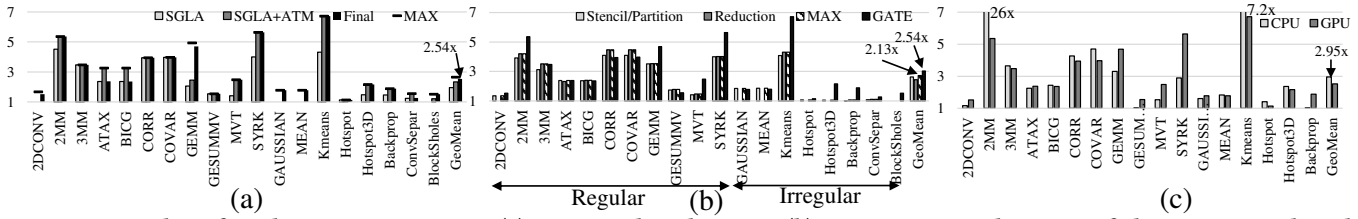


Figure 7: Speedup of applications at TOQ 90% (a) Using each techniques, (b) Comparison with a state-of-the-art approach and (c) In a CPU and a GPU

ing data are shown in the form of a bar graph: speedup when Paraprox's stencil/partition approaches are applied, speedup when Paraprox's reduction approaches are applied, maximum speedup with both techniques, and speedup of GATE. As shown in Figure 7 (b), Paraprox's approach achieved a speedup of 2.13x whereas GATE achieved a speedup of 2.54x.

More specifically, Paraprox achieved a mean speedup of 2.80x for regular pattern benchmarks (Polybench) whereas GATE achieves a mean speedup of 3.10x. GATE achieved similar performance gains in most benchmarks and showed better performance than Paraprox when ATM is applied. Especially for GEMM, when the ATM is applied, a partial subgraph-level approximation makes the performance of GATE better than that of Paraprox.

Paraprox achieved a mean speedup of 1.45x for irregular pattern benchmarks whereas GATE achieved a mean speedup of 1.93x. In the irregular pattern benchmarks, GATE can cover not only evident patterns such as loops and regular data accesses, but also code regions where the patterns are not clearly visible. For example, GATE finds several similar subgraphs containing multiple compute instructions without loops, whereas Paraprox cannot cover.

5.2.3 Performance Improvement on CPUs. To evaluate the effectiveness across multiple target architectures, we measured the speedups of all the applications on a CPU. As shown in Figure 7 (c), although the same approximation techniques are applied, the performance varies significantly due to the differences in the effectiveness of each approximation opportunity on CPUs and GPUs. For example, ATM is more effective on CPUs because it is used to convert thread-level parallelism, which is suitable to GPUs, into instruction-level parallelism, which is suitable to CPUs.

However, despite the performance variance, GATE shows satisfactory speedups over most benchmarks. GATE can successfully tolerate architecture-specific features by using a generalized approach. GATE achieved a speedup of 2.95x on a CPU.

6 RELATED WORKS

Approximate computing is a well-known technique in a wide range of applications, and automatic approximation with runtime quality monitoring has already been introduced in research areas [2, 15]. SAGE [15] is the closest related work to our work because it performs approximation on GPU systems. However, there are two noticeable differences between SAGE and GATE: 1) GATE attempts to maximize the coverage of approximation by using more generalized dataflow graph analysis, whereas most approaches including SAGE and Paraprox [14] depend on special patterns or hardware features; 2) we can measure a substantial amount of minimum performance gain by detecting input-insensitive opportunities.

7 CONCLUSION

In this paper, we proposed a novel approximation framework called *GATE* with two general dataflow-level techniques i.e., SGLA and ATM. By replacing instructions based on subgraphs, *GATE* framework increases the approximation coverage and also realizes a higher performance gain over than conventional pattern-based approaches. In addition, GATE provides high performance using aggressive tuning and runtime calibration processes, while ensuring minimum performance gain using a conservative tuning process. On average, *GATE* shows an average speedup of 2.54x with less than 10% quality degradation for 19 widely used data-parallel workloads.

ACKNOWLEDGMENTS

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP)(No. 2017R1A4A1015498), ICT R&D program of MSIP/IITP (No.2017-0-00142), and the R&D program of MOTIE/KEIT (No.10077609). Yongjun Park is the corresponding author.

REFERENCES

[1] 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.

[2] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 198–209, 2010.

[3] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009.

[4] J. Kessenich, B. Ouriel, and R. Krisch. Spir-v specification provisional (version 1.1, revision 4), 2016.

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

[6] 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.

[7] 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.

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

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

[10] G. Nvidia. computing sdk. *GPU computing SDK," https://developer.nvidia.com/gpu-computing-sdk*, 22(07):2013, 2013.

[11] Y. Park, S. Seo, H. Park, H. K. Cho, and S. Mahlke. Simd defragmenter: Efficient ilp realization on data-parallel architectures. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 363–374. ACM, 2012.

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

[13] 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.

[14] 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.

[15] 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.