Exploiting half precision arithmetic in Nvidia GPUs

Nhut-Minh Ho, Weng-Fai Wong
Department of Computer Science
National University of Singapore, Singapore
{minhhn,wongwf}@comp.nus.edu.sg

***Abstract*—With the growing importance of deep learning and energy-saving approximate computing, half precision floating point arithmetic (FP16) is fast gaining popularity. Nvidia's recent Pascal architecture was the first GPU that offered FP16 support. However, when actual products were shipped, programmers soon realized that a naïve replacement of single precision (FP32) code with half precision led to disappointing performance results, even if they are willing to tolerate the increase in error precision reduction brings. In this paper, we developed an automated conversion framework to help users migrate their CUDA code to better exploit Pascal's half precision capability. Using our tools and techniques, we successfully convert many benchmarks from single precision arithmetic to half precision equivalent, and achieved significant speedup improvement in many cases. In the best case, a $3\times$ speedup over the FP32 version was achieved. We shall also discuss some new issues and opportunities that the Pascal GPUs brought.**

I. Introduction

Nvidia recently introduced native *half precision* floating point support (FP16) into their Pascal GPUs[1]. This was mainly motivated by the possibility that this will speed up data intensive and error tolerant applications in GPUs. Nvidia GPUs support half precision as *storage* format starting from CUDA 7.5. Applications such as deep learning training and inference that can take advantage of FP16 as a storage format enjoyed a reduction in the data transfer time from the host (CPU side) to device (GPU side). However, the half precision floating point operations in the older architectures had to be promoted to `float` (FP32) inside the floating point compute units as they did not support half precision arithmetic in hardware. Rather than introducing new FP16 arithmetic units, Nvidia modified the single precision floating point unit (FPU) inside the CUDA cores such that it can either perform a single `float` operation, *or two* FP16 operations. In this paper we use the term 'FPU' to refer to the unit inside CUDA cores that performs both FP16 and FP32 operations while the unit that performs double precision floating (FP64) point operations as the 'DPU'. The arithmetic and representations of these data types follows the IEEE 754 standard [1]. However, this cost-effective implementation meant that the throughput for FP16 operations can be up to twice that for FP32 operations – but only if we use `half2` datatype [2]. For optimal utilization of the FPU, FP16 data must be paired and packed into 32 bit registers as a new `half2` (FP16x2) datatype. In other words, Pascal's new FPU has the capability of executing one single precision floating point instruction, or one 2-way SIMD (Single instruction, multiple data) instruction of the type `half2` in the same amount of time. With this design, the performance advantage of FP16 is twofold: reduced data transfer times and parallel computation. The support for half precision in CUDA is summarized in Figure 1.

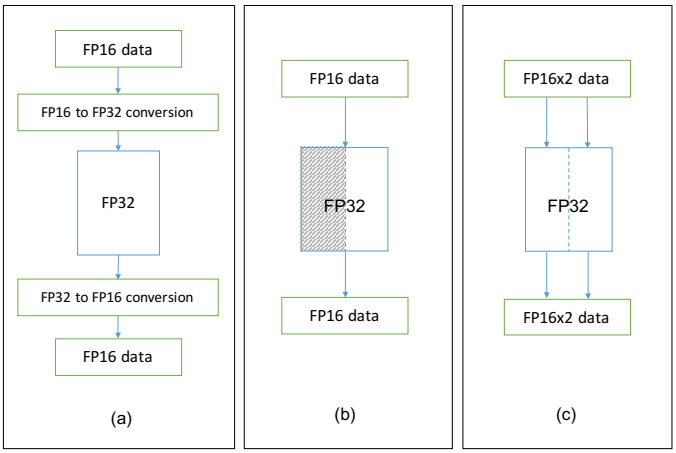

Fig. 1. Evolution of FPU's support for half precision. The blue boxes represent CUDA cores.
(a) Older architectures where FP16 is only a storage type, all computations are promoted to FP32.
(b) Pascal FPU, FP16 is natively supported with the same throughput as FP32.
(c) Pascal FPU, FPU16x2 can execute two FP16 instructions at a time.

There are two approaches of using half precision in Pascal:

- When the `half` datatype (Figure 1b) is used, the FPU takes the same amount of time to execute both FP16 and FP32 instructions. This approach is simple for code migration, but it fails to take full advantage of the new FPU. The performance gain is slightly better than when `half` is used merely as a storage datatype because the data is not implicitly converted to and from `float` during each operation as in Figure 1a. We will show in Section V that the majority of programs showed only marginal speedup, sometimes even slow down when using this approach.
- FP16 hardware is fully exploited using the `half2` datatype (Figure 1c) as the FPU runs in SIMD mode that can execute two identical operations on a `half2` value

[1]To be more exact, the Pascal GPU is not the first to introduce half precision computation natively in hardware. Jetson TX1 with Maxwell cores also provided support for FP16. However, Pascal is the first architecture that supports FP16 natively in the whole product line.

in register. In theory, the program can be twice as fast. Unfortunately, for the latter, users must rewrite their code and Nvidia provides minimal guidelines to doing this. The process of migrating one's code from using `float` to `half`, or `half2`, with the purpose of achieving better performance is tedious and error prone. The problem mainly comes from (a) half precision data type is not a primitive data type in both sides, GPU and CPU and (b) marshalling of data into `half2`.

The process of converting CUDA program from using `float` to `half2` is equivalent to vectorizing a program to a 2-way SIMD version. However, most vectorizing techniques are for CPU code, and targets the *innermost* loops, with no guarantee of always being successful. The structure of CUDA makes the situation worse. In this paper, we shall describe simple yet powerful techniques to deal with this code migration problem in CUDA by attempting to vectorize the *outermost* loop of CUDA threads. The method achieves a speedup of approximately $2\times$ compared to the FP32 versions when most of the code is amenable to our techniques. Interestingly, when we count in other empirical factors in hardwares and cache effects, the speedup can be over $2\times$ in some benchmarks. In fact, our techniques maybe the only simple way to relieve `nvcc` from vectorizing workload at the time being. To the best of our knowledge, we are the first to study the performance of half precision in GPU, and provide conversion solutions to enable the use of `half2` instructions in CUDA threads.

A. Related Work

Our work is related to vectorization. Since it is a new problem in the context of CUDA, we shall review some of the techniques for vectorizing code on other architectures, which, unfortunately, are not suitable for CUDA code. Vectorization is a well-known technique that is implemented in `gcc` and `icc` compilers [3], [4], [5] targeting SIMD instructions sets such as AVX-512, SSE. Apart from what is implemented in the popular compilers, there are advanced vectorization methods to deal with difficult-to-analyze code [6], [7], [8].

The large body of works on vectorization target sequential code with loops running on general CPU as well as some specialized architectures. They try to find steady-state access patterns in inner loops [9], [10], [11], [12] with some techniques applicable also to outer loops [13], [14], [12]. The techniques cannot be applied to CUDA code because they work mainly on loops in sequential programs, and CUDA is naturally parallelized in a single instruction, multiple thread manner. Some of the techniques were used to convert sequential code to CUDA code [15], [16]. However, in practice, most programmers choose to develop and optimize CUDA code manually. Given that the code is already written in CUDA, attempting to reuse such vectorization-based tools may end up complicating the workflow. For example, the CUDA code may have to be re-transformed back into CPU code, with the loops regenerated before such an existing CPU vectorization tool can proceed to regenerate the CUDA version that uses `half2`. Furthermore, with the vector size of 2, it may not be easy to amortize the cost of the scatter-gather operations needed for SIMDization. In our approach, we maximize performance gain by vectorizing the entire program, thereby avoiding the data marshalling overhead of transiting between vectorized and unvectorized portions of the code.

For performance analysis, there are works targeting CUDA which explore GPU architecture via microbenchmarking for older architectures [17], [18]. Part of our work uses microbenchmarking to pinpoint performance bottlenecks, then providing solutions for some of the problems that arise in the use of the Pascal GPU. With the introduction of `half2`, `nvcc` has been extended to optimize for the `half2` API. However, this requires `half2` code to be present in the first place, thus motivating our work.

The paper is organized as follows. Following the introduction, Section II will give details about the challenges of migrating FP32 code to FP16x2. Section III gives an overview of our conversion tool, while Section IV will describe other problems associated with the use of half precision in the Pascal architecture. Section V evaluates the effectiveness of our proposed techniques. This is followed by a conclusion.

II. Code migration to Pascal: the challenges

In this section, we will briefly introduce CUDA, and discuss the challenges of converting CUDA program from using FP32 to FP16 and FP16x2. Note that we assume the user is fully aware of the increase in error due to precision reduction, which falls outside the scope of this work.

A. CUDA programming model

CUDA has been used extensively for parallel program in the last decade. The fine-grain parallel threads executed in *single instruction, multiple thread* (SIMT) model of GPU achieving massive speedup over conventional CPU code [19]. When a program is executed in GPU, it will launch many threads that are grouped into a hierarchy of *warps*, *blocks* and *grids*. A warp is the smallest unit that GPU will issue instructions from. Inside a warp, every thread will execute identical sequence of instructions on different data. If there is a branch, at each branch some threads in a warp that has the branching condition satisfied will execute while the rest will be inactive. Multiple warps will form a block, in which they share the *streaming multiprocessors*' (SM) shared memory. Multiple blocks then form a grid which share the global memory of the GPU. The SIMT model is somewhat similar to SIMD with lesser restrictions on data location (where data can be scattered, and not strictly in vector form) and branching.

B. Pascal 2-way FP16 SIMD FPU

In the Pascal architecture, the FPU is capable of executing 2-way SIMD instruction of the `half2` (FP16x2) data type. Figure 2 shows the example for adding two values in `half2` using API from `cuda_fp16.h` provided starting from CUDA 7.5.

The API for the current half precision data type is not user friendly. Since it is not a primitive data type, every arithmetic operator must be explicitly called as in Figure 2. This reduces

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

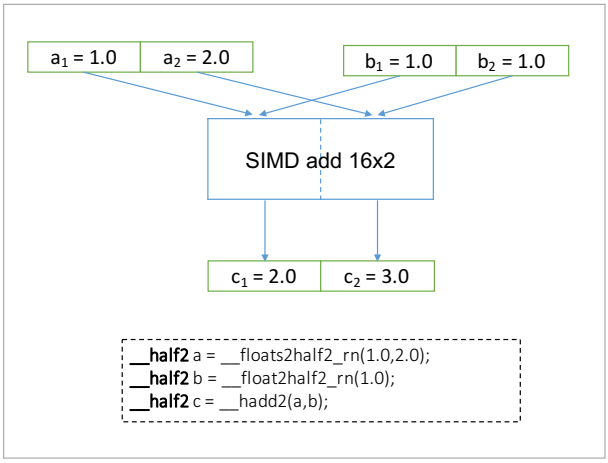

Fig. 2. An example of executing 2-way SIMD instruction in a CUDA thread

the readability of the converted code, and also makes the conversion prone to error. The conversion to use the `half` data type is fairly straightforward. It is not necessary to change the original CUDA code extensively other than the data type and conversion at host (CPU) side and device (GPU) side. In contrast, `half2` requires vectorizing the code so that each thread will execute 2-way SIMD code. To issue a `half2` instruction, both halves must be located in the same 32 bit register. This means that they must already be adjacent in memory before they are fetched into the register. Any data movement before issuing `half2` instructions will only reduce the potential performance gain from using `half2` instructions.

C. Threads pairing and array access rewriting

There are many techniques to convert code for SIMD execution. Most of them targeting loops and similar code sequences. The input data needs to be coalesced to enable SIMD instructions because of the high cost for data movement. Thus, many CPU techniques try to analyze memory access pattern of array inside loops so as to vectorize the code. Although these techniques can generalize well to any vector length, they cannot fully vectorize a full function if there are no loops in the function. In case of nested loop, they will try to process the innermost loop. These vectorizing techniques have limited efficacy on CUDA code since CUDA function body does not always have a loop inside.

CUDA code is executed by all threads with the only difference being their thread indexes at runtime. CUDA programs are mostly compatible to SIMD processor with vector width equals warp size (usually 32) with the exception that branches and memory operations are handled differently. The memory hierarchy is also different. Based on these observations, we develop a technique to integrate 2-way SIMD into the SIMT execution model of CUDA, thereby helping programmers to quickly convert their code for the `half2` datatype correctly, and with little effort.

We chose to target the implicit outermost loop that is always present in any CUDA program - the 'loop' over all the CUDA threads. This way, we can take advantage of any CUDA optimization that can be applied inside the code. By observing that most of CUDA programs are already optimized to use coalesced memory accessing, we fuse the two *adjacent* CUDA threads into one single thread. Using this method, the number of threads is reduced to half that of the original programs. All other things being equal, the new SIMD thread will take approximately the same amount of time to execute as the original thread. Therefore, a potential $2\times$ speedup can be expected.

The process is straight-forward when threads perform memory read and write into array element using their IDs as index. However, we will face problems when threads access multi-dimensional array, strided access, or when the array index is determined during runtime. The latter is hard to solve, and one can always find degenerate inputs to defeat any scheme. We therefore provide the solution for the first two access patterns which in any case accounts for the majority of CUDA programs.

1) Array access rewriting: On CPU side, the data is prepared in exactly the same way for both `half` and `half2` versions if the original data is a floating point array. In the case of an array of structures, we would group two adjacent data elements of the same field from two structures into a new structure with our helper functions.

On GPU side, suppose the thread index of CUDA code is T_x after translating from higher dimension into one dimension. The access pattern for an array M is $M[f(T_x)]$, where $f(T_x)$ is a function of thread's index. We can rewrite the array reference $f(T_x)$ to reflect our thread pairing decision. To fully exploit the `half2` type, there is one condition that the old threads before fusing into SIMD threads must satisfy, namely that $f(T_x)$ must be in form of $A \cdot T_x + B$ where A and B are constants. The following forms the basis of our conversion algorithm:

- If $A = 1$: we can do a straight-forward conversion where we can leave the array access as it is in the original version, and run the program with `half2` data type, and half number of threads.
- If $A = 0$: one element is accessed by all threads, so we need to convert that value which is in `half` type to `half2`. This is done automatically in our function overload header for operators between `half` and `half2`.
- If $A \neq 1$: here we have strided access. We can transpose M before copying it to the GPU so that $f(T_x)$ becomes: $T_x + A \cdot B$, which reduced to the first case.

For higher dimensional space, we can always translate the accessing address into one dimension. In fact, most CUDA programs use $M[\#\text{cols} \cdot i + j]$ to access the element $M[i][j]$, where #cols is the number of columns of the matrix. These techniques are applicable to most CUDA programs. All the program we tested on from Rodinia benchmarks use these access patterns for their floating point array. This is to be expected since other, especially more complicated, access patterns are not easily parallelizable in the CUDA execution model, and will not perform well as CUDA code.

III. Overview of the tool

We developed a tool built on LLVM Clang's LibTooling [20] that rewrites CUDA code. For now, it is one of the few choices available to parse CUDA code. The tool can work in two modes: (1) calling the original API provided by Nvidia for some specific functions, or (2) using our operator overload headers for the majority of operators and functions to improve readability. On the CPU side, data is converted into half precision using an open source half math library from [21].

On the GPU side, our tool is fully automatic for `half` type, and semi-automatic for `half2`. The workflow is divided into two phases. The first phase will run through the original CUDA code to identify all floating point variables, and list them in a *configuration file*. The second phase will read variables listed in the configuration file and change their types, calling the APIs of the floating point variables and function accordingly. This will allow users to recover some accuracy in the program by deleting variables that are critical to output accuracy in the configuration file. By default, however, we will convert all floating point variables from float or double to `half` and `half2` versions. Our tool outputs two versions for the original CUDA kernel, one uses `half` without SIMD support, and the other uses `half2`.

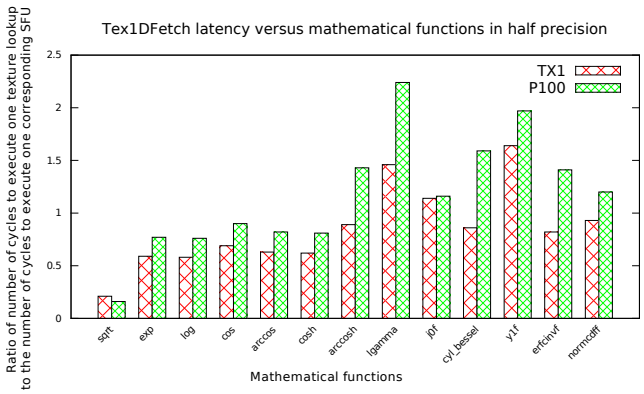

Fig. 3. Relative latency for accessing global memory randomly compared to calling half precision mathematics functions.

IV. New issues with using half precision

The use of `half2` data type leads to some issues when migrating application from old architecture using our thread pairing method, and other vectorizing techniques in general. First, by grouping threads together, the branching condition now will be more complex in each thread as it introduces a new 2-way SIMD thread divergence problem. Second, on applications that extensively use the mathematics library of CUDA, the Pascal GPU will promote these values to single precision before calling the Special Function Units (SFU) to compute the result. In other words, the SFU of Pascal supports neither half precision nor SIMD natively. Third, there is no support for half precision immediate values at the source code level and `ptx` levels. We had to add immediate value conversion support to our tool. Lastly, there will be address misalignment error when we convert an array of `half` into `half2` by the method of grouping adjacent values. This only happens in a few of the benchmarks, but it requires careful processing at the array boundaries. We shall now tackle each of these problems in turn.

A. Thread divergence

Thread divergence is a significant concern when writing a CUDA program that any CUDA developer will strive to avoid. Consider a program with two branches, each thread in a warp will have to wait the same amount of time as both branches will be executed. Our techniques of threads pairing will have doubled the branches. For each 2-way SIMD thread, there will be two similar conditional statements, each of them for one of the old thread before fusion. By replacing any value changed inside each condition with their masked variable, we can remove all the branching instructions. This masking technique is similar to masking in SIMD used in AVX or SSE instruction sets which is not available in CUDA.

1) Code rewrite solution: We rewrite the code to mitigate the thread divergence problem. The trick is to keep the code region that needs to be in branch as smallest as possible. All the code in branches will be converted to simple predicated instructions by `nvcc` instead of a branch instruction. To do so, we keep masked values for each branch of the conditional statements, and execute instruction of the SIMD version on the masked values. All the instructions in branches then can be converted to simple SIMD instructions, and the number of predicated instructions will be reduced to the smallest set needed to copy the value of masked values back to the original variables. Due to limited space, we present the code and explanation of this in [22]. This technique is extremely effective when the body of conditional statement contains read and write operations to the global memory since `nvcc` does not optimize half precision memory access as we will discuss in Section V.

B. SFU Performance

The *special functional unit* (SFU) is a hardware accelerator for the complex functions found in the CUDA mathematical function library [23]. The SFU of previous architecture operates on single precision and double precision values, with support for approximate version invoked by means of the `"--use_fast_math"` compiler flag. There is barely any mention about support of SFU in the Nvidia architecture for FP16. We had to investigate this issue for the Pascal GPUs.

Listing 1. Math functions in half precision

```
LDG.E.U16 R2, [R2]; //load from memory
F2F.F32.F16 R0, R2; //promoted to float
....
RRO.EX2 R4, R0; //range reduction
MUFU.EX2 R0, R4; //compute exp()
....
F2F.F16.F32 R0, R0; //convert back to half
....
STG.E.U16 [R2], R0; //store result to memory
```

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

Listing 1 shows the SASS (assembly code that is disassembled from binary file after all optimization done by `nvcc`) version of the device function to compute `exp(x)`. We can clearly see that the data will be promoted to `float` before a call is made to the SFU functions. For `half2` version of the code, the subroutine simply duplicates the `half` version, except that the epilogue and prologue will use SIMD instructions. Based on the above observation, we can conclude that the throughput of FP16 mathematics functions will be less than `float` with `"fast_math"` in general, and the latency of `half2` mathematics functions is nearly $2\times$ that of the `half` version.

1) Simple lookup table for complex math functions: The SFU can be a bottleneck for some applications if complex mathematics functions are called extensively. We mitigate this problem with the use of table lookup. Full table lookup is not possible for `float` functions because the table size will simply be too large. This is not the case for FP16. Due to the shorter length of 16 bits, table lookup is feasible and potentially yields better performance compared to the SFU hardware. In half precision, it is possible to store a table for each half precision math function (without any optimization) in the global memory: 2 bytes per entry $\times\ 2^{16} = 131$ kilobytes. We used texture lookup to implement this. To test the feasible of this technique, we measure the clock cycle latency of both using `"fast-math"` SFU, and a single texture lookup at a random index. For functions that are not currently available in CUDA, we use the fastest version of the equivalent in `float`, and simulate the same procedure that were used for available `half` math functions in CUDA. For texture read, we precompute the outputs of the function for all 2^{16} possible `half` numbers and make it available for lookup at runtime. The latency measured for table lookup in the worst case when the data is completely random with no locality or cache to help to improve the performance. Figure 3 estimates the relative speedup if we replace a function call to the SFU by a texture fetch from global memory on two different GPUs that support FP16 natively. While the speedup is less than 1 for simple math functions, table lookup can accelerate the computation of complex expressions involving a single variable. For example, the FP16 computation of $\exp(2x) + \sin(3x^2) - 3\cos(x)$ can be done using a single table lookup.

C. Floating point constant

Floating point constants are used widely in programs, and CUDA code is not an exception. The PTX instruction set allows users to load immediate value to floating point register (32 bits) easily. Furthermore, it also supports immediate values as operands for basic operators. In contrast, half operators currently do not enjoy such support. To mitigate this, we added function calls for direct data conversion when one of the operands is constant. When both of the operands are constant, we optimize the code by computing the results, and combining them into a single 32-bit value.

D. Address misalignment

Data types require address alignment because of caching, atomic and memory operations. `half2` datatype is aligned by 4 bytes, i.e., the starting address of any array must be divisible by 4. When we convert an array pointer of type `half` into `half2`, this alignment constraint must be met. A problem arises when the original program access an element starting at odd index in the array. The starting address if we reuse that array for `half2` computation will cause an address misalignment error. We solve this problem by making the first thread of the entire program do some extra work if the starting index is not aligned. In particular, it needs to compute on the first element of the array in FP16 without SIMD support. Then the rest of the code will be converted smoothly as the addresses are aligned. We implemented this solution in `gaussian`.

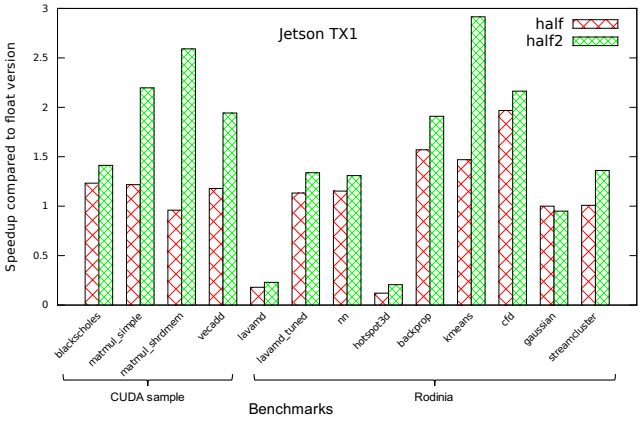

Fig. 4. Performance comparison on the Jetson TX1.

V. Experiments

We implemented our proposed techniques, and evaluated them on two different Nvidia GPUs that support half precision floating point arithmetic natively:

1) Jetson TX1 with compute capability 5.3. The TX1 fully supports `half2` instructions with $2\times$ throughput compared to `float`. Although TX1 belongs to Maxwell architecture, its FPUs supports `half2` as in Pascal.
2) Tesla P100 with compute capability 6.0, P100 is similar to TX1 for `half2` support.

Comparison using two different architectures that support half precision floating point arithmetic allows us to somewhat control for idiosyncrasies of these architectures that may be orthogonal to half precision.

The experiment results reported in this section are obtained from GPUs in which, all things being equal, `half2` operations in hardware can achieve $2\times$ speedup compared to `float` throughput. We do not consider GPUs that only support half precision as storage type, or the Pascal GP104 (GTX1080) where the hardware support for half precision is slow and incomplete, and mainly for compatibility purposes [24]. We ran our tool to convert 12 benchmarks with 4 of them from the

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

Nvidia sample code, and the rest being the Rodinia benchmark suite [25]. We use the original `makefiles` and compiler flags of these benchmarks, except adding `"-arch=sm_xx"` flag where `"xx"` is the compute capability of the target GPU. The execution times were measured by running the programs with data set provided five times, and then taking the average value. For benchmarks that required more memory than is available on our Jetson TX1 development board (which is 4GB), we reduced the problem size to fit the TX1's memory accordingly. The speedups reported are for the CUDA kernels execution time. We did not include the time for memory transfer between host and device since it is obvious that either `half` or `half2` will enjoy faster data transfer times compared to `float` when the size of the array is large enough. We also carefully checked the output values to ensure that the programs executed correctly. The amount of error that comes from using half precision floating point depends on the problem size at runtime and algorithms in the programs. For small problem sizes, the relative output error of `half` version is usually within 1% of the original. However, for larger problem sizes, and if the algorithm uses accumulators repeatedly, the output may completely different from the `float` version because of accumulated error, value overflow and underflow. This is an inherent problem of using lower precision. The impact of using half precision on the error of the output is an issue beyond the scope of this paper. We only used the smaller size inputs to check the correctness of our conversion technique.

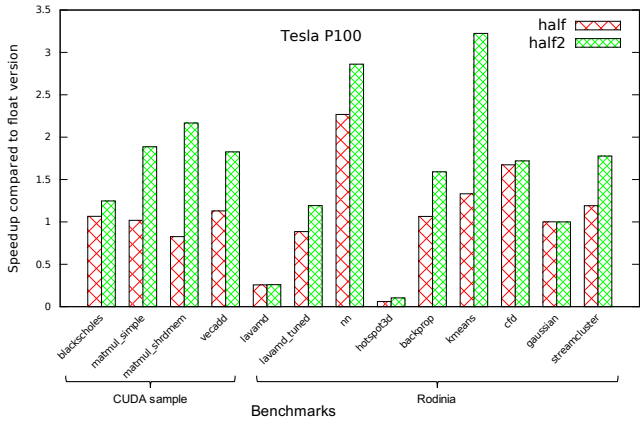

Fig. 5. Performance comparison on the Tesla P100.

A. Discussion

As we can see from Figures 4 and 5, most applications achieved more than 1.5× speedup when using the `half2` datatype via our code rewriting techniques. For the `half` type, when the code conversion is simple, the gain in speedup is rather marginal, or even worse than `float` in some benchmarks. Interestingly, some benchmarks can gain above 2× speedup. We attribute this to (1) the original version using non-optimized mathematics functions extensively, such as the case in `nn`, (2) the internal optimizations done by `nvcc`, and/or (3) cache effects and increased memory throughput. The latter was observed in `kmeans`, and verified by profiling using the Nvidia tool Nsight. The conversion produced a new program with a different memory requirement, cache usage pattern, and instruction schedule that nevertheless produced the compatible results.

Inspecting the `ptx` and `cubin` code generated, we found that `nvcc` optimizes memory access well by introducing temporary variables for accessing and writing to the same location in memory over several loops inside each thread. The same optimization is not available to half precision instructions. This is the cause of the slowdown observed in `lavamd`. We improve memory access in `lavamd` by rewriting (by hand) the code using the `nvcc` optimization technique, and the speedup became significant in `lavamd_tuned`.

In `streamcluster`, both the `half` and `half2` versions took much longer to complete because the algorithm converges on a value that needs to be correct by more than the third digit after the decimal point. Both versions were unable to reach this condition, and had to stop after being timed out. For the sake of comparison, we relaxed the convergence condition in such a way that all versions of the code converge, and took the same number of iterations to do so.

Not all applications will benefit from full `half2` support. These include:

- Applications that have irregular access pattern or runtime index reference that require marshalling data at runtime for `half2` execution.
- Applications that converge on some amount of error (e.g. `streamcluster`), or in other ways are very sensitive to the potential error in the computation. Some algorithmic adjustments, such as tweaking the convergence condition, may alleviate the situation.

Fortunately, these are the minority of CUDA applications. The former, for example, would not perform well in CUDA using any precision.

VI. Conclusion

Support for half precision floating point arithmetic in Nvidia's Pascal paves the way for further research and application in error tolerant and data intensive algorithms on GPUs. However, naïve replacement of FP32 with FP16 will not yield good performance. To exploit FP16x2 well, developers must significantly rewrite the code. To ease this burden, we proposed techniques to automate the conversion process. The conversion takes into consideration the structure of CUDA and the entirely new demands of the `half2` datatype. While our tool may not be able to always convert any FP32 code to FP16x2, it is usable on many benchmarks, including those from Rodina. In the best case, a 3× speedup was achieved. Developers must still make the decision of whether or not to trade accuracy for performance by using FP16 instead of FP32 or FP64. Our tool will make the conversion task effective and easy if they decide to do so. But even without the tool, we believe the insights and techniques on using `half2` we described in this paper will enable them to rewrite their code more effectively. The tool is available at [26].

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

REFERENCES

[1] IEEE Standards Committee, "754-2008 IEEE standard for floating-point arithmetic," *IEEE Computer Society Std*, vol. 2008, 2008.

[2] "NVIDIA Tesla P100 whitepaper," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, accessed: 2017-05-10.

[3] "Auto-vectorization in GCC," https://gcc.gnu.org/projects/tree-ssa/vectorization.html, accessed: 2017-05-10.

[4] "A guide to vectorization with Intel C++ compilers," https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf, accessed: 2017-05-10.

[5] C. García, R. Lario, M. Prieto, L. Piñuel, and F. Tirado, "Vectorization of multigrid codes using SIMD ISA extensions," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, pp. 8–pp.

[6] H. Chang and W. Sung, "Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2008, pp. 167–176.

[7] S. Kim and H. Han, "Efficient SIMD code generation for irregular kernels," in *ACM Sigplan Notices*, vol. 47, no. 8. ACM, 2012, pp. 55–64.

[8] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "SIMD vectorization of histogram functions," in *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*. IEEE, 2007, pp. 174–179.

[9] A. E. Eichenberger, P. Wu, and K. O'brien, "Vectorization for SIMD architectures with alignment constraints," in *Acm Sigplan Notices*, vol. 39, no. 6. ACM, 2004, pp. 82–93.

[10] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for SIMD," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 132–143, 2006.

[11] N. Sreraman and R. Govindarajan, "A vectorizing compiler for multimedia extensions," *International Journal of Parallel Programming*, vol. 28, no. 4, pp. 363–400, 2000.

[12] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 2009, pp. 327–337.

[13] D. Nuzman and A. Zaks, "Outer-loop vectorization: revisited for short SIMD architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 2–11.

[14] R. Karrenberg, "Whole-function vectorization," in *Automatic SIMD Vectorization of SSA-based Control Flow Graphs*. Springer, 2015, pp. 85–125.

[15] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 54, 2013.

[16] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," in *International Conference on Compiler Construction*. Springer, 2010, pp. 244–263.

[17] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.

[18] A. Li, S. L. Song, M. Wijtvliet, A. Kumar, and H. Corporaal, "SFU-driven transparent approximation acceleration on GPUs," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 15.

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[20] "Clang LibTooling," https://clang.llvm.org/docs/LibTooling.html, accessed: 2017-05-10.

[21] "IEEE 754-based half-precision floating point library," http://half.sourceforge.net/half_8hpp.html, accessed: 2017-05-10.

[22] "Gist.github link," https://gist.github.com/minhhn2910/d76dbb67c02bb2c6ad3b5819bb71dd8f.

[23] S. F. Oberman and M. Y. Siu, "A high-performance area-efficient multifunction interpolator," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*. IEEE, 2005, pp. 272–279.

[24] "FP16 throughput on GP104," http://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/5, accessed: 2017-05-10.

[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 44–54.

[26] "Github link," https://github.com/minhhn2910/cuda-half2.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE