A Framework for Automated and Controlled Floating-Point Accuracy Reduction in Graphics Applications on GPUs

ALEXANDRA ANGERD, ERIK SINTORN, and PER STENSTRÖM,
Chalmers University of Technology

Reducing the precision of floating-point values can improve performance and/or reduce energy expenditure in computer graphics, among other, applications. However, reducing the precision level of floating-point values in a controlled fashion needs support both at the compiler and at the microarchitecture level. At the compiler level, a method is needed to automate the reduction of precision of each floating-point value. At the microarchitecture level, a lower precision of each floating-point register can allow more floating-point values to be packed into a register file. This, however, calls for new register file organizations.

This article proposes an automated precision-selection method and a novel GPU register file organization that can store floating-point register values at arbitrary precisions densely. The automated precision-selection method uses a data-driven approach for setting the precision level of floating-point values, given a quality threshold and a representative set of input data. By allowing a small, but acceptable, degradation in output quality, our method can remove a significant amount of the bits needed to represent floating-point values in the investigated kernels (between 28% and 60%). Our proposed register file organization exploits these lower-precision floating-point values by packing several of them into the same physical register. This reduces the register pressure per thread by up to 48%, and by 27% on average, for a negligible output-quality degradation. This can enable GPUs to keep up to twice as many threads in flight simultaneously.

CCS Concepts: • **Computer systems organization** → **Single instruction, multiple data**;

Additional Key Words and Phrases: Approximate computing, LLVM, GPU, register file, floating-point precision

ACM Reference format:

Alexandra Angerd, Erik Sintorn, and Per Stenström. 2017. A Framework for Automated and Controlled Floating-Point Accuracy Reduction in Graphics Applications on GPUs. *ACM Trans. Archit. Code Optim.* 14, 4, Article 46 (December 2017), 25 pages.
<https://doi.org/10.1145/3151032>

1 INTRODUCTION

Approximate computing promises to trade a small and acceptable output-quality loss for significant gains in performance and energy efficiency. It is applicable to a wide range of applications including inherently error-resilient applications, media processing, applications with analog input and/or output, and physics simulations.

This work is supported by the Swedish Research Council under the ACE project.

Authors' addresses: A. Angerd, E. Sintorn, and P. Stenström, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden; emails: {angerd, erik.sintorn, per.stenstrom}@chalmers.se.

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 the author(s) 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.

2017 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 1544-3566/2017/12-ART46 \$15.00

<https://doi.org/10.1145/3151032>

Narrowing the width of floating-point values is an effective approach to both achieve higher performance [6] as well as higher energy efficiency [8, 16, 22], especially for GPUs, which now are supporting 16-bit floating-point standards [14]. A substantially narrower width of floating-point values can open up many novel optimization approaches at the hardware level, such as more resource-efficient register files, data-paths, functional units, and cache memory subsystems. However, to leverage such optimizations, two issues must be addressed. First, the width of each and every floating-point value must be established at the instruction level. Second, architectural support is needed to use the established widths to utilize register file, data path, functional unit, or cache resources more efficiently. The goal of this article is to provide such a framework.

Programming language models that enable approximate computing, such as EnerJ [20] and FlexJava [15], take a binary view to declare a variable as either approximable or precise. Hence, they cannot deal with an arbitrary width of floating-point variables. Even if there were support for specifying precision, it would be laborious or nearly impossible for programmers to use it efficiently. Also, it would need support at the instruction-set architecture level, such as in Quora [24], to specify error-bounds at the instruction level.

Precimonious [18] provides a framework to automatically select among IEEE754-compliant data types for each source-level floating-point variable to improve performance while generating an output within some guaranteed error bound. There are three limitations: First, the level of approximation is constrained to IEEE754-compliant data types, and, second, the optimizations target source-level floating-point variables instead of instruction-level floating-point values. Finally, the algorithm has limited scalability ($O(an^2)$, where n is the number of analyzed floating-point values and a is the number of precision levels).

Chisel [13] also supports automatic precision-tuning, but, like Precimonious, it does not target instruction-level registers. In addition, Chisel employs a static interval analysis, giving an upper bound on the required precision that yields conservative results.

In this article, we first propose a method for automatically, and transparently to the programmer, classifying the amount of approximation of floating-point registers possible for individual *low-level instructions*. We then explore how this method can be used to optimize the resource efficiency of a floating-point register file and propose a precision-aware register file for GPUs. While we focus on graphics algorithms generating images with readily available methods for automatically assessing the quality [26], we argue that our proposed method is applicable to any class of algorithms where such a measurement can be established.

A set of input parameters, a reference output, and a quality threshold are provided as input to our proposed auto-tuning method. It then automatically chooses the precision of each individual *floating-point register* used in the program. Unlike Precimonious and Chisel, our proposed method operates immediately on the compiled and optimized assembly code and selects an appropriate approximate format for each virtual floating-point register at the instruction level. We apply this method to three low-precision floating-point formats: IEEE754 half precision, mantissa truncation, and dynamically selected exponent and mantissa width. By lowering the quality threshold, we can, with fine granularity, trade output quality in a controlled fashion for a reduction in the number of bits needed to store the floating-point values. Empirically, we show that, even with an extremely high-quality threshold, we can reduce the average number of bits needed per register from 32 bits to between 13 and 26 bits. When allowing for a reduced quality output, which still can be useful in many cases, the average number of bits per register can be reduced to between 11 and 22 bits.

Next, we propose a novel GPU register file organization together with a register-allocation algorithm that can tightly pack registers in the proposed register file. We show that by only approximating floating-point registers, and maintaining an extremely high output quality, the maximum

register pressure of our kernels can be reduced by 12–48%, 27% on average, potentially allowing for substantially higher throughput of memory-bound kernels.

To summarize, our contributions are the following:

- A method for automating the process of selecting the precision of each floating-point value given a certain quality threshold in a controlled fashion. The worst-case complexity is bounded by $O(an)$ as opposed to $O(an^2)$ in previous work, where a is the number of precision levels and n is the number of analyzed values. This is important, since we target virtual registers, of which there can be hundreds or thousands, even in small kernels.
- A novel GPU register file organization and a register file allocation algorithm that can leverage lower precision of floating-point numbers for increased capacity and performance.
- An evaluation of the benefits in terms of reduced register file pressure, showing that up to twice as many threads in a GPU can be active simultaneously.

The rest of the article is organized as follows. Section 2 gives motivational examples of how much precision can be lowered with a negligible deterioration in quality. In Section 3, we propose our method for automating selection of the precision level of each floating-point register, allowing mixed floating-point register precision. Then, Section 4 presents our proposed register file organization capable of storing floating-point registers with different precision levels. Sections 5 and 6 present our evaluation methodology and results, respectively, given by the precision-selection and register-allocation algorithms. Finally, we put our work in context of related works in Section 7 before we conclude in Section 8.

2 MOTIVATION

In this study, we explore the potential benefit of reducing the precision of floating-point register values in programs where a small loss in output quality can be accepted. Specifically, we compile the kernel to a hardware-independent assembly format (LLVM IR [11]) and use our proposed method to annotate the kernel with an appropriate width for every floating-point register, given a certain output quality threshold.

Register-width annotations can be used to enable optimizations to, for instance, functional units (e.g., SIMD-style parallelism [14]), cache systems [6], bandwidth utilization [21], and register file organizations [12]. The register file is of particular interest in GPU architectures. Unlike traditional CPU architectures, which rely heavily on cache hierarchies to hide memory latency, GPUs hide latency by keeping a large number of threads in flight simultaneously, so a new thread can be scheduled whenever a running thread stalls on a memory operation. Since context switches happen at the cycle level, each active thread's state must be live in the register file until it terminates. Consequently, the maximum register pressure of a kernel dictates how many threads that can be simultaneously active. Reducing the space required for floating-point registers can therefore allow for more parallelism and thus improved latency-hiding capability, leading to higher performance.

While GPUs are increasingly being used for general High Performance Computing tasks, the most common task by far, and the task that must motivate any hardware modification, is to render images in real time for, for example, video games, industrial visualization, and medical imaging. Whatever the application, producing a frame for a real-time graphics application invariably consists of a number of *rendering passes*, that is, a number of kernels that are launched in sequence where the final kernel assembles the final output image. The render passes that are required differ from application to application but can include, for instance, shading of each visible pixel, screen-space raytracing of reflections, screen space ambient occlusion, tonemapping, and final compositing. Most of these passes have in common that their output is an image that will be used as input to a subsequent pass, and thus the quality of the output compared to a reference can

Table 1. Distribution of Exponent and Mantissa Bits for Each Format and Bit-width

	Bit-width [exponent bits / mantissa bits]						
	32 bits	28 bits	24 bits	20 bits	16 bits	12 bits	8 bits
IEEE754-compliant	[8 / 23]	–	–	–	[5 / 10]	–	–
Mantissa truncation	[8 / 23]	[8 / 19]	[8 / 15]	[8 / 11]	[8 / 7]	[8 / 3]	–
IEEE754-style	[8 / 23]	[7 / 20]	[6 / 17]	[5 / 14]	[5 / 10]	[4 / 7]	[3 / 4]

Note: One sign bit is used in all configurations.

be measured with established image quality metrics, for example, Structural Similarity [26]. We have investigated a number of such kernels and evaluated how resilient they are to reduction in floating-point accuracy. Our investigation shows that, in all tested kernels, the precision can be lowered significantly with a negligible loss in quality.

In Section 2.1, we first introduce the floating-point formats assumed in this study. Then, in Section 2.2, we show an example of how much the floating-point register widths can be reduced given a certain quality threshold.

2.1 Floating-point Formats

We use three different formats to represent floating-point values at lower precision:

IEEE754-compliant. This format considers single-precision (32-bit) and half-precision (16-bit) floats as specified in the IEEE754-standard. It is the least flexible format in our investigation, but it is of interest, since state-of-the-art GPUs can already make use of these formats.

Mantissa truncation. This format assumes a single-precision floating-point value where mantissa bits are dropped to lower the precision. This gives a more flexible format compared to the IEEE754-compliant format, as there can be a much finer granularity in the tradeoff between width and precision. The exponent width is 8 bits, so the numerical range is the same as for a 32-bit float.

IEEE754-style. This format improves the utilization of available number of bits, at the cost of a lower range for lower-precision numbers, by dynamically setting the width of both the mantissa and the exponent as shown in Table 1.

In Table 1, we list, for each format, the ordered set of possible floating point number representations. We call these the *levels of approximation*. The table shows the distribution of exponent and mantissa bits investigated for each format. All formats use the rounding mode round-to-zero.

2.2 A Motivating Example

In this section, we highlight the results obtained for a common graphics kernel called DEFERRED. This kernel calculates the reflected radiance for each pixel by evaluating a Bidirectional Reflectance Distribution Function (BRDF) for a point-light source as well as for a pre-convolved environment map. For this kernel, we reduced the precision of each floating-point register, using the algorithm described in the next section, such that the output image quality does not differ from the reference by more than a given threshold (as described in detail in Section 5.2). We refer to the lowest acceptable output quality as the *quality threshold*. We measure quality using the Structural Similarity (SSIM) index, as described in detail in Section 5.2.

Figure 1 shows the output for DEFERRED given three different quality thresholds (HIGH, MEDIUM, and Low), and Figure 2 shows the distribution of the obtained register widths after optimization.

Fig. 1. The resulting image from three different quality settings. HIGH is almost indistinguishable from reference. MEDIUM has small deviations. Low has clearly visible differences, but its quality can still be acceptable. The right part of each image visualizes the Structural Similarity (SSIM) index for each pixel.

Fig. 2. The distribution of register widths after optimization.

The x -axis shows the different quality thresholds. Each threshold has three bars, one for each precision format (left: **IEEE754-compliant**; middle: **Mantissa truncation**; right: **IEEE754-style**). Each bar shows the distribution of the register widths: The y -axis shows the number of registers, and each fraction of the bar corresponds to a certain register width.

At the highest quality threshold (HIGH threshold), **IEEE754-compliant** can reduce the width of about half of the registers below 32 bits. At MEDIUM quality, almost all registers can be reduced to the 16-bit format, so there is little more to gain by allowing the Low quality setting. In contrast, the fine-grained formats (**Mantissa Truncation** and **IEEE754-style**) reduce *all* registers below 32 bits already at the HIGH threshold, and a more graceful degradation is obtained as the quality threshold is reduced. At the Low threshold, a large number of registers have been reduced to a width as low as 8 bits.

As will be seen in Section 6, these trends are also visible when comparing average and maximum register pressure. The part of the register file that is occupied by floating-point registers can be reduced to approximately 80% at HIGH quality with the simpler **IEEE754-compliant** format, and to approximately 60% with **IEEE754-style**, the best performing fine-grained format. At Low quality, the **IEEE754-compliant** format reduces the footprint in the register file to approximately 50%, and using the **IEEE754-style** reduces the footprint by an *additional* 60%.

It is clear that it is possible to significantly increase the bit-efficiency by using low-precision floating-point formats that depart from the IEEE754 standard, but to efficiently utilize the fine-grained formats presents a challenge. To this end, in Section 4, we propose a novel GPU register

file organization and a register-allocation algorithm that can efficiently pack architectural registers denser to require fewer physical registers per thread. For the above example, this organization lowers the maximum register pressure per thread by between 38% and 53% for the **IEEE754-style** format, between 28% and 47% for the **Mantissa truncation** format, and between 12% and 40% for the **IEEE754-compliant** format, while maintaining a **HIGH** quality.

That said, even with suitable micro-architectural solutions in place, a challenge is *which* floating-point registers can be represented by lower precision formats and *how much* the precision of these registers should be lowered to meet a certain quality threshold. This is laborious, if not impossible, at the register-level for a programmer to solve. Therefore, we present in the next section an automated heuristic-based method to individually set the precision of each floating-point register.

3 AN AUTOMATED AND CONTROLLED PRECISION-SELECTION METHOD

The goal of our algorithm is to identify *which* floating-point registers can be represented by lower precision formats and *how much* the precision of these registers can be lowered while meeting a certain output quality threshold, given a representative set of application inputs. In short, we need to determine *one* suitable approximation level from a fixed set of *a* approximation levels for each of the *r* registers used in the kernel. As this is carried out at the LLVM IR level, a register refers to a *virtual* register in Single Static Assignment (SSA) form, that is, each register corresponds to one single value definition.

To find the *optimal* set of all registers and all approximation levels, an exhaustive search that evaluates all combinations is needed. Such an approach would have a complexity of $O(a^r)$, which is not practically viable. As a solution to this problem, Rubio-González et al. [18] suggest Precimonious, which uses an algorithm with a worst-case complexity of $O(r^2)$ per approximation level. While this may be an acceptable complexity for tuning a small number of variables it is not feasible for tuning all related virtual registers, of which there can be orders of magnitude more. Instead, we propose a heuristic algorithm, with an acceptable worst-case complexity, that is likely to find a solution close to the optimal. The algorithm is presented in Section 3.1, a worst-case analysis is given in Section 3.2, and we present the implementation in Section 3.3. A short discussion on the limitations of this approach is given in Section 3.4.

3.1 Algorithm

One strategy to consider is that when an approximation level for one register that does *not* meet the requirements is found, there is no reason to investigate lower approximation levels for that particular register. A first naïve approach is to lower the approximation level of each register in turn until the quality does not meet the requirement. However, the level of approximation that can be achieved with this approach is highly order dependent. If a specific register, then r_i , is a function of a set of other registers, S , and r_i is aggressively approximated, it might not be possible to approximate any of the registers in S , resulting in a suboptimal optimization overall. Our strategy is instead to lower the precision breadth-first, that is, to lower as many registers as possible to the first approximation level before continuing to the next. We attempt to lower the precision of many registers in one go: If this gives an acceptable quality, then the registers are subject to further precision-reduction. Otherwise, the registers whose precision could not be reduced are singled out.

The pseudocode in Algorithm 1 shows how the precision-selection is carried out. Ultimately, we want each register to be locked to one of a discrete set of approximation levels defined by the format that is considered (see Table 1). To reach this, our algorithm iterates through the set of considered approximation levels (lines 5 to 9) until either all approximation levels have been visited or until all registers have been locked to an approximation level. The result of each iteration is a subset of unlocked registers, subject to further precision-reduction, if there exist more approximation levels.

ALGORITHM 1: Precision-Selection Algorithm

Input: Approximation levels, a (sorted with highest precision first), All registers in kernel, $registers$, Quality threshold, qt
Globals: A mapping between registers and approximation levels, $precision_of_registers$
Output: $precision_of_registers$

```

1 Algorithm setPrecisions( $a, registers, qt$ )
2    $previous\_precision = pop(a)$ 
3   Set  $registers$  in  $precision\_of\_registers$  to  $previous\_precision$ 
4    $unlocked\_registers = registers$ 
5   repeat
6      $current\_precision = pop(a)$ 
7      $unlocked\_registers = LowerOneStep(current\_precision, previous\_precision, unlocked\_registers, qt)$ 
8      $previous\_precision = current\_precision$ 
9   until  $unlocked\_registers$  is empty  $\vee a$  is empty
10  return  $precision\_of\_registers$ 
11 Procedure LowerOneStep( $current\_precision, previous\_precision, unlocked\_registers, qt$ )
12   Set  $unlocked\_registers$  in  $precision\_of\_registers$  to  $current\_precision$ 
13   Execute kernel with  $precision\_of\_registers$ 
14   if output quality  $< qt$  then
15     if size of  $unlocked\_registers$  is 1 then
16       Set  $unlocked\_registers$  in  $precision\_of\_registers$  to  $previous\_precision$ 
17       clear  $unlocked\_registers$ 
18     else
19        $subset1 =$  upper half of  $unlocked\_registers$ 
20        $subset2 =$  lower half of  $unlocked\_registers$ 
21        $unlocked1 = LowerOneStep(current\_precision, previous\_precision, subset1, qt)$ 
22        $unlocked2 = LowerOneStep(current\_precision, previous\_precision, subset2, qt)$ 
23        $unlocked\_registers = unlocked1 \cup unlocked2$ 
24   return  $unlocked\_registers$ 

```

In each iteration, all *unlocked* registers, that is, registers *not* locked to an approximation level, are considered (lines 11–24). The unlocked registers are adjusted to the current level of approximation, the kernel is executed, and the output quality is tested against the quality threshold (lines 12–14). If the output quality lies above the allowed threshold, then all unlocked registers can be considered for a lower level of approximation, so all unlocked registers are returned.

Otherwise, the algorithm has to identify which of the registers cannot use the current level of approximation. This is achieved by dividing the set of unlocked registers into two parts and recursively evaluating these. Note that this is done sequentially: The precision setup that was found to be acceptable in the first subset is used in the second subset.

If a subset containing a single register causes a quality below the threshold, then this register is set to the *previous* precision, and the register is removed from the unlocked registers (lines 16 and 17).

3.2 Worst-Case Analysis

In each iteration, the set of unlocked registers is recursively split into two. This results in a binary tree, where each node corresponds to one kernel execution. The worst case for *the first* iteration,

iteration 0, arises when all leaf nodes correspond to one single register. This happens when *at least* every other register has to be locked to the current approximation level. Assuming L_0 is the set of registers that need to be locked in iteration 0, we evaluate, in the last level of the binary tree, *at most* each register in L_0 as well as the sibling of each register. We also evaluate *at most* the parent and the parent's sibling of the already evaluated registers. Hence, for each register in L_0 , we do *at most* two evaluations per level. Because we have $\log_2 r_0$ levels, where r_0 is the number of unlocked registers, the total number of evaluations for iteration 0 is bounded by $2|L_0| \cdot \log_2 r_0$. This is a *very* conservative bound, since at some level in the binary tree, the registers in L_0 will share parents.

In the next iteration, L_1 is the set that need to be locked. With the same reasoning as above, the number of evaluations is bounded by $2|L_1| \cdot \log_2 r_1$, where $r_1 = r_0 - |L_0|$. Similarly, the number of evaluations for iteration i is bounded by $2|L_i| \cdot \log_2 r_i$. As r_0 is equal to the *total* number of registers r , $\log_2 r_i < \log_2 r$. Consequently, if a is the number of approximation levels, then the total number of evaluations is bounded by:

$$\sum_{i=0}^{a-1} 2|L_i| \cdot \log_2 r = 2\log_2 r \sum_{i=0}^{a-1} |L_i| < 2\log_2 r \cdot r,$$

because the sum of all locked registers can never exceed the number of total registers r . Hence, the worst-case complexity is *bounded* by, but will *never* be as high as, $O(r \cdot \log_2 r)$.

Another bound on the worst-case complexity is found by realising that the largest possible number of evaluations in one iteration is also bounded by $2r_i - 1$ (the number of nodes in a full traversal tree). Since $r_i < r$, the total number of evaluations is also bounded by $O(ar)$.

It is difficult to prove a tighter bound, but either of the two worst-case bounds presented is a considerable improvement from $O(r^2)$, the worst-case complexity of the Precimonious algorithm. For instance, one of our investigated kernels has approximately 1,500 virtual registers. In the worst case, this corresponds to millions of evaluations with the Precimonious algorithm, compared to a few thousand with our algorithm.

3.3 Implementation

The precision-selection algorithm is implemented at the LLVM IR level, because our goal is to tune the precision of each *register* in contrast to each high-level language *variable* as done in Precimonious [18]. Furthermore, we use a data-driven approach. If a kernel is annotated with the required precision for input and output data and variables in a high-level language, then static analysis can be performed to obtain an *upper bound* on the precision required for the actual intermediate register values, but the result will be conservative, since this approach does not take into account the values stored in these registers. We instead start from optimized LLVM IR and select a precision based on the actual values that each register contains.

To emulate registers of varying precision, we inject code after each instruction whose target is a float register. The injected code modifies the value stored in the floating-point register to the desired level of approximation. An example of how we inject code can be seen below. The framework sets the precision by modifying the constant c , which is set individually for each float register.

Original Code	After Truncation
<code>%reg0 = fmul float %0, %1</code>	<code>%temp0 = fmul float %0, %1</code> <code>%reg0 = call float @trunc(float %temp0, i32 c)</code>

Since all formats we consider (see Table 1) can be exactly represented by the IEEE754 format, the resulting value is stored in a standard 32-bit floating-point register. This correctly emulates the behavior of a system where floating-point values are stored with fewer bits but are expanded to the IEEE754 format prior to any ALU operation.

Fig. 3. The baseline GPU register file is banked. A register access fetches one warp register, which is further divided into thread registers.

3.4 Discussion

We note two limitations of this approach. Since our precision-selection method is *data driven*, it relies on the developer to provide a set of representative inputs. As the optimization is based on these inputs, no quality guarantees are made for other inputs. To choose a set of inputs that span the practical input space is a problem in all test-driven approaches, where it is impossible to iterate through all possible inputs. This problem is orthogonal to the questions we consider in this article.

Although we focus on graphics algorithms, we claim that our approach can be generalized to all applications where it is possible to establish a metric to quantify the quality of the output. Establishing such a metric is not trivial, and often has to be made with care by a domain expert, as the result from the precision-selection step will only be as good as the quality metric.

4 A REGISTER FILE ORGANIZATION WITH CONFIGURABLE PRECISION

The automated precision-selection method results in a specification of register widths that can be used for efficiency improvements at the microarchitecture level. This section presents a GPU register file organization with the ability to store registers at different precision levels with the goal of extending the capacity of the register file by tightly packing the architectural registers.

The baseline register file organization is inspired by that of a typical NVIDIA GPU. Its execution model groups multiple threads into execution units called *warps*. The threads in each warp run in lockstep, that is, all threads in the warp execute the same instruction but with different register values in a SIMD fashion. Hence, the register file feeds the instruction with *warp registers*, which consist of the thread registers in the warp. All architectural warp registers have a one-to-one mapping to physical warp registers using the warp and register identity to index into the register file. Also, to minimize latency during warp context-switches, when a warp is in flight, the registers needed for its entire execution are readily available in the register file. In addition, the register file is banked to reduce read and write conflicts while keeping the number of read and write ports to a minimum.

Figure 3 shows a register fetch where warp one accesses register r_2 . Here, each warp uses eight registers in total, so each warp adds an offset, denoted $R = 8$ to access the correct physical register. This means that r_2 of warp one corresponds to the physical warp register $8 * 1 + 2 = 10$, which is fetched from bank 2. This location contains r_2 of the threads belonging to warp one.

In Section 4.1, we introduce the new register file organization concept, and Section 4.2 discusses the challenges in implementing such a register file. Then, in Section 4.3, we present how architectural registers are allocated in our proposed register file organization.

4.1 Register File Organization

Our proposed register file organization introduces two extensions to the baseline organization. First, each physical register is divided into a number of *slices*, and each architectural register is

Fig. 4. Our proposed register file organization consists of an indirection table (in center) and a sliced and banked register file (to the right).

allotted the number of slices needed to store its content. This way, multiple architectural registers can be placed within one physical register, which allows the register file to tightly pack the architectural registers. Second, the proposed organization has a virtualization mechanism that allows for a one-to-many mapping between architectural and physical warp register locations in the register file, as a physical register is decomposed into slices. A temporal one-to-one mapping is supported by live analysis at the compiler level. The proposed register file organization allows for reduction of the *register pressure* of each thread to free up physical register space that can be used for keeping more threads in-flight. In this article, the proposed register file organization is intended as a target to show the potential of floating-point accuracy reduction rather than fully analyzing the properties of it.

The register file comprises two parts: an indirection table and a banked and sliced register file, as shown in Figure 4. The figure shows an example of a register fetch, the same as in Figure 3, but considering the new organization. Each physical register is divided into eight slices, and each architectural register can be split into two parts that can reside in arbitrary slices of two physical registers. A mask for each register is used to identify the slices containing the data.

The register fetch in Figure 4 proceeds as follows. As in the previous example, warp one accesses architectural register $r2$. Before the fetch, the indirection table dictates that the content is split into physical registers 2 and 4. As in the baseline case, an offset based on the total number of registers per warp $R = 6$ and the warp number $i = 1$ is used to correctly index into the register file and fetch the correct physical registers, which means that register $r2$ of warp one is located in the physical registers $6 * 1 + 2 = 8$ and $6 * 1 + 4 = 10$. Two masks, also specified in the indirection table, are used to extract the slices containing register $r2$ from two physical registers.

As each architectural register is mapped to slices of physical registers using the indirection table, the ISA is oblivious to the required precision of each register. This entails that no information about register width is included directly in the instructions, and the ISA does not have to be modified. However, as the mapping between the architectural registers and the physical registers is generated specifically for each kernel, the content of the indirection table needs to be loaded before the kernel is executed. We assume that this is done at the same time as the GPU is programmed with the kernel code. Additionally, we assume that the configuration of all registers is kept in the indirection table during the entire execution of the kernel, so the indirection table only has to be reloaded when the GPU launches a new kernel. Because the register allocation is made by the compiler, only a lookup in the indirection table is required by the hardware.

The indirection table has one entry per architectural register, meaning that it is small compared to an NVIDIA Pascal register file of 256kB [14]. Assuming each physical register identity is eight bits (to be able to identify 256 physical registers per warp, as possible in the Pascal architecture),

each mask is eight bits (for eight slices per register), and the limit of architectural registers is 256 (as in the Pascal architecture), it takes $256 \times (8 + 8 \times 2)$ bits $\Rightarrow 8192$ bits $\Rightarrow 1$ kB of memory to store the content of the indirection table, corresponding to $\frac{1\text{kB}}{256\text{kB}} \approx 0.4\%$ of the register file.

4.2 Discussion

There are several architectural issues to address to implement the proposed register file organization, but we argue that it is possible to mitigate all these challenges. First, the indirection table lookup, which has to be performed for each register access, may introduce an unacceptable latency that can have a negative impact on performance. However, as shown by Lee et al. [12], an extra access that adds latency at the register fetch stage can be mitigated by pipelining, at least as long as the additional access has a latency that does not exceed that of the register file. As the indirection table is less than 1% in size compared to the register file, this seems viable.

Second, the indirection table might be a bottleneck when several read accesses are performed simultaneously if the number of read ports are less than the number of accesses. As many read ports to the same memory requires a lot of physical space, it is not viable to assume a multitude of read ports. Instead, the indirection table can be banked. This is the same solution as is in place to solve the equivalent issue for conventional GPU register files.

4.3 Allocation of Registers

The precision-selection step described in Section 3 provides the compiler with a width for each virtual register. The number of virtual registers is unbounded for an arbitrary kernel, as each value definition requires its own virtual register. However, the number of *architectural* registers is limited, meaning that multiple virtual registers have to share one architectural register if the number of virtual registers exceeds the maximum number of architectural registers. The merging of virtual registers has to be carried out carefully, since it might result both in an unoptimal utilization of bits and physical allocation problems. A method for mitigating these problems is described below.

After the virtual registers are merged into architectural registers, they must be allocated to a physical location in the register file such that the *maximum* register pressure is minimized. How we achieve this is described in Section 4.3.2.

4.3.1 From Virtual to Architectural Registers. One established method for assigning virtual registers to architectural registers is to create and color a graph representing the interference between all virtual registers in the program. If the graph is colored in such a way that no adjacent nodes have the same color, then the colors represent a valid register allocation. This approach takes into consideration the *mergingexample(b)* of each virtual register, but it is unaware of the required bit-widths. The result is that virtual registers of unequal widths might end up in the same architectural register. Because our suggested indirection table demands a static allocation to physical registers, and because we need to maintain the quality guarantee given by the precision-selection step, an architectural register containing virtual registers of unequal widths has to adopt the width of the *largest* virtual register. This is not desirable, as some virtual registers would utilize more bits than necessary.

One solution to this problem is to split the registers acquired by graph coloring according to their width in such a way that all architectural registers represent both a single width and color. However, this might cause problems when allocating the architectural registers to a physical location if virtual registers that are live in different parts of the program are merged. An example of this problem is visualized in Figure 5. An example program and its virtual registers are shown in Figure 5(a) and 5(b). The optimal allocation in the physical register file takes up nine slices (Figure 5(c)), and in Figure 5(d), the registers have been divided into four different colors. Each architectural register

Fig. 5. An example of merging virtual registers into architectural registers.

contains only one color and width. These registers utilize 11 slices, 22% more than in the optimal case, because of unfortunate merging of virtual registers: register seven prevents register two from being placed in the two first slices. Similarly, registers three and five prevent register six from being placed into the two first slices. To alleviate this kind of problems, we split the architectural registers according to which *basic block* they were created in (Figure 5(e)).

Unfortunately, after splitting, the number of architectural registers might exceed the acceptable limit. For instance, assume the limit of architectural registers is six. While the merging conducted in Figure 5(e) makes it possible to find an allocation that only uses nine slices, the number of architectural registers is seven instead of four (Figure 5(d)). Hence, we have exceeded the acceptable limit and need to re-merge some of the registers we just created.

We re-merge the registers based on the highest *optimal register pressure*, that is, the summed width of all live registers, they experience during their lifetime. Since we optimize for *maximum* register pressure, it is important to have many allocation options at the *highest* optimal register pressure. Therefore, we merge registers experiencing as *low* optimal register pressure as possible. For instance, register eight experiences a maximum optimal register pressure of six, the lowest register pressure any of the registers experience. Hence, we select register eight to re-merge. The registers to merge have to be of the same color and width, so we pick register four and re-merge it with register eight. This way, we achieve an optimal allocation using six architectural registers instead of eight, as was the case in Figure 5(c).

An overview of the just-described steps are presented in Figure 6. In addition to the steps described above, note that if there are still architectural registers to spare after the first three splitting passes, we continue splitting the architectural registers containing the largest number of virtual registers until we reach the maximum number of architectural registers. This is because using as many architectural registers as possible will give more possibilities in the next step, when the architectural registers are assigned to physical locations. Our approach does not guarantee an optimal merging of virtual registers, but it works well in practise, as can be seen in Section 6.3.

Fig. 6. The steps taken to merge virtual registers into architectural registers. #Regs and MaxRegs denote the *current* number of and the *maximum* number of architectural registers, respectively.

4.3.2 From Architectural to Physical Registers. To use our proposed register file, we must statically allocate physical register slices to each architectural register and create an indirection table (see Figure 4). Optimally, physical register slices should be allocated such that, at any point in the program, the space required in the register file (for each warp) is equal to the width of the currently *live* registers. Finding such a globally optimal solution would be very complex, and instead we present a simple greedy algorithm that, in our experiments (see Section 6.3), is sufficient to ensure that the *maximum* register pressure is close to the optimal.

We iterate through all instructions in the kernel, sorted by their optimal register pressure, and whenever a previously unallocated register is assigned a value, we consider *all* possible combinations of placements within the available physical registers, assign a score to each combination, and then select the combination with the best score. Thus, the worst-case complexity is bounded by $O(R * P^2)$, where R is the number of registers and P is the maximum register pressure.

The register allocation algorithm is applied at the LLVM IR level. First, we perform a liveness analysis to identify, for each instruction, which new architectural registers need to be allocated and which can be released at this point in the program. We then iterate through all instructions to assign physical slices to each live architectural register. When a new architectural register needs to be allocated, *all* possible combinations of one or two physical registers, with sufficient free slices, are considered and a score is assigned to each combination. The main objective of the allocation algorithm is to avoid fragmentation and, to that end, we assign scores for a combination based on the number of free slices that are left in each used register. It is, for instance, better to choose a combination that fills the slices of two partially filled physical registers than one which leaves one slice of a register unoccupied, since that slice may be hard to occupy in later allocations. Given a combination of registers, r_0 and r_1 , with s_0 and s_1 unoccupied slices each, we calculate the score for the combination as:

$$\text{score} = \text{cost}(s_0) + \text{cost}(s_1) - \text{cost}(s'_1)$$

$$\text{cost}(x) = \begin{cases} 0, & \text{if } x = 0 \\ S - x, & \text{otherwise,} \end{cases}$$

where s'_1 is the number of unoccupied slices in r_1 after the allocation, and S is the total number of slices of one register. For the combinations of only one register, r_0 , with s_0 unoccupied slices, the score is simply $\text{cost}(s_0) - \text{cost}(s'_0)$, where s'_0 is the number of unoccupied slices after the allocation.

If no valid combinations were found among the already partly occupied registers, an empty register is added which guarantees that we find a combination in a second pass. The combination with the best score is chosen and the corresponding slices are marked as occupied by this architectural register. When deallocating an architectural register, the slices are simply marked as unoccupied.

Fig. 7. An overview of the evaluation steps.

5 EVALUATION METHODOLOGY

In the previous sections, we have described a method for automatically selecting the required precision for each register in a kernel (Section 3) and a reconfigurable register file organization (Section 4). We have evaluated both of these within the same framework, illustrated in Figure 7. We have run the experiment on a few nodes of a cluster that is built on Intel Haswell (2650v3) CPUs. In total, the cluster has 315 compute nodes with 20 cores each and 26TB of RAM.

The steps taken by the framework are as follows. First, the kernel, written in C++, is compiled and optimized using standard tools from the LLVM package (clang version 3.5.0 and opt version 3.5.0) which produces standard LLVM IR.

Second, we automatically inject instructions that make it possible to dynamically set the precision of the contents of each floating-point register. We then run our precision-selection algorithm to find the optimal set of precisions for a given quality threshold and a representative set of sample inputs (step 3). We have measured the resulting required widths of all architectural registers for a wide range of thresholds, as presented in Section 6.

Next, we use these results to evaluate the proposed reconfigurable register file organization. We first perform a liveness analysis on the LLVM IR (step 4) to find the static liveness for all instructions. Using this information and the list of required register widths, the register allocation algorithm (see Section 4.3) produces the content of the indirection table that maps architectural registers to physical register slices (step 5). We measure the actual occupancy of the register file for each instruction of the kernels, along with the optimal register pressure.

In the next two sections, we will describe the kernels used for all evaluations and the quality metric used for deciding when an approximate result is good enough.

5.1 Evaluation Kernels

We have evaluated our method using four different kernels. The first two, DEFERRED and SSAO, are standard rendering passes used in most modern real-time applications. ELEVATED and PATHTRACER are both larger kernels taken from the popular *shadertoy*s [17] web site and are chosen because they make use of several techniques that are common to many graphics algorithms. Each kernel is briefly described below and Figure 8 shows the output images at full precision.

DEFERRED. This kernel performs the *shading* pass in a deferred rendering framework. The input is a camera and light-source position, a set of pre-convolved environment maps, and textures containing the position, normal, and material parameters for each pixel. The task of the kernel is to evaluate the radiance reflected towards the camera for each pixel by evaluating

Fig. 8. Sample outputs from all kernels at maximum quality.

a Torrance-Sparrow [23] Bidirectional Reflection Distribution Function (BRDF). This is a very common task, required in almost any real-time rendering application.

SSAO. This algorithm is also very common in real-time applications. It estimates how occluded the surface at each pixel sample is from incoming environment illumination and outputs an *ambient occlusion* value per pixel. The specific technique used is Horizon-Based Ambient Occlusion [2].

ELEVATED. Using only an input image containing uniformly distributed random numbers, this kernel generates an image of a fractal landscape with terrain, clouds, atmospheric scattering, and soft shadows, through ray marching. While this is not how images are generally created, the techniques used (e.g., ray marching, evaluation of fractals, and perlin noise) are common in many other algorithms.

PATHTRACER. Finally, this kernel (also from shadertoy) implements a standard path-tracing algorithm on a minimal virtual scene. We believe this kernel to be an interesting indicator of how a full GPU pathtracer could benefit from our method, although it should be noted that it does not implement ray intersections against large scene hierarchies as would be required for a complete renderer. This kernel also makes use of techniques common to many graphics algorithms (e.g., ray-triangle intersections and importance sampling of BRDFs).

5.2 Measuring Quality of Output

An important part of approximate computing is how to select a proper quality metric for determining when the output is good enough. It must be possible to quantify whether the *final* output is perceived as good enough by the end user; hence, the chosen metric should be application dependent. In this article, all of the evaluated kernels produce images as their final output and we use the SSIM [26] index to measure quality. The SSIM index is a score, calculated per pixel, which has been validated against a user-study and is a well-established method for comparing the quality of, for example, compressed images. It should be noted that the metric only compares gray-scale images and therefore we consider the color channels separately and take the minimum value as each pixel's index. Additionally, it is common to use the *average* SSIM value over all pixels to get a single overall quality-metric for a distorted image. Since, in our case, this would allow small image regions to deviate strongly from the reference image, as long as the image is of good quality overall, we instead conservatively use the *minimum* SSIM value as the overall metric.

6 EXPERIMENTAL RESULTS

This section presents the results from using our precision-selection and register allocation algorithms. The kernels are optimized for a range of quality thresholds between $SSIM = 0.5$ and 1.0 , and for the precision formats presented in Table 1. We evaluate the results of the precision-selection

Fig. 9. The distribution of width and total bit reduction for four computer graphics kernels. Each quality threshold has three bars that represent the formats, from left to right: IEEE754-compliant, Mantissa truncation, IEEE754-style. The connected-dot lines correspond to the average number of bits used per register and format.

algorithm by looking at the register width distribution, the optimal per-instruction register pressure, and the maximum register pressure. The register allocation algorithm is evaluated by comparing the maximum register pressure obtained after allocation with the optimal maximum register pressure (the sum of the widths of all live registers at the instruction with the highest pressure).

6.1 Register Width Distribution

To evaluate how effective the precision-selection algorithm is at identifying floating-point registers that require less than 32 bits for a given quality threshold, we first show (in Figure 9) how all floating-point registers in the kernel are distributed for different levels of approximation. To give an intuition for the amount of savings each format allows for, the figure also shows the average number of bits needed per register. While the kernels are fundamentally different and vary greatly in complexity, we identify trends for four ranges of quality thresholds, which we call *Pixel perfect*, *Extremely high quality*, *Good quality*, and *Usable quality*. Outputs from the SSAO kernel using these ranges as thresholds can be seen in Figure 10.

Pixel perfect. An SSIM index value of 1.0 indicates that the image is *pixel perfect*, meaning that the image is identical to the reference image. Here, only a few registers can be reduced below 32 bits with either method. Considering that each kernel will be run literally millions of times for varying inputs, it is understandable that most registers will, when approximated even slightly, at some point cause a single pixel to deviate by a tiny amount from the reference value.

Extremely high quality. In this range the output image is perceptually indistinguishable from the reference, but we do allow for very small errors. Using the fine-grained methods, the

Fig. 10. Output from the SSAO kernel using the Mantissa truncation format and four quality thresholds.

majority of registers (*all* registers in the DEFERRED kernel) can be reduced from 32 bits, and most of the registers are represented by 20–28 bit values for all kernels. The IEEE754-compliant format is less efficient than the other formats, but at the bottom of this range (SSIM = 0.99) even this format can sufficiently well approximate the majority of registers in the DEFERRED and SSAO kernel. It should be noted that while the Mantissa truncation format almost consistently removes more bits than the IEEE754-compliant format in this range, there is one point in the SSAO graph (SSIM = 0.992) where the IEEE754-compliant format removes more bits than Mantissa truncation. This is because a half-precision float value corresponds roughly to a 20-bit Mantissa truncation value, and the Mantissa truncation format has not been able to reduce many registers below 16 bits.

Good quality. In this range, errors are noticeable but not distracting. Looking at DEFERRED and SSAO, we see that we have already exhausted most of what can be achieved when using the IEEE754-compliant format. The few remaining registers that are represented by 32 bits cannot be represented by 16 bits. In contrast, the fine-grained formats can gradually use smaller registers. In the ELEVATED and PATHTRACER kernels, which contain a large number of very sensitive registers, there is a gradual improvement for all methods.

Usable quality. In this range, artifacts are clearly visible, but the image quality may still be sufficient in some use cases. As we allow for higher errors, we see that even the fine-grained formats will not yield many more bits in most kernels. At this point, almost all registers are represented by so few bits that further approximating either of them will lead to a completely erroneous pixel color. That all registers are now very sensitive to further approximations is also visible in that, for instance, the results of PATHTRACER at SSIM = 0.6 are worse than at 0.7, simply because of the order in which the optimization was done.

In summary, the IEEE754-compliant format makes it possible to reduce the average number of bits needed per register to 16–17 bits, that is, use half-precision floats almost exclusively, for two out of four kernels when using a good or usable quality threshold. In these cases, the fine-grained formats can further remove one to two bits (Mantissa truncation) or three to five bits (IEEE754-style). When using an extremely high-quality threshold, the IEEE754-compliant format allows for an average number of bits per register between 18 and 26 bits depending on the kernel. The Mantissa truncation format can remove additionally 1 to 4 bits depending on the kernel, while the IEEE754-style format can additionally remove between 4 and 7 bits.

It is clear that, if even an extremely small error is allowed for, the vast majority of registers can be represented by fewer than 32 bits in all of our examples. It is preferable to allow for a more fine-grained reduction in precision than the formats specified in the IEEE754-standard, since this allows for both a larger reduction overall and a graceful degradation as we move from one quality level to the next.

6.2 Per-instruction Register Pressure

In this section we evaluate how the register widths affect what needs to be stored in the register file at any given time during the lifetime of the kernel. Studying the amount that register pressure can be reduced at all instructions (and not just the maximum register pressure) can be interesting for, for instance, power optimizations such as shutting down unused banks [12] or for performance optimizations such as sharing unused physical registers [7].

In Figure 11, we have plotted the *optimal register pressure*, that is, the summed width of all *live* registers, for each instruction in the kernel, at three different quality thresholds, each representing one of the SSIM ranges defined above (see Section 6.1). The *x*-axis represents the program line number, and the *y*-axis shows the number of live register bits for that particular point in the program. The top line in each graph shows the unoptimized register pressure.

In general, the trends reflect the results seen in Section 6.1: A lower-quality threshold means a lower overall register pressure. On average, across all kernels, the IEEE754-compliant format reduces register pressure by 87%, 77%, and 71% for extremely high quality, good quality, and usable quality, respectively. The corresponding numbers for the Mantissa truncation and IEEE754-style format are 81%, 72%, 67% and 76%, 66%, 60%, respectively. Hence, while the IEEE754-compliant format reduces the register footprint in all cases, the Mantissa truncation format performs better, reducing the total register pressure additionally 5%, and the IEEE754-style format gives the best performance in terms of register pressure reduction, with on average 10% more total reduction than the IEEE754-compliant format.

When the fraction of bits that represent floating-point values is reduced, the registers containing non-float values become more prominent and eventually dominate the number of bits that make up the register pressure. This is most obvious in the SSAO graphs where the number of non-float register bits make up about half of the total number of bits in the unoptimized case. To more clearly show the yields of approximating floating-point numbers with our method, Figure 12 shows the register pressure of only the floating-point registers, normalized to the unoptimized case. These results clearly show the benefit of using the more fine-grained formats: on average, Mantissa truncation lowers the register pressure with approximately 10% compared to the IEEE754-compliant format, and the IEEE754-style format improves the average result with an additional 10%.

In summary, the IEEE754-style format always performs better than the IEEE754-compliant and the Mantissa truncation format. The IEEE754-style format can remove up to 65% on average in the best case (the DEFERRED kernel with a usable quality threshold), while the same number is 48% and 42% for the Mantissa truncation format and IEEE754-compliant format respectively.

6.3 Maximum Register Pressure

Finally, we evaluate the improvements obtained by our reconfigurable register file organization and register allocation algorithm. We focus on the impact the reduced-width registers have on the maximum register pressure. As the maximum register pressure directly affects how many threads the reconfigurable register file can keep in-flight, this is a key metric.

To evaluate the allocation algorithm, we compare the *optimal* maximum register pressure, that is, the maximum register pressure achievable without any fragmentation, to the register pressure obtained with our allocation algorithm.

Figure 13 shows the maximum register pressure (on the *y*-axis) for SSIM thresholds between 0.5 and 1.0 (on the *x*-axis). The solid lines show the register pressure *after* register allocation using the algorithm described in Section 4.3, while the dashed lines show the corresponding *optimal* register pressure. For DEFERRED, SSAO, and PATHTRACER, the register allocation algorithm performs optimally or very close to optimally, that is, the registers can be tightly packed at the highest

Fig. 11. Width of live registers per instruction. The gray area at the bottom represents the registers that are *not* approximated (i.e., the registers that do not contain floating-point values).

register pressure lines. This is, however, not the case for the ELEVATED kernel, where, for most quality thresholds, the maximum register pressure is between one and five registers higher than the optimal register pressure for all formats.

The slack occurring for ELEVATED is a result of the merging of virtual registers (Section 4.3.1): This kernel is the one most affected, since it has about 6 times more virtual registers than architectural registers. In comparison, the same number for DEFERRED is about 2. For SSAO and

Fig. 12. Fraction of live registers per instruction compared to the reference, disregarding non-float registers.

PATHTRACER, all their virtual registers can be allocated to separate architectural registers. When merging many virtual registers, the allocation possibilities in the physical allocation step are constrained. Hence, a less optimal register allocation is to be expected in cases where the number of virtual registers greatly exceeds the number of architectural registers. Nevertheless, the maximum achieved register pressure of ELEVATED shows the same trends as the other kernels.

Let us put these graphs into the context of a throughput processor. For instance, the original maximum register pressure of the DEFERRED kernel is 33 registers. By allowing a quality threshold

Fig. 13. Maximum register pressure for four different kernels, at quality thresholds between 0.5 and 1.0 (non-linear scale). Dotted line: maximum register pressure of the unoptimized kernel. Dashed line: *optimal* maximum register pressure. Solid line: *allocated* maximum register pressure.

of, for example, SSIM = 0.97, both the IEEE754-compliant and the Mantissa truncation formats lower the register pressure to 21 registers, a decrease of 36%. Thus, 36% of the physical space in the register file can be used to increase the number of warps in-flight, leading to a higher utilization of the resources and more efficient latency hiding. Similarly, if the IEEE754-style method is used, the number of registers is reduced to 16, a decrease of 51%. In summary, there is most to gain in the beginning of the quality threshold range; a small deviation means a large difference in maximum register pressure. PATHTRACER is an exception: No difference between thresholds is seen before the end of the threshold range. This is because the SSIM index is a very conservative measurement for the PATHTRACER kernel, which outputs a rather noisy image. A different quality measurement would probably produce a result more similar to the other kernels.

Since the non-float registers have an increasingly bigger impact on the register pressure when the portion of bits that make up the float registers is reduced, it would be interesting to investigate the possibility to decrease the bits needed to describe, for example, the integer registers. This would further reduce the maximum register pressure. While out of scope for this article, we believe it is possible to reduce the widths of also these registers, but through the removal of zeros at MSB bits rather than approximating LSB bits (as investigated for CPUs by Ergin et al. [4]). Because of that, we also present the maximum register pressure reduction of only the fraction of floats in Figure 14. Note that for DEFERRED and ELEVATED, this simplifies the merging of registers, since fewer virtual registers are considered. By allowing a quality threshold of 0.99 (the lower region of the “Extremely high quality”-range), all formats reduce the fraction of bits to at least 70% in all kernels. The corresponding number for the IEEE754-style format is 60%. Hence, the IEEE754-style format performs *at least* 10% better than the IEEE754-compliant method for this threshold.

In summary, the performance of the register allocation algorithm is optimal or close to optimal in most cases. Furthermore, the maximum register pressure can be lowered by using all three

Fig. 14. Maximum register pressure for four different kernels, non-float registers disregarded, at quality thresholds between 0.5 and 1.0 (non-linear scale). The maximum register pressure is normalized to the maximum register pressure of the unoptimized kernel.

precision formats, but the IEEE754-style format provides the highest reduction: In all cases, it can reduce the maximum register pressure with *at least* 10% if the threshold is set to be in the good quality range. In most cases, the Mantissa truncation format performs better or slightly better than the IEEE754-compliant format, but there are cases, for instance, for the PATHTRACER kernel, where the Mantissa truncation format provides no advantage over the IEEE754-compliant format.

7 RELATED WORK

Automatically exploiting lower precision. In the reconfigurable hardware domain, a well-studied problem is how to optimize the word-length of internal variables in a fixed-point hardware implementation. One example is the work of Constantinides [3]. However, as it is tailored for differentiable nonlinear systems and fixed-point numbers, it cannot be applied to our problem.

More general efforts have also been made to tune the precision of floating-point values, both on the variable level [18] and the register level [10]. However, these studies only consider IEEE754-compliant formats, whereas we study a range of different floating-point formats ranging from 8 to 32 bits. Additionally, these attempts do not consider any architectural optimizations as they target already existing architectures. Furthermore, they aim to provide the programmer with hints about which source-code floating-point variables are targets for precision lowering. In contrast, we provide a completely transparent method, with no need to modify source-code after the analysis.

Studies where the floating-point format departs from the IEEE754-standard have also been made: Tong et al. [22] study the impact of lowering precision by modifying both

the exponent and mantissa of floating-point numbers on signal processing applications. In contrast, our study provides results for graphics applications and use it to explore architectural optimizations. Pool et al. [16] investigate the tradeoff between floating-point precision and ALU energy savings in pixel shaders by using the Mantissa truncation format. We instead investigate the tradeoff between floating-point precision and performance gains focusing on register files. Also, we investigate the precision *per instruction*, while they set the precision *per shader*.

The work by Jain et al. [6] also investigates the Mantissa truncation format, but in the context of optimizations in the CPU memory hierarchy. Since they target the memory hierarchy, this is orthogonal to our work. We believe that their techniques can be applied to our results.

Misailovic et al. [13] proposes a reliability- and accuracy-aware framework for optimizing approximate computational kernels, including interval analysis of source-code variables to bound the variables to minimum-required precisions. In contrast, our approach optimizes on actual data on assembly level, which makes it less likely to overestimate the required number of bits.

Approximate computing for GPUs. Samadi et al. [19] propose SAGE, a self-tuning approximation approach for GPUs, which uses off-line compilation to compile a number of kernels with a varying degree of approximations and runtime management to monitor and control the accuracy of the output. In contrast to our work, this is implemented entirely in software and does not address the impact of reducing floating-point variable precision. On the other hand, Satish et al. [21] study floating-point variable approximations for improving I/O link bandwidth of GPGPU workloads. In contrast to our work, however, they do not factor in quantitative assessments of the quality and do not focus on register file efficiency.

Register file optimizations based on narrow integer registers. Several studies have targeted how to improve register file efficiency by exploiting narrow-width integer registers, for both GPUs and general-purpose CPUs. This approach exploits the fact that many benchmarks contain a large fraction of narrow-width integers, which can be stored in fewer than 32 or 64 bits. Concerning CPUs, Ergin et al. [4] present register packing, where several register values can be placed simultaneously inside one physical register. Our reconfigurable register file organization is inspired by this idea, but extends it by allowing for one value in multiple registers. Furthermore, we investigate the impact on floating-point values while they investigate the impact on integers.

Kondo and Nakamura [9] propose bit-partitioning of register file banks and Wang et al. [25] investigate asymmetrically sized register banks. Furthermore, the work by Gilani et al. [5] proposes partitioning of the datapath and register file, in GPUs, into two parts. The approach of exploiting narrow-width integers is orthogonal to our proposal as we target floating-point operands. In fact, our proposed register file could use narrow-width integers to further increase its efficiency in decreasing the maximum register pressure.

GPU register file optimizations. Register file optimization techniques targeting GPUs include a method which compresses the register file data using BDI [12], virtualization of the register file [7], and a partitioned register file that places rarely accessed registers in a low-energy, high-latency register bank [1]. All of these approaches target power efficiency rather than performance efficiency, and none of them exploit approximations. A work by Jeong et al. [8] does, however, exploit approximations of mantissa bits by lowering the refresh rate register bits in a eDRAM-based register file. They do not, however, allow different approximations between registers, but all float registers are equally approximated.

8 CONCLUSION

Approximate computing has opened an avenue for using computational resources more efficiently. However, *what* to approximate and *to what degree* are still open questions. In this article, we investigate approximation of floating-point values in computer graphics kernels using three different low-precision formats. We do this by presenting an automated method of narrowing the architectural float registers to meet a given output quality threshold. Furthermore, we introduce a novel GPU register file organization together with a register allocation algorithm that can take advantage of narrow architectural registers and pack them tightly together, freeing up physical registers that can be used to keep more threads in-flight. We show that only by targeting floating-point registers, the maximum register pressure can be reduced up to 48%, 27% on average, while still maintaining an extremely high quality of the output. We believe that our results show that approximation of floating-point values is a viable way of controlling approximation and trade bits for efficiency, especially if the float formats are allowed to deviate from the IEEE754 standard to allow for fine-grained precision tuning. However, more work has to be done to solve the microarchitectural challenges that arise when it comes to efficiently taking advantage of such formats. Nevertheless, our work shows that it is possible to considerably increase the number of threads in-flight in GPUs by accepting a small loss of output quality and even use the accepted quality as a knob for controlling the number of threads in-flight.

REFERENCES

- [1] M. Abdel-Majeed, H. Jeon, A. Shafaei, M. Pedram, and M. Annavaram. 2017. Pilot register file: Energy efficient partitioned register file for GPUs. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture*. 589–600. DOI : <http://dx.doi.org/10.1109/HPCA.2017.47>
- [2] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. 2008. Image-space horizon-based ambient occlusion. In *Proceedings of the ACM SIGGRAPH 2008 Talks (SIGGRAPH'08)*. ACM, New York, NY, Article 22, 1 pages. DOI : <http://dx.doi.org/10.1145/1401032.1401061>
- [3] George A. Constantinides. 2006. Word-length optimization for differentiable nonlinear systems. *ACM Trans. Des. Autom. Electron. Syst.* 11, 1 (Jan. 2006), 26–43. DOI : <http://dx.doi.org/10.1145/1124713.1124716>
- [4] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev. 2004. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*. 304–315. DOI : <http://dx.doi.org/10.1109/MICRO.2004.29>
- [5] S. Z. Gilani, N. S. Kim, and M. J. Schulte. 2013. Power-efficient computing for compute-intensive GPGPU applications. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. 330–341. DOI : <http://dx.doi.org/10.1109/HPCA.2013.6522330>
- [6] A. Jain, P. Hill, S. C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. 2016. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–13. DOI : <http://dx.doi.org/10.1109/MICRO.2016.7783744>
- [7] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. 2015. GPU register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, 420–432. DOI : <http://dx.doi.org/10.1145/2830772.2830784>
- [8] D. Jeong, Y. H. Oh, J. W. Lee, and Y. Park. 2016. An eDRAM-based approximate register file for GPUs. *IEEE Des. Test* 33, 1 (Feb. 2016), 23–31. DOI : <http://dx.doi.org/10.1109/MDAT.2015.2500185>
- [9] M. Kondo and H. Nakamura. 2005. A small, fast and low-power register file by bit-partitioning. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. 40–49. DOI : <http://dx.doi.org/10.1109/HPCA.2005.3>
- [10] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. 2013. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS'13)*. ACM, New York, NY, 369–378. DOI : <http://dx.doi.org/10.1145/2464996.2465018>
- [11] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Washington, DC.

- [12] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. 2015. Warped-compression: Enabling power efficient GPUs through register compression. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 502–514. DOI: <http://dx.doi.org/10.1145/2749469.2750417>
- [13] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*. ACM, New York, NY, 309–328. DOI: <http://dx.doi.org/10.1145/2660193.2660231>
- [14] NVIDIA. 2016. NVIDIA Tesla P100. *White Paper*. (2016).
- [15] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. 2015. FlexJava: Language support for safe and modular approximate programming. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, New York, NY, 745–757. DOI: <http://dx.doi.org/10.1145/2786805.2786807>
- [16] Jeff Pool, Anselmo Lastra, and Montek Singh. 2011. Precision selection for energy-efficient pixel shaders. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG'11)*. ACM, New York, NY, 159–168. DOI: <http://dx.doi.org/10.1145/2018323.2018349>
- [17] I. Quilez and P. Jeremias. 2017. Shadertoy. Retrieved March 27, 2017 from <https://www.shadertoy.com/>.
- [18] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM, New York, NY, Article 27, 12 pages. DOI: <http://dx.doi.org/10.1145/2503210.2503296>
- [19] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, 13–24. DOI: <http://dx.doi.org/10.1145/2540708.2540711>
- [20] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, 164–174. DOI: <http://dx.doi.org/10.1145/1993498.1993518>
- [21] Vijay Sathish, Michael J. Schulte, and Nam Sung Kim. 2012. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, 325–334. DOI: <http://dx.doi.org/10.1145/2370816.2370864>
- [22] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. 2000. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Trans. VLSI Syst.* 8, 3 (Jun. 2000), 273–286. DOI: <http://dx.doi.org/10.1109/92.845894>
- [23] K. E. Torrance and E. M. Sparrow. 1967. Theory for off-specular reflection from roughened surfaces*. *J. Opt. Soc. Am.* 57, 9 (Sep. 1967), 1105–1114. DOI: <http://dx.doi.org/10.1364/JOSA.57.001105>
- [24] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, 1–12. DOI: <http://dx.doi.org/10.1145/2540708.2540710>
- [25] S. Wang, H. Yang, J. Hu, and S. G. Ziavras. 2007. Asymmetrically banked value-aware register files. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'07)*. 363–368. DOI: <http://dx.doi.org/10.1109/ISVLSI.2007.27>
- [26] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Trans. Image Process.* 13, 4 (Apr. 2004), 600–612. DOI: <http://dx.doi.org/10.1109/TIP.2003.819861>

Received April 2017; revised September 2017; accepted October 2017