Functional Unit Balancing in Graphical Processing Units (GPUs)

Taylor Lloyd, Martin Ichilevici de Oliveira

December 16, 2016

1 Background

GPUs are increasingly being used to accelerate scientific computing applications. The parallel execution model exposed by GPUs allows a massive improvement in FLOPS/Watt when compared to CPUs. In fact, the upcoming supercomputer *Summit* [7] will use multiple GPUs per compute node, and expects to deliver over 200 PFLOPS of performance for only 10 megawatts. That represents a 5x performance boost over the previous top US supercomputer *Titan* [8] while using only 10% more power.

GPUs deliver this performance through significant architectural differences from traditional CPUs. In particular, they rely on a heirarchical parallelism model: Each GPU contains many streaming multiprocessors (SMs), which resemble highly parallel CPU cores. SMs execute instructions in units of *warps* of 32 threads, with the limitation that every thread within a warp must execute the same instruction on a given cycle. Modern SMs are capable of executing 2 independent instructions from 2 warps every cycle, for a maximum instruction throughput of 256 instructions per cycle per SM. The recently released Pascal GP100 by Nvidia contains 56 independent SMs, allowing a maximum throughput of 14336 instructions per cycle [6].

The theoretical limit is, however, rarely reached. In order to reach this maximum, a number of conditions must be met:

1. There needs to be sufficient parallelism to distribute warps to all SMs
2. Each SM needs 2 warps each with 2 independent instructions
3. Those warps need both instructions ready to execute (not waiting for data)
4. Those warps need to have no threads predicated off due to differing control flow
5. The SM needs to have sufficient functional units (FUs) to execute the instructions for all threads

If any of these conditions are not met, at least a portion of the SM will stall for that cycle, reducing throughput. A unique design feature of GPU architectures simplifies satisfying these conditions: Each SM keeps the registers loaded for all currently executing threads, allowing context-switches between warps with no penalty. Given enough threads, each SM can typically find a warp ready to at least partially execute on a given cycle. We concentrate on issue 5 in this work, attempting to better select instructions at compile-time to reduce stalls due to insufficient functional units.

Each SM in a GPU contains a variety of independent FU pipelines, and the exact count of each has been tuned over time to best match workloads traditionally offloaded to GPUs. Table 1 shows the width of pipelines for each category of instruction on an NVidia Kepler GPU in instructions per cycle. Most notably, no pipeline has sufficient width to handle 256 simultaneous instructions per cycle. Programs should therefore attempt to balance instructions to fit within the pipelines provided. We refer to attempts to overuse a particular functional unit pipeline as exhibiting *pipeline bias*.

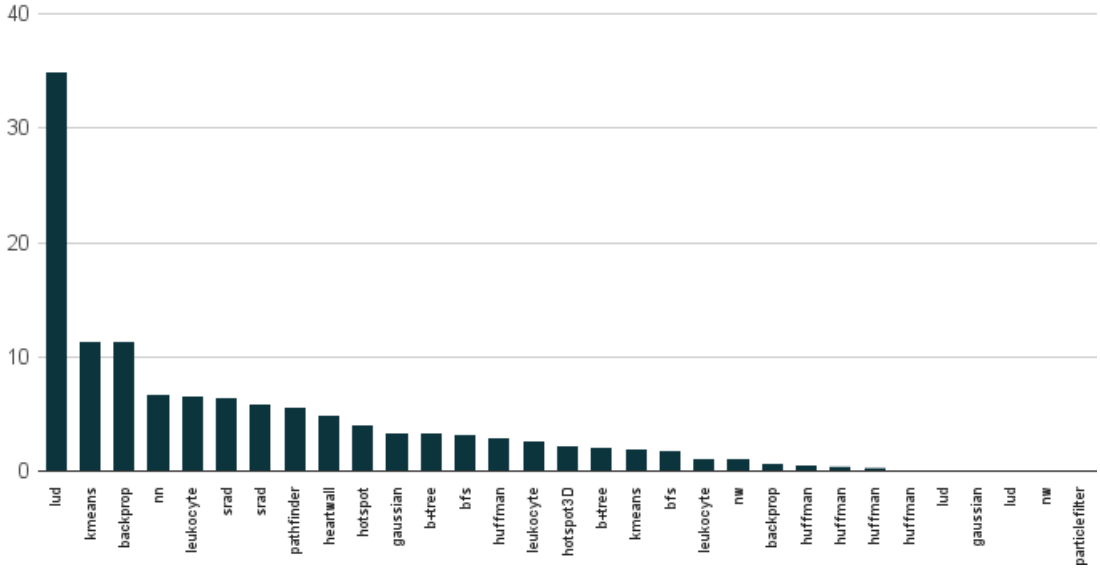

Figure 1: Cycles with at least partial stalling as a percentage of all executed cycles

32-bit floating-point add, multiply, multiply-add	192
64-bit floating-point add, multiply, multiply-add	64
32-bit floating-point reciprocal, reciprocal square root, base-2 log, base 2 exp, sine, cosine	32
32/64-bit integer add, subtract, minimum, maximum	160
32/64-bit integer multiply, multiply-add 32-bit sum of absolute difference, population count, count leading zeros	160
32-bit integer shift	64
32-bit integer bit reverse, bit field extract/insert	64
Logical Operations (AND, OR, XOR, NOT)	160
Warp Shuffle	32
Type Conversions from 8/16-bit integers to 32-bit float or int	128
Type Conversions to and from 64-bit types	32
All other type conversions	32

Table 1: Independent arithmetic functional pipelines on Kepler GPUs (Compute Capability 3.5) [2]

2 Exploration

To identify cases where pipeline bias causes performance issues, the Rodinia Heterogenous computing benchmark suite [4] was inspected by profiling. Execution profiles for 16 of the benchmarks were collected, with the remaining benchmarks excluded due to either profiling time greater than 3 hours, or an inability to compile and run with Clang. Figure 1 shows each kernel in the executed benchmarks, labeled with the benchmark containing the kernel. Some benchmarks contain multiple kernels, and are therefore present multiple times in the graph. Some kernels have pipeline stalls as much as 35% of total execution time, while others have none at all. The distribution suggests that pipeline stalls are highly program-dependent.

Next we investigated whether balancing pipeline bias could result in faster code, using Clang to compile CUDA programs. Figure 2 shows our CUDA kernel function, dramatically overutilizing 32-bit floating point operations. We generated the parallel thread execution (PTX) assembly for this kernel through Clang, resulting in the snippet shown in Figure 3, which is one of the 32 unrolled iterations of the loop. The ptx was then manually edited to

```
__global__ void kernel(float *arr) {
  double sum;
  float a,b,c,d,e;
  for(int i=0; i<64; i++) {
    a = arr[i];
    b = a*a;
    c = a+b;
    d = b*c;
    e = b+c;
    sum += (a + b + c + d + e);
  }
}
```

Figure 2: CUDA kernel overutilizing 32-bit floating point operations

```
ldu.global.f32  %f1, [%rd2];
mul.f32         %f2, %f1, %f1;
add.f32         %f3, %f1, %f2;
add.f32         %f4, %f2, %f3;
add.f32         %f5, %f3, %f3;
fma.rn.f32      %f6, %f2, %f3, %f5;
add.f32         %f7, %f4, %f6;
cvt.f64.f32     %fd4, %f7;
add.f64         %fd5, %fd19, %fd4;
```

Figure 3: Original PTX loop iteration

```
ldu.global.f32  %f1, [%rd2];
mul.f32         %f2, %f1, %f1;
add.f32         %f3, %f1, %f2;
add.f32         %f4, %f2, %f3;
add.f32         %f5, %f3, %f3;
fma.rn.f32      %f6, %f2, %f3, %f5;
cvt.f64.f32     %fd20, %f4;
cvt.f64.f32     %fd21, %f6;
add.f64         %fd4, %fd20, %fd21;
cvt.f64.f32     %fd4, %f7;
add.f64         %fd5, %fd19, %fd4;
```

Figure 4: Modified PTX loop iteration, with extra add.f64

produce Figure 4 by converting a 32-bit addition to a 64-bit addition, and the two executables were compared. We would expect to see a speedup in this scenario since more instructions can be executed in a given cycle when some use different FU pipelines. On a CPU the extra latency of the 64-bit operation would hurt performance, but the GPU in most cases can hide these latencies simply by executing other instructions until the 64-bit operation is complete.

Using the profiler, and 5 executions each, the original and modified kernels run in 14.3 us and 12.8 us respectively on average, for a speedup of 11.7%. In order to verify that this result was from the transformation, the instruction mixes of each application were profiled. The modified version used 2% fewer FP32 instructions, and 2% more FP64 instructions. Given that transformations seem to produce expected speedups under ideal conditions, and benchmarks appear to share these issues, it is a reasonable conclusion that reducing pipeline bias should improve performance, all else held equal.

3 Analysis

To identify pipeline bias, we first develop a mapping from LLVM instructions to the Functional Unit pipelines they utilize. For most instructions, this is a trivial 1-1 mapping, but there are a few exceptions to handle:

- **Fused Multiply-Add(FMA):** GPUs support fused multiplication and addition for both floating-point and integer operations. To identify cases where a FMA will be used, whenever mapping an addition operation we check that (1) it is used by a single instruction, and (2) that instruction is a multiplication. If both of those conditions are true, we consider the addition to use no functional units.
- **GetElementPtr(GEP):** LLVM defines an instruction that performs element lookups through a pointer, and takes a variable number of parameters to calculate the exact offset. We consider all GEP instructions to be equivalent to a 64-bit integer multiply-add.

- **NVidia Intrinsics:** Nvidia provides a number of specialized intrinsic functions to provide direct access to hardware functionality. These include access to threadIds, specialized conversion operations, and transcendental function accesses such as reciprocals, cosines, or logarithms. These are all approximated using the appropriate functional units.

The *pipeline usage* of an LLVM instruction is the result of the mapping operation defined above. The pipeline usage of a basic block is the sum of the pipeline usage of each instruction in that basic block.

To simplify the following analysis we make the observation that instructions executed within a loop are likely to execute much more often than instructions outside the same loop. We thus consider only instructions within innermost loops of a given loop nest when analyzing a kernel function, and analyze each loop nest in isolation.

To generate accurate pipeline usage for the innermost body of a loop, it is necessary to know how any conditional control-flow will evaluate. As this analysis is needed at compile time, that information is unavailable. The LLVM BasicBlockFrequency analysis, a compiler-provided heuristic measure, is used to weight the pipeline usage of each basic block within an innermost loop nest.

The *pipeline usage estimate* of an innermost loop body is a weighted sum of pipeline usages for each basic block b within the loop body, weighted by $\frac{BasicBlockFrequency(b)}{BasicBlockFrequency(entry)}$, where *entry* is the entry block of the loop body.

The pipeline usage estimate provides a summary of the behaviour of a particular application, which can be compared against the hardware's available pipelines. *Overuse rate* is a numeric metric which can be used to rank how well matched a pipeline usage estimate is to the pipelines available. Algorithm 1 shows how the overuse rate is calculated. It is a measure of fractional overuse for each pipeline, and is strictly greater than zero. A greater overuse rate indicates additional overutilized units, which indicates that more pipeline stalls can occur at runtime. Because the overuse rate can be calculated using only a pipeline usage estimate, the effect of various transformations can be measured by applying the transformation's effect on the pipeline usage estimate and recalculating the overuse rate.

Algorithm 1: OveruseRate

Input: *usage*: A map from FUs to their absolute usage
Input: *hw*: A map from FUs to their hardware availability count
Input: *maxIPC*: The maximum number of instructions that can be issued per cycle
Output: *overuse*: A floating-point value indicating how badly overused FUs may be

$usage_{total} = 0$;
for *fu in values(hw)* **do**
  $usage_{total} += usage_{fu}$;
end
$overuse = 0$;
for *fu in values(hw)* **do**
  if $usage_{fu}/usage_{total} > hw_{fu}/256$ **then**
    `/* Add the rate at which this fu is overused */`
    $overuse += \frac{usage_{fu}/usage_{total}}{hw_{fu}/maxIPC}$;
  end
end
return *overuse*

4 Transformations

Transformations are applied to each innermost loop body as the final LLVM transformation before conversion to PTX.

First, the pipeline usage estimate and overuse rate are obtained. Then, each instruction in the loop body is inspected to see if any transformation can be applied, and if so the effect of the transformation on overuse rate is calculated. The transformation that furthest reduces the overuse rate is applied, and the entire process is repeated until either no transformation can be applied, or no transformation reduces the overuse rate. Each currently implemented transformation is discussed in detail below.

4.1 Shift Left to Integer Multiplication

Shifting an integer n right by m is equivalent to multiplying n by 2^m. During compilation, two conditions must be met to allow for this transformation. The first is that n is an integer and the second is that m is a constant. The second condition is necessary because we want to be able to compute at compile time the equivalent multiplication operand, so that an extra instruction is not necessary. This transformation decreases the usage of `Shifts` by 1 and increases the usage of `IntMul` by 1.

4.2 Integer Multiplication to Shift Left

This is the reverse transformation of the transformation described in Section 4.1. It requires that one of the operands be an integer power of two (constant) and that the other be an integer. Given that the opimization described in this report is applied in the end of the compilation chain, it is unlikely that this mulitplication pattern will naturally appear, as it will have already been converted to a shift left in previous steps. However, it can be produced by a *Shift Left to Integer Multiplication* transformation.

4.3 Shift Right to Integer Divide

This transformation is analagous to *Shift Left to Integer Multiplication*. It converts a shift right by a constant integer m to a division by 2^m. However, the GPU does not have a functional unit capable of divisions and instead uses multiple other instructions to do the division. Hence, applying this transformation would mean adding several other instructions, what, in practice, means that it is very unlikely that this transformation would ever pay off.

4.4 FP32 to FP64

Every 32-bit floating point value is representable using 64-bit floating-point. Therefore, it is possible to use functional units that perform 64 floating point operations to perform 32 bit floating point operations without a loss of precision. The variables, however, must be converted to double precision format before the operation can be computed. To avoid adding extra instructions to perform the conversion, this transformation is restricted to FP32 operations whose operands are constants or the result of conversion operations. If these operands are not used elsewhere in the program, then they can be safely converted to 64 bits instead.

The rest of the program expects the resulting value to be a 32-bit float, so a conversion operation must follow the 64-bit operation for correct program behaviour. In the optimal case, the result of the binary operation is converted again in the original code, in which case it is possible to just fix the original cast to match the new source type. However, to allow greater applicability, this transformation also allows a cast to be introduced following the operation if necessary. Figure 5 shows the structure of the transformation being applied.

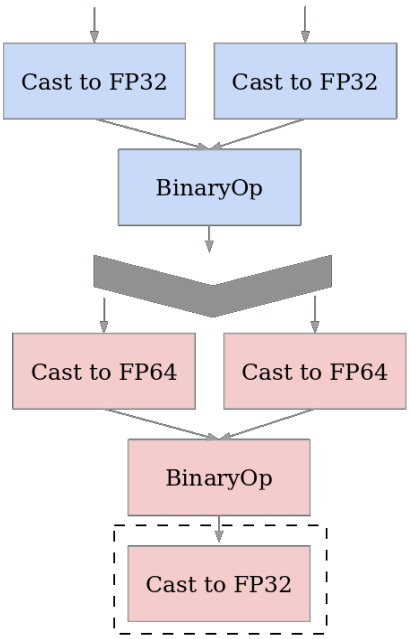

Figure 5: FP32 to FP64

5 Experimental Setup

Experiments were performed in a machine with a dual Intel Xeon E5-2630-v2 processor, Nvidia Tesla K20m GPU and 64GB of DDR3 RAM. We used Clang 4.0.0 (under active development) and Cuda compilation tools version 7.5.17.

Figure 6 shows the compilation model used. Notable, Clang itself only goes as far as generating PTX (Nvidia's pseudo-assembly language). To generate the actual device binary, it relies on Nvidia's proprietary tools `ptxas` and `fatbin`[1]. Our optimization runs after all other analysis and optimizations have been performed in the LLVM toolchain.

6 Experimental Results

We attempted to apply the transformations in the Rodinia benchmarks we had profiled. However, many kernels do not have loops and use a grid geometry to achieve parallelism. Furthermore, some kernels had loops with a constant trip count and were fully unrolled, so our analysis was unable to find any loop bodies on which to operate. The remaining kernels either did not have enough pipeline overutilization or they did not match any of the patterns required by the transformations. Therefore, we were unable to apply the transformation to any of the benchmarks we tested.

Another challenge we faced to apply the transformations is due to our analysis' position in the compilation flow. Consider for example Figure 7, that has a kernel function designed to overutilize the shift functional unit. It can benefit from the `ShlToMul` transformation and our optimization applies it, replacing a shift by a constant with a multiplication.

However, when `ptxas` is invoked, this is reverted back to a shift by performing strength reduction. This happens because `ptxas` is called with optimization level `-O3`. To overcome this limitation, we had to change the invocation to use `-O0` instead, since all optimization levels transform the multiplication back to a shift.

With this modification in the compilation cycle, the transformation was able to decrease the kernel's running time from (297.4 ± 3.8)

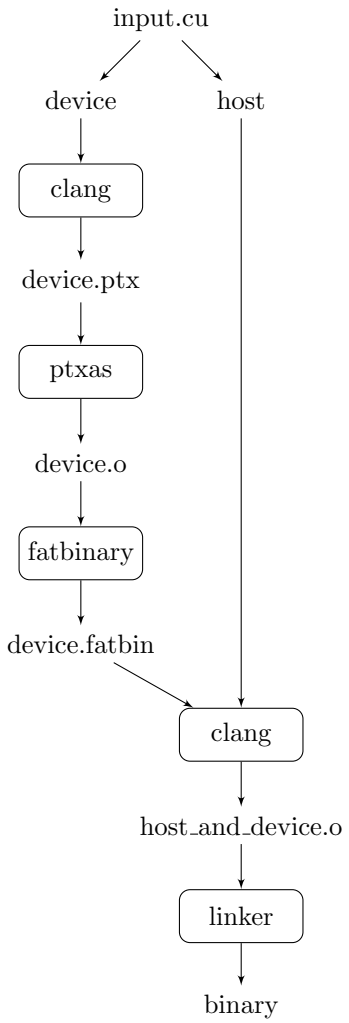

Figure 6: Clang GPU Compilation Flow

```
__global__ void kernel2(int a) {
  for(int i=0; i<1000; i++) {
    a = a<<1;
    a = 1<<a;
    a = a<<1;
  }
}
```

Figure 7: CUDA kernel overutilizing shift operations

us to (268.6 ± 2.4) us on average, representing a speedup of 10.7%. This experiment was repeated 15 times.

7 Future Work

Finding valid transformations and matching them to real programs proved to be a challenging task. To increase the applicability of this technique, some steps could be taken.

First, to explore the space of possible transformations, a promising approach would be to use superoptimizers[5, 3] to find equivalent instruction patterns that occur in real code. Our transformations were only rarely applicable, so having additional transformations could dramatically improve performance.

Additionally, we could apply our transformations to all regions between inter-thread synchronization points, instead of just innermost loops, as many programs we inspected either did not make use of loops or had those loops fully unrolled. This approach would require a more sophisticated functional unit analysis, as each instruction could have potentially very different real frequencies, and a poor estimate of pipeline usage could cause our transformations to harm performance dramatically.

Finally, to increase the accuracy of our functional unit estimator, we should implement this optimization as a `MachineFunction` pass. LLVM's Intermediate Representation (IR) is very high-level and has few instructions when compared to a machine language. One IR instruction can represent multiple instructions in the actual machine language (an IR `add` can be translated to a PTX floating point `add`, double precision floating point `add` or integer `add`, for example). Furthermore, some machine instructions are a combination of multiple IR instructions (a PTX `fma` is composed of an IR `add` and a `mul`). On a `MachineFunction` pass, the code has already been translated to a machine-dependent representation equivalent to the final PTX. Additionally, using a `MachineFunction` pass would mean that the optimization is in an even later step of the compilation process, thus reducing the opportunity for other compiler components (such as the instruction selector) to undo any transformations we perform.

8 Related Work

To the best of our knowledge, there has been no attempt to balance functional unit usage in scalar and vector processors, mainly because the cost of context switching means that using an instruction with greater latency will almost always reduce program performance. Superscalar processors, on the other hand, have a greater diversity of functional units and they rely on independent instructions that use different functional units and can be issued simultenously to achieve maximum performance. While some architectures determine which instructions to issue at runtime, an alternative is to recognize them during compilation. One prominent example is HP's Explicitly Parallel Instruction Computing (EPIC) [9], which Intel used as a basis for its Itanium architecture. EPIC's compiler reordered instructions to match functional units each cycle, but did not attempt to translate them to equivalent instructions.

Superoptimization is a term coined by Massalin [5] and it is the task of finding the optimal sequence of instructions to perform a computation. It finds and replaces a sequence of instructions with an equivalent one that has the lowest cost, as defined by a cost function (usually, number of instructions). Because the search space is very large, it relies heavily in pruning techniques [3]. Superoptimizations, however, do not target pipeline and functional unit balancing.

References

[1] Compiling CUDA with clang. `http://llvm.org/docs/CompileCudaWithLLVM.html#id12`. Accessed: 2016-12-13.

[2] CUDA C programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Accessed: 2016-09-23.

[3] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Sys-*

tems, ASPLOS XII, pages 394–403, New York, NY, USA, 2006. ACM.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[5] Henry Massalin. Superoptimizer: A look at the smallest program. In *Architectual Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, 1987.

[6] NVidia. Gp100 pascal whitepaper. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`. Accessed: 2016-12-11.

[7] Oak Ridge National Laboratory. Summit. `https://www.olcf.ornl.gov/summit/`. Accessed: 2016-11-27.

[8] Oak Ridge National Laboratory. Titan Cray XK7. `https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/`. Accessed: 2016-11-27.

[9] Michael Schlansker and Ramakrishna Rau. Epic: An architecture for instruction-level parallel processors. Technical report, HP, February 2000.