Software-based Branch Predication for AMD GPUs

Ryan Taylor
University of Delaware
Newark, DE 19716
rtaylore@udel.edu

Xiaoming Li
University of Delaware
Newark, DE 19716
xli@udel.edu

ABSTRACT

Branch predication is a program transformation technique that combines instructions of multiple branches of an if statement into a straight-line sequence and associates each instruction of the sequence with a predicate. The branch predication improves the execution of branch statements on processors that support predicated execution of instruction, e.g., Intel IA-64, because such transformation improves the instruction scheduling and might help cache performance. This paper proposes a novel software-based branch predication technique for GPU. The main motivation is that branch instructions can easily become a performance bottleneck for a GPU program because of the cost of branch instructions compared to ALU instructions and the possibility of low ALU utilization due to separation of ALU instructions within control flow blocks. Due to the SIMD nature and massive multi-threading architecture of the GPU, branching can be costly if more than one path is taken by a set of concurrent threads in a kernel. In this paper we reveal that branch predication can enable instruction packing, a VLIW-like GPU feature that is designed to increase the parallel execution of independent instructions, and can also decrease the number of control flow instructions thereby improving the performance of GPU kernels with both single and multiple branch paths. The key of our novel branch predication technique is a set of transformation rules that takes into consideration the specialties of the GPU architecture and implements software-based predicated execution of instruction on the GPU with little to no overhead. Furthermore, we identify architectural and program factors that affect the effectiveness of our technique and build a benefit analysis model for the transformation. The implementation of our technique on synthetic benchmarks and real-world application proves its effectiveness.

1. INTRODUCTION

The Graphic Processing Unit (GPU) has became a major force in the high performance computing area. Its rapid increase in popularity as the platform for computing intensive applications can be generally attributed to its massive parallel processing architecture and the introduction of general purpose program frameworks for the GPU. Currently there are two major GPU vendors, NVIDIA and AMD, that both have GPU chips supporting general purpose programming. On the architecture side, NVIDIA's latest GTX 200 series GPU has up to 240 cores per chip, and AMD's current Raedon 5000 series provides up to 1600 cores per chip. The huge number of processing cores is the source for the impressive raw performance. Although NVIDIA and AMD have developed two very different approaches for general purpose computing on their GPU's, their programming frameworks still share similarities in the organization and the execution model of threads. Programs written for one kind of GPU cannot directly run on the other kind, however, the optimizations of program for the two architectures frequently follow similar guidelines. This paper focuses on the program optimization technique for AMD's GPU.

GPU hardware provides great potential performance. However, realizing the full performance potential of a GPU program is a real challenge. The main difficulty comes from that the GPU architecture has very different memory organizations and the execution of threads on the GPU is controlled by program control structure in a different way than how multi-thread programs run on CPU. For the first, the GPU has a memory organization that cannot find an easy mapping from the traditional CPU memory hierarchy based on cache/memory concepts. Both NVIDIA's and AMD's GPU has a global memory that is not cached and slow, and a separate faster memory that is software controlled. A GPU program must employ explicit memory management to achieve efficient usage of GPU memory. For the second factor, both GPU families organize and execute threads of a program in Single-Instruction-Multiple-Thread (SIMT) style. SIMT is similar to SIMD in that a bunch of threads must execute the same instruction at the same time. When the bunch of threads hit a control statement such as the *if* statement, the threads can diverge, though threads that do not take a branch must still step through instructions of the branch taken by other threads. Such a bunch of threads is called a *thread block* for NVIDIA GPU and a *wavefront* for AMD GPU. From a very high-level point of view, to optimize a program for the SIMT thread execution model, the unnecessary overhead of thread divergence must be minimized.

Prior works have extensively shown how to improve the memory performance of GPU programs. In particular global memory coalescing and reducing shared memory bank conflict have been addressed in application-specific environments such as image processing and computational flow dynamics, and for general programs such as [6]. However, work that addresses improving the efficiency of the SIMT execution of threads on GPU is relatively few. [5] discusses the employment of loop-unrolling to reduce the number of branch statements. Siegel et.al. [3] proposes a program transformation technique for NVIDIA GPU that splits GPU program segment with branch statement into multiple kernels for different branches so that the divergence in the execution of multiple threads can be reduced and hence the overall performance improved. In this paper, we identify a unique instruction scheduling feature called "instruction packing" on AMD GPU that can be exploited to reduce the overhead branch statement, model the program characteristics that are most important for the AMD GPU when considering the effects of branching and branch divergence on performance, and develop a software-based predication technique to enable the generation of the "packed" instructions in an AMD GPU program. Furthermore, this technique also reduces control flow instructions and therefore the latency associated with them.

Predication for a processor is not a new concept and is a common feature found in processors such as Intel IA-64. Predication can improve program performance because it basically transforms control flow into data flow, and therefore can improve instruction scheduling, reduce control flow instructions and possibly help the cache behavior of program. A good overview of the architectural support of predication in IA-64 can be found in [4]. The basic principles of compiling for the predication mechanism in IA-64 is described in detail in [2]. Furthermore, compilation techniques that can generate code for processors with predicated instruction execution have been extensively studied. As an example, August et.al. [1] proposed the basic program optimizing framework that balances the control flow of a program and the potential benefit of transforming part of the control flow into predicated execution. Recently, Smith et.al. [7] developed predication technology for dataflow computer architectures. Predication, in hardware, is found both in AMD GPUs and Nvidia GPUs; however, instructions are predicated by the compiler, require hardware support and the methodology is not clearly exposed. This technique requires no hardware support for predication and allows the user direct control over the instructions that are predicated.

A key insight of this paper is that by using a purely software-based approach to the predicated execution of instructions in branches, the special instruction scheduling and execution feature of the AMD GPU can be exploited to greatly reduce the adverse effect of a branch statement on the SIMT execution of a multi-thread GPU program. More specifically, this paper presents a software based predication technique to increase performance on AMD GPUs. The main contribution of this paper is a novel technique that is designed to both reduce control flow in a kernel, thereby reducing the number of control flow instructions and reducing the overhead associated with clause switching, to allow for the possibility of increased VLIW-like instruction packing. Our technique does not have a window size and so can be applied to both branches with a large number of instructions and to kernels with multiple branches. Also, our technique, in some cases, offers the possibility of reduced register pressure and can, in some cases, offer the possibility to reduce fetches.

In the remaining part of this paper, we first describe the background of the AMD GPU hardware and the AMD Stream SDK and the motivation of this software based predication technique. Next, we describe how this software based predication technique differs from the current predication techniques used for the GPU, describe the source to source transformation and how it effects the produced assembly and discuss the rules governing this technique. Furthermore, we describe why this technique is independent of the API being used (OpenCL, Brook+ or CAL/IL) and how the optimization is embedded in the hardware and not tied to a particular API. Finally, we show performance results for both a real world application and synthetic benchmarks.

2. AMD GPU

2.1 Hardware

The current AMD GPUs have a large number of ALUs, texture fetch units and a large register file. The speeds listed in Table 1 were obtained from the information listed in AMD's Catalyst Control Center. For the purpose of this paper the RV770 was tested using a 4870 video card and the RV870 was tested using a 5870 video card.

GPU	SPs	Texture Units	SIMD Engines
RV770	800	40	10
RV870	1600	80	20

GPU	Core Clock	Mem Clock	Mem Type
RV770	750Mhz	900Mhz	DDR5
RV870	850Mhz	1200Mhz	DDR5

Table 1: GPU Hardware Features

The RV770 for example has 800 ALUs, 40 texture fetch units and register file size of 16k(256x64) 128-bit wide registers. The RV770 consists of 10 SIMD engines, each having 16 * 5-wide VLIW (stream cores) stream processors and 4 texture fetch units (this is true for all of the current AMD GPU generations listed above). There are 4 general stream cores which can execute basic ALU operations and 1 transcendental stream core which is capable of executing both basic and transcendental operations. The texture fetch units are capable of fetching up to 128-bits each.

The AMD GPU streaming model consists of a group of threads called a wavefront running on a SIMD engine. The wavefront is split into quads which are groups of 2x2 threads, each quad executing on a thread processor. For example, the RV770's wavefront has 64 threads (as does the RV870) and each quad executes on one of the 16 thread processors/SIMD engine, refer to Figure 1. Each thread in a quad is interleaved over the thread processor to help hide latency. Depending on resource usage, multiple wavefronts can be running on one SIMD in parallel, each also interleaved to help hide latency. In addition, each thread processor has an odd and even slot such that one wavefront is assigned to run in each slot. If there is only one wavefront only half the thread processor is used. If there are two wavefronts then the entire thread processor is used. This is important because there exists "temporary clause registers" which are taken from the global purpose register pool for each slot (a maximum of two per slot). ALU and texture fetch instructions are grouped into clauses called ALU and TEX clauses respectively and the temporary clause registers are only live inside these clauses, they do not hold their value across clauses. Wavefronts hide latency by switching between these clauses when a stall occurs as shown in Figure 2. In the example ISA, TEX, ALU and EXP_DONE are all clauses. Under each clause there are only instructions of that clause type. For example, the TEX clause has texture sampling instructions while the ALU clause has ALU instructions. The ALU instructions are packed together in a VLIW instruction. In this example, this code only used the x, y, z, and w cores; however, use of the t core can also be packed in with the same VLIW instruction (instructions scheduled to run on the cores of a thread processor within the same cycles), also called a bundle. In this example ISA code you can also see the use of both the clause temporary registers (named T0 and T1) and the previous vector register (named PVx). The underline means that the result is going into the previous vector register to be used in the following instruction. In this code there are three inputs and one output and there are three global purpose registers used (named Rx).

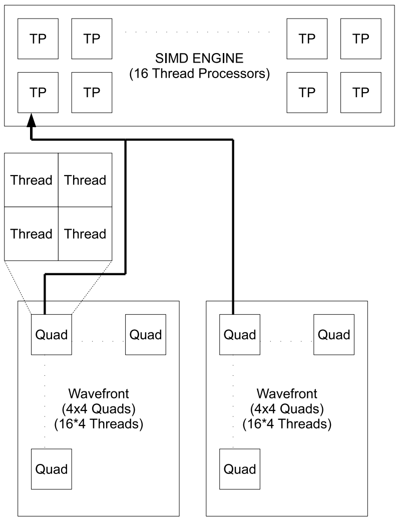

Figure 1: Thread Organization

```
; ———————  Disassembly ——————————————
00 TEX: ADDR(128) CNT(8) VALID_PIX
   0  SAMPLE R1, R0.xyxx, t0, s0
UNNORM(XYZW)
   1  SAMPLE R2, R0.xyxx, t1, s0
UNNORM(XYZW)
   2  SAMPLE R3, R0.xyxx, t2, s0
UNNORM(XYZW)
01 ALU: ADDR(32) CNT(88)
   8   x: ADD          ____,   R1.w,   R2.w
       y: ADD          ____,   R1.z,   R2.z
       z: ADD          ____,   R1.y,   R2.y
       w: ADD          ____,   R1.x,   R2.x
   9   x: ADD          ____,   R3.w,   PV1.x
       y: ADD          ____,   R3.z,   PV1.y
       z: ADD          ____,   R3.y,   PV1.z
       w: ADD          ____,   R3.x,   PV1.w
  14   x: ADD          T1.x,   T0.w,   PV2.x
       y: ADD          T1.y,   T0.z,   PV2.y
       z: ADD          T1.z,   T0.y,   PV2.z
       w: ADD          T1.w,   T0.x,   PV2.w
02 EXP_DONE: PIX0, R0
END_OF_PROGRAM
```

Figure 2: Example ISA

3. SOFTWARE-BASED PREDICATION

This software based branch predication technique requires a source to source transformation changing the code from control flow to data flow. The transformation takes advantage of the AMD GPU's high FLOPs performance to reduce the amount of control flow, changing control flow statements into ALU operations. Figure 3 shows the transformation for this technique which can be applied unconditionally to any type of branch, including nested brances with the addition of more predicated variables. This method takes advantage of the conditional set operations included in the AMD GPU ISA. The StreamSDK compiler turns the pre-transformed code into several clauses and tries to use hardware predication to optimize performance. On the other hand, the StreamSDK compiler turns the transformed code into ALU instructions since the conditional blocks have only one assignment statement, and can be done within one bundle, there is no need for the compiler to create additional clauses and use it's hardware predication. Furthermore, since the conditional blocks are not broken up into separate clauses (as in the pre-transformed code) the compiler is free to try and optimize the bundles (instruction packing). This transformation also allows for packing between the conditional block(s) and any code that is part of the kernel but is not included in the conditional block(s).
This transformation works for multiple branch paths in the same manner. The overhead associated with this method are the possible extra ALU operations required for setting the predicate variables (assuming they don't get packed in), adding two multiplies and an add and the register space needed to hold the predicate values (assuming new registers are needed). The technique does not significantly increase register pressure since registers are allocated per wavefront regardless of divergence. Like all ALU operations, the compiler attempts to pack the overhead ALU operations into stream cores in bundles not currently being used, as it does with the case of the synthetic benchmarks, thus hiding the associated overhead of the transformation. Since all the operations are being executed in the transformed code, the conditional set operation that sets the predicate values can occur anywhere in the code prior to setting the output, this allows for the reuse of the registers holding the predicate values in the case of multiple outputs. These factors show that there is very little direct overhead associated with this transformation.

This transformation uses software predication to reduce/eliminate control flow and therefore increase VLIW instruction packing and is most beneficial when the packing percentage of a kernel is low. This transformation can cause degradation when performed on a kernel with branches that already have a high ALU packing percentage and whose divergence is low.

3.1 Synthetic Benchmarks

The synthetic benchmarks attempt to express and isolate the use of the software-based branch predication technique for the explicit purpose of enabling instruction packing (increasing the number of operations in each VLIW instruction). All the operations in the benchmarks are vector-wide which depends on the float vector size being used. For each benchmark, different float vector sizes are used: float, float2, float3 and float4 to give the impact of the technique on performance for varying packing percentages. The data dependency between instructions within a control block also allows the synthetic benchmarks to show varying cases of possible instruction packing. For example, the float data type has a packing percentage of 20%, float2 40%, float3 60% and float4 80%. The high number of ALU operations used in each conditional block of the benchmarks insures

```
if (cond)
        ALU_OPs1;
        output = ALU_OPs1;
else
        ALU_OPs2;
        output = ALU_OPS2;
```

(a) Before Transformation

```
pred1 = 0, pred2 = 0;
if (cond)
        pred1 = 1;
else
        pred2 = 1;
ALU_OPS1;
temp = ALU_OPS1;
ALU_OPS2;
output = temp*pred1+ALU_OPS2*pred2;
```

(b) After Transformation

Figure 3: Transformation

that the kernels are ALU bound. The synthetic benchmarks are intended to show the performance increase of different percentages of ALU packing between two or more conditional blocks. For example, each synthetic benchmark is an attempt to mimic a real world kernel that utilizes the same packing percentage as the non-predicated synthetic benchmark (it is not an attempt to actually mimic a real world algorithm), thus showing the percent improvement that can be gained through this software based branch predication technique. The data dependency of the real world application is irrelevant compared to the packing percentage (low packing can be attributable to effects other than data dependency), it is used in the synthetic benchmarks only to gain control over the packing percentages.
Figures 4b and 4a show the code used for the benchmarks. The first two instructions depend on the inputs and every instruction after that is dependent on the previous two calculations, creating a high data dependency within each conditional block, giving a controlled test kernel. Figure 4 shows the resulting assembly code for the pre-transformed code and the transformed code for the float data type on the Radeon 4870. The pre-transformed assembly in Figure 4c uses hardware predication but does not effectively pack the instructions between the two conditional blocks. Instead, it creates two separate clauses, one for each conditional block thus not allowing for any instruction packing to occur. The transformed assembly in Figure 4d does not use the hardware predication, hides the overhead associated with the technique inside the first few bundles and then merges the second conditional block in with the first conditional block.

3.2 Lattice Boltzmann Method

The Lattice Boltzmann Method code for the AMD GPU was adapted from a C++ version written for a sequential processor. The code was then optimized for parallelism and for the AMD GPU architecture and StreamSDK. The code was optimized over several different versions to obtain very good performance results using several different methods such as: vectorization, register reduction, minimizing fetches through calculation, combining several functions to minimize kernel overhead and other small optimizations. The GPU code still used branching similar to the C++ code for large conditional blocks. There are four kernels being called in the LBM code and each kernel has some form of branching; however, not each kernel's conditional statement is based on the same parameters. For example, the *mcollid* and *stream* kernels' branching is based on the shape of the solid (the geometry) while *adv2* and *adv3* is based on the shape of the boundary, so more performance was gained from *mcollid* and *stream* then the *adv2* and *adv3* kernels.

4. EVALUATION

In this section we show the performance improvements for both our synthetic benchmarks and the real world application, the Lattice Boltzmann Method along with some other preliminary results. In particular, we present timing analysis, produced instruction types for both the non-predicated code and the predicated code and improvement percentage.

4.1 Synthetic Benchmarks

The synthetic benchmarks were ran over several thread divergences and for varying packing percentages, as shown in Figures 5 and 6. The divergence shown indicates how many threads divergerge on average based on a random distribution. The divergence is how many threads take the else path instead of the if path. The 'Packing Percent' shown in Table 2 is the pre-transformed packing percentages, this is also the case for all following tables and figures. Since the synthetic benchmark only utilizes two conditional blocks (if and else), the highest performance possible for both 20% and 40% is 100%. For example, if there was a similar benchmark with the same data dependency that had an if-else if-else code segment and each conditional block had the same dependency as the synthetic benchmarks and the float vector size was used, then optimally we could expect a near 400% increase in VLIW packing, from utilizing one stream core to utilizing four stream cores (the same number of conditional blocks). Theoretically, the maximum speedup for instruction packing is 5x, going from utilizing one stream core to utilizing all five stream cores. Conversely, if for the same example float4 was used instead of float, then the maximum speedup that could be obtained would be 20% because four of the five stream cores would be utilized in the non-predicated code, leaving only one stream core available for packing purposes. The CF instructions have also been reduced, thereby reducing any latency that exists from branching and switching clauses. The percent performance increases are given in Table 3.
Figure 4 shows the beginning assembly in both the predicated and non-predicated versions. In the non-predicated version the compiler attempts to use hardware predication to help reduce the branching effects; however, the hardware predication does not increase instruction packing and does not fully reduce the CF instructions. The software predicated version does not use the hardware predication and manages to both reduce the CF instructions and optimally utilize the ALU bundles.
Figures 5 and 6 show the run times of the varying benchmark tests. The run times for the non-predicated benchmarks for no divergence do not change because the number of ALU operations remains constant for each data type. The non-predicated branch divergence stays the same from 1/2 to 1/8 and then slowly begins to decrease from 1/16 to 1/128 and so the run times begin to decrease as the divergence reaches zero. The predicated times remain the same over the thread

```
kernel void step1(float a<>, float b<>, int cf<>, out float e<>)
{
        float t0, t1, t2, end;
        if (cf == 0)
        {
                t0 = a + b;
                t1 = t0 + a;
                t2 = t1 + t0;
                ...
                ...
                t0 = t2 + t1;
                t1 = t0 + t2;
                end = t1 + t0;
        }
        else
        {
                t0 = a - b;
                t1 = t0 - a;
                ...
                ...
                t0 = t2 - t1;
                t1 = t0 - t2;
                end = t1 - t0;
        }
        e = end;
}
```

(a) Pre-Transformed Synthetic Benchmark

```
kernel void step2(float a<>, float b<>, int cf<>, out float e<>)
{
        float t0, t1, t2, end, pred1, pred2;
        t0 = a + b;
        t1 = t0 + a;
        t2 = t1 + t0;
        ...
        ...
        t0 = t2 + t1;
        t1 = t0 + t2;
        end = t1 + t0;

        t0 = a - b;
        t1 = t0 - a;
        t2 = t1 - t0;
        ...
        ...
        t0 = t2 - t1;
        t1 = t0 - t2;
        if (cf == 0)
                pred1 = 1.0f;
        else
                pred2 = 1.0f;
        e = (t1-t0)*pred2 + end*pred1;
}
```

(b) Transformed Synthetic Benchmark

```
; ---------- Disassembly ------------------------
00 TEX: ADDR(176) CNT(3) VALID_PIX
      0  SAMPLE R1.x___, R0.xyxx, t2, s0  UNNORM(XYZW)
      1  SAMPLE R2.x___, R0.xyxx, t0, s0  UNNORM(XYZW)
      2  SAMPLE R0.x___, R0.xyxx, t1, s0  UNNORM(XYZW)
01 ALU_PUSH_BEFORE: ADDR(32) CNT(2)
      3  x: SETE_INT      R1.x,  R1.x,   0.0f
      4  x: PREDNE_INT    ____,  R1.x,   0.0f UPDATE_PRED
02 ALU_ELSE_AFTER: ADDR(34) CNT(66)
      5  y: ADD           T0.y,  R2.x,   R0.x
      6  x: ADD           T0.x,  R2.x,   PV5.y
      7  w: ADD           T0.w,  T0.y,   PV6.x
      8  z: ADD           T0.z,  T0.x,   PV7.w
.....
.....
03 ALU_POP_AFTER: ADDR(100) CNT(66)
     71  y: ADD           T0.y,  R2.x, -R0.x
     72  x: ADD           T0.x, -R2.x,   PV71.y
     73  w: ADD           T0.w, -T0.y,   PV72.x
     74  z: ADD           T0.z, -T0.x,   PV73.w
     75  y: ADD           T0.y, -T0.w,   PV74.z
...
...
```

(c) Pre-Transformed Synthetic Benchmark Assembly

```
; ---------- Disassembly ------------------------
00 TEX: ADDR(176) CNT(3) VALID_PIX
      0  SAMPLE R2.x___, R0.xyxx, t0, s0  UNNORM(XYZW)
      1  SAMPLE R1.x___, R0.xyxx, t1, s0  UNNORM(XYZW)
      2  SAMPLE R0.x___, R0.xyxx, t2, s0  UNNORM(XYZW)
01 ALU: ADDR(32) CNT(121)
      3  y: ADD           T0.y,  R2.x, -R1.x
         z: SETE_INT      ____,  R0.x,   0.0f         VEC_201
         w: ADD           T0.w,  R2.x,   R1.x
         t: MOV           R3.y,  0.0f
      4  x: ADD           T0.x, -R2.x,   PV3.y
         y: CNDE_INT      R1.y,  PV3.z,  (0x3F800000, 1.0f).x,  0.0f
         z: ADD           T0.z,  R2.x,   PV3.w
         w: CNDE_INT      R1.w,  PV3.z,  0.0f,  (0x3F800000, 1.0f).x
      5  y: ADD           T0.y,  T0.w,   PV4.z
         w: ADD           T0.w, -T0.y,   PV4.x
      6  x: ADD           T0.x,  T0.z,   PV5.y
         z: ADD           T0.z, -T0.x,   PV5.w
      7  y: ADD           T0.y, -T0.w,   PV6.z
         w: ADD           T0.w,  T0.y,   PV6.x
.....
```

(d) Transformed Synthetic Benchmark Assembly

Figure 4: Synthetic Benchmark Code and Assembly Code

divergence (but vary from 20% to 80% for the same reason as above) because of the elimination of branch instructions.

	Instruction Type and Number		
Packing Percent	ALU	TEX	CF
20	135/68	3	6/4
40	135/68	3	8/5
60	135/83	3	8/6
80	134/109	3	9/7

Table 2: Non-Predicated/Predicated Instruction Count

	Packing Percent			
Divergence	20	40	60	80
No Divg	0/0	0/0	0/0	-22.5/-6.5
1/2 Threads	93.5/89.6	92/85	55.7/26.9	57.7/20.3
1/4 Threads	93.5/89.6	92/85	55.7/26.9	57.7/20.3
1/8 Threads	93.5/89.6	92/85	55.7/26.9	57.7/20.3
1/16 Threads	92.6/88.9	90.9/88.3	55/26.5	21.7/15
1/64 Threads	61.9/61	59.5/58	31/9.5	2.4/3.7
1/128 Threads	39.4/36.6	39.1/37.3	13.8/.3	-11/-6.5

Table 3: Synthetic Benchmark Performance Increase 4870/5870

4.2 Lattice Boltzmann Method

A two dimensional LBM solution was ported from CPU code to GPU code. The domain sizes listed are squared, so

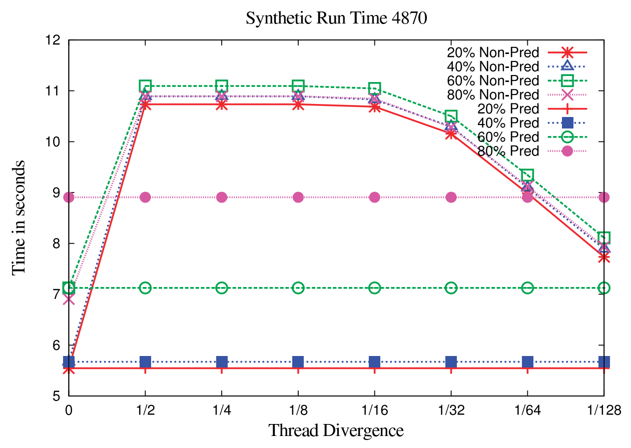

Figure 5: Synthetic Benchmark Run Time 4870

2048 = 2048x2048. The LBM code was executed for very course grain (water running over a perfectly square rock, low divergence) and for very fine grain (water running through sand, high divergence). The LBM GPU code consists of four kernels, each kernel has one if statement with no accompanying else statement (one conditional block). The transformation was applied to each if statement. Two of the four kernels did not benefit much from the transformation because the conditionals for those kernels are the boundary

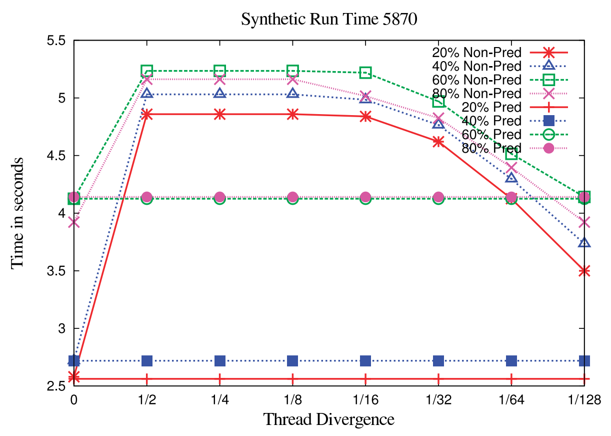

Figure 6: Synthetic Benchmark Run Time 5870

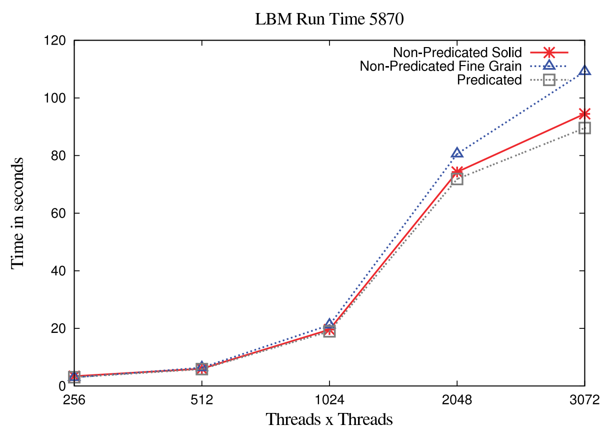

Figure 8: LBM Problem Size Run Time 5870

conditions. The other two kernels had good improvement after the transformation was applied; however, better improvement would be expected from a multiple path branch, since that would allow for a greater increase in instruction packing.
The Table 4 shows the percent increase in performance for each geometry over the given domain sizes for each GPU and Figures 7 and 8 show the execution times for the given domain sizes. In both kernels that saw a performance improvement there was an increase in instruction packing (approx. 10%), a decrease in control flow and a slight decrease in fetches. These last two improvements contribute to the increasing performance improvement with increasing thread count. Cache hit percentage is unknown for the kernels but could also contribute to the increase in performance improvement with the increasing thread count.

GPU	256	512	1024	2048	3072
Course Grain 4870	3.2	3.3	3.3	3.3	3.6
Fine Grain 4870	7.3	7.3	7.3	7.5	14.2
Course Grain 5870	2.4	3.4	3.3	3.3	5.3
Fine Grain 5870	2.6	7.9	11.3	11.9	18

Table 4: Performance Increase for All Solid and Fine Grain Geometries (in Percent)

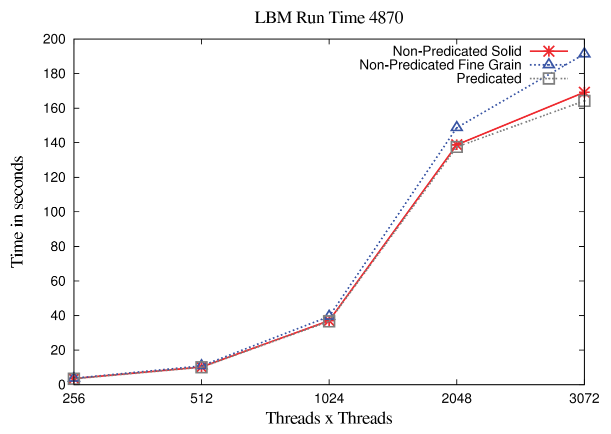

Figure 7: LBM Problem Size Run Time 4870

4.3 Other Results - Preliminary

This transformation was also performed on some OpenCL applications using both the OpenCL profiler and StreamKernelAnalzyer to collect the data. The first application was an OpenCL N-Queen solver with dimensions of 17x17 utilizing 32k threads. The N-Queen solver source code was non-vectorized and the transformation was only applied to one of the two application kernels. The packing percentage for that kernel went from 35.2% to 52% while the CF instruction count went from 22 to 9. The kernel execution time reduced from 74.3ms to 47.2ms, a substantial decrease in time.
The other applications include the DwtHaar1D, Eigenvalue and Bitonic Sort from the AMD OpenCL SDK samples. The DwtHaar1D saw an increase in packing percentage from 42.6% to 52.44% while both the Eigenvalue and Bitonic Sort samples saw a reduction in the number of average global writes per wavefront, the Eigenvalue went from 6 to 2 and the Bitonic Sort went from 4 to 2. These two latter examples show that the benefit from this transformation can also impact memory operations as well as increase packing percentage.

5. CONCLUSION

In this paper we describe a novel software-based branch predication technique for AMD GPUs. The main novelty of our approach is that it reveals several unique architectural features of GPU that can be enabled by "simulating" the predicated execution of branches, thereby improving program performance. In particular, this technique maximizes the 5-wide VLIW thread processor of the AMD GPU while reducing control flow instructions by combining branches through software based branch predication allowing for optimal utilization of the stream cores while also possibly reducing memory operations. We show synthetic benchmarks to prove the possible performance improvements from this technique. We also show performance improvement in a real world application, the Lattice Boltzmann Method along with several other sample kernels. This paper explains both the advantages and disadvantages of this technique as well as when and how to apply it, allowing for the possibility of this technique to be auto-generated.

6. REFERENCES

[1] D. I. August, W.-m. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 92–103, Washington, DC, USA, 1997. IEEE Computer Society.

[2] J. Bharadwaj, W. Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce. The intel ia-64 compiler code generator. *IEEE Micro*, 20(5):44–53, 2000.

[3] S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for gpgpu. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, pages 147–150, New York, NY, USA, 2009. ACM.

[4] C. Dulong. The ia-64 architecture at work. *Computer*, 31(7):24–32, 1998.

[5] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. W en mei. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[6] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM.

[7] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–102, Washington, DC, USA, 2006. IEEE Computer Society.