High Performance by Exploiting Information Locality through Reverse Computing

Mouad Bahi, Christine Eisenbeis

High Performance by Exploiting Information Locality through Reverse Computing

Mouad Bahi
INRIA Saclay - Ile-de-France, Orsay, France
LRI, Univ. Paris-Sud 11, Orsay France
mouad.bahi@inria.fr

Christine Eisenbeis
INRIA Saclay - Ile-de-France, Orsay, France
LRI, Univ. Paris-Sud 11, Orsay, France
christine.eisenbeis@inria.fr

***Abstract*—In this paper we present performance results for our register rematerialization technique based on reverse recomputing. Rematerialization adds instructions and we show on one specifically designed example that reverse computing alleviates the impact of these additional instructions on performance. We also show how thread parallelism may be optimized on GPUs by performing register allocation with reverse recomputing that increases the number of threads per Streaming Multiprocessor (SM). This is done on the main kernel of Lattice Quantum ChromoDynamics (LQCD) simulation program where we gain a 10.84% speedup.**

***Index Terms*—thread level parallelism, rematerialization, reversible computing**

I. Introduction

Over the years, processor cycle time is decreasing much faster than memory access times, and the architectural design of processors has improved with the development of pipelining and multiple instruction issue. These factors have influenced the increasing technological gap between processor speed and the speed of the memory hierarchy. Figure 1 shows processor and memory speedup during the years, and the processor-memory performance gap.

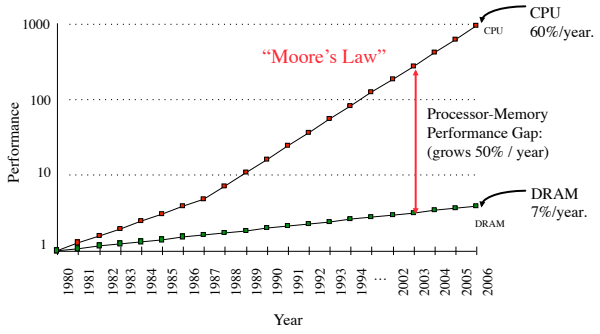


Fig. 1. Memory Access vs CPU Speed

But, no matter how fast a CPU is, it has to request data from different levels of some memory hierarchy. Hence CPU speed is hindered by the slow read/write memory speed. This is why memory together with communication optimizations are today the most important room for performance.

The purpose of this paper is twofold. First we study the effect of using reverse computing for register materialization [1]. Register materialization generates additional instructions and we have to trade instruction count against register usage. We show on one specifically designed example that rematerialization with reverse computing is less sensitive that ordinary rematerialization with direct recomputing. Second we show that register optimization can improve different levels of parallelism namely instruction level and thread level. This latter level is typically present in GPUs and we give experimental results that take advantage of reverse computing-based register rematerialization for optimizing the performance of a physics application (Lattice Quantum ChromoDynamics - LQCD) [6]. In this case of thread-level parallelism minimizing register usage increases the number of threads that can concurrently share the same streaming multiprocessor and therefore the occupancy [1].

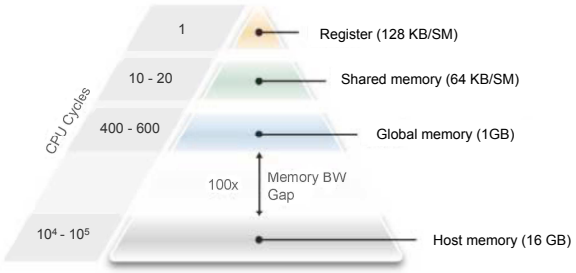


Fig. 2. GPU architecture: Memory Access vs CPU Speed

The reason why we use the term "information locality" instead of "data locality" is because information may be carried not only by a data but also by the process and input data that generate it. Therefore information can be present in different forms. When computations are reversible there are more ways for retrieving information, not only from input data and computation but also from other already computed data and reverse computation. Hence reversible computing improve information locality. We do not elaborate further on this concept in that paper but information locality is a notion we are currently working on, in the context of reverse computing in relationship with power consumption and heat dissipation.

[1]Occupancy is the ratio of active threads to the maximum number of threads supported on a multiprocessor

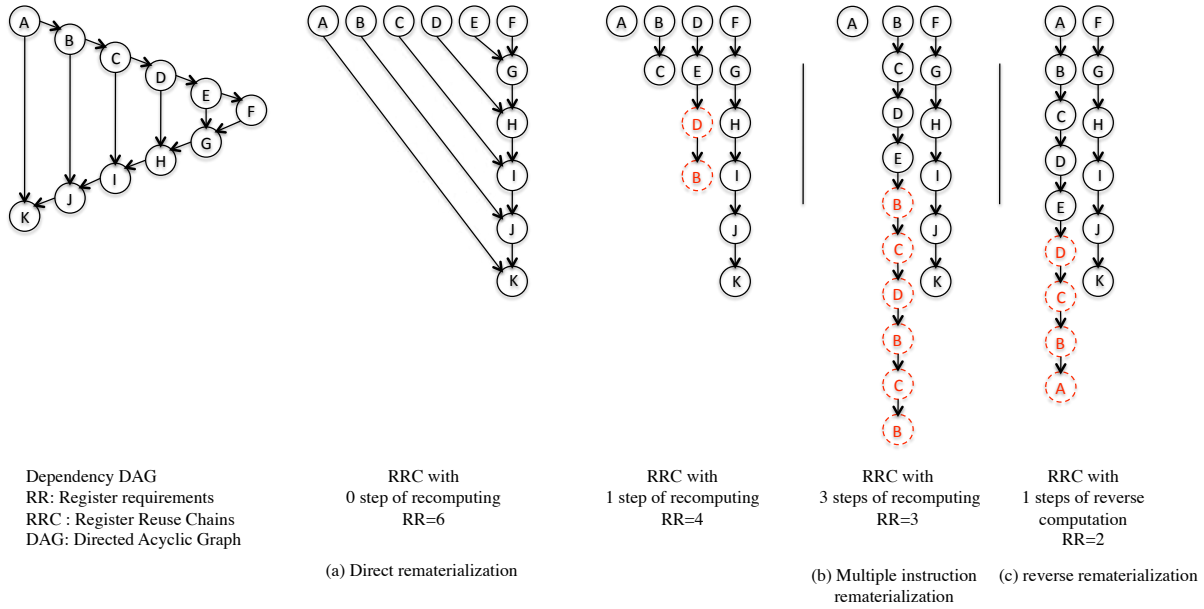


Fig. 3. Recomputing Vs Storage

II. Reverse Rematerialization

In the domain of register allocation when the register pressure is too high one can spill the data of one register into memory and reload it later when it is needed, or alternatively one can try to recompute it from currently still alive registers. This process is called rematerialization. We have designed a register allocation algorithm that uses rematerialization by reverse computing with one or more recomputing instructions [1]. Instead of recomputing from available input date one can also recompute reversely from results by using reversible computing. Our algorithm is based on the generation of register reuse chains [2]. The evaluation of register requirements uses a reuse DAG, which indicates which instruction can reuse a register used by a previous instruction.

This algorithm is based on the observation that sometimes several data carry the same information, so there is no need to keep all of them in memory. For example, for the instruction *c:=a+b*, the information in *(a,b,c), (a,b), (a,c)* or *(b,c)* is the same, so no need to keep the three values live at the same time, because we can always compute one value from the two others. *a* can be retrieved from *(b,c)* and *b* from *(a,c)* by a simple subtraction. Hence, values that carry the same information can share the same register. We consider that all functions can be made reversible, therefore, a value v might be rematerializable by a direct operation from source operands, or by a reverse operation from the result and the rest of operands of the operation. We have published details of this algorithm elsewhere[1]. It relies on the observation that:

v is directly rematerializable iff $\forall p \in input(v) \Rightarrow p\ is\ live$

v is reversibly rematerializable iff
$\exists q \in output(v) \forall p \in input(q)$ s.t. $p \neq v \Rightarrow p\ and\ q\ are\ live$

We call $R\text{-}input(v)$ the set of sets of operands allowing to recompute v either by a direct or reverse operation.

v is rematerializable by multiple instructions iff
$\exists S \subset R\text{-}input(v)\ \forall p \in S \Rightarrow p\ is\ live$ or $p\ is\ rematerializable.$

Based on register reuse chains we give conditions allowing value w to reuse the register of value v in case of high register pressure and recompute v later when we achieve low register pressure, knowing that live-ranges of w and v overlap.

v is rematerializable $\wedge$ w can reuse v iff
$\exists S \subset R\text{-}input(v)\ \ \forall p \in S \Rightarrow$ p stays live after w

If at the next use of v we still have high register pressure, we add the condition $output(w) \cap output(v) = \phi$

Figure 3 demonstrates rematerialization with one and multiple instructions using direct and reverse operations. In the example, six registers are required to perform the DAG according to the initial reuse DAG (Figure 3(a), middle). Live-ranges of A, B, C, D, E and F overlap, but as B and C stay live during computing all outputs of C and E respectively, and C can be directly computed from B, and E from D, we can allow C to reuse the register of B, and E the register of D, and recompute them later just before computing H and J respectively, as shown on the right of Figure 3(a). We can increase register reuse more than that. A stays live during all the computation. Thus, we can rematerialize B, C, D based on A directly. In the Figure 3(b), the register requirements using rematerialization with multiple instructions is three with six additional operations but helps to avoid three spill operations, in case of three registers available.

In Figure 3(c) only 2 registers are required using reverse rematerialization, and each loaded value is rematerialized by one instruction which avoids four load/store operations for the whole DAG in case of 2 available registers. The register pressure is high after computing C, D, E and F causing A, B, C and D to be spilled. A simple way to avoid inserting four load operations before computing H, I, J and K is to rematerialize them from theirs outputs with one instruction by recomputing D from E, C from D, B from C and A from B.

To achieve this goal, we propose register rematerialization based on register reuse chains after register allocation. Figure 4 shows an overview of the algorithm. It takes a source code as an input and constructs data dependence graphs (DDG) for the basic blocks. The measurement of register requirements uses a reuse DAG indicating which instruction can reuse a register used by a previous instruction. Once rematerialization decision is taken, we proceed to the graph transformation of the original DDG. The rematerialization algorithm we propose is iterative, after each graph transformation we re-call the algorithm till there will be no rematerializable value.

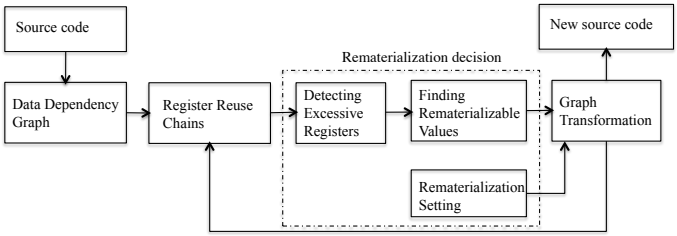


Fig. 4. Overview of the rematerialization algorithm

A. Register usage, rematerialization and performance

We have implemented the code in Figure 9, on NVIDIA GTX 470 (the processor architecture is described in Section IV-A). The body of the outer loop is a simulation of the above example in Figure 3. We aim to study the impact of reverse rematerialization on performance and how additional operations could effects the execution time. We try also to find a trade-off between register usage, number of rematerialization instructions and performance. We apply reverse rematerialization till there will be no rematerializable value. In this example the register requirements is lowered to two as shown in Figure 3(c). Figure 5 shows execution time as we vary the number of simultaneously live values. By comparing the two plots of Figure 5, we notice that the positive impact of reducing register usage per thread by using reverse rematerialization and rematerialization in general is not immediate. This can be explained by, first, the limited number of concurrently running threads per SM, 1024 on GTX 470 comparing to the large number of registers, 32 768 32-bit registers, which allows assigning up to 32 registers per threads. So no speedup is expected with a number of live values less than 32. Second, the number of additional operations increases by increasing the number of simultaneously live values when the number of available registers is fixed, as shown in Figure 6.

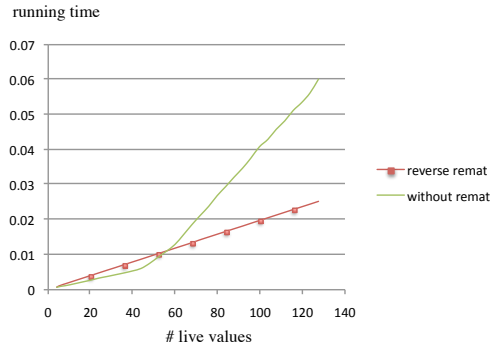


Fig. 5. Reverse rematerialization : performance vs. register requirements

After the crossing of the two curves, we start getting speedup and when the number of live values is getting larger, the performance gap between the two versions is getting larger too. In this simulation, the register requirements per thread using rematerialization is constant, only the number of operations is increasing. The plot of reverse rematerialization is almost linear with the number of simultaneously live values because the register requirement and the number of active threads are constant and only the number of operations is increasing. Thus, there is no extra spill of variables or use of local memory contrary to the original code.

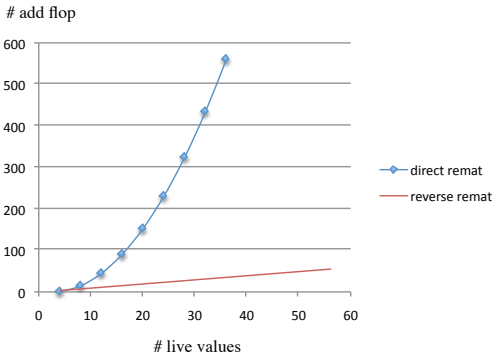
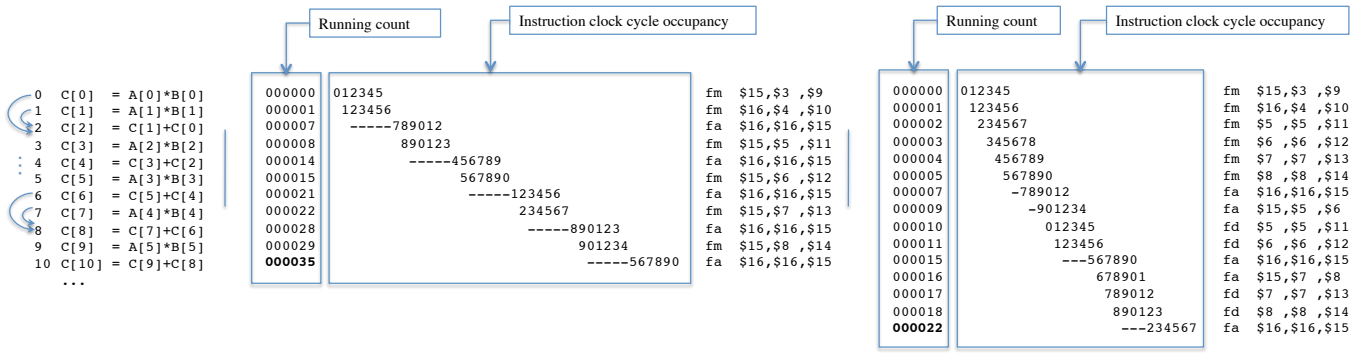


Fig. 6. Cost of reversibility: Additional operations

Now we choose a configuration with 48 simultaneously live values that requires 48 registers. We apply again reverse rematerialization and we vary the number of available registers in the algorithm, knowing that the number additional operations decreases by increasing the number of available registers and vice versa. Figure 8 shows that the execution time decreases gradually by increasing the number of available registers. However the optimal running time is reached when the number of registers is less than that required for the original version.

In the future, in order to take advantage of this approach systematically, we would like to find ways in which a compiler can automatically predict with which rate reducing register demands per thread, using recomputing, will provide higher performance. This allows high performance CUDA programs to be built with minimum time and effort.



(a) 3 address code (b) Assembly code: without rematerialization (c) assembly code: using reverse rematerialization

Fig. 7. Using reverse computing to increase instruction level parallelism - within loop SIMDization

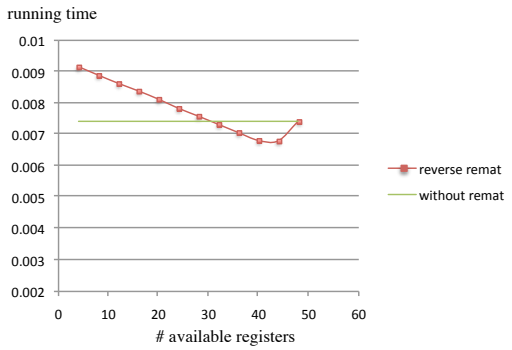


Fig. 8. Reverse rematerialization : performance vs. available registers

B. Limitations

Currently, our reverse rematerialization has several limitations. First, there are data precision issues especially with floating point operations and round-off problems. Next, division operation, the inverse of multiplication, when implemented in software, will require more non pipelined stages, and each will require temporary storage of a few intermediate results, so we only apply reverse rematerialization when we don't exceed the number of registers of the original operation.

III. Using Reverse Computing to Increase Instruction-Level Parallelism

A program is, in essence, a set of instructions, that can be grouped together to accomplish a task. These instructions can be scheduled according to the data and resources dependences among them, and then executed in parallel or in concurrency without changing the result of the program, to speed up the execution. This is called Instruction-level parallelism (ILP). This is achieved by performing different stages of the pipeline through a number of execution units within the processor.

Parallelism is limited among instructions. First, by data dependencies between pairs of instructions which are executing in parallel. Second, by resource dependencies when an instruction requires a hardware resource which is still being used by a previously issued instruction.

Recall that the usefulness of reverse rematerialization is not only decreasing register pressure but also increasing register reuse to allow more instructions to be performed simultaneously.

Two different kinds of techniques for increasing the instruction-level parallelism that a processor can be exploited; *within a basic block* and *across basic blocks*.

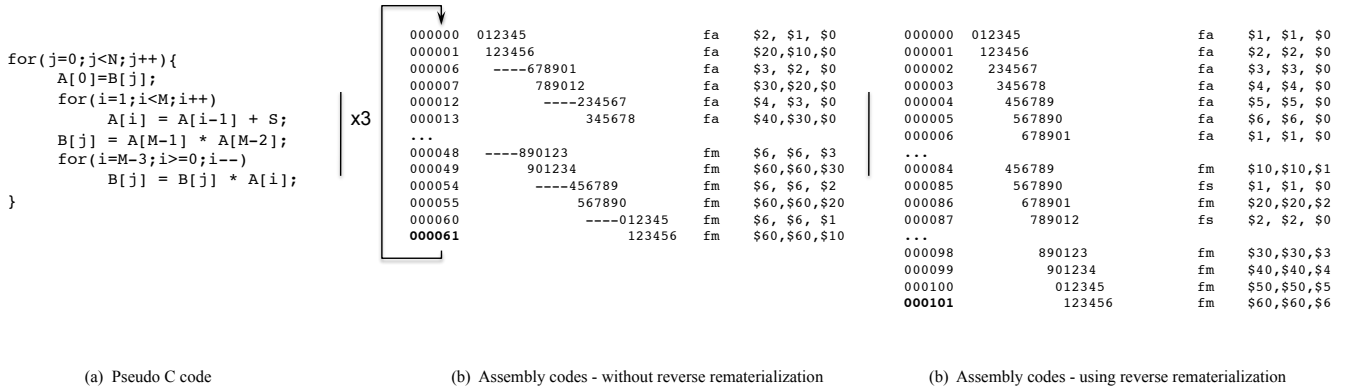
A. Instruction-level parallelism within basic blocks

Consider the following code corresponding to a part of `_su3_multiply` macro: multiplication of a 3x3 complex matrix by a 3 complex vector. As the assembly code in Figure 7(a) shows, the added operations using reverse rematerialization do not increase the cycle time since they are overlapped with synchronized operations and replace stall cycles.

Operation 2 depends on the results of operations 0 and 1, so it cannot be calculated until both of them are completed. Knowing that input data stay live during the whole computation, with only two available registers, the processor stalls for six cycles before computing the result of operation 2. However operations 0, 1, 3, 5, 7, 9 do not depend on any other operation, so they can be calculated simultaneously if there are enough available registers. As can be seen in Figure 7(c), with reverse rematerialization we can use registers of input data to perform the six operations simultaneously and replace most of stall cycles. We rematerialize input data once intermediate values are used. The performance gain in this example is up to 31.7%

B. Instruction-level parallelism across basic blocks

Sometimes, even if the register file can hold all intermediate values of one iteration, the processor might occasionally stall as a result of data dependencies and branch instructions. Rematerialization through reverse operations helps to reduce register usage per iteration to keep pipeline full by unrolling loops without any spill operations and exploit parallelism among instruction by finding sequences of unrelated instructions from different iterations that can be overlapped in the pipeline.



(a) Pseudo C code (b) Assembly codes - without reverse rematerialization (b) Assembly codes - using reverse rematerialization

Fig. 9. Using reverse computing to increase instruction level parallelism - across loop SIMDization

Consider the C code in Figure 9(a), where the body of the outer loop is a simulation of the example in Figure 3(a). The first inner loop generates a sequence of numbers where each term is found by adding the previous one with a fixed number S. The second inner loop returns the product of all previous elements in reverse way and store the result in array B in memory. Here, the compiler can exploit a minimum of M registers, the sequence's size, which is the number of simultaneously live values. As shown in Figure 3(c), using reverse rematerialization, we can reduce the register requirements to only two registers.

To avoid stalls, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction. By assuming the number of the inner loop iterations is 6, and the number of available registers is 12, the compiler can select the unroll factor that fits the register requirements into the available registers, we can unroll the outer loop twice, without using rematerialization, and 6 times using reverse rematerialization, to avoid the stalls and extra tests and branches. After unrolling there are 2 copies of the original loop body and 4 copies of modified loop with additional operations. By assuming the number of j-loop iterations is 6, the estimated running time of this loop is running time of the body loop after unrolling, times the new number of iterations as shown in Figure 9(b). In this example it is estimated to 67x3 = 201 cycles, compared to only 107 cycles using rematerialization.

The assembly code in Figure 9(c) shows this optimization and the number of cycles, statically timed using the spu-timing tool of IBM on Cell BE. As can be seen, with reverse rematerialization, we can use available registers to perform six independent operations simultaneously and replace all stall cycles. The performance gain in this example is x2.

However, this optimization is traded off on Cell BE due to the small size of SPE's local store (LS), against the potential penalty caused by increased code size on the larger loop body to fit both code and data.

IV. IMPLEMENTATION AND EXPERIMENTAL RESULTS

This section presents our experiments in parallelizing a real scientific simulation code from the computational nuclear physics domain. This application performs complex operations on a 4-dimensional space-time lattice of size n that describes the strong force which binds protons and neutrons forming the atomic nucleus. The core kernel `Hopping_Matrix` involves *O(n)* computation over two sequential and synchronized loops `Hopping_Matrix_k` and `Hopping_Matrix_l`, single or double precision floating point latices, where each loop iteration involves complex-vector multiplications, vector-matrix multiplications, vector additions and vector subtractions.

Considering that an input value should be loaded at most one time, and once a final output is computed it will be directly stored in the memory, thus:

- One complex-complex multiplication requires 6 floating point operations and 6 registers.
- One complex-vector multiplication requires 18 floating point operations and 10 registers at least.
- One vector-matrix multiplication amounts to 66 floating point operations and 14 registers at least by performing three independent vector-vector multiplications.

This results in 768 flops for the first loop and 840 flops for the second one. An optimized use of data lets reduce the memory usage to 48 registers per iteration in both first and second loop.
For a maximum instruction level parallelism, the register requirements are:

- 6 registers for complex-complex multiplication.
- 14 registers for complex-vector multiplication.
- 18 registers for vector-vector multiplication.
- 42 registers for vector-matrix multiplication.

Thus, the memory size required for the kernel scales linearly with loop size, but register requirements and the number of floating point operations are fixed *O(1)*.

Choosing between increasing instruction level parallelism (ILP) and reducing register pressure depends of resources specifications like size of register file, size and latency of the

shared memory, etc. The allocation process based on reuse chains consists of two closely interacting subtasks: scheduling and allocation. The allocator's main objective is to define a reuse politics of registers that deals with resources and a schedule that deals with latencies. A scheduling before register allocation will ensure that each pair of independent instructions will dual-issue and may ignore registers availability. A register allocation before scheduling will ensure a best register allocation but may penalize instruction level parallelism. The register allocation based on register reuse chains in our implementation ensures that scheduling and register allocation are solved simultaneously.

In the following, we use our tool to study and optimize this application. We give priority to reduce register requirements per thread without inserting any spill instruction. Figure 10, shows a code fragment of the LQCD application and its corresponding data dependency graph.

```
_su3_multiply(chi0, U, psi0);
_complex_times_vector(phi0, ka, chi0);
_su3_multiply(chi1, U, psi1);
_complex_times_vector(phi1, ka, chi1);
```

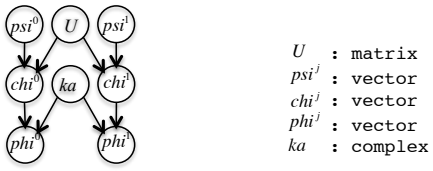


Fig. 10. HMC code fragment - original implementation

We divide the above instructions into sub-instructions. The major improvement is reordering the computational instructions, so that, we decrease the number of simultaneously live values. A vector of matrix U is used for the computation of two complex 3x3 matrix vector complex multiplications for two different vectors. We compute the $i\text{-}th$ complex of each vector instead of computing one vector after another, as shown in Figure 11. Thus, a matrix U is never loaded entirely, but only the six values on each line that will be replaced by the next six values of the next line. As a result, we can save up to 6 registers in this code fragment.

A. GPU Implementation

GPU is a massively parallel architecture intended for a range of complex algorithms in both commercial and scientific fields. Its highly parallel structure makes it more effective than general-purpose CPUs to accelerate a variety of data parallel applications. Figure 12 shows a block diagram of the NVIDIA Fermi Streaming Multiprocessor (SM). Fermi architecture has been designed to support a broad range of application by featuring up to 448 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 448 CUDA cores are organized in 14 Streaming Multiprocessors (SMs) of 32 cores each. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Each SM has 16 load/store units, allowing

```
_vector_times_vector(chi00, U0, psi0);
_complex_times_complex(phi00, ka, chi00);
_vector_times_vector(chi01, U0, psi1);
_complex_times_complex(phi01, ka, chi01);

_vector_times_vector(chi10, U1, psi0);
_complex_times_complex(phi10, ka, chi10);
_vector_times_vector(chi11, U1, psi1);
_complex_times_complex(phi11, ka, chi11);

_vector_times_vector(chi20, U2, psi0);
_complex_times_complex(phi02, ka, chi20);
_vector_times_vector(chi21, U2, psi1);
_complex_times_complex(phi21, ka, chi21);
```

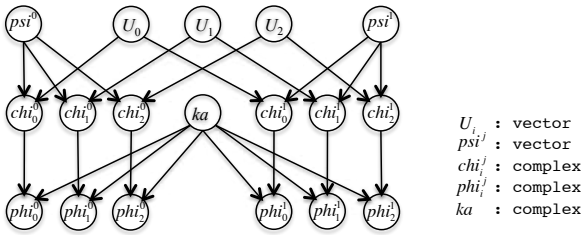


Fig. 11. HMC code fragment: code splitting and reordering - optimized implementation

source and destination addresses to be calculated for sixteen threads per clock. Four Special Function Units (SFUs) execute transcendental instructions such as sin, cosine, reciprocal, and square root. Each SFU executes one instruction per thread, per clock. Up to 16 double precision fused multiply-add operations can be performed per SM, per clock. Each SM has a 128 KB of shared register file and a 48 KB of shared memory. The SM schedules threads in groups of 32 parallel threads called warps. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. For existing applications that are bandwidth constrained, reducing the shared memory and register usage yields significant performance improvements. First to avoid higher-level memory usage, so automatically benefit from fast on-chip registers, and second increase the multiprocessor warp occupancy. In the following, we study the effects of register requirement via direct measurement of time.

The SM's 32 768-entry, 32-bit register file allows running 1024 thread simultaneously which is the maximum number of threads per block running on a single streaming multiprocessor. More thread blocks may run on the same SM. If one thread block stalls while waiting for global memory, etc., then another thread block may run. This can be used to hide the latency. The number of registers and shared memory requires by the kernel affects the number of threads per block and the number of thread blocks per streaming multiprocessor. If a thread uses more "variables" than registers allocated for each thread, the extra variables will automatically be spilled to a special region of global memory, called local memory, which may decrease performance. The optimization we found to be important, as Fermi runs most efficiently with a large numbers of threads, was a concept "`recomputing or rematerialization`", that helps to minimize register

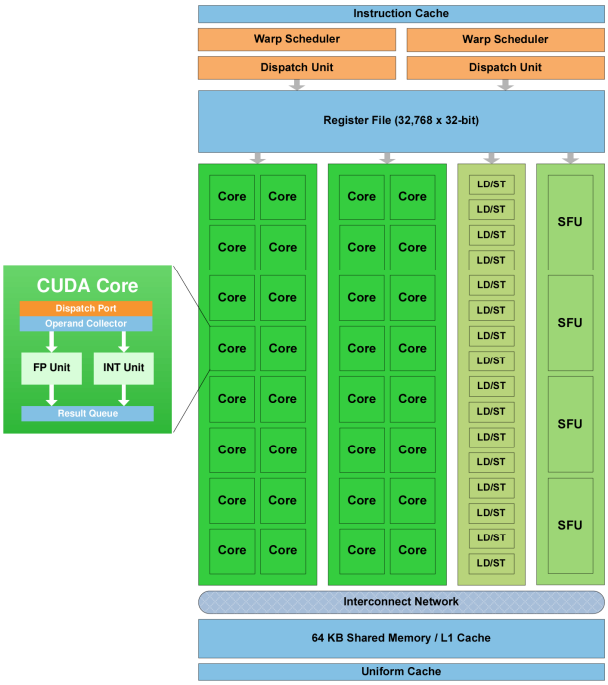


Fig. 12. Fermi Streaming Multiprocessor (SM)

requirements per thread. We mean that all registers of values that can be recomputed can be re-used. This increases register reuse and reduces register demands per thread that should increase the number of active threads per SM.

We wrote a parser for the kernel that extracts basic blocks and build their data dependency graphs. Our tool analyses divergences in graphs that correspond to multi-use of data and tries to decrease them by transforming the original graphs by applying rematerialization algorithm. In general, more divergences in a graph mean more simultaneous live values.

In our previous work [1], we showed that rematerialization through reverse computing can reduce the register usage with significant rate. Recomputing increases the number of instructions, but this should not penalize performance since threads of one warp are running in parallel comparing to the gain from reducing register demands that frees more space for more threads. The total space gain is the register gain per thread times the number of threads per SM.

In parallelizing across multiple cores of GPU, we have taken the simplest approach by assigning one thread to each lattice site. In other words, we assigned one thread per loop iteration Hopping_Matrix. We implemented a new versions of `Hopping_Matrix`, called `Hopping_Matrix_opt`, that requires less registers than the original code using our register allocation tool based on register reuse chains. We performed the above optimization for both original and optimized versions as described in table 1 that shows a gain of 6% and 14% in register requirements respectively, and an increase of 20% in the optimized implementation using recomputing on the number of thread per block. This optimization increases the number of instructions of the GPU kernel by 7%, but recomputing using `reverse operations` from output operands amortizes the cost of recomputing to 4.68%.

	min reg. req.	max thread/block
Hopping_Matrix_k	50	640
Hopping_Matrix_k_remat	47	640
Hopping_Matrix_k_opt	43	704
Hopping_Matrix_k_opt_remat	37	768
Hopping_Matrix_l	54	576
Hopping_Matrix_l_opt	50	640

TABLE 1
CONTRIBUTION OF REVERSE REMATERIALIZATION TO INCREASE THREAD LEVEL PARALLELISM

We measure the performance of the original and optimized code without and with using recomputing. Tables 2 and 3 present results for: (1) original code without any kernel's optimization (2) code optimized using our register allocation tool. This result demonstrated the efficiency of recomputing to improve performance by increasing memory space availability and thread level parallelism. It provides improvements of 5.75% and 10.83% over original code for simple and double precision respectively, and -1.33% and 10.60% over the optimized implementation for simple and double precision versions respectively.

		without remat	with remat	%gain
Hopping_Matrix_k	float	19.12	20.22	5.75%
	double	9.69	10.74	10.83%

TABLE 2
CONTRIBUTION OF REVERSE REMATERIALIZATION TO INCREASE PERFORMANCE ON NVIDIA GPU (GFLOPS) - ORIGINAL IMPLEMENTATION -

		without remat	with remat	%Perf
Hopping_Matrix_k	float	20.19	19.92	-1.33%
	double	11.88	13.14	10.60%

TABLE 3
CONTRIBUTION OF REVERSE REMATERIALIZATION TO INCREASE PERFORMANCE ON NVIDIA GPU (GFLOPS) - OPTIMIZED IMPLEMENTATION -

Our results in table 4 illustrate that on a NVIDIA GTX 470 GPU, recomputing can even improve single-thread performance. Among the four different simple precision versions, applying recomputing on the original code takes the least amount of time that increases performance up to 22.8% to 0.14 gflops. when converting to double precision Only an improvement of 6.37% can be observed by using recomputing on the original code. Our optimization has a negative impact on the optimized versions.

	original	remat	opt	opt + remat
float	0.1163	0.1429	0.1115	0.1097
double	0.0565	0.0601	0.0747	0.0737

TABLE 4
SINGLE THREAD PERFORMANCE ON NVIDIA GPU FOR HOPPING_MATRIX_K (GFLOPS)

B. Analysis of Results

Table 4 shows a weak single-thread performance. This is justified by the slower memory, lower clock frequency and short-vector data parallelism. The power of the GPU comes from the fact that it can keep thousands of threads running concurrently. Constantly swapping different threads, with no context switch overhead, hides global memory latency and stall time due to dependencies among instructions, which is approximately 24 cycles. Thus, increasing the number of active threads per SM makes GPU able to run a lot of threads in essentially the same time as a single thread would execute.

NVIDIA GPU today demonstrates approximately twice the performance for single precision execution when compared to double precision. We know double precision rules in High Performance Computing for problems that require higher precision and to minimize the accumulation of round-off error. Because of lack of 64-bit registers, recomputing to reduce register usage has greater advantages for 64-bit floating point execution.

V. Related Works

NVIDIA's Experience with Open64 [8] reveals that 90% of the performance improvement is just due to keeping things in registers. When every thread has to do a slow memory access, the performance degrades quite a bit, so there is a large benefit from simply putting local variables in registers. Various approaches have been proposed to decrease register pressure in compilers, including different algorithms of register live range splitting [3], register rematerialization [5], and register pressure sensitive instruction scheduling before register allocation [7]. First who have efficiently conceived rematerialization were Briggs et al. in [4], their approach focuses on rematerialization in the context of Chaitin's allocator [5]; where the problem was discussed briefly. Punjani in [9] has implemented rematerialization in GCC, the experimental results indicated a gain of 1-6% in code size and 1-4% improvement in execution performance. Zhang in [11] proposed an aggressive rematerialization algorithm to reduce security overhead that uses multiple instruction to recompute a value and extends the live-ranges of depending values deliberately to make the values alive through the point of rematerialization. Most of these previous works target rematerialization between basic blocks and ignore it inside basic blocks, and many of rematerialization algorithms are invoked before register allocation [10], which makes rematerialization decision less efficient because of the lack of information concerning register requirements, excessive registers and rematerializable values, and which can create extra dependencies to extend live-range of inputs of the rematerialized value, and this can increase register pressure. We presented in [1] how to use reversible computing to reduce register pressure and spill code and showed that there are more opportunities for rematerialization through reverse operations than direct operations.

VI. Conclusion and future works

We have shown experimental results for our approach using reverse computing based register rematerialization. A number of previous works have addressed the interaction between instruction scheduling and register allocation, but using reverse computing gives news degrees of freedom and we have shown that is can still improve performance despite an increase in instruction count. In our algorithm the maximum number of steps required for recomputing a value is a parameter hence in the future, the user could flexibly play with this parameter to find the best tradeoff between number of threads, register count and additional instruction count.

We have also shown that register allocation is still an important issue on the recent GPU processors as different threads share a common register file and the number of simulaneously concurrent threads depends on the number of registers of each individual thread. In this case too reverse computing helps gaining performance as it provides more opportunities for register rematerialization.

We will have to extensively check whether precision issues can be overcome, this will be done on the LQCD application that is a specially well adapted benchmark for that purpose as it requires very high precision at least in some parts.

We are aware that more systematic experiments and benchmarks should be done in order to prove that reverse computing based rematerialization is effective in the general case. But we believe that as the gap between communication/memory latency and CPU latency increases we possibly will have to consider that computation is for free and only communication matters, very much in the spirit of research on communication avoiding algorithms. Recomputing together with reverse recomputing may then be an alternative to communicating.

Acknowledgments

This work was partially supported by the ANR PetaQCD project.

References

[1] Mouad Bahi and Christine Eisenbeis. Register Reverse Rematerialization. Research report, INRIA Saclay - Ile de France, July 2011.

[2] D. A. Berson, R. Gupta, and M. L. Soffa. Ursa: A unified resource allocator for registers and functional units in vliw architectures. In *Conf. on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, Amsterdam, The Netherlands, 1993.

[3] Preston Briggs. *Register allocation via graph coloring*. PhD thesis, Rice University, Houston, TX, USA, 1992.

[4] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *the ACM SIGPLAN conference on Programming language design and implementation*, pages 311–321, New York, NY, USA, 1992. ACM.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN symposium on Compiler construction*, NY, USA, 1982. ACM.

[6] Angles D'Auriac et al. Towards the petaflop for Lattice QCD simulations the PetaQCD project. In *Journal of Physics Conference Series*, volume 219, pages 052–021, 2010.

[7] R. et al Govindarajan. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Trans. Comput.*, 52:4–20, January 2003.

[8] Mike Murphy. Nvidia's experience with open64. *Open64 Workshop at CGO 2008*.

[9] Mukta Punjani. Register rematerialization in gcc. In *GCC Developers' Summit 2004*.

[10] Loren Taylor Simpson. *Value-driven redundancy elimination*. PhD thesis, Rice University, Houston, TX, USA, 1996.

[11] T. Zhang, X. Zhuang, and S. Pande. Compiler optimizations to reduce security overhead. In *the Int. Symposium on CGO*, pages 346–357, Washington, DC, USA, 2006. IEEE Computer Society.