Aspect-Driven Mixed-Precision Tuning Targeting GPUs

Ricardo Nobre[*,+], Luís Reis[*,+], João Bispo[*,+], Tiago Carvalho[*,+], João M.P. Cardoso[*,+], Stefano Cherubin[×], Giovanni Agosta[×]
{rjfn,luis.cubal,jbispo}@fe.up.pt,jmpc@acm.org,stefano.cherubin@polimi.it,agosta@acm.org
[*]University of Porto, Portugal [+]INESC-TEC, Portugal [×]Politecnico di Milano, Italy

ABSTRACT

Writing mixed-precision kernels allows to achieve higher throughput together with outputs whose precision remain within given limits. The recent introduction of native half-precision arithmetic capabilities in several GPUs, such as NVIDIA P100 and AMD Vega 10, contributes to make precision-tuning even more relevant as of late. However, it is not trivial to manually find which variables are to be represented as half-precision instead of single- or double-precision. Although the use of half-precision arithmetic can speed up kernel execution considerably, it can also result in providing non-usable kernel outputs, whenever the wrong variables are declared using the half-precision data-type. In this paper we present an automatic approach for precision tuning. Given an OpenCL kernel with a set of inputs declared by a user (i.e., the person responsible for programming and/or tuning the kernel), our approach is capable of deriving the mixed-precision versions of the kernel that are better improve upon the original with respect to a given metric (e.g., time-to-solution, energy-to-solution). We allow the user to declare and/or select a metric to measure and to filter solutions based on the quality of the output. We implement a proof-of-concept of our approach using an aspect-oriented programming language called LARA. It is capable of generating mixed-precision kernels that result in considerably higher performance when compared with the original single-precision floating-point versions, while generating outputs that can be acceptable in some scenarios.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Source code generation**;

KEYWORDS

GPGPU, aspect-driven, mixed-precision

This work was partially supported by the TEC4Growth project, NORTE-01-0145-FEDER-000020, financed by NORTE 2020, under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (ERDF), and by Fundação para a Ciência e a Técnologia (FCT) through PD/BD/105804/2014 and SFRH/BPD/118211/2016. In addition, this work was partially supported in part by EU H2020-FET HPC program through the ANTAREX project under grant 671623. Finally, we acknowledge support from the HiPEAC4 Network of Excellence, financed by the EU H2020 research and innovation programme under grant agreement number 687698.

PARMA-DITAM'18, January 2018, MAN, United Kingdom

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

ACM Reference Format:
Ricardo Nobre[*,+], Luís Reis[*,+], João Bispo[*,+], Tiago Carvalho[*,+], João M.P. Cardoso[*,+], Stefano Cherubin[×], Giovanni Agosta[×] . 2018. Aspect-Driven Mixed-Precision Tuning Targeting GPUs. In *Proceedings of PARMA-DITAM Workshop (PARMA-DITAM'18).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

High Performance Computing (HPC) is evolving towards a stage where resources, in particular energy, cannot be considered expendables. Exascale HPC systems challenge the boundaries of the power supply, and therefore energy and power efficiency are paramount to exploit effectively such huge amount of resources. To address this issue, heterogeneity is currently the most interesting option from an architectural point of view. Nowadays most supercomputers support heterogeneous accelerators and often they feature General Processing Units (GPUs). GPUs allow HPC systems to significantly improve their FLOPS/Watt metrics, as testified by their strong presence in the Green500. In this scenario, it is worth to consider the impact of data precision on GPU computations.

Modern GPUs with support for native half-precision (i.e., 16-bit floating point) arithmetic with up to 2× the peak arithmetic throughput of single-precision include the NVIDIA P100 (Pascal architecture) and V100 (Volta architecture), the AMD Vega 10 XT/XL (Vega architecture), and integrated GPUs within the Intel CPUs with the Skylake, Kaby Lake, and Coffee Lake micro-architectures (Gen9+ architecture). These GPUs can be used to accelerate a number of applications that can work with 16-bit floating-points, which have less precision and range than 32-bit and 64-bit precision floating-points.

The use of data-types smaller than 32-bit can be useful in a broad selection of applications such as deep neural network training [7, 29] and inferencing [30], some image processing kernels [28] and radioastronomy [16].

The benefit potential is not limited to the higher arithmetic throughput of the GPUs, we should also consider the benefit of performing memory operation on a reduced amount of data. This consideration is valid for various memory levels, from the local memory to all the levels of the cache, from the GPU memory to the host memory. For instance, using a 16-bit float instead of a 32-bit float can make data accesses faster, both for reading and for writing.

In this paper, we present a LARA-based [17] approach for precision-tuning and we evaluate it in the context of compilation of OpenCL kernels for a GPU. We present our work-in-progress in terms of the development of a tool for automatically generating and evaluating (considering both performance and the quality of the generated outputs) alternative versions of given GPU code. The purpose of this easy-to-use tool is to help GPGPU programmers to better use the floating-point capabilities of their GPUs.

Section 2 presents our approach. The methodology behind the experiments presented in this paper is described in detail in section 3. Section 4 presents experimental results and a brief discussion. Related work in the context of precision tuning is presented in section 5. Finally, section 6 presents concluding remarks about the work presented in this paper.

2 PROPOSED APPROACH

We use the LARA language [17] to develop a dedicated tool. Our tool takes an OpenCL kernel as input, it generates and evaluates multiple versions of the input kernel. Those versions exploit mixed-precision data types to achieve performance improvements over the original kernel, while they satisfy a user-defined constraint on the quality of the output. The tool supports specifying the following parameters:

- A set (or multiple sets) of inputs to the kernel;
- A metric for evaluating the quality of the outputs of the generated kernels;
- A threshold value which represents the maximum difference between the outputs of the original kernel and the candidates generated during exploration which is considered acceptable;
- A goal metric, such as execution performance.

The overall flow for the user is as follows. First, the user decides which metric they want to use to measure the quality of the output. They can pick a metric available from our library, or they can write a custom metric. Then, the user has to define an initialization routine for the GPU kernel under inspection. Our tool runs the original GPU kernel to generate a reference for the output. Then, it uses a Design Space Exploration (DSE) loop to evaluate several versions of the kernel. Our tool uses the selected goal metric as the target of a DSE optimization. Different OpenCL versions are automatically generated by our tool by mixing different combinations of data types for the declarations of variables in the GPU kernel. We target half-, single-, and double-precision floating-point. At the end, the tool reports the solution(s) that achieves best score according to the given quality metric. The tool excludes from the report the solution(s) that exceeds the desired precision threshold.

Our current implementation of the tool only supports automatic generation of code for the different OpenCL kernel versions, whereas the generation of the corresponding host-side code is still work-in-progress. For this reason, for the experiments we report in this paper we manually modified the host code for each set of kernels.

The automatic transformation of already existing OpenCL host code can be too complex in the general case. Therefore, the approach we suggest for the host code generation requires to start from an host code version which features a call to a function written in C. Such C function should hold the code of the OpenCL kernel. This allows the source-to-source compiler to easily recognize and modify the data types in the OpenCL code, and to properly insert code to perform the data type conversion at runtime.

Let us consider the example of a kernel where all floating-point variables are declared as single-precision. Some of the inputs and/or outputs of the kernel should be converted to a different precision level, in this case half-precision floating point. This process entails three stages of conversion. First, input data must be converted from original data type to the destination data type before their transfer from the host to the GPU. In this case, the 32-bit floating-point input data must be converted to the 16-bit floating-point representation, which is used by the `half` OpenCL data-type. Then, the code of the OpenCL kernel is adapted to match the desired precision level. Finally, a conversion of the output data back to the original single precision floating-point representation is required. The latter conversion is performed after transferring the output data from the GPU to the host. In the host code, we rely on functions from the *half* project – which is an open-source half-precision floating point library [25] – for the conversion from 32-bit floating points to 16-bit floating points and vice-versa. The representation we use to store 16-bit floats is based on the standard IEEE 754. Therefore, it is compatible with the `half` OpenCL data type.

To generate the various OpenCL kernel versions we exploit *Clava*[1], which is an aspect-oriented source-to-source compiler. It allows the programmer to use the LARA language to describe analyses, transformations, and instrumentations of C, C++, and OpenCL code. Figure 1 shows the LARA aspect[2] that we use to generate the several versions of the OpenCL kernel. This aspect has three inputs parameters `numIndependentParams`, `maxVersions` and `combinationFilter`.

The `numIndependentParams` represents the first N parameters of the kernel that we will freely combine. We use it to group the generated kernels in folders according to their interface: kernels with the same parameter signature will be in the same folder. This classification allows us to group all kernels that correspond to the same host code. When the automatic host code generation will be implemented, this input will no longer be required.

The parameter `maxVersions` indicates the maximum number of kernel versions that we want to generate. For the experiments in the paper we used the value `undefined`, which means we do not restrict the number of versions to be generated.

The `combinationFilter` is an array where each element represents a variable declaration in the OpenCL code. They are sorted by the order they appear in the kernel. They can have one of three values: `0`, `1`, or `undefined`. During code generation, a value of `0` means that the corresponding variable should maintain its original type in all versions. A value of `1` requires that all the generated versions have that variable declared as *half*. The `undefined` value means that the variable may preserve its original data type or it may have its data type changed to *half* in the generated kernel versions. An empty array means that all values are `undefined`. The `combinationFilter` parameter helps to reduce the number of kernels to generate. However, in our experimental campaign we choose not to introduce any constraint in the `combinationFilter` and we generate all possible combinations.

From the code in Figure 1 we can analyze the behaviour of the aspect. First, it obtains the list of variable declarations in the OpenCL kernel that use a `float` or a `double`. These declarations are not restricted to built-in data types, it also detects arrays, `typedefs` and pointers. Then, it creates generator for the variable combinations we will test. In this case, we sequentially generate all possible

[1]The source-code code of the Clava compiler is available at https://github.com/specs-feup/clava/tree/master/ClavaWeaver

[2]An extended version of this example is available at https://github.com/specs-feup/specs-lara/tree/master/ANTAREX/HalfPrecisionOpenCL

```
aspectdef HalfPrecisionOpenCL
 input
  numIndependParams,
  maxVersions = undefined,
  combinationFilter = []
 end

 // Get the list of float and double variables
 // declared in the OpenCL kernel (including parameters)
 call result : OpenCLVariablesToTest;
 var variablesToTest = result.variablesToTest;

 // Generates combinations of variables
 var sequenceGenerator = new SequentialCombinations(
     variablesToTest.length, maxVersions);

 var counter = 0;
 while(sequenceGenerator.hasNext()) {

  // Get a new combination of variables
  var combination = sequenceGenerator.next();

  // Number that was used to generate the sequence
  var lastSeed = sequenceGenerator.getLastSeed();

  // Check if combination passes the filter
  if(!isCombinationValid(lastSeed, combinationFilter)) {
   continue;
  }

  // Save original kernel version
  Clava.pushAst();

  // Change the builtin type of the variables
  for(var index of combination) {
   var $vardecl = Clava.findJp(variablesToTest[index]);
   changeTypeToHalf($vardecl);
  }

  // Add to the OpenCL kernel the pragma that enables half-precision
  call addHalfPragma();

  // Choose output folder based on the
  // number of independent parameters of the kernel
  var baseFolder = getBaseFolder(lastSeed, numIndependParams);
  var kernelFolder = "half_version_" + counter;
  var outputFolder = Io.getPath(baseFolder, kernelFolder);

  // Generate code
  Clava.writeCode(outputFolder);

  // Discard modified kernel and restore original one
  Clava.popAst();

  counter++;
 }

end
```

Figure 1: LARA aspect that generates alternative versions of OpenCL kernels using different types for the input parameters and the kernel body variables.

combinations. The aspect performs a loop through all combinations. For each combination, the aspect tests if it is a valid combination, according to the given `combinationFilter`. If it passes the test, the aspect saves the original version of the kernel, so that it can be loaded again before testing the next combination, and change all the variable types of the combination to half-types. Then, it adds to the OpenCL kernel the pragma required to support half-precision, writes the modified kernel version, and restores the original kernel.

3 EXPERIMENTAL METHODOLOGY

In this section we describe the setup of the experiment we present in this paper. It includes a description of the hardware, the benchmarks, the validation approach, and validation metrics.

3.1 Target Platform

We execute our experiments on a Sapphire Radeon Vega 64 Limited Edition graphics card (part number 21275-01-20G), which features an AMD *Vega 10 XT* GPU [2]. The GPU has a peak compute performance of 12,759, 12,874, and 815 GFLOPS respectively with single-, half-, and double-precision scalar data-types. The use of vector data-types with width 2 – i.e., `float2`, `half2` and `double2` – results in performance improvement that brings the previous metrics to 12,690, 24,294 and 808 GFLOPS. As expected, the use of vector data-types results in a peak performance boost of 2× for half-precision computations, and a non-improvement for the others. In the case of single-precision and of double-precision data types, the peak compute performance does not increase with the use of any of the vector data-types. It is also worth mentioning that using the `half` scalar data-type results in similar peak compute performance as using `float`. The use of `half4`, `half8`, and `half16` results in performance close to when using `half2`, as operating on two half-precision floats at a given time per OpenCL kernel instance saturates the FPUs. We extract the peak performance metrics through *clpeak* [4], a tool for profiling OpenCL devices.

Notice that, on GPUs such as the one we used, half-precision arithmetic peak performance can only be achieved if vectorization is used. Otherwise, the performance of single- and half-precision versions of the same code can be identical for some OpenCL kernels. Although vectorization is an important part of accelerating half-precision applications – i.e., the use of the packed half-precision floating-point arithmetic capabilities of the GPU– we consider auto-vectorization to be outside the scope of this paper. Nevertheless, a tool implementing the approach described in this paper can be a key component in a mapping strategy for generating high-performance vectorized implementations of OpenCL kernels, as using half-precision data-types is required for vectorization to have a considerable impact on GPU kernel code.

The graphics card is installed on a desktop computer with an Intel Xeon E3-1245 V3 @3.6GHz and 16GB of DDR3 @2400MHz in a dual channel configuration. We use Ubuntu 16.04 64-bit with the ROCm [1] driver stack (ver. 1.6). The GPU is set to *high* performance mode using the ROCm System Management Interface (*rocm-smi*), in order to reduce the variability introduced by the more aggressive dynamic frequency scaling of other more energy efficient modes.

3.2 Kernels and Goal Metric

In this paper we use kernels from the Polybench/GPU benchmark suite [26] to evaluate our approach. We selected this particular benchmark suite as it is freely available and thus contributes to making the results presented in this paper reproducible.

Polybench/GPU is a collection of codes implemented for GPUs using CUDA, OpenCL, and HMPP. This benchmark suite includes kernels from benchmarks from different domains which represent computations that would be performed on GPUs in the context

Table 1: Number of floating-point (FP) input parameters of the OpenCL kernel, number of floating-point variables in the kernel body, number of possible kernel signatures, and number of possible versions for each of the benchmarks used in the experiments.

Benchmark	# FP params.	# FP vars.	# signatures	# versions
2DCONV	2	9	4	2,048
GEMM	5	0	32	32

of HPC. We rely on the OpenCL kernels from the `2DCONV` and the `GEMM` benchmarks for the purpose of demonstrating our approach.

Table 1 shows a number of metrics related with the design space of the possible solutions. For the experiments presented in this paper, we allow each floating-point variable to be either a `float` or an `half`.

For instance, the OpenCL kernel of the `2DCONV` benchmark has 2 floating-point parameters, matrices `A` and `B`. Therefore, there are four possible kernel signatures, as such: `A` and `B` float, `A` and `B` half, `A` float and `B` half and vice-versa. The 2 floating-point inputs, combined with the fact that there are 9 floating-point variables in the kernel body, and that each variable is allowed to assume one of two different data-types (`float` or `half`) results in 2048 ($= 2^{2+9}$) possible solutions. The `GEMM` kernel has no floating-point variables in it's body, therefore, considering it has 5 floating-point parameters (matrices `A`, `B` and `C`, and `alpha` and `beta`) the number of possible OpenCL kernel versions is 32 ($= 2^5$), each having a unique function signature.

In this paper, when representing an OpenCL kernel signature, we use 0 or 1 at a given position n to denote that input number n of the kernel is of data type `float` or `half`, respectively. For instance, sig_{01101} represents a function kernel where the first and fourth inputs are of data type `float` and all other are of data type `half`.

The goal metric in the experiments presented in this paper is the minimization of execution time of a given OpenCL kernel, through evaluation of alternative versions of the kernel. The performance reported in this paper are averaged over 30 executions. We kept the computer load as low as possible to avoid interferences from other processes.

To determine whether the obtained error is within an acceptable margin, we compute the *relative solution error* [12], computed as:

$$\eta = \frac{\|A_{approx} - A\|_F}{\|A\|_F} \quad (1)$$

Where $\|X\|_F$ is defined as the Frobenius matrix norm, computed as the square root of the sum of the squares of all elements of the specified matrix:

$$\|X\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}(X_{i,j})^2} \quad (2)$$

Note that $\|X\|_F$ is 0 when the matrix is composed entirely of zeros, so if $A_{approx} = A$ (i.e., if the results are exact), then the *relative solution error* will be 0.

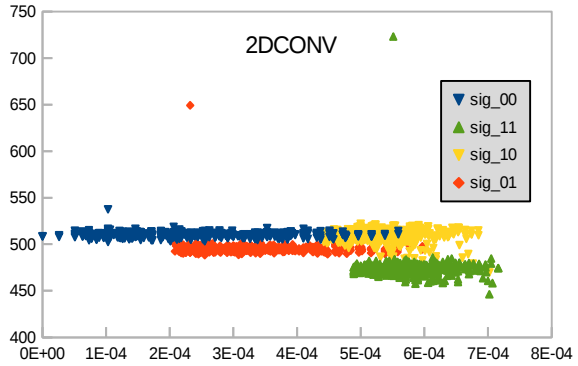

Figure 2: Output error (X axis) and performance in microseconds (Y axis) for the 2,048 variations of the 2DCONV kernel.

4 RESULTS

In this section, we present results that can be used to infer the performance/error trade-offs of the different versions generated by our tool from the OpenCL codes considered for demonstrating our approach.

Figure 2 depicts the results for `2DCONV`, for the four possible signatures this OpenCL convolution kernel can have when each of its floating-point parameters is allowed the freedom to be a `float` or a `half`. The cases when both `A` and `B` input matrices are half-precision (i.e., signature sig_{11}) results generally in higher performance than the original solution (data-point most at left with error of 0.0) at the cost of a larger difference between the output from execution of the original kernel and the output from execution of the modified kernels (see, section 3.2). However, some of these solutions result in a smaller output error than some of the solutions with each of the other possible signatures, therefore increasing the merit of performing an exploration on the data-types of both the kernel parameters and the variables in the kernel body. Some of the solutions that have some of the variables declared as half might be useful in scenarios where some loss of accuracy in the output of the kernel is allowed, resulting in a speedup up to 1.13× (0.46 seconds for the fastest version and 0.51 seconds for the original version). The OpenCL version with all variables (both input and kernel body variables) declared as `half` was the solution measured to achieve higher performance. However, it results in a large error (see, section 3) of 0.000702 when compared with the other versions (error of 0.000716 was the highest registered). Other solutions that differ from the former by having some of the kernel body variables declared as `float` can result in smaller output error while having close to the same performance. For instance, there is a solution that results on an output error of 0.000489 while having close to the same execution time as the fastest solution (0.47 vs. 0.46 seconds).

Compared with `2DCONV`, the `GEMM` kernel lead to more problems when using half-precision when using the original PolyBench/C initialization data. There were many instances where some half-precision floating-point variables represented `inf`, which is indicative of an overflow. This resulted in `inf` being also reported as error measurement (see, section 3.2).

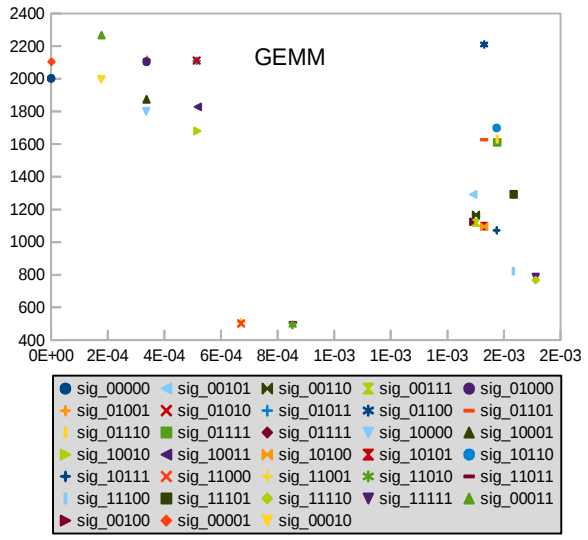

Figure 3: Output error (X axis) and performance in microseconds (Y axis) for the 32 variations of the GEMM kernel.

We opted to, for the experiments presented here, reduce both the initialization values for the positions of matrices `A`, `B` and `C`, as well as `alpha` and `beta` by a factor of 1000×. Figure 3 depicts the results for `GEMM`, for the 32 possible versions the OpenCL matrix multiplication kernel can have.

In the case of `GEMM`, all four mixed-precision variations of the original kernel that have matrix `A` and matrix `B` as `half` and matrix `C` as `float` (i.e., sig_{11000}, sig_{11001}, sig_{11010} and sig_{11011}) take close to 0.5 seconds to execute, while incurring in a smaller output error than many other versions in the solution space. This experimental result is particularly interesting: with respect to these four mixed-precision versions not only result in smaller error, but also have considerably higher performance and than the four versions where all three input matrices are of the data type `half`. This consideration includes the version where all input floating-point input matrices and scalars of data type `half` (i.e., sig_{11111}), which is arguably the solution that at first would be intuitively expected to result in the highest performance compared with other versions. This occurs due to *strict aliasing*, a concept from the C language standard which indicates that two pointers to different base types never alias. When matrix `C` points to `half` values and matrices `A` and `B` point to `float` values, or vice-versa, then the compiler may assume that matrix `C` never aliases with the other pointers (as doing so would be undefined behavior). The `GEMM` benchmarks contains a loop where the same position of `C` is continuously overwritten. If `C` is known to never alias with `A` and `B`, then the store instruction can be moved out of the loop, resulting in significantly fewer memory instructions. The same optimization can be enabled by explicitly marking `C` as *restrict*. When this happens, the version where all vectors are of data type *half* (i.e., sig_{11100}) executes in roughly the same performance as sig_{11000} (approximately 0.5 seconds). Versions using only single-precision floating point also benefit from *restrict*. For instance, the version where all inputs are of data type *float* takes approximately 1.3 seconds to execute if the `C` matrix is declared with the `restrict` keyword (the original takes 2.0 seconds).

The version measured to execute faster (sig_{11010}) (although very close in terms of execution performance to the other three versions with signature sig_{110XX}) results in an improvement of 4.06× over the original version (the generated mixed-precision version takes 0.49 seconds and the original single-precision version takes 2 seconds), in cases the error that results from using that version is acceptable. With respect to the effect of the data-type of the two input scalars on the quality of the output of this specific matrix multiplication kernel, the declaration of `alpha` as a `half` floating-point always results in considerably increasing the output error, while declaring `beta` as a `half` has a much smaller effect.

5 RELATED WORK

Approximate computing have been investigated at several levels, from energy-efficient hardware design [15] to software tools for high performance computing [12, 21]. Researchers developed entire frameworks [5] to analyze the effects of several approximate computing techniques. However, the effectiveness of such frameworks has yet to be proved.

A large share of the work in this field is related to the adaptation of the data to hardware and energy constraints. In the embedded system domain, floating point values are usually converted to fixed point one [24] to limit the area cost and power consumption of the architecture. Hardware designers developed tools to support the automatic conversion of floating point values to fixed point ones [3, 13, 19, 27]. More recent work in this field aims at exploiting fixed point arithmetics on FPGA accelerators to speedup floating point applications, such as [20].

In high performance computing the effects of reduced precision has been investigated in [12] on the Intel x86 architecture. Fixed point arithmetics has been exploited with good results on GPUs in the field of computer physics simulations. An interesting mixed precision approach involving both fixed point and floating point data types has been applied to molecular dynamics simulations in [14].

A common approach to apply reduced and mixed precision on GPU requires the programmer to manually substitute the data type in the problem as presented in [22]. More complex frameworks can provide an automated approach to reduced precision. Lam introduces in [21] an automated procedure to identify portions of code which can be safely computed with reduced floating point precision without affecting result accuracy. In order to be able to select the best precision configuration, the work presented in [21] requires to generate and execute several versions of the same program.

Analysis and profiling tools, such as `Precimonius` [6] and later works [11], provide hints to the programmer on which data type is the most efficient while satisfying a given error threshold on the output. However, their implementations focus only on CPUs and do not deal with accelerators neither they support OpenCL.

An holistic approach described in [8] merges the error requirements introduced by reduced precision with a contract-based programming model which supports both floating point and fixed point

arithmetics. According to the methodology described in [8], the programmer has to specify annotations on the source code to describe the precision requirements through a DSL specifically designed for that purpose.

Our approach leverages an existing DSL which is already used within the embedded system and HPC domains (see, e.g., [9, 10, 18, 23]). Although our work focus only on the HPC use case, our approach can be generalized to whichever target domain.

The effort we require from the programmer is limited to a description of the transformation's scope, the precision threshold, and the metric to evaluate the quality of the output. Moreover, our approach allows the evaluation metric and the precision threshold to be selected from an existing preset.

6 CONCLUSIONS

We presented a LARA-based approach for precision-tuning and we evaluated it in the context of compilation of OpenCL kernels for GPU. Our tool generated alternative versions of the GPU kernels resulted in improving performance over the original single-precision versions on an AMD Vega 10 XT GPU. These versions also produced outputs which have a difference from the original outputs that might be acceptable for some use-cases. Moreover, some of the solutions obtained in our experiments clearly demonstrate the importance of using mixed-precision when striving to increase the performance of a given OpenCL kernel while also having requirements in terms of the quality of the generated output.

We are confident that our approach and the tool we are currently developing can be useful, especially given the ease-of-use, to GPGPU programmers. It helps programmers to better use the floating-point capabilities of modern GPUs with support for native half-precision arithmetic with 2× peak performance throughput compared with peak the performance arithmetic throughput with single-precision, such as the NVIDIA P100 (Pascal architecture) and V100 (Volta architecture), the AMD Vega 10 XT/XL (Vega architecture) and current Intel integrated GPUs that are part of the Intel CPUs with the Skylake, Kaby Lake and Coffee Lake microarchitectures.

In the current implementation, our tool generates all OpenCL versions for evaluation, which corresponds to performing a full search in the design space during the exploration. As a future direction, we will work on integrating with specialized DSE tools to allow for faster exploration. Finally, we plan to add support for CUDA in our tool in a near future.

REFERENCES

[1] 2017. ROCm, a New Era in Open GPU Computing. (2017). https://rocm.github.io/

[2] AMD. 2017. Radeon's next-generation Vega architecture. (2017). https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf

[3] P. Belanovic and M. Rupp. 2005. Automated floating-point to fixed-point conversion with the fixify environment. In *16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*. 172–178.

[4] Krishnaraj Bhat. 2017. clpeak: A tool which profiles OpenCL devices to find their peak capacities. (2017). https://github.com/krrishnarraj/clpeak

[5] Christos Sakalis et al. 2017. A Software Framework for Investigating Software and Hardware Approximate Computing. In *ACACES 2017 Poster Abstracts (ACACES 2017)*. HiPEAC, 225–228.

[6] Cindy Rubio-González et al. 2013. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 27.

[7] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Low precision arithmetic for deep learning. *CoRR* abs/1412.7024 (2014). http://arxiv.org/abs/1412.7024

[8] Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 235–248.

[9] Cristina Silvano et al. 2016. The ANTAREX Approach to Autotuning and Adaptivity for Energy Efficient HPC Systems. In *International Conference on Computing Frontiers (CF '16)*. ACM, 288–293.

[10] Cristina Silvano et al. 2016. Autotuning and adaptivity approach for energy efficient Exascale HPC systems: The ANTAREX approach. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 708–713.

[11] Cindy Rubio-González et al. 2016. Floating-point Precision Tuning Using Blame Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1074–1085.

[12] Stefano Cherubin et al. 2017. Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error. In *to appear in Proceedings of the 2017 International Conference on Parallel Computing (ParCo 2017)*.

[13] Gaël Deest et al. 2014. Toward Scalable Source Level Accuracy Analysis for Floating-point to Fixed-point Conversion. In *International Conference on Computer-Aided Design (ICCAD)*. San Jose, United States, 726–733.

[14] Scott Le Grand, Andreas W. Götz, and Ross C. Walker. 2013. SPFP: Speed without compromise - A mixed precision model for GPU accelerated molecular dynamics simulations. *Computer Physics Communications* 184, 2 (2013), 374 – 380.

[15] J. Han and M. Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*. 1–6.

[16] Mark Harris. 2016. Mixed-Precision Programming with CUDA 8 | Parallel Forall. https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/. (Dec. 2016). Accessed: November 11th, 2017.

[17] João M. P. Cardoso et al. 2012. LARA: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM, 179–190.

[18] José G. F. Coutinho et al. 2013. Deriving Resource Efficient Designs Using the REFLECT Aspect-oriented Approach. In *Proceedings of the 9th International Conference on Reconfigurable Computing: Architectures, Tools, and Applications (ARC'13)*. Springer-Verlag, Berlin, Heidelberg, 226–228.

[19] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. 2000. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 47, 9 (Sep 2000), 840–848.

[20] L. Gan et al. 2013. Accelerating solvers for global atmospheric equations through mixed-precision data flow engine. In *2013 23rd International Conference on Field programmable Logic and Applications*. 1–6.

[21] Michael O Lam and Jeffrey K Hollingsworth. 2016. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications* (2016), 1–15.

[22] M.A. Clark et al. 2010. Solving lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications* 181, 9 (2010), 1517 – 1528.

[23] Martin Golasowski et al. 2017. *Expressing and Applying C++ Code Transformations for the HDF5 API Through a DSL*. Springer International Publishing, Cham, 303–314.

[24] Daniel Menard, Daniel Chillet, and Olivier Sentieys. 2006. Floating-to-fixed-point Conversion for Digital Signal Processors. *EURASIP J. Appl. Signal Process.* 2006 (Jan. 2006), 77–77.

[25] Christian Rau. 2017. half: IEEE 754-based half-precision floating point library. (2017). http://half.sourceforge.net/

[26] Louis-Noel Pouchet Scott Grauer-Gray. 2012. PolyBench/GPU: Implementation of PolyBench codes for GPU processing. (2012). http://web.cs.ucla.edu/~pouchet/software/polybench/GPU/index.html

[27] N. Simon, D. Menard, and O. Sentieys. 2011. ID.Fix-infrastructure for the design of fixed-point systems. In *University Booth of the Conference on Design, Automation and Test in Europe (DATE)*, Vol. 38. http://idfix.gforge.inria.fr

[28] SiSoftware. 2017. FP16 GPGPU Image Processing Performance & Quality. (2017). http://www.sisoftware.eu/2017/04/14/fp16-gpgpu-image-processing-performance-quality/

[29] Suyog Gupta et al. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 1737–1746. http://dl.acm.org/citation.cfm?id=3045118.3045303

[30] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. 2016. Accelerating Deep Convolutional Networks using low-precision and sparsity. *CoRR* abs/1610.00324 (2016). http://arxiv.org/abs/1610.00324