Critical Points Based Register-Concurrency Autotuning for GPUs

Ang Li*, Shuaiwen Leon Song†, Akash Kumar‡, Eddy Z. Zhang§, Daniel Chavarría-Miranda† and Henk Corporaal*
*Eindhoven University of Technology, The Netherlands. Email: {ang.li, h.corporaal}@tue.nl
†Pacific Northwest National Laboratory, USA. Email: {shuaiwen.song, daniel.chavarria}@pnnl.gov
‡Technische Universität Dresden, Center for Advancing Electronics Dresden, Germany. Email: akash.kumar@tu-dresden.de
§The State University of New Jersey, USA. Email: eddy.zhengzhang@cs.rutgers.edu

***Abstract*—The unprecedented prevalence of GPGPU is largely attributed to its abundant on-chip register resources, which allow massively concurrent threads and extremely fast context switch. However, due to internal memory size constraints, there is a tradeoff between the per-thread register usage and the overall thread concurrency. This becomes a design problem in terms of performance tuning, since the performance "sweet spot" which can be significantly affected by these two factors is generally unknown beforehand.**

In this paper, we propose an effective autotuning solution to quickly and efficiently select the optimal number of registers per-thread for delivering the best GPU performance. Experiments on three generations of GPUs (Nvidia Fermi, Kepler and Maxwell) demonstrate that our simple strategy can achieve an average of 10% performance improvement while a max of 50% over the original version without modifying the user code. Additionally, to reduce local cache misses due to register spilling and further improve performance, we explore three optimization schemes (i.e. bypass L1 for global memory access, enlarge local L1 cache and spill into shared memory) and discuss their impact on performance on a Kepler GPU.

I. Introduction

The extraordinary emergence of general-purpose Graphic Processing Units (GPGPUs) is well-known for their massive thread-level-parallelism (TLP). To accommodate such amount of active threads, GPU has to encapsulate a large register file. Moreover, to mitigate the negative impact from the memory wall, GPU adopts the "latency hiding" technique by keeping the contexts of all the active threads in the register files, which enables fast switching when stalls are encountered. Although the GPU register files are quite large compared to those on CPUs, such utilization can still impose great pressure on them. As the limited registers are evenly distributed among the active threads, the performance tradeoff between the per-thread register usage and the overall thread volume appears: for the applications that are bounded by the limited register resource, although more registers per thread indicate superior single-thread performance without register spills, fewer registers per thread could increase thread concurrency, which may eventually result in aggregated performance improvement. Therefore, finding the optimal per-thread register usage that delivers the best performance becomes an important issue for GPU software developers. Efficient register usage management is also considered as one of the biggest remaining issues of the current CUDA toolchain [7].

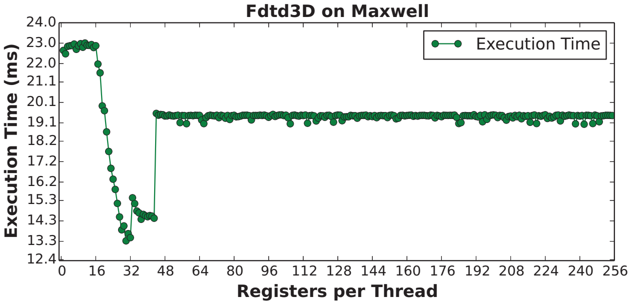

Fig. 1. Profiling for different register number for *Fdtd3d* on Maxwell GPU.

Figure 1 illustrates an example to describe the problem. It shows the execution time of *Fdtd3d* with respect to per-thread register usage for a Maxwell GPU[1]. On the left, the execution time decreases with a higher register utilization. However, the curve is interrupted at $r = 33$ and $r = 44$ with a sudden and significant increase. The task is to find the r that corresponds to the shortest execution time. Although in this example, it is obvious that $r_{\text{opt}} = 32$, it is impractical to plot such figure for every application we study, since the register range can be very large (e.g. 1 to 255 for Maxwell GPUs) and the position of the optimal point may also be input-dependent. Furthermore, not all applications show such an ideal curve, as will be seen later. Therefore, the problem is how to find an effective way to shrink the search space for r_{opt} and then efficiently locate it. This paper thus makes the following contributions:

- We study the underlying relationship between register count, concurrency and performance, based on which we propose the idea of critical points (Section III).
- We propose an efficient autotuning scheme to find the optimal register usage per thread. It is tractable, effective, and general for benefiting all GPU generations (Section III).
- We explore three optimizations to further improve performance and reduce local cache conflicts due to register spills (Section V).

II. Background

In this section, we briefly introduce the GPU **thread organization** and the **local memory** access.

A GPU kernel, which is a device function executed on the GPU hardware, contains thousands or tens of thousands concurrent threads that are primarily partitioned into multiple *thread blocks* (TBs). When a kernel is launched, all

[1] This paper only focuses on Nvidia GPUs as AMD GPUs do not provide the compiler option required (i.e. *-maxrregcount*) to tune the register bound.

978-3-9815370-6-2/DATE16/©2016 EDAA

the TBs are distributed the streaming multiprocessors (SMs). It is possible that several TBs are distributed to the same SM simultaneously, depending on the size of SM on-chip resources, such as the registers and the scratchpad memory (i.e. shared memory). These resources are evenly divided among the concurrent TBs. The threads of a TB are further grouped into a number of execution vectors, called *warps*, that perform the same operations on different data in a lockstep manner. A warp is the basic unit for instruction issuing, executing, L1 cache access and so on. The utilization of the warp-slots is defined as **occupancy** of an SM, which is proportional to the **thread concurrency** that describes the number of active threads of an SM.

In addition to the register file, a GPU thread has several types of memory to access, including global (off-chip, the GPU main memory, L1 and L2 cached), local (off-chip, L1 and L2 cached), shared (on-chip, shared in a TB), texture (on-chip, read-only and cached) and constant (on-chip, read-only and cached). The local memory is not essentially a physical memory but rather an abstraction of the global memory. Its scope is thread-private, the same as for the register file. It is generally used for temporal spilling when there are insufficient registers to hold all the required variables or the arrays that are declared inside the kernel but the compiler cannot resolve the indexing. It is also L1- and L2-cached, for both read and write. Register spilling in local memory may hurt the performance as it introduces extra instructions and memory traffic, especially for cache miss (i.e. long access latency to the global memory).

III. Methodology

In this section, we present our critical points (CP) based autotuning method. We call it "auto" because the entire tuning process can be accomplished automatically without user intervention. All the required information can be extracted from the output of the CUDA compiler and the profiler. The method is based on the following **key observations**:

1) On one hand, a GPU kernel requires at least a number of registers to be successfully compiled (i.e. the lower bound of the register usage: r_{min}). On the other hand, a GPU kernel needs at most a number of registers so that all the intermediate data are located in registers (i.e. the upper bound of the register usage: r_{max}). Beyond r_{max}, allocating more registers is wasteful.
2) For a single GPU thread, more register contributes to spill reduction and locality exploitation. Therefore, more registers can lead to better single thread performance.
3) For the massive TLP on GPUs, the concurrency or occupancy may *impact* performance significantly. Although more threads generally lead to better latency hidding and pipeline utilization thereby a higher performance, it is not always true under certain scenarios: if a subsystem is already saturated (e.g. the scalar processors (SPs) are fully leveraged by exploiting instruction-level-parallelism (ILP) [11]), adding more threads brings no further performance gains. Even worse, excessive threads to an overloaded system may lead to dramatic conflicts and contention, degrading the overall performance [5].

Obviously, there is a performance tradeoff between register usage per thread and thread concurrency: *can the benefits from higher thread concurrency (i.e. fewer registers assigned to each thread) offset the drawbacks from register spills?* To answer this question, we first discuss the relationship between register usage and performance. We denote r as the number of registers per thread, and based on observation-(1) we have

$$r_{\text{min}} \leq r \leq r_{\text{max}} \tag{1}$$

We label this region $[r_{\text{min}}, r_{\text{max}}]$ as the *Register Effective Region* (**RER**). Based on observation-(2), with a larger r, more spill loads and stores can be avoided, which contributes to a higher performance. If we use $g(r)$ to denote the performance function with respect to the per-thread register count, then

$$\text{Performance} = g(r) \propto r \tag{2}$$

Note that $g(r)$ is continuously increasing as every one more register eliminates a fraction of spills until all spills are eliminated.

Now let us turn to thread concurrency and explore why the change of r can lead to concurrency drop. Since the number of registers per TB is fixed, **the only factor that can directly impact concurrency is the maximum number of TBs that can be dispatched simultaneously on an SM at runtime**. This TB number is limited by the hardware restrictions and availability of on-chip resources, one of which is just the amount of registers. Therefore, if we use w to denote the number of warps per TB, then the number of TBs that can be dispatched simultaneously on an SM is:

$$\begin{aligned} N = min\{ \lceil \frac{\text{All\_TBs}}{\text{SMs}} \rceil, N_{\text{TBs/SM}}, \lfloor \frac{N_{\text{warps/SM}}}{w} \rfloor, \\ \lfloor \frac{N_{\text{regs/SM}}}{\lceil \frac{N_{\text{regs/TB}}}{\text{unit}_{\text{reg}}} \rceil * \text{unit}_{\text{reg}}} \rfloor, \lfloor \frac{N_{\text{smem/SM}}}{\lceil \frac{N_{\text{smem/TB}}}{\text{unit}_{\text{smem}}} \rceil * \text{unit}_{\text{smem}}} \rfloor \} \end{aligned} \tag{3}$$

The five terms in the function are number of TBs per kernel, GPU restricted amount of TBs per SM, GPU restricted amount of warps per SM, register limitation and shared memory limitation per SM. The ceiling in the last two items are because a GPU allocates registers/shared memory to TBs by a unit size, which is 64/128B for Fermi and 256/256B for both Kepler and Maxwell. In general, a kernel includes thousands of TBs, so the first term is very large. $N_{\text{TBs/SM}}$ is 8 for Fermi, 16 for Kepler and 32 for Maxwell. If here we temporarily assume that the shared memory is not the bottleneck, then the formula becomes:

$$N = min\{N_{\text{TB/SM}}, \lfloor \frac{N_{\text{warps/SM}}}{w} \rfloor, \lfloor \frac{N_{\text{regs/SM}}}{\lceil \frac{32*w*r}{\text{unit}_{\text{reg}}} \rceil * \text{unit}_{\text{reg}}} \rfloor \}$$

in which $N_{\text{TB/SM}}$, $N_{\text{warps/SM}}$, $N_{\text{regs/SM}}$ and unit_{reg} are constants while w is predefined by the application. The only variable left in the equation is the register number (r). If we use $f(\text{concurrency})$ to denote the performance function with respect to to thread concurrency, then

$$\begin{aligned} \text{Performance} &= f(\text{concurrency}) = f(N_{\text{thds/TB}} * N_{\text{TB/SM}}) \\ &= f(N_{\text{thds/TB}} * \lfloor \frac{N_{\text{regs/SM}}}{\lceil \frac{32*w*r}{\text{unit}_{\text{reg}}} \rceil * \text{unit}_{\text{reg}}} \rfloor) \end{aligned} \tag{4}$$

Based on observation-(3) that a higher concurrency in general contributes to a better performance, we have

$$\text{Performance} \propto 1/r \tag{5}$$

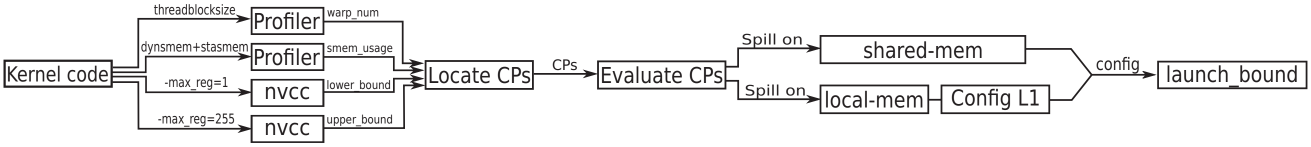

Fig. 2. Autotuning Framework

When observing Eq.2 and Eq.5, there is a clear conflict. Note that, unlike $g(r)$, the correlation between f(concurrency) and r shows only a few discrete steps due to the *floor()* function in Eq.4. In fact, with the *floor()* function, an increment of r does not necessarily lead to a decrement of $\lfloor \frac{N_{\text{regs/SM}}}{\lceil \frac{32*w*r}{\text{unit}_{\text{reg}}} \rceil * \text{unit}_{\text{reg}}} \rfloor$. But once the increment of r triggers a drop of $\lfloor \frac{N_{\text{regs/SM}}}{\lceil \frac{32*w*r}{\text{unit}_{\text{reg}}} \rceil * \text{unit}_{\text{reg}}} \rfloor$, the concurrency degrades by a significant factor of $N_{\text{thds/SM}}$. We label the last points (i.e. register usage) before such drops as the **critical points** (CPs). These significant changes in concurrency or occupancy may lead to drastic variations in performance, which forms a series of stages (we label them **concurrency levels**). Such a performance curve is the result of a typical combination of effects from $g(r)$ and $f(1/r)$.

Therefore, the basic idea for the CP-based autotuning is: In the range of RER, different concurrency levels separate the performance curve with respect to the register count into several regions. Within each region, the performance at the CP is likely the optimal or very close to the optimal (see next section for details). Since a different concurrency level impacts performance but not necessarily leads to a better performance, we need to evaluate all the CPs to locate the global optimal in the autotuning process.

Our proposed autotuning framework is shown in Fig.2. First, we need to decide the boundaries of RER. This information can be extracted from the GPU compiler *nvcc* when passing the *-maxrregcount=1* and *-maxrregcount=max_reg_per_thd* (the values shown in Table I) flags respectively. The compiler will output the minimum and maximum registers required (i.e. r_{min} and r_{max}). We then profile the kernel to obtain the warp number and shared memory usage per TB. Together with the hardware information, we are able to locate the CPs for a specific application based on Eq.3. After that, the framework measures the execution time of each CP and reports the optimal point.

IV. Experiment

In this section, we validate the CP-based autotuning method on three generations of GPUs: Fermi, Kepler and Maxwell. The platform information is listed in Table I. We selected 12 representative applications from the Rodinia [1], SDK [9] and Parboil[10] benchmarks, as listed in Table II. We also show the number of warps and amount of shared memory allocated per TB in each application to compute the CPs. As discussed in Section III, the flags *–maxrregcount=1* and *–maxrregcount=255 (63 for Fermi)* are passed to the *nvcc* compiler to acquire the lower (r_{min}) and upper bound (r_{max}) for the register usage of an application. We also obtain the default register usage chosen by the compiler as the "**Baseline**" for performance comparison. The results for Fermi, Kepler and Maxwell are shown in Fig.3, 4 and 5 respectively. "**Proposed**" is the performance achieved by CP-based autotuning. "**Optimal**" is the performance improvement upper-bound given by exhaustive searching within the RER region. We also show the occupancy change, the register usage points that have to be searched and the geometric mean for performance improvement across all applications in the figures. As can be seen, our autotuning approach achieves 7.9%, 8.8% and 5.5% speedup on average for Fermi, Kepler and Maxwell GPUs over the baseline, while the optimal results reported by exhaustive searching are 9%, 10% and 7%, respectively. However, compared with exhaustive searching, our CP-based method reduces the search space for r_{opt} by a factor of 15x, 20x and 13x on geometric average, which corresponding to time reduction from 53s, 337s and 150s to 3.5s, 17s and 11.2s respectively, when using the dataset in Table II. (Note, the exhaustive searching here is within the RER region, which has already reduced the search space substantially. A complete exhaustive search takes 85s, 1449s and 650s otherwise).

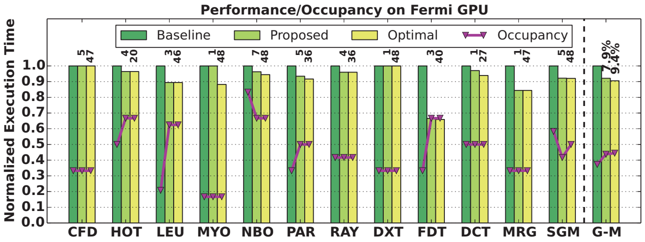

Fig. 3. Performance Improvement on Fermi GPU. The black numbers on top of the application bars indicate the size of search space.

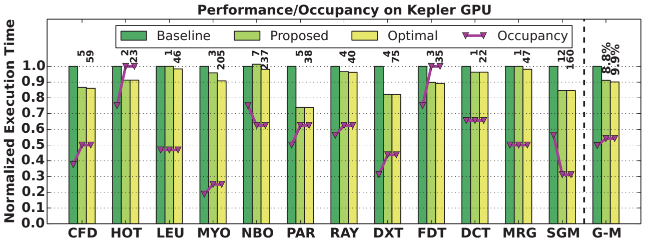

Fig. 4. Performance Improvement on Kepler GPU.

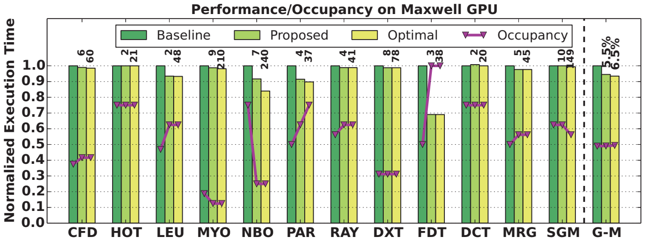

Fig. 5. Performance Improvement on Maxwell GPU.

TABLE I
Experiment Platforms. Dri/Rtm means the CUDA driver version and toolkit version. M(TBs/SM) indicates the maximum allowable number of thread blocks per SM. M(Thds/SM) is the maximum number of threads per SM. M(Regs/Thd) is the maximum number of registers per thread. (Shared+L1)/SM is the volume of shared memory and L1 cache per SM.

GPU	Arch	Dri/Rtm	SMxSP	M(TBs/SM)	M(Thds/SM)	Regs/SM	M(Regs/Thd)	(Shared+L1)/SM
GTX570	Fermi-2.0	6.5/6.5	15x32	8	1536	32K	63	(48+16)KB
Tesla K40	Kepler-3.5	6.0/6.0	15x192	16	2048	64K	255	(48+16)KB
GTX750Ti	Maxwell-5.0	6.5/6.5	5x128	32	2048	64K	255	(64+0)KB

TABLE II
Experiment Applications. U/L/D indicates the register upper- and lower-bound per thread as well as the default register number chosen by the compiler on a specific architecture. Fm stands for Fermi. Kp stands for Kepler. Mx stands for Maxwell. Shared is the consumption of shared memory per SM.

Application	Abbr.	Kernel	Warps	Shared	U/L/D (FM)	U/L/D (KP)	U/L/D (MX)	Source
cfd	CFD	cuda_compute_flux()	8	0	62/16/62	74/16/68	75/16/70	Rodinia[1]
hotspot	HOT	calculate_temp()	8	3072B	35/16/35	38/16/38	36/16/35	Rodinia[1]
leukocyte	LEU	IMGVF_kernel()	10	14586B	61/16/52	61/16/61	63/16/63	Rodinia[1]
myocyte	MYO	solver_2()	1	0	63/16/63	220/16/149	225/16/133	Rodinia[1]
nbody	NBO	integrateBodiesIf	8	4096B	63/16/24	252/16/38	255/16/37	SDK[9]
particles	PAR	collideD()	8	0	51/16/51	52/16/52	52/16/52	SDK[9]
ray-tracing	RAY	render()	4	0	51/16/50	55/16/49	56/16/56	SDK[9]
dxtc	DXT	compress()	2	2048B	63/16/63	90/16/89	93/16/90	SDK[9]
fdtd3d	FDT	FiniteDifferencesKernel()	16	3840B	55/16/45	50/16/40	53/16/45	SDK[9]
dct8x8	DCT	CUDAkernel2IDC()	3	3136B	42/16/35	37/16/33	35/16/34	SDK[9]
mri-gridding	MRG	gridding_GPU()	2	1536B	62/16/56	62/16/62	60/16/59	Parboil[10]
sgemm	SGM	mysgemm()	4	512B	63/16/33	175/16/53	164/16/48	Parboil[10]

One interesting observation is that not every application's thread concurrency or occupancy increases after the optimization (e.g. NBO and SGM), which indicates that a higher occupancy does not necessarily lead to a better performance. It also confirms the necessity to evaluate each different concurrency level (i.e. each CP). Also note that CFD shows very different behaviors on the three architectures (i.e. CFD shows significant performance improvement on Kepler, but very little on Fermi and Maxwell).

To further explore why in certain applications the CP set cannot capture the optimal (e.g. MYO and MRG in Fig.4) and why in NBO, the performance of CP is even worse than the baseline, we plot the execution time with respect to register number and concurrency level for the 12 applications on Kepler, shown in Fig.6. We also draw the curves for normalized spilled loads & stores reported by compiler and the cache hit rate for local access (see Section II) measured by profiler. Regarding this result, we have the following observations:

(1) Though we only plot the figures in the range of RER (using the lower- & upper-bound in Table II), we can clearly observe that the point at which the spilled-load and store disappears (also the point where the local cache hit rate reduces to zero) is always less than the upper-bound of RER. We call this point the **spill-disappear-point (SDP)**. Although at this point, no spill occurs, there is still some rematerialization, because the compiler is able to reduce the register usage by recomputing the values of some intermediate variables based on the alternative registers. Nonetheless, such rematerialization incurs unnecessary computation overhead. Only beyond the RER upper-bound, all the intermediate data is stored in the registers, and there is neither spill nor redundant computation.

(2) The trends that execution time drops with more threads confirm the Observation-(1). However, not all the applications are concurrency sensitive, e.g. MYO and SGM. Meanwhile, some applications such as LEU, DCT and MRG are limited by other on-chip resources, changing the register usage does not impact occupancy or concurrency. For example, LEU and DCT are limited by the shared memory usage. As each TB in LEU requires 14586B shared memory space (see Table II), 48KB shared memory can afford up to 3 TBs. With 10 warps per TB, the occupancy keeps constant at $3 * 10/64 \approx 0.47$. For DCT, each TB counts 3136B, 48KB thus is theoretically sufficient for 15 TBs. However, as shared memory is allocated in a unit of 256B on Kepler (see Eq.3 in Section III), eventually only 14 TBs are initiated per SM, which contributes to an occupancy of $14 * 2/64 \approx 0.44$. On the other hand, MRG is restricted by the maximum number of TBs per SM (hardware limitation), which is 16 for Kepler (see Table I). The occupancy thus stays at 0.5. From Kepler to Maxwell, as an SM supports more TBs (from 16 to 32), we can observe that the occupancy changes as expected and the performance increases for MRG in Fig.5.

(3) The baseline point, or the default register usage number imposed by the compiler is neither the SDP nor the upper-bound of RER. It is calculated by an unknown algorithm of the compiler. Additionally, the number of CPs for each application is generally around 5, which is much smaller than the RER range. The optimal point for performance is mostly captured by our approach for each application. The exceptions are MYO, MRG and NBO. It can be observed that dramatic performance oscillation occurs within a concurrency level (especially MRG). This may due to the variation of register bank conflict degree from different register allocation strategies adopted by *nvcc*.

(4) Although in general the normalized spill LD&ST curves drop with increased number of registers until the SDP, the curves for local cache hit rates are far more intractable. They commonly start at lower hit rate because there are many variables that have to be spilled due to significant shortage of registers. At the same time, a higher occupancy also implies more inter-TB conflicts in the L1 cache. As more registers are allocated and less TBs share the cache, the hit rates increase,

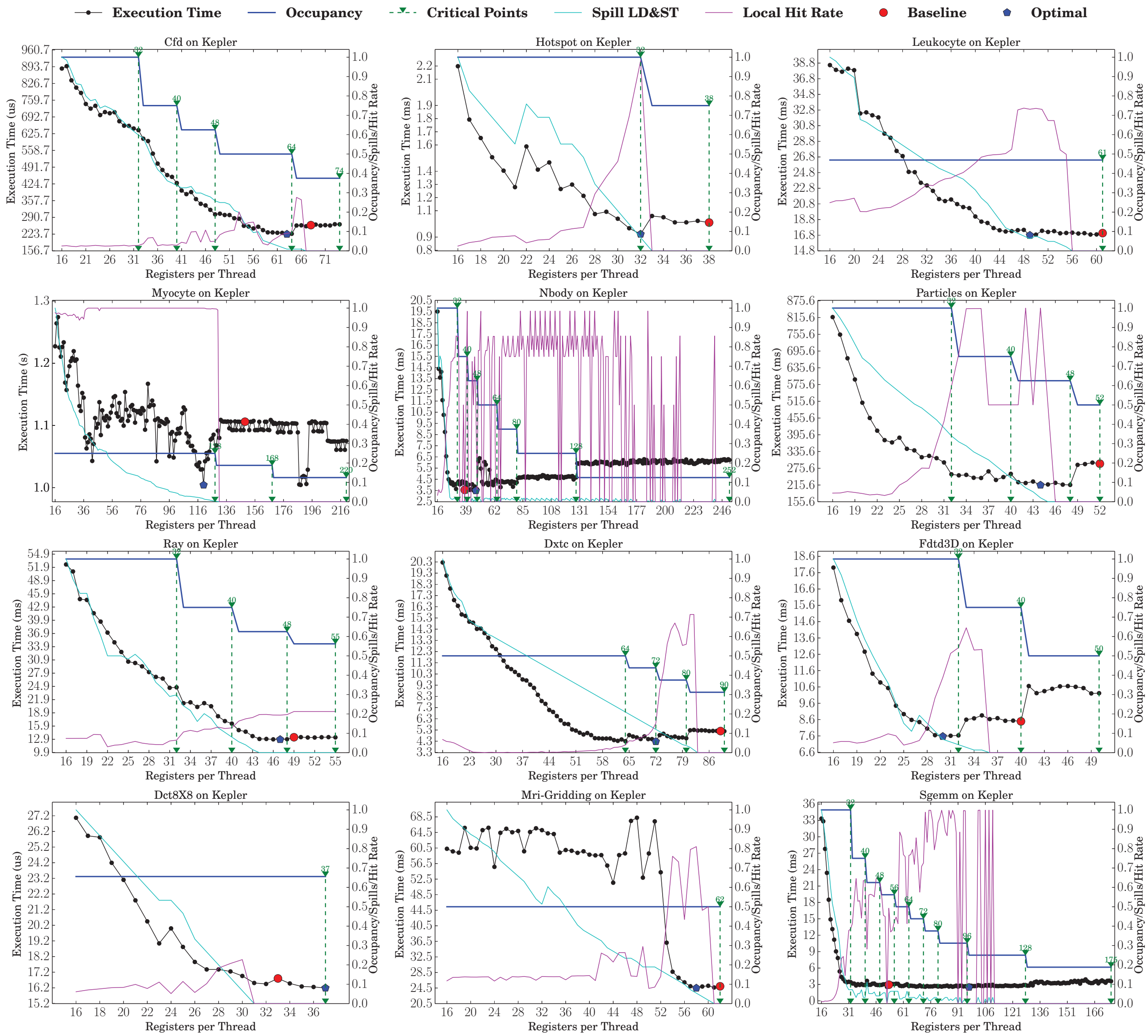

Fig. 6. Detailed Application Profiling on Kepler GPU. Local hit rate is only for local cache hit rate of L1 not the overall L1 hit rate.

and drop to zero at the SDPs because there is no local memory access any more. Additionally, some steep fluctuation in NBO and SGM can be observed. This is because with different register numbers, the compiler algorithm may occasionally enforces some 4B to 16B local memory spills, which translate to a very high hit rates. Thus, the curves oscillate sharply within certain regions (e.g. register range between 90-110 for SGM). Also note that the local cache hit rates may suffer from global memory accesses, as they are sharing the same cache storage.

V. Discussion

Shown in Figure 6, overall the local cache hit rates for the applications are not quite high. Possible reasons include compulsory misses (i.e. first-time spill), capacity misses (i.e. many registers from multiple active threads need to spill to a very small cache size of 16KB per SM), and conflict misses (i.e. shared by multiple TBs and meanwhile shared with the global accesses). To mitigate or further eliminate the latter two, we apply the following three optimizations:

- We configure a larger L1 cache (e.g. 32KB or 48KB, instead of 16KB) upon kernel invocation.
- We apply software-level strategies [4] to spill in the shared memory instead of the local memory .
- We bypass the L1 cache to avoid possible conflicts from global memory access by setting "*-dlcm=cg*".

The results are shown in Fig.7. As can be seen, a larger L1 cache size enhances local cache hit rate for CFD, RAY, DXT and FDT, which improves performance for CFD, RAY and FDT. The scenario for DXT is interesting, as a 32KB L1 increases performance but a larger 48KB L1 degrades performance drastically. This is because, although a 48KB entirely avoids L1 cache miss, the larger L1 cache capacity is achieved at the expense of a smaller shared memory (L1

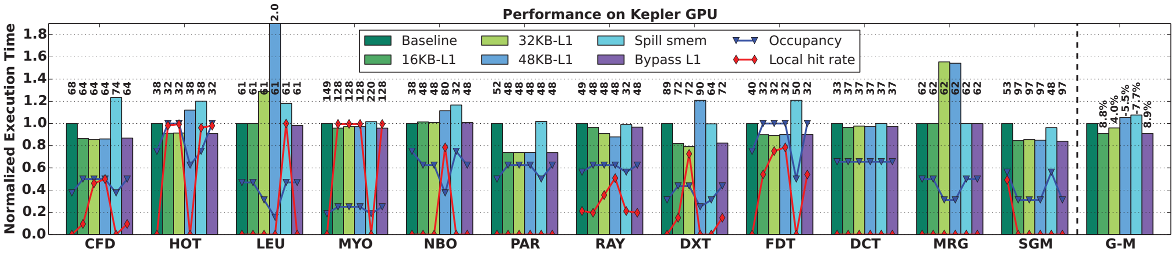

Fig. 7. Test different L1 cache configurations, the design of spilling on shared memory and bypassing global access at L1 on Kepler GPU. The numbers on top of the histograms are the obtained register number by each scheme.

cache and shared memory share the same storage in an SM). The reduced shared memory capacity limits the number of TBs that can be allocated simultaneously per SM (see Eq.3 in Section III), which eventually degrades the concurrency and performance. Besides, spill on shared memory is not shown to be a good solution in our test, as it always delivers the lowest performance. Finally, bypassing global access does not impact local cache hit rate or performance (watch that the time and local hit rate for “16KB-L1” and “Bypass L1” are the same); this is because on Kepler, all global memory access bypass L1 by default [8]. However, this is not the case for Fermi. In fact, we observed performance improvement for every applications except MYO on Fermi with L1 cache being bypassed for global memory access.

VI. Related Work

Previous work related to GPU register file mostly focuses on architectural improvement, seeking to reduce chip area and energy consumption [2], [12], [3], [6]. Gebhart et. al. [2] placed a small register cache on top of GPU’s main register file so that the small register cache can filter a large portion of the accesses before going to the main register file. In this way, significant power consumption can be avoided. They also combined their register cache with a novel two-level warp scheduler for further energy reduction. Yu et. al. [12] integrated eDRAM into the SRAM based GPU register file to reduce energy. Later, Gebhart et. al. [3] combined register file, L1 cache and scratchpad memory of GPU as a unified storage space and dynamically tuned the partitioning among them. Recently, Lee et. al. [6] found that values written by threads in the same warp show great similarity therefore can be compressed to reduce power.

The work that most related to ours is proposed by Hayes and Zhang [4]. Their work also concentrated on the tradeoff between register usage and concurrency while wrapped the on-chip scratchpad memory as a supplementary register file. A metric based on computation/memory interleaving degree is proposed to predict the best concurrency level at compile-time. However, their design is concurrency-centric. The calculation of the predicted concurrency (i.e. the metric) requires complicated parsing and analysis of the binary while some of the input parameters are architecture-dependent and are very difficult to measure (e.g. the dispatch interval). Their work also presumed that local memory access is detrimental and should be all eliminated. However, migrating the latency sensitive data from L1&L2-cached local memory to the shared memory with extra software management overhead may not be beneficial eventually (see Fig.7 in Section V).

VII. Conclusion

In this paper, we proposed an autotuning approach to resolve the conflict between concurrency and register usage for GPUs. We discovered that the performance impact from register is continuous but from concurrency is discrete. The tradeoff between the two factors forms a special relationship such that a series of critical points can be precomputed. These CPs denote the best performance of each concurrency level, and the global optimum is then selected among them. Our approach is **tractable**, **effective** and **general**. It leverages the existing features of the hardware and demonstrates immediate speedup for all three generations of GPUs over a dozen of real applications. The improvement is very close to the optimal one achieved by exhaustive searching. Our method reduces the search space for the optimal register usage by up to 20x and enhances the overall GPU performance by up to 1.5x without changing the user code. More importantly, our tuning method is fully automatic and can be easily integrated into the compiler toolchain.

Acknowledgment

This work is supported in part by the German Research Foundation (DFG) within the Cluster of Excellence Center for Advancing Electronics Dresden (cfaed).

References

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *IISWC*. IEEE, 2009.

[2] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, “Energy-efficient mechanisms for managing thread context in throughput processors,” in *ISCA*. ACM, 2011.

[3] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, “Unifying primary cache, scratch, and register file memories in a throughput processor,” in *MICRO*. IEEE, 2012.

[4] A. B. Hayes and E. Z. Zhang, “Unified on-chip memory allocation for SIMT architecture,” in *ICS*. ACM, 2014.

[5] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, “Neither more nor less: optimizing thread-level parallelism for GPGPUs,” in *PACT*. IEEE, 2013.

[6] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, “Warped-compression: enabling power efficient GPUs through register compression,” in *ISCA*. ACM, 2015.

[7] M. Murphy, “NVIDIAs Experience with Open64,” in *Open64 Workshop at CGO*, 2008.

[8] C. Nvidia, “CUDA Programming Guide,” 2015.

[9] C. Nvidia, “SDK Code Samples,” 2015.

[10] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” *Center for Reliable and High-Performance Computing*, 2012.

[11] V. Volkov, “Better performance at lower occupancy,” in *GTC*, 2010.

[12] W.-K. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, “SRAM-DRAM hybrid memory with applications to efficient register files in fine-grained multi-threading,” in *ISCA*. ACM, 2011.