Spill Code Placement for SIMD Machines

Diogo Nunes Sampaio[1], Elie Gedeon[2], Fernando M. Q. Pereira[1] and Sylvain Collange[1]

Departamento de Ciência da Computação – UFMG – Brazil
{diogo,fernando,sylvain.collange}@dcc.ufmg.br,elie.gedeon@ens-lyon.fr

Abstract. The Single Instruction, Multiple Data (SIMD) execution model has been receiving renewed attention recently. This awareness stems from the rise of graphics processing units (GPUs) as a powerful alternative for parallel computing. Many compiler optimizations have been recently proposed for this hardware, but register allocation is a field yet to be explored. In this context, this paper describes a register spiller for SIMD machines that capitalizes on the opportunity to share identical data between threads. It provides two different benefits: first, it uses less memory, as more spilled values are shared among threads. Second, it improves the access times to spilled values. We have implemented our proposed allocator in the Ocelot open source compiler, and have been able to speedup the code produced by this framework by 21%. Although we have designed our algorithm on top of a linear scan register allocator, we claim that our ideas can be easily adapted to fit the necessities of other register allocators.

1 Introduction

The increasing programmability, allied to the decreasing costs of graphics processing units (GPUs), is boosting the interest of the industry and the academia in this hardware. Today it is possible to acquire, for a few hundred dollars GPUs with a thousand processing units on the same board. This possibility is bringing together academics, engineers and enthusiasts, who join efforts to develop new programming models that fit the subtleties of the graphics hardware. The compiler community is taking active part in such efforts. Each day novel analyses and code generation techniques that specifically target GPUs are designed and implemented. Examples of this new breed include back-end optimizations such as Branch Fusion [10], thread reallocation [29], iteration delaying [7] and branch distribution [17]. Nevertheless, register allocation, which is arguably the most important compiler optimization, has still to be revisited under the light of graphics processing units.

Register allocation is the problem of finding locations for the values manipulated by a program. These values can be stored either in registers, few but fast, or in memory, plenty but slow. Values mapped to memory are called *spills*. A good allocator keeps the most used values in registers. Register allocation was already an important issue when the first compilers where designed, sixty years ago [2]. Since then, this problem has been explored in a plethora of ways, and today an industrial-strength compiler is as good as a seasoned assembly programmer at assigning registers to variables. However, GPUs, with their Single Instruction, Multiple Data (SIMD) execution model, pose new challenges to traditional register allocators. By taking advantage of explicit data-level parallelism, GPUs provide about ten times the computational throughput of comparable CPUs [19]. They run tens of thousands of instances (or *threads*) of a program at the same time. Such massive parallelism causes intense register pressure, because the register bank is partitioned between all threads. For instance, the GeForce 8800 has 8,192 registers per multiprocessor. This number might seem large at first, but it must be shared with up to 768 threads, leaving each thread with at most 10 registers. It is our goal, in this paper, to describe a register allocator that explores the opportunity to share identical data between threads to relieve register pressure.

In this paper we propose a *Divergence Aware Spilling Strategy*. This algorithm is specifically tailored for SIMD machines. In such model we have many threads, also called *processing elements* (PEs), executing in lock-step. All these PEs see the same set of virtual variable names; however, these names are mapped into different physical locations. Some of these variables, which we call *uniform*, always hold the same value for all the threads at a given point during the program execution. Our register allocator is able to place this common data into fast-access locations that can be shared among many threads. When compared to a traditional allocator, the gains that we can obtain with our divergence aware design are remarkable. We have implemented the register allocator proposed in this paper in the Ocelot open source CUDA compiler [12], and have used it to compile 46 well-known benchmarks to a high-end GPU. The code that we produce outperforms the code produced by Ocelot's original allocator by almost 21%. Notice that we are not comparing against a straw-man: Ocelot is an industrial quality compiler, able to process the whole PTX instruction set, i.e., the intermediate format that NVIDIA uses to represent CUDA programs. The divergence aware capabilities of our allocator have been implemented as re-writing rules on top of Ocelot's allocator. In other words, both register allocators that we empirically evaluate use the same algorithm. Thus, we claim in this paper that most of the traditional register allocation algorithms used in compilers today can be easily adapted to be divergence aware.

2 Background

C for CUDA is a programming language that allows programmers to develop applications to NVIDIA's graphics processing units. This language has a syntax similar to standard C; however, its semantics is substantially different. This language follows the so called Single Instruction, Multiple Thread (SIMT) execution model [14, 15, 20, 21]. In this model, the same program is executed by many virtual threads. Each virtual thread is instantiated to a physical thread, and the maximum number of physical threads simultaneously in execution depends on the capacity of the parallel hardware. In order to keep the hardware cost low, GPUs resort to SIMD execution. Threads are bundled together into groups called *warps* in NVIDIA's jargon, or *wavefronts* in ATI's. Threads in a warp execute in lockstep, which allows them to share a common instruction control logic. As an example, the GeForce GTX 580 has 16 Streaming Multiprocessors, and each of them can run 48 warps of 32 threads. Thus, each warp might perform 32 instances of the same instruction in lockstep mode.

Regular applications, such as scalar vector multiplication, fare very well in GPUs, as we have the same operation being independently performed on different chunks of data.

However, divergences may happen in less regular applications when threads inside the same warp follow different paths after processing the same branch. The branching condition might be true to some threads, and false to others. Given that each warp has access to only one instruction at each time, in face of a divergence, some threads will have to wait, idly, while others execute. Hence, divergences may be a major source of performance degradation. As an example, Baghsorkhi *et al.* [3] have analytically showed that approximately one third of the execution time of the prefix scan benchmark [18], included in the CUDA software development kit (SDK), is lost due to divergences.

Divergence Analysis. A divergence analysis is a static program analysis that identifies variables that hold the same value for all the threads in the same warp. In this paper we will be working with a divergence analysis with affine constraints, which we have implemented previously [25]. This analysis binds each integer variable in the target program to an expression $a_1\mathtt{T}_{id}+a_2$, where the special variable $\mathtt{T}_{id}$ is the thread identifier, and a_1, a_2 are elements of a lattice C. C is the lattice formed by the set of integers $\mathbb{Z}$ augmented with a top element $\top$ and a bottom element $\bot$, plus a meet operator $\wedge$. We let $c_1 \wedge c_2 = \bot$ if $c_1 \neq c_2$, $c \wedge \top = \top \wedge c = c$, and $c \wedge c = c$. Similarly, we let $c + \bot = \bot + c = \bot$. Notice that C is the lattice normally used to implement constant propagation; hence, for a proof of monotonicity, see Aho *et al* [1, p.633-635]. We define A as the product lattice $C \times C$. If (a_1, a_2) are elements of A, we represent them using the notation $a_1\mathtt{T}_{id} + a_2$. We define the meet operator of A as follows:

$$(a_1\mathtt{T}_{id} + a_2) \ \wedge \ (a'_1\mathtt{T}_{id} + a'_2) = (a_1 \wedge a'_1)\mathtt{T}_{id} + (a_2 \wedge a'_2) \qquad (1)$$

A divergence analysis with affine constraints classifies the program variables in the following groups:

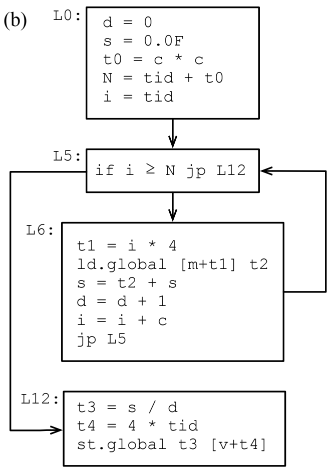
- **Constant:** every processing element sees the variable as a constant. Its abstract state is given by the expression $0\mathtt{T}_{id} + c, c \in \mathbb{Z}$.
- **Uniform:** the variable has the same value for all the processing elements, but this value is not constant along the execution of the program. Its abstract state is given by the expression $0\mathtt{T}_{id} + \bot$.
- **Constant affine:** the variable is an affine expression of the identifier of the processing element, and the coefficients of this expression are constants known at compilation time. Its abstract state is given by the expression $c\mathtt{T}_{id} + c', \{c, c'\} \subset \mathbb{Z}$.
- **Affine:** the variable is an affine expression of the identifier of the processing element, but the free coefficient is not known. Its abstract state is given by the expression $c\mathtt{T}_{id} + \bot, c \in \mathbb{Z}$.
- **Divergent:** the variable might have possibly different values for different threads, and these values cannot be reconstructed as an affine expression of the thread identifier. Its abstract state is given by the expression $\bot\mathtt{T}_{id} + \bot$.

Figure 1 illustrates how the divergence analysis is used. The kernel in Figure 1(a) averages the columns of a matrix `m`, placing the results in a vector `v`. Figure 1(b) shows this kernel in assembly format. We will be working in this representation henceforth. It is clear that all the threads that do useful work, e.g., that enter the gray area in 1(a) iterate the loop the same number of times. Some variables in this program always have

(a)

```
__global__ void avgColumn(float* m, float* v, int c) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid < c) {
    int d = 0;
    float s = 0.0F;
    int N = tid + c * c;
    for (int i = tid; i < N; i += c) {
      s += m[i];
      d += 1;
    }
    v[tid] = s / d;
  }
}
```

(b)



```
L0:  d = 0
     s = 0.0F
     t0 = c * c
     N = tid + t0
     i = tid

L5:  if i ≥ N jp L12

L6:  t1 = i * 4
     ld.global [m+t1] t2
     s = t2 + s
     d = d + 1
     i = i + c
     jp L5

L12: t3 = s / d
     t4 = 4 * tid
     st.global t3 [v+t4]
```

(c)

$[\texttt{c}] = 0 \times \texttt{tid} + \perp$	$[\texttt{t0}] = 0 \times \texttt{tid} + \perp$	$[\texttt{t1}] = 4 \times \texttt{tid} + \perp$
$[\texttt{d}] = 0 \times \texttt{tid} + \perp$	$[\texttt{N}] = 1 \times \texttt{tid} + \perp$	$[\texttt{t3}] = \perp \times \texttt{tid} + \perp$
$[\texttt{s}] = \perp \times \texttt{tid} + \perp$	$[\texttt{i}] = 1 \times \texttt{tid} + \perp$	$[\texttt{t4}] = 4 \times \texttt{tid} + 0$

Fig. 1. (a) A kernel, written in C for CUDA, that fills a vector `v` with the averages of the columns of matrix `m`. (b) The control flow graph (CFG) of the gray area in the kernel code. (c) The results of a divergence analysis for this kernel.

the same value for all the threads. The constant `c`, and the base addresses `m` and `v`, for instance. Furthermore, variable `d`, which is incremented once for each iteration is also uniform. Other variables, like `i`, do not have the same value for all the processing elements, but their values are functions of the thread identifier $\texttt{T}_{id}$; hence, these variables are classified as affine. The limit of the loop, `N`, is also an affine expression of $\texttt{T}_{id}$. Because `i` and `N` are both affine expression of $\texttt{T}_{id}$ with the same coefficient 1, their difference is uniform. Therefore, the divergence analysis can conclude that the loop is non-divergent; that is, all the threads that enter the loop iterate it the same number of times. Finally, there are variables that might have a completely different value for each processing element, such as `s`, the sum of each column, and `t3`, the final average which depends on `s`. Figure 1(c) summarizes the results of the divergence analysis.

3 Divergence Aware Register Allocation

In this section we explain why register allocation for GPUs differs from traditional register allocation. We also show how the divergence analysis can improve the results of register allocation. Finally, we close this section discussing some important design decisions that we chose to apply in our implementation.

3.1 Defining the register allocation problem for GPUs

Similar to traditional register allocation we are interested in finding storage area to the values produced during program execution. However, in the context of graphics processing units, we have different types of memory to consider. Thus, in the rest of this paper we assume that a value can be stored in one of the following locations:

- **Registers:** these are the fastest storage regions. A traditional GPU might have a very large number of registers, for instance, one streaming multiprocessor (SM) of a GTX 580 GPU has 32,768 registers. However, running 1536 threads at the same time, this SM can afford at most 21 registers (32768 / 1536) to each thread in order to achieve maximum hardware occupancy.
- **Shared memory:** this fast memory is addressable by each thread in flight, and usually is used as a scratchpad cache. It must be used carefully, to avoid common parallel hazards, such as data races. Henceforth we will assume that accessing data in the shared memory is less than 3 times slower than in registers [24].
- **Local memory:** this off-chip memory is private to each thread. Modern GPUs provide a cache to the local memory, which is as fast as the shared memory. We will assume that a cache miss is 100 times more expensive than a hit.
- **Global memory:** this memory is shared among all the threads in execution, and is located in the same chip area as the local memory. The global memory is also cached. We shall assume that it has the same access times as the local memory.

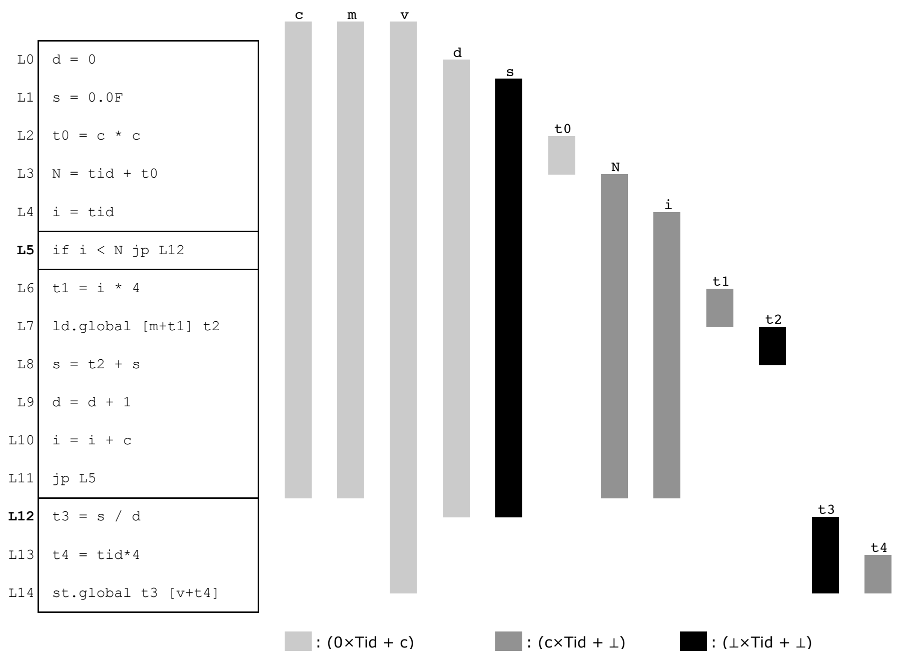
As we have seen, the local and the global memories might benefit from a cache, which uses the same access machinery as the shared memory. Usually this cache is small: the GTX 580 has 64KB of fast memory, out of which 48KB are given to the shared memory by default, and only 16KB are used as a cache. This cache area must be further divided between global and local memories. Equipped with these different notions of storage space, we define the divergence aware register allocation problem as follows:

Definition 1. *Given a set of variables V, plus a divergence analysis $D : V \mapsto A$, find a mapping $R : V \mapsto M$ that minimizes the costs paid to access these variables. The possible storage locations M are registers, shared memory, local memory and global memory. Two variables whose live ranges overlap must be given different positions if they are placed on the same location.*

Figure 2 shows the instance of the register allocation problem that we obtain from the program in Figure 1. We use bars to represent the *live ranges* of the variables. The live range of a variable is the collection of program points where that variable is *alive*. A variable v is *alive* at a program point p if v is used at a program point p' that is reachable from p on the control flow graph, and v is not redefined along this path. The colors of the bars represent the abstract state of the variables, as determined by the divergence analysis.

3.2 A quick glance at traditional register allocation

Figure 3 shows a possible allocation, as produced by a traditional algorithm, such as the one used in `nvcc`, NVIDIA's CUDA compiler. In this example we assume that a warp is formed by only two threads, and that each thread can use up to three general purpose registers. For simplicity we consider that the registers are type agnostic and might hold either integer or floating point values. Finally, we assume that the parameters



```
L0   d = 0
L1   s = 0.0F
L2   t0 = c * c
L3   N = tid + t0
L4   i = tid
L5   if i < N jp L12
L6   t1 = i * 4
L7   ld.global [m+t1] t2
L8   s = t2 + s
L9   d = d + 1
L10  i = i + c
L11  jp L5
L12  t3 = s / d
L13  t4 = tid*4
L14  st.global t3 [v+t4]
```

Fig. 2. An instance of the register allocation problem in graphics processing units.

of the kernel, variables c, m and v are already stored in the global memory. A quick inspection of Figure 2 reveals that only three registers are not enough to provide fast storage units to all the variables in this program. For instance, at label L8 we have eight overlapping live ranges. Therefore, some variables must be mapped to memory, in a process known as *spilling*. The variables that are assigned memory locations are called *spills*. Minimizing the number of spilled values is a well-known NP-complete problem [8, 22, 26]. Furthermore, minimizing the number of stores and loads in the target program is also NP-complete [13]. Thus, we must use some heuristics to solve register allocation.

There exist many algorithms to perform register allocation. In this paper we adopt an approach called *linear scan* [23], which is used in many industrial strength compilers. The linear scan algorithm sees register allocation as the problem of coloring an interval graph, which has polynomial time solution [16]. However, this correspondence is not perfect: live ranges might have holes, which the intervals in an interval graph do not have. Thus, the linear scan algorithm provides an approximation of the optimal solution to the register allocation problem. This algorithms starts by linearizing the control flow graph of the program, finding an arbitrary ordering of basic blocks, in such a way that each live range is seen as an interval. The left side of Figure 2 shows a possible linearization of the program given in Figure 1(b). After linearizing the program, the allocator scans the live ranges, from the beginning of the program towards the end, as-

	Program	register file						local						global		
		PE0			PE1			PE0			PE1					
		r0	r1	r2	r0	r1	r2	0	1	2	0	1	2	0	1	2
L0	d = 0													c	m	v
L1	**st.local d [1]**	d			d									c	m	v
L2	s = 0.0F	d			d				d			d		c	m	v
L3	**st.local s [0]**	d	s		d	s			d			d		c	m	v
L4	**ld.global [0] c**	d	s		d	s		s	d		s	d		c	m	v
L5	t0 = c * c	d	s	c	d	s	c	s	d		s	d		c	m	v
L6	N = tid + t0	t0	s	c	t0	s	c	s	d		s	d		c	m	v
L7	**st.local N [2]**	t0	s	N	t0	s	N	s	d		s	d		c	m	v
L8	i = tid	t0	s	N	t0	s	N	s	d	N	s	d	N	c	m	v
L9	**ld.local [2] N**	i	s	N	i	s	N	s	d	N	s	d	N	c	m	v
L10	if i < N jp L24	i	s	N	i	s	N	s	d	N	s	d	N	c	m	v
L11	t1 = i * 4	i	s	N	i	s	N	s	d	N	s	d	N	c	m	v
L12	**ld.global [1] m**	i	s	t1	i	s	t1	s	d	N	s	d	N	c	m	v
L13	ld.global [m+t1] t2	i	m	t1	i	m	t1	s	d	N	s	d	N	c	m	v
L14	**ld.local [0] s**	i	m	t2	i	m	t2	s	d	N	s	d	N	c	m	v
L15	s = t2 + s	i	s	t2	i	s	t2	s	d	N	s	d	N	c	m	v
L16	**st.local s [0]**	i	s	t2	i	s	t2	s	d	N	s	d	N	c	m	v
L17	**ld.local [1] d**	i	s	t2	i	s	t2	s	d	N	s	d	N	c	m	v
L18	d = d + 1	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
L19	**st.local d [1]**	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
L20	**ld.global [0] c**	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
L21	i = i + c	i	s	c	i	s	c	s	d	N	s	d	N	c	m	v
L22	jp L9	i	s	c	i	s	c	s	d	N	s	d	N	c	m	v
L23	**ld.local [1] d**	i	s	c	i	s	c	s	d	N	s	d	N	c	m	v
L24	t3 = s / d	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
L25	t4 = tid*4	t3	s	d	t3	s	d	s	d	N	s	d	N	c	m	v
L26	**ld.global [2] v**	t3	t4	d	t3	t4	d	s	d	N	s	d	N	c	m	v
L27	st.global t3 [v+t4]	t3	t4	v	t3	t4	v	s	d	N	s	d	N	c	m	v

Fig. 3. Traditional register allocation, with spilled values placed in local memory.

signing variables to registers in the process. If a spill must happen, then a conventional approach is to send to memory the variable with the furthest use from the spilling point onwards. This approach is known as *Belady's Heuristics*, as it has been first described by Belady in the context of virtual page management in operating systems [4].

Current register allocators for graphics processing units place spilled values in the local memory. Figure 3 illustrates this approach. In this example, variables `s`, `d` and `N` had to be spilled. Thus, each of these variables receive a slot in local memory. The spilled data must be replicated once for each processing element, as each of them has a private local memory area. Accessing data from the local memory is an expensive operation, because this region is off-chip. To mitigate this problem, modern GPUs provide a cache to the local and to the global memories. However, because the number of threads using the cache is large – in the order of thousands – and the cache itself is small, e.g., 16KBs, cache misses are common. In the next section we show that it is possible to improve this situation considerably, by taking the results of the divergence analysis into consideration.

	Program	register file						local		shared		global		
		PE0			PE1			PE0	PE1					
		r0	r1	r2	r0	r1	r2	0	0	0	1	0	1	2
L0	d = 0											c	m	v
L1	**st.shared d [0]**	d			d							c	m	v
L2	s = 0.0F	d			d					d		c	m	v
L3	**st.local s [0]**	d	s		d	s				d		c	m	v
L4	**ld.global [0] c**	d	s		d	s		s	s	d		c	m	v
L5	t0 = c * c	d	s	c	d	s	c	s	s	d		c	m	v
L6	N = tid + t0	t0	s	c	t0	s	c	s	s	d		c	m	v
L7	**st.shared t0 [1]**	t0	s	N	t0	s	N	s	s	d		c	m	v
L8	i = tid	t0	s	N	t0	s	N	s	s	d	t0	c	m	v
L9	**ld.shared [1] t0**	i	s	N	i	s	N	s	s	d	t0	c	m	v
L10	**N = tid + t0**	i	s	t0	i	s	t0	s	s	d	t0	c	m	v
L11	if i < N jp L24	i	s	N	i	s	N	s	s	d	t0	c	m	v
L12	t1 = i * 4	i	s	N	i	s	N	s	s	d	t0	c	m	v
L13	**ld.global [1] m**	i	s	t1	i	s	t1	s	s	d	t0	c	m	v
L14	ld.global [m+t1] t2	i	m	t1	i	m	t1	s	s	d	t0	c	m	v
L15	**ld.local [0] s**	i	m	t2	i	m	t2	s	s	d	t0	c	m	v
L16	s = t2 + s	i	s	t2	i	s	t2	s	s	d	t0	c	m	v
L17	**st.local s [0]**	i	s	t2	i	s	t2	s	s	d	t0	c	m	v
L18	**ld.shared [0] d**	i	s	t2	i	s	t2	s	s	d	t0	c	m	v
L19	d = d + 1	i	s	d	i	s	d	s	s	d	t0	c	m	v
L20	**st.shared d [0]**	i	s	d	i	s	d	s	s	d	t0	c	m	v
L21	**ld.global [0] c**	i	s	d	i	s	d	s	s	d	t0	c	m	v
L22	i = i + c	i	s	c	i	s	c	s	s	d	t0	c	m	v
L23	jp L9	i	s	c	i	s	c	s	s	d	t0	c	m	v
L24	**ld.shared [0] d**	i	s	c	i	s	c	s	s	d	t0	c	m	v
L25	t3 = s / d	i	s	d	i	s	d	s	s	d	t0	c	m	v
L26	t4 = tid*4	t3	s	d	t3	s	d	s	s	d	t0	c	m	v
L27	**ld.global [2] v**	t3	t4	d	t3	t4	d	s	s	d	t0	c	m	v
L28	st.global t3 [v+t4]	t3	t4	v	t3	t4	v	s	s	d	t0	c	m	v

Fig. 4. Register allocation with variable sharing.

3.3 Divergence Aware Spilling as a Set of Rewriting Rules

Figure 4 shows the code that we generate for the program in Figure 2. The most apparent departure from the allocation given in Figure 3 is the fact that we have moved to shared memory some information that was originally placed in local memory. Our divergence aware register allocator is basically a system of rewriting rules built on top of a host algorithm. We have identified four different ways to rewrite the code produced by the traditional allocator, given the information made available by the divergence analysis. These rules are described in Figure 5. In the rest of this section we will describe each of these rules, and, in the process, explain how we arrived at the allocation given in Figure 4.

Constant Propagation. The divergence analysis discussed in Section 2 marks, as a byproduct, some variables as constants. Thus, it enables us to do constant propagation, a well-known compiler optimization [28]. Indeed, as mentioned before, the lattice that underlies this analysis is the same structure that grounds constant propagation. Variables

(a) $[v] = 0 \times \texttt{tid} + c$

Original spill code:		Constant Propagation:
`st.local` v [@]	$\longrightarrow$	{}
`ld.local` [@] v	$\longrightarrow$	$v = c$

(b) $[v] = c \times \texttt{tid} + c'$

Original spill code:		Rematerialization:
`st.local` v [@]	$\longrightarrow$	{}
`ld.local` [@] v	$\longrightarrow$	$v = c \times \texttt{tid} + c'$

(c) $[v] = 0 \times \texttt{tid} + \perp$

Original spill code:		Sharing:
`st.local` v [@]	$\longrightarrow$	`st.shared` v [@']
`ld.local` [@] v	$\longrightarrow$	`ld.shared` [@'] v

(d) $[v] = c \times \texttt{tid} + \perp$

Original spill code:		Sharing:
`st.local` v [@]	$\longrightarrow$	$t = v - c \times \texttt{tid}$ `st.shared` t [@']
`ld.local` [@] v	$\longrightarrow$	`ld.shared` [@'] t $v = c \times \texttt{tid} + t$

Fig. 5. Rules that rewrite the code produced by a divergent aware register allocator in order to take benefit from divergence information.

that are proved to be constant do not need to be mapped into memory. As we see in the Figure 5(a), constant propagation can eliminate all the memory accesses related to the spilled value, cutting the stores off, and replacing the loads by simple variable assignments. In many cases it is possible to fold the constant value directly in the instruction where that value is necessary; thus, even avoiding the copy that replaces loads. In our experiments we did not find many opportunities to do constant propagation, simply because the code that we received from the NVIDIA compiler had already been optimized. However, we found many situations that benefit from the next rewriting rules that we describe.

Rematerialization. Variables that the divergence analysis identifies as affine constants can be rematerialized. Rematerialization is a technique proposed by Briggs *et al.* [6] to trade memory accesses by recomputation of values. If all the information necessary to reconstruct a spilled value is available in registers at the point where that value is needed, the register allocator can recompute this value, instead of bringing it back from memory. Like constant propagation, rematerialization is an optimization that completely eliminates all the memory accesses related to the spilled value. Figure 5(b) shows the rewriting rules that we use to rematerialize spilled values. Loads can be completely eliminated. Stores can be replaced by a recomputation of the spilled value, given the thread identifier.

Sharing of uniform variables. Threads inside a warp can share uniform variables. Figure 5(c) shows the rewriting rules that we use in this case. Accesses to the local memory are replaced by analogous accesses to the shared memory. In Figure 4 variable `d` has been shared in this way. Notice how the store in labels `L1` and `L19` in Figure 3 have been replaced by stores to shared memory in labels `L1` and `L20` of Figure 4. Similar changes happened to the instructions that load `d` from local memory in Figure 3.

Sharing of affine variables. The last type of rewriting rule, describing the sharing of affine variables, is shown in Figure 5(d). If the spilled variable v is an affine expression of the thread identifier, then its abstract state is $[\![v]\!] = c\mathrm{T}_{id} + t$, where c is a constant known statically, and t is a uniform value. In order to implement variable sharing in this case, we must extract t, the unknown part of v, and store it in shared memory. Whenever necessary to reload v, we must get back from shared memory its dynamic component t, and then rebuild v's value from the thread identifier and t. Notice that only one image of the value t is stored for all the threads in the warp. Thus, the sharing of affine and uniform variables produce the same number of accesses to the shared memory. The difference is that a multiply-add operation is necessary to reconstruct the affine value. Variable `N` has been spilled in this way in Figure 4. In line `L7` we have stored its dynamic component. In lines `L9` and `L10` we rebuild the value of `N`, an action that replaces the load from local memory seen at line `L9` of Figure 3.

3.4 Implementation Details

Handling Multiple Warps: There is an important implementation detail that deserves attention: a variable is uniform per warp; however, many warps integrate a GPU application. In fact, modern GPUs are implemented as multiple SIMD units [15]. In order to do variable sharing, we partition the shared memory among all the warps that might run simultaneously. This partitioning avoids the need to synchronize accesses to the shared memory between different warps. On the other hand, the register allocator requires more space in the shared memory. That is, if the allocator finds out that a given program demands N bytes to store uniform variables, and the target GPU runs up to M warps simultaneously, then the divergent aware register allocator will need $M \times N$ bytes in shared memory. We had to face an additional difficulty: we do not know, at compilation time, how many warps will run simultaneously. To circumvent this obstacle, our implementation assumes the existence of 32 warps in flight, the maximum number that our hardware supports. If the shared memory does not provide enough space for spilling, then our allocator falls back to the default execution mode, mapping spilled variables to local memory. This kind of situation will happen if the original program is already using too much of the shared memory space.

Spilling policy. A good spilling policy for a divergent aware register allocator must consider the data type of the spilled variable and this variable's access frequency. For instance, the cost to rematerialize a variable depends on its size. Operations involving 64-bit integer values, on a NVIDIA's Fermi GPU, can be as much as four times slower than similar operations with 32-bits operands. Thus, the re-writing rule that replaces loads in Figure 5(b) and (d) can cost up to eight times more when applied onto doubles. In addition to the variable's data time, its access frequency also plays an important role in the overall cost of spilling it. The access frequency is more important when we consider the spilling of affine variables, as described in Figure 5(d). Each load of an affine variable has a fixed cost that includes reading the shared memory and performing a multiply-add operation to reconstruct the spilled value. If the variable is kept in the

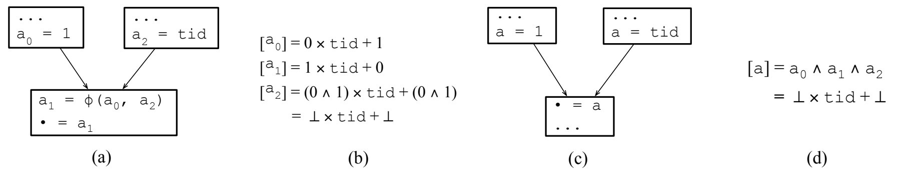

...
a_0 = 1
...
a_2 = tid
$a_1 = \phi(a_0, a_2)$
• = a_1
(a)
$[a_0] = 0 \times \texttt{tid} + 1$
$[a_1] = 1 \times \texttt{tid} + 0$
$[a_2] = (0 \wedge 1) \times \texttt{tid} + (0 \wedge 1)$
$= \bot \times \texttt{tid} + \bot$
(b)
...
a = 1
...
a = tid
• = a
...
(c)
$[a] = a_0 \wedge a_1 \wedge a_2$
$= \bot \times \texttt{tid} + \bot$
(d)

Fig. 6. (a) Program before SSA elimination. (b) Divergent status of the variables before SSA elimination. (c) Program after SSA elimination. (d) Divergent status after SSA elimination.

local memory, loading it might require an expensive trip to the off-chip memory space. However, if the variable is frequently accessed, then it is likely to be kept in cache from one load to the other. Thus, the cost of reading it from the local memory is amortized over the many times the variable is read or updated. On the other hand, if it is stored in the shared memory, not only the data access fee, but also the multiply-add cost must still be paid whenever the variable is loaded or stored.

SSA Elimination. Many compilers use the Static Single Assignment (SSA) form [11] as the default intermediate representation. Examples include gcc, LLVM, Jikes, and Ocelot, our target compiler. Programs in this format provide the core property that any variable has only one definition site. To ensure this property, the SSA format relies on a notational abstraction called ϕ-function, which merges the live ranges of different definitions of the same variable, as we show in Figure 6(a). It is possible to perform register allocation on SSA form programs [5]. However, it is more common to precede register allocation with a step called *SSA Elimination*, which replace the ϕ-functions by instructions usually found in assembly languages. There are different ways to perform SSA Elimination. A typical approach is to replace all the variables related by ϕ-functions by the same name [27]. This is the solution that we outline in Figure 6(c). Independent on the strategy used to eliminate ϕ-functions, the compiler must propagate the divergent status of variables when merging variable names. This propagation follows the meet operator that we defined for the lattice A in Equation 1. Continuing with our example, Figure 6(b) shows the divergent status of the variables before SSA Elimination, and Figure 6(d) shows it after. As the divergence analyses are done over SSA intermediate representation no coallesced variable will finish with a undefined value.

4 Experiments

Compiler and Hardware: we have implemented our divergence analysis and divergence aware register allocator in the Ocelot [12] open source compiler, SVN revision 1824 of April 2012. We have tested our implementation on an Intel Xeon X3440 with 8GB RAM equipped with a GPU Geforce GTX 470 with Nvidia's Cuda toolkit 4.1 and Device driver 295.41 (4.2).

Register allocators: we have implemented two divergence aware register allocators, as re-writing rules on top of Ocelot's original linear scan register allocator. Thus, in this section we compare three different implementations. The first, which we use as a baseline, is the linear scan implementation publicly available in Ocelot. The second, which we call the *divergent* allocator uses Ocelot's default divergence analysis [10]. This analysis only classifies variables as divergent or uniform; hence, it can only use Rule (c) from Figure 5. The other divergence aware register allocator, which we call *affine*, uses the divergence analysis with affine constraints that we describe in [25]. It can use all the four rules in Figure 5. In our experiments we give each allocator only eight registers. For the two divergent aware implementations, this number includes the register necessary to hold the base of the spilling area in the shared memory.

Benchmarks: We have compiled 177 CUDA kernels from 46 applications taken from the Rodinia [9] and the NVIDIA SDK benchmarks, which are publicly available. In this paper we show results for the 23 applications that gave us more than one hundred PTX instructions. We convert these applications from C for CUDA to PTX using NVIDIA's compiler, `nvcc`. We then use the Ocelot compiler to perform register allocation on these PTX files, lowering the register pressure to eight registers. In order to obtain runtime numbers, each tested application was executed 11 times in sequence. For each application we discarded the results of the first, the fastest and the slowest runs, and averaged the remaining eight results. The time is given in CPU ticks, and is taken right before each kernel call and right after it exits and all threads are synchronized. Some applications consists of many kernels, and some kernels are called more than once per application. Thus, we present the sum of the times taken by each kernel that constitutes an application. The total average numbers, like 21% of speedup, have been obtained by averaging the sum of the total absolute numbers.

Runtime comparison: Figure 7 shows the speedup that the different divergence aware register allocators provide over Ocelot's original linear scan. Overall, the divergence aware allocator with affine constraints speeds up the applications by 20.81%. The register allocator that only classifies variables as uniform or divergent provides a speed up of 6.21%. We cut the negative scale of the figure in -20%, but for two applications, Rodinia's nw and NVIDIA's SobolQRNG the affine allocator provides substantial slowdown: -380.55% and -146.55%. We believe that this slowdown is caused by the fact that the affine allocator must reserve two registers to handle spills: one for rematerializing values, and another for the base of the spill stack. The simple divergent aware allocator only spares one register for the spill stack, and Ocelot's linear scan can use all the eight registers. Hence, there are more spills in the affine allocator. Furthermore, these applications deal with 64 bit values, and the cost of rematerializing them, as discussed before, is substantially higher than if 32-bit values were used instead.

Static results. Figure 8 shows how often each re-writing rules of Figure 5 have been used by the divergent aware register allocator with affine constraints. By analyzing Figure 8(a) we see that, on the average, 56% of the variables were classified as divergent. 28% of the variables were classified as uniform, and thus could be handle by the re-writing rules in Figure 5(c). 9% of the variables were classified as affine, and could be handled by Rule(d). 5% of the variables were shown to be constant affine; hence, fit Rule(b) of Figure 5. Finally, 2% of the variables were constants, and could be handled by Rule(a). The low number of constants is due to `nvcc` already optimizing the programs that we give to Ocelot.

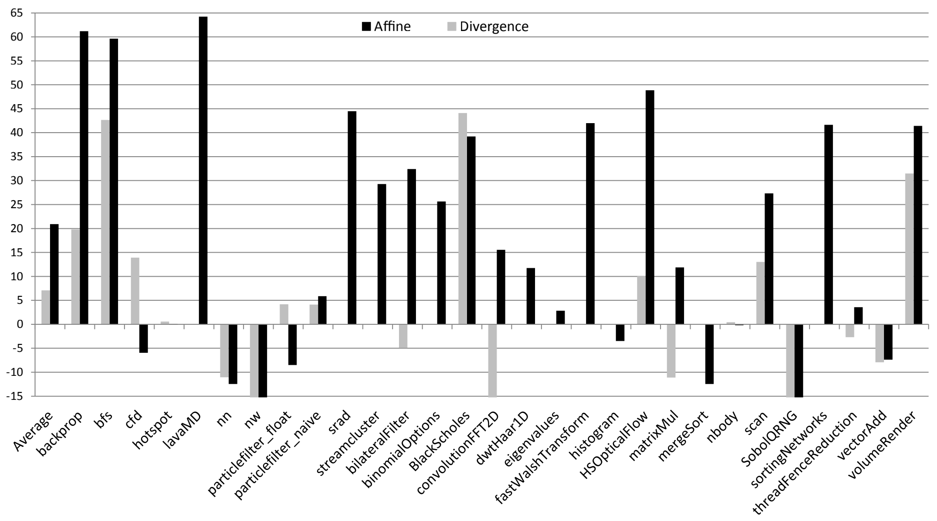


Fig. 7. Runtime results. Each bar gives the percentage of gain of the new allocator over Ocelot's original linear scan. Dark bars: divergence aware allocation with affine constraints – all the four rules in Figure 5. Light gray bars: allocation with simple divergence analysis – only Rule (c) of Figure 5.

Comparing the charts in part (a) and (b) of Figure 8 we see that with eight registers, about 13.89% of all the program variables had to be spilled, and thus mapped to memory. From Figure 8(b) we infer that most of the spill code, 47% uses the local memory, reflecting the fact that the majority of the program variables are divergent. 37% of the spilling code uses the shared memory according to Rule (c) from Figure 5. We could replace less than one percent of the spill code by constants, what is due to the low concentration of constants in the code that we obtain from `nvcc`. The other rules, for affine and constant affine variables account for 16% of the spill code. Figure 8(c) and (d) further distriminate between rules used to re-write stores, and rules used to re-write loads. Looking at these last two pies we observe a proportion of 1.69 uses of spilled variables for each definition. Ocelot adopts a spill-everywhere approach to place loads. According to this policy, each use of a spilled variable is replaced by a load instruction. In the case of divergence aware register allocation, some of these load and store instructions are re-written by the rules in Figure 5.

5 Conclusion

This paper has described what we believe is the first register allocator specifically tailored for the Single Instruction, Multiple Data execution model ever discussed in the compiler related literature. We have implemented the proposed allocator as a set of re-writing rules on top of the linear scan allocator used in an open source PTX compiler. Our code is available for download at `http://simdopt.wordpress.com`. This web-

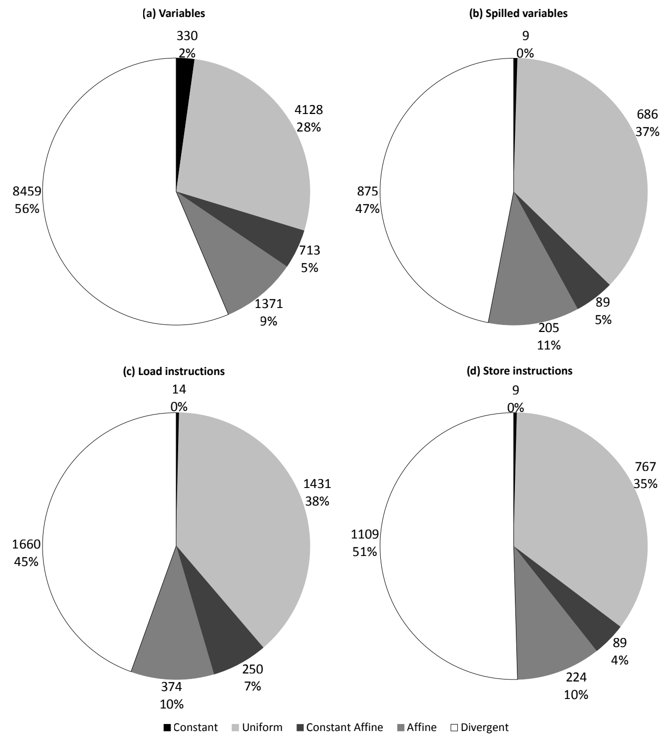


Fig. 8. Overall distribution, into the 5 types of affinity, of divergent states. (a)Variables (b)Spilled variables (c)Load instructions (d)Store instructions

page also contains raw data, like the absolute numbers that we have used to produce the charts in Section 4. We have presented an extensive evaluation of the proposed allocator, but there are still work to be done in this area. In particular, we are interested in trying different spill policies that take into consideration more information related to the divergent state of program variables. We are also interested in identifying uniform variables that do not need to be replicated among every warp in the target program.
Acknowledgement: This project was sponsored by FAPEMIG, grant 01/2010, CAPES and CNPq.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (2006)
2. Backus, J.: The history of fortran i, ii, and iii. SIGPLAN Not. 13(8), 165–180 (1978)
3. Baghsorkhi, S.S., Delahaye, M., Patel, S.J., Gropp, W.D., Hwu, W.M.W.: An adaptive performance modeling tool for gpu architectures. In: PPoPP. pp. 105–114. ACM (2010)
4. Belady, L.A.: A study of replacement algorithms for a virtual storage computer. IBM Systems Journal 5(2), 78–101 (1966)

5. Bouchez, F.: Allocation de Registres et Vidage en Mémoire. Master's thesis, ENS Lyon (October 2005)
6. Briggs, P., Cooper, K.D., Torczon, L.: Rematerialization. In: PLDI. pp. 311–321. ACM (1992)
7. Carrillo, S., Siegel, J., Li, X.: A control-structure splitting optimization for gpgpu. In: Computing frontiers. pp. 147–150. ACM (2009)
8. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer Languages 6, 47–57 (1981)
9. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IISWC. pp. 44–54. IEEE (2009)
10. Coutinho, B., Sampaio, D., Pereira, F.M.Q., Meira, W.: Divergence analysis and optimizations. In: PACT. IEEE (2011)
11. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS 13(4), 451–490 (1991)
12. Diamos, G., Kerr, A., Yalamanchili, S., Clark, N.: Ocelot, a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In: PACT. pp. 354–364 (2010)
13. Farach-colton, M., Liberatore, V.: On local register allocation. Journal of Algorithms 37(1), 37 – 65 (2000)
14. Garland, M.: Parallel computing experiences with CUDA. IEEE Micro 28, 13–27 (2008)
15. Garland, M., Kirk, D.B.: Understanding throughput-oriented architectures. Commun. ACM 53, 58–66 (2010)
16. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. Elsevier, 1st edn. (2004)
17. Han, T.D., Abdelrahman, T.S.: Reducing branch divergence in GPU programs. In: GPGPU-4. pp. 3:1–3:8. ACM (2011)
18. Harris, M.: The parallel prefix sum (scan) with CUDA. Tech. Rep. Initial release on February 14, 2007, NVIDIA (2008)
19. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: ISCA. pp. 451–460. ACM (2010)
20. Nickolls, J., Dally, W.J.: The GPU computing era. IEEE Micro 30, 56–69 (2010)
21. Nickolls, J., Kirk, D.: Graphics and Computing GPUs. Computer Organization and Design, (Patterson and Hennessy), chap. A, pp. A.1 – A.77. Elsevier, 4th edn. (2009)
22. Pereira, F.M.Q., Palsberg, J.: Register allocation after classic SSA elimination is NP-complete. In: Foundations of Software Science and Computation Structures. Springer (2006)
23. Poletto, M., Sarkar, V.: Linear scan register allocation. TOPLAS 21(5), 895–913 (1999)
24. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., mei W. Hwu, W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPoPP. pp. 73–82. ACM (2008)
25. Sampaio, D., Martins, R., Collange, S., Pereira, F.M.Q.: Divergence analysis with affine constraints. Tech. rep., École normale supérieure de Lyon (2011)
26. Sethi, R.: Complete register allocation problems. In: 5th annual ACM symposium on Theory of computing. pp. 182–195. ACM Press (1973)
27. Sreedhar, V.C., Gao, G.R.: A linear time algorithm for placing ϕ-nodes. In: POPL. pp. 62–73. ACM (1995)
28. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. TOPLAS 13(2) (1991)
29. Zhang, E.Z., Jiang, Y., Guo, Z., Tian, K., Shen, X.: On-the-fly elimination of dynamic irregularities for GPU computing. In: ASPLOS. pp. 369–380. ACM (2011)