Register Packing for Cyclic Reduction: A Case Study

Andrew Davidson
University of California, Davis
aaldavidson@ucdavis.edu

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

ABSTRACT

We generalize a method for avoiding GPU shared communication when dealing with a downsweep pattern. We apply this generalization to Cyclic Reduction, a tridiagonal solver with this pattern. Previously, Cyclic Reduction suffered poor performance when compared to other tridiagonal solvers on the GPU due to performance issues stemming from shared-memory bandwidth bottlenecks and step-efficiency. We address this problem by applying our downsweep shared-memory communication-reducing methodology. Our re-mapping also allows Cyclic Reduction to solve larger systems directly in a virtual block. By using our generalized mapping, we improve Cyclic Reduction's performance on a GPU by a factor of 3–4.5x over the original CR implementation, making it 1.5–3x faster than other GPU tridiagonal solvers.

1. INTRODUCTION

GPUs have become a popular platform for a wide variety of methods and applications. However, it is still difficult to determine the optimum mapping of an algorithm to a GPU. These difficulties are amplified by the added memory hierarchy existing in modern GPUs. Identifying computational patterns, and their complementary best GPU mappings, has therefore become an interesting topic for the GPU computing community.

We focus on one such computational pattern, which has an hourglass-shaped active-thread pattern, combined with a stride-doubling communication pattern. From now on we will refer to this type of communication pattern as a *downsweep*. At every stage in a downsweep pattern, communication between elements (and operations) are halved, while the stride to access adjacent elements doubles. We focus our analysis on Cyclic Reduction, a tridiagonal solving algorithm that utilizes this downsweep computation pattern.

Tridiagonal linear systems arise in many scientific and engineering problems. Examples of these include spectral Poisson solvers [6], alternating direction implicit (ADI) methods [10], and numerical ocean models [4]. Typical applications may require solving up to hundreds or thousands of tridiagonal systems. Due to its importance, there have been a number of algorithms designed to solve tridiagonal systems that have been mapped to the GPU. Cyclic Reduction is one such common tridiagonal solver that fits this downsweep communication pattern.

We make two major contributions in this work. First, we analyze downsweep patterns and generalize a mapping onto the GPU that reduces shared memory communication. We discuss the tradeoffs for register usage and shared memory usage. Next we contribute a variation of the Cyclic Reduction method that utilizes this generalization and outperforms previous known tridiagonal solvers by a factor of 2–4x. This method is also able to solve much larger systems directly in a block (up to 16 times the size) without the use of global PCR splitting stages [2].

The rest of this paper is organized as follows: We will discuss the benefits and motivation of register packing in Section 2. Section 3 will review Blelloch's formulation of parallel-prefix sum [1], a simple feed-forward downsweep pattern. Section 4 will examine our case study, a GPU-based tridiagonal solver with a communication pattern that matches our requirements. Then we will consider two similar algorithms in Section 5 that do not fit our mapping. Finally we show our results and discuss the broader implications of this work.

2. REGISTER PACKING

NVIDIA GPU's memory system is organized in a hierarchical model that includes global memory, shared memory and registers. Each virtual block has access to a local set of registers and a shared memory space as well as access to global memory. Shared memory can be used in order to communicate information between threads much faster than using global memory (DRAM). This has been vital for the speedups that the GPU computing community has witnessed recently (global memory bandwidth is about an order of magnitude slower than shared memory bandwidth). However, in order to reach near-peak performance, a program must have a significantly higher number of register-to-register operations than shared memory or global memory operations. Otherwise, shared or global memory bandwidth limits performance.

Register packing refactors parallelized code to have fewer threads do more work directly in registers. It reduces the total number of threads doing work, giving each thread more register-based work. If this reorganization is done correctly,

GPGPU-4 Mar 05-05 2011, Newport Beach, CA, USA

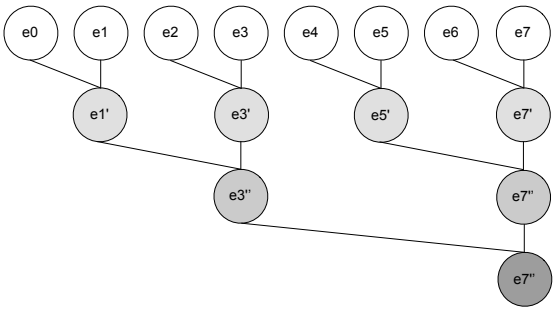


Figure 1: Communication pattern of Parallel-Prefix Sum, with a feed-forward downsweep pattern. Harris et al. [5] mapped this algorithm onto the GPU with register packing.

it can have two major benefits. First, register-register communication has much higher bandwidth than shared memory communication (nearly an order of magnitude difference in bandwidth). Also, because the amount of shared memory per block is often the major factor in determining the number of simultaneous blocks that can be run on a processor, our reduction in shared memory use can potentially increase the number of blocks that can be run at the same time. However, if we pack too much, we reduce the available thread parallelism and introduce too much register pressure, lowering occupancy and throughput.

In related work, Volkov and Demmel showed the benefits of register communication as opposed to shared memory communication in GPU-based dense linear algebra routines such as SGEMM and LU decomposition [9]. Their SGEMM implementation stores each block in shared memory, resulting in $O(n^2)$ operations from $O(n)$ loads. Volkov and Demmel hand-tuned the block sizes for their routines, and due to the high number of register arithmetic instructions per shared memory load, found that packing each thread with the maximum number of registers before overflow resulted in the best performance.

For the algorithms we address that are structured as the hourglass-sized downsweep, the benefits of register packing are immediately apparent. However, we see a tradeoff with the number of elements per thread that we pack in registers. We must preserve intermediate results in registers for future upsweep stages. Therefore, each extra element we pack adds register pressure per thread. However, if the correct balance is struck, our re-ordering helps to reduce both shared memory communication and storage, resulting in much better performance (Section 4). The communication pattern of a simple downsweep algorithm can be demonstrated with parallel-prefix sum.

3. PARALLEL-PREFIX SUM

Parallel-prefix sum (also known as scan) is a feed-forward downsweep algorithm that matches the two requirements for this computational pattern: 1) the total number of elements being shared at each iteration is halved at each stage; 2) the communication stride doubles after each iteration. Figure 1 illustrates this communication pattern, which is organized as a binary tree. Given an input array of elements, we can organize the downsweep computation in several ways.

As shown in Figure 2a, we could assign a thread to each element, and halve the number of active threads at each stage. However, this leads to maximal shared memory communication with little register usage. Harris et al.'s feed-forward downsweep implementation [5] instead packs four elements per thread (Figure 2b). Note that the first two steps of this implementation communicate only between registers, reducing the shared memory traffic. Also note that the binary-tree nature of the communication pattern implies that all communication is only in one direction (to the right).

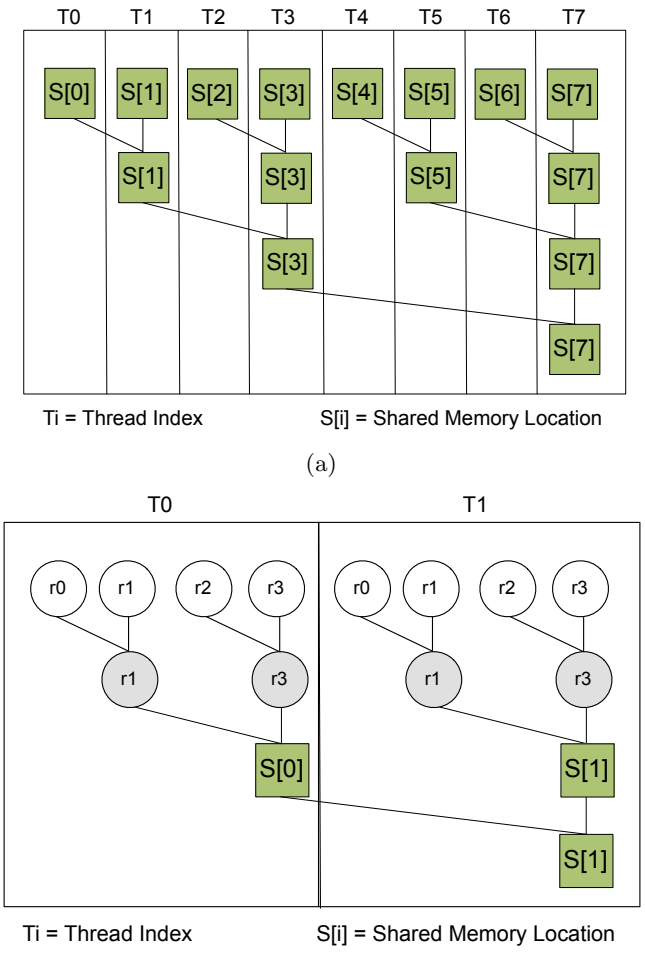


Figure 2: Comparison of a (a) non-packed and (b) 4-deep register packed implementation of Parallel-Prefix Sum. Columns hold threads; registers are in circles and shared memory is in squares. While the packed implementation may have less occupancy, it has more register communication and less shared memory communication, increasing overall throughput.

Sengupta et al. [7] addressed both scan and a cyclic-reduction-based tridiagonal solver in their work. While they were able to use this technique to improve scan's performance, they could not optimize CR because its communication pattern was more complex. CR requires bidirectional communication and a more generalized downsweep formulation, requiring barrier synchronizations and shared memory communication after each iteration to update neighboring threads

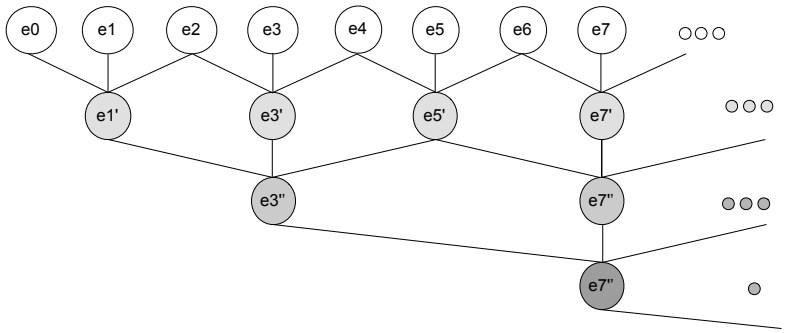


Figure 3: Communication pattern of cyclic reduction. In each stage, each element requires information from both directions. The straightforward implementation thus stores all elements in shared memory and halves the number of threads active at each step.

(Figure 2b cannot be directly applied). Our next section will discuss this method, and how we create a register-packed implementation.

4. THE CR ALGORITHM

Cyclic reduction consists of two inverted phases: the downsweep communication pattern (known as forward substitution), as well as an inverted up-sweep pattern (backward substitution). We can apply the same register packing principles we discover for the downsweep pattern, to the upsweep pattern. A tridiagonal system consists of three bands of elements (labeled a, b and c), and a solution matrix d. CR's downsweep phase consists of these sets of equations [6]:

$$
\begin{aligned}
k_1 &= \frac{a_i}{b_{i-stride}} \\
k_2 &= \frac{c_i}{b_{i+stride}} \\
a'_i &= -k_1 \cdot a_{i-stride} \\
b'_i &= b_i - k_1 \cdot c_{i-stride} \\
c'_i &= -k_2 \cdot c_{i+stride} \\
d'_i &= d_i - k_1 \cdot d_{i-stride} - k_2 \cdot d_{i+stride}
\end{aligned}
$$

Using this formulation, after every iteration, the number of active elements being updated is halved, and the stride is doubled (Figure 3). The increased efficiency of parallel-prefix sum due to register packing motivates applying the same technique to our CR implementation.

In related work, Zhang et al. [11] studied and mapped a number of tridiagonal solvers onto the GPU, including a version of CR that communicates at each stage through shared memory. It suffered from shared-memory bank conflicts, but even without these shared-memory bank conflicts, all of their solvers were shared-memory-bound. Their work used a similar communication pattern to Figure 2a, halving the number of active threads at each iteration. Göddeke et al. developed an improved version of Cyclic Reduction which matched performance to many of Zhang et al.'s other hybrid tridiagonal solvers [3].

The major contribution of this work is to extend the feed-forward register-packing method to the bidirectional communication pattern required by algorithms like CR. The bidirectional communication pattern requires a necessary shared communication and synchronization step at every stage. At every stage, border values from one side must be shared with the previous thread. As the stride doubles, the register that must be *shared* changes (r_{save}). However, the register that is being *updated* stays the same (r_{update}). This results in $O(1)$ shared memory updates in one direction at every iteration per thread, as opposed to no shared memory communication needed in parallel-prefix sum (Figure 2b).

An example of this transformation can be seen in Figure 4 for an eight-deep register packing implementation. At every stage, each thread must save its top-most register. For the first stage r_{save} is equal to r_0 (which represents the values a_0, b_0, c_0 and d_0). This value is saved so that its neighbor can update the appropriate bottom-most elements r_{update} (r_7 in our example). This is repeated at each stage for different registers until each thread only has one update left. We then use shared memory to complete the final $\log(numThreads)$ stages.

5. COUNTER-EXAMPLES: Parallel Cyclic Reduction and Fast Fourier Transform

Hybrid techniques that use Parallel Cyclic Reduction (PCR) outperform a completely CR or PCR implementation [11] because of higher step efficiency and a lack of shared memory bank conflicts. However, PCR and the similarly-structured Fast Fourier Transform are unsuitable for this register-packing technique.

Both FFT and PCR require communication between elements in a strided pattern that doubles at every iteration. For PCR, if we pack n elements into each thread, we require n communications per thread on the first step, then $2n$ on the second step, $4n$ on the third step, and so on. Register packing, then, does not significantly reduce the amount of overall communication.

FFTs can be organized into a doubled-strided $\log(n)$-step communication pattern. Volkov et al. [8] packed small sets of elements into each thread, with each thread independently solving small FFT routines. Afterwards, the intermediate results were re-shuffled through shared memory, and the process was repeated until the FFT was complete. However, this method still requires $O(n)$ shared memory shuffles after each mini-FFT is complete, and is also a poor match for register packing.

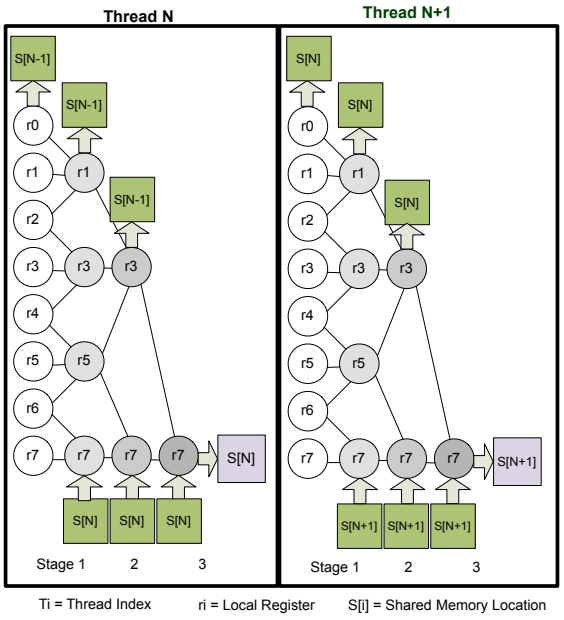
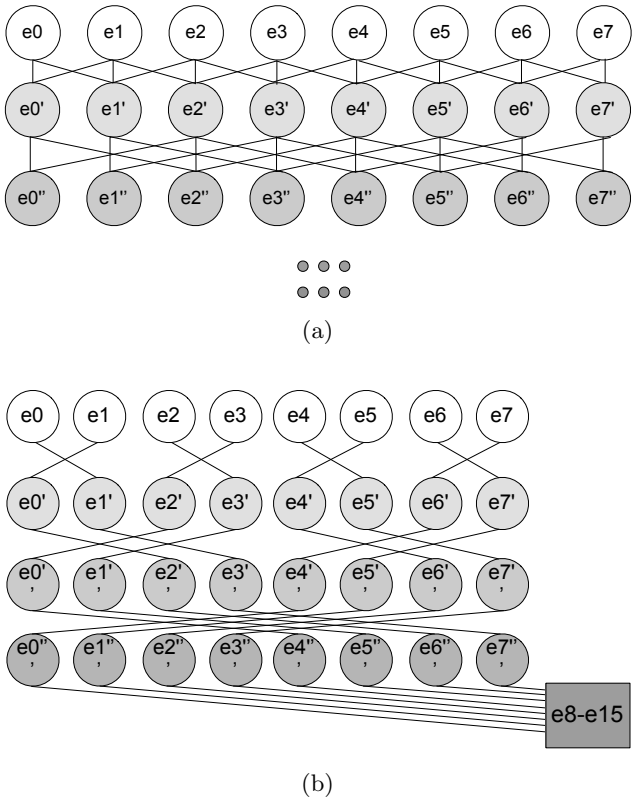


Figure 4: Our reworked mapping of the CR algorithm onto the GPU utilizes more elements in registers (8 sets per thread in this example) and less shared memory communication(3 words per thread). At every iteration, we must store r_{save} (r_0 in stage 1, r_1 in stage 2, etc.) and update r_{update} (r_7 in this case).



(a)

(b)

Figure 5: Communication pattern of (a) Parallel Cyclic Reduction (PCR) and (b) Fast Fourier Transform (FFT). These methods have a stride-doubling communication pattern, but no hourglass shape. Therefore shared memory requirements are $O(n)$ and we are unable to apply similar register packing techniques.

6. TRADEOFF ANALYSIS: Registers vs Shared Memory

Though register communication is much faster than shared communication and therefore generally preferred, we must consider the decreased occupancy and increased register pressure that results from having fewer threads. Previous CR methods saturated the shared memory bandwidth, so register packing can potentially improve performance. We analyze the tradeoffs in using register packing with a model-based approach, and compare against kernels with varying register packing depths.

6.1 Model Based

We wish to model three things: register usage per thread (which will limit occupancy), shared memory per block (which will limit the size of the system we can process), and shared memory communication per block (which will limit performance). As highlighted in Section 2, in order to get close to the optimal throughput, we must have enough register operations to amortize the cost of shared memory communication.

We model each initial thread as having a register penalty of α for temporary registers while processing each down-sweep stage. For each set of intermediate values (packing depth) needed, we apply a cost of β registers. Therefore, the number of registers needed for a given variation of our register packing technique becomes:

$$\text{Number of Registers} = \alpha + \beta \cdot \text{Packing Depth} \quad (1)$$

At every stage, each thread needs to share one set of elements with one other thread. Therefore the amount of shared memory needed and shared memory communication being performed at every stage should scale linearly with the number of threads:

$$\text{Shared Memory Per Block} = \gamma \cdot \text{Number of Threads} \quad (2)$$

$$\text{Shared Memory BW Per Stage} = \lambda \cdot \text{Number of Threads} \quad (3)$$

From these equations we can assume that as we increase the number of elements processed per thread (and decrease the total number of threads), we decrease shared-memory communication at the cost of extra registers per thread. Communication through registers gives us potential speedup, but if shared-memory communication is already significantly hidden by register operations, increasing packing depth may introduce register pressure and lower occupancy, resulting in lower performance.

6.2 Experimental Confirmation

Table 1 shows the results from examining NVIDIA's compiler (nvcc)-generated cubin files, and the amount of allocated shared memory. All of these were generated from compiling and running on a NVIDIA GTX 460 (Fermi) us-

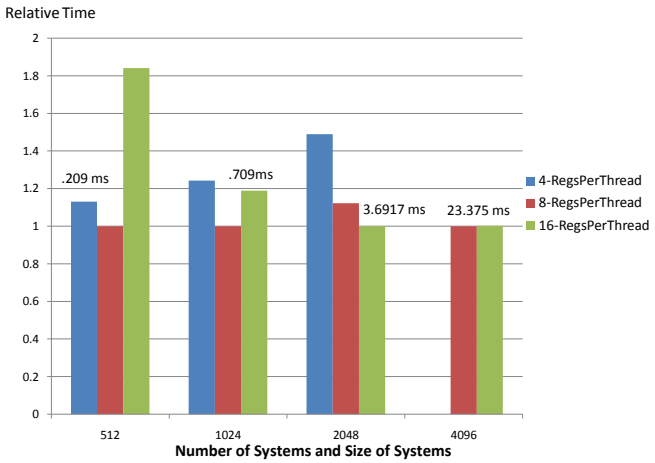


Figure 6: Experimental results showing performance as we vary the amount of register packing per thread, normalized to the highest performing variation. Compare with Table 1 for register usage and shared memory usage per test.

ing CUDA 3.1. As we assumed, shared memory increases linearly with the number of threads, but decreases with the number of elements processed per thread.

Our model approach appears to map quite well from our limited sample data. Using our register prediction model, we find that $\alpha \approx 15$ and $\beta \approx 4$. From this data, we see that moving from four elements to eight elements per thread increases register usage per thread by only about 50% while halving the amount of shared memory needed for storage and communication. However, if we again double the number of elements packed in registers, α plays less of a role for further packing (sixteen elements per thread), and register usage nearly doubles per thread, threatening the total possible occupancy on our GTX 460.

From the timing results, on this GPU, we see that in general the sweet spot is eight elements processed per thread. At this point, the reduction of registers from the next stage (sixteen elements) is nearly linear, and having fewer registers per thread leads to less register pressure. However, this may change with future cards, and we believe provides proper motivation for future auto-tuning work.

7. EXPERIMENTAL RESULTS

We ran a set of benchmarks comparing our CR register-packing method against a set of other GPU tridiagonal solvers. Our testing environment used an NVIDIA GTX 460 with CUDA 3.1. Our results indicate that using our register-packing method on our Cyclic Reduction algorithm proved effective. Packing more elements into registers also allows us to solve larger systems without using global memory to split systems. For systems that now can fit within a single block, this represents a significant performance boost.

For systems larger than 1024 elements, previous methods run into a shared-memory size limit. Therefore, we instead compared against a hierarchical PCR-Thomas method that splits larger systems into smaller chunks and then processes each chunk separately [2]. Figure 7 shows our performance versus a set of GPU tridiagonal solvers. Our CR-register-packed version outperforms all of these methods (even hybrid step-efficient methods such as CR-PCR and PCR-Thomas). Our method outperforms all instances, with a minimum speedup of about 50%, and a maximum speedup of 4.5x against non-hybrid methods such as a shared-memory-bound CR or PCR.

System Size	CR Depth (Elements)	Shared Mem Per Block	Regs Per Thread	Number of Threads
512	4 Elements	2.5 KB	31	128
	8	1.25 KB	47	64
	16	.625 KB	80	32
1024	4	5 KB	31	256
	8	2.5 KB	47	128
	16	1.25 KB	80	64
2048	4	10 KB	31	512
	8	5 KB	47	256
	16	2.5 KB	80	128
4096	4	20 KB	31	1024
	8	10 KB	47	512
	16	5 KB	80	256

Table 1: Shared memory usage, register usage, and thread count for each of our implementations given varying system sizes.

8. DISCUSSION

As our model showed in Section 6, the initial cost α for registers was one of the key motivations for packing more elements locally. However, as we pack more elements, the benefits become less significant and the penalty in register pressure begins to grow. For our application and our GPU, we found the sweet spot to be around a depth of eight.

We emphasize that the best depth is both algorithmic-dependent and machine-dependent. Algorithms that require more local communication per shared (or global) memory read will have a peak-performance advantage, and may benefit from more packing. Tuning algorithms can help to identify the best depth for a variety of algorithms and platforms, though we do not address this subject in our work.

The Downsweep Pattern. In this work we characterize the downsweep pattern as an hourglass shape that is halved in size at each iteration. However, we can extend this definition to patterns with a larger radix. For example, Figure 8 shows communication patterns for radixes 2 and 4. The key requirement for our technique here is that the shared memory communication must remain constant per stage.

9. CONCLUSION

In this work we analyzed a specific communication pattern and studied its complementary mapping onto the GPU. We used a Cyclic Reduction tridiagonal solver as our case study, one that before this work had shown poor performance on GPUs when compared to step-efficient or hybrid methods. However, with our generalized downsweep mapping, we were able to substantially improve on CR's performance, making it faster than other known current GPU tridiagonal solvers.

We believe this mapping and generalization can be useful

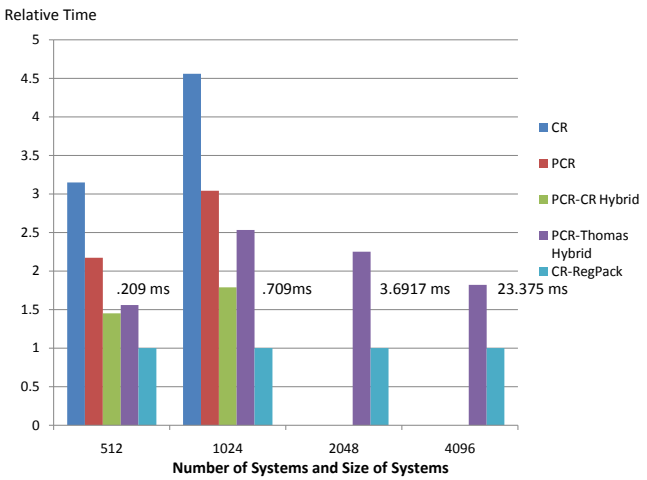


Figure 7: Experimental results comparing our CR with packed registers against the previous highest-performing GPU tridiagonal solvers, normalized to our register-packing implementation. Due to shared-memory limitations, the CR and PCR solvers developed by Zhang et al. cannot be applied to larger systems. Therefore, we instead compared against a PCR-Thomas hybrid method designed for large systems.

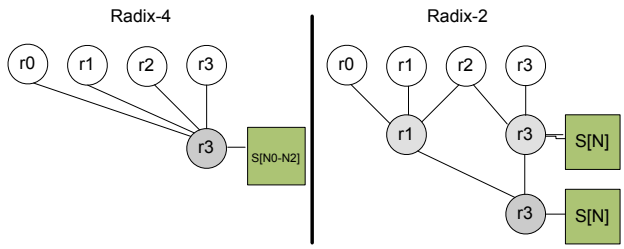


Figure 8: We can apply the same methodology to a variety of downsweep types. Pictured here are conceptualized versions of a radix-2 and radix-4 downsweep.

for all algorithms that have this downsweep pattern. However, the core motivation for our method (reduced shared communication, higher register throughput) can be applied to a number of communication patterns. Therefore we feel identifying and classifying sets of communication patterns, and then analyzing their optimal mapping to the GPU is an important field of research. Once some of these patterns and mappings are better understood, it will be possible to develop automated tuning methods for them to optimize register re-usage and shared memory communication.

10. ACKNOWLEDGEMENTS

The authors would like to thank Dominik Göddeke for insightful feedback on earlier versions of this paper. Thanks also to the SciDAC Institute for Ultrascale Visualization and the National Science Foundation (Awards 0541448, 1017399, and 1032859) for funding, and to NVIDIA for equipment donations.

References

[1] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[2] Andrew Davidson, Yao Zhang, and John D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2011.

[3] Dominik Göddeke and Robert Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):22–32, January 2011.

[4] G. R. Halliwell. Evaluation of vertical coordinate and vertical mixing algorithms in the HYbrid-Coordinate Ocean Model (HYCOM). *Ocean Modelling*, 7:285–322, 2004.

[5] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Herbert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[6] R.W. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM*, 12(1):95–113, 1965.

[7] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106, August 2007.

[8] V. Volkov and B. Kazian. Fitting FFT onto the G80 architecture. Technical report, University of California at Berkeley, 2008.

[9] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, 2008.

[10] Ting-Yuan Wang and Charlie Chung-Ping Chen. Thermal-ADI—a linear-time chip-level dynamic thermal-simulation algorithm based on alternating-direction-implicit (ADI) method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(4):691–700, August 2003.

[11] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, pages 127–136, January 2010.