Improving Memory Subsystem Performance using ViVA: Virtual Vector Architecture

Joseph Gebis[1][2],Leonid Oliker[1][2], John Shalf[1], Samuel Williams[1][2],Katherine Yelick[1][2]

[1] CRD/NERSC, Lawrence Berkeley National Laboratory Berkeley, CA 94720
[2] CS Division, University of California at Berkeley, Berkeley, CA 94720
{*JGebis, LOliker, JShalf, SWWilliams, KAYelick*}*@lbl.gov*

Abstract. The disparity between microprocessor clock frequencies and memory latency is a primary reason why many demanding applications run well below peak achievable performance. Software controlled scratchpad memories, such as the Cell local store, attempt to ameliorate this discrepancy by enabling precise control over memory movement; however, scratchpad technology confronts the programmer and compiler with an unfamiliar and difficult programming model. In this work, we present the Virtual Vector Architecture (ViVA), which combines the memory semantics of vector computers with a software-controlled scratchpad memory in order to provide a more effective and practical approach to latency hiding. ViVA requires minimal changes to the core design and could thus be easily integrated with conventional processor cores. To validate our approach, we implemented ViVA on the Mambo cycle-accurate full system simulator, which was carefully calibrated to match the performance on our underlying PowerPC Apple G5 architecture. Results show that ViVA is able to deliver significant performance benefits over scalar techniques for a variety of memory access patterns as well as two important memory-bound compact kernels, corner turn and sparse matrix-vector multiplication — achieving 2x–13x improvement compared the scalar version. Overall, our preliminary ViVA exploration points to a promising approach for improving application performance on leading microprocessors with minimal design and complexity costs, in a power efficient manner.

1 Introduction

As we enter the era of billion transistor chips, computer architects face significant challenges in effectively harnessing the large amount of computational potential available in modern CMOS technology. Although there has been enormous growth in microprocessor clock frequencies over the past decade, memory latency and latency hiding techniques have not improved commensurately. The increasing gap between processor and memory speeds is a well-known problem in computer architecture, with peak processor performance improving at a rate of 55% per year, while DRAM latencies and bandwidths improve at only 6% and 30% respectively [13]. To mask memory latencies, current high-end computers now demand up to 25 times the number of overlapped operations required of supercomputers 30 years ago. This "memory wall" is a primary reason why high-end applications cannot saturate the system's available memory bandwidth, resulting in delivered performance that is far below peak capability of the system.

Numerous techniques have been devised to hide memory latency, including out-of-order superscalar instruction processing, speculative execution, hardware multithreading, and stream prefetching engines; nevertheless, these approaches significantly increase core complexity and power requirements [9, 12] (the "power wall") while offering only modest performance benefits. This is particularly true of irregularly-structured and data-intensive codes, which exhibit poor temporal locality and receive little benefit from the automatically managed caches of conventional microarchitectures. Furthermore, a significant fraction of scientific codes are characterized by predictable data parallelism that could be exploited at compile time with properly structured program semantics; superscalar processors can often exploit this parallelism, but their generality leads to high costs in chip area and power, which in turn limit the degree of parallelism. This benefit is as important for multicore chips as it is for chips in the area of exponential clock frequency scaling.

Two effective approaches to hiding memory latency are vector architectures [5] and software controlled memories [8]. These techniques are able to exploit regularity in data access patterns far more effectively than existing prefetching methods using minimal hardware complexity. However, vector core designs are costly due to the limited market and limited applicability, while software controlled memories require radical restructuring of code and the programming model, and are currently incompatible with conventional cache hierarchies.

In this work we extend the Virtual Vector Architecture (ViVA[1]), which combines these two concepts to achieve a more effective and practical approach to latency hiding. ViVA offers the hardware simplicity of software controlled memory hardware implementations, with familiar vector semantics that are amenable to existing vectorizing compiler technology. Additionally, our approach can coexist with conventional cache hierarchies and requires minimal changes to the processor core, allowing it to be easily integrated with modern microprocessor designs, in a power-efficient fashion.

Overall results, measured by a series of microbenchmarks as well as two compact numerical kernels, show that ViVA offers superior performance compared to a microprocessor using conventional hardware and software prefetch strategies. ViVA thus offers a promising, cost-effective approach for improving latency tolerance of future scalar processor chip designs, while employing a familiar programming paradigm that is amenable to existing compiler technology.

2 ViVA Architecture and Implementation

In this section we present the ViVA programming model and design philosophy. As shown in Figure 1, ViVA adds a software-controlled memory buffer to traditional microprocessors. The new buffer logically sits between the L2 cache and the microprocessor core, in parallel with the L1 cache. Block transfer operations move data between DRAM and the ViVA buffer, and scalar operations move individual elements between the ViVA buffer and existing scalar registers in the microprocessor core. Extensive details of the ViVA infrastructure are provided in a recent PhD thesis [6].

[1] The ViVA acronym was developed with IBM during the BluePlanet collaboration [4]

Programming Model The block transfers, between DRAM and the ViVA buffer, are performed with new instructions that have vector memory semantics: unit-stride, strided, and indexed (gather/ scatter) are all supported. In order to fully take advantage of the benefits of ViVA, and maximize the amount of concurrency available to the memory system, most programs should use ViVA in a double-buffered approach whenever possible. A thorough exploration of techniques is given in [6].

The basic ViVA programming model is conceptually simple:

```
do_vector_loads;
for(all_vector_elements) {
   transfer_element_to_scalar_reg;
   operate_on_element;
   transfer_element_to_buffer;
}
do_vector_stores;
```

A major advantage of the ViVA approach is that it can leverage vector compiler technology with only minor modifications, since the new ViVA instructions have vector memory semantics. The compiler can generate regular vector code, with one straightforward exception: the arithmetic vector operations have to be replaced with a scalar loop that iterates over a vector's worth of elements. Since vector compilers are a mature, well-understood technology, real applications could benefit from ViVA immediately.

VIVA Buffer Hardware Details Our approach allows the ViVA buffer to act as a set of vector registers, but without the associated datapaths that accompany registers in full vector computers. This reflects one of the main goals of ViVA: efficient memory transfer at low cost. Since there are no datapaths associated with the ViVA buffer, no arithmetic operations can be directly performed on elements stored within the buffer. In order to perform arithmetic operations, elements must first be transferred to existing scalar (integer or floating-point) registers.

As with most vector register files, the ViVA register length is not fixed in the ISA. Instead, control registers are used for hardware to describe the lengths of the registers to software. This allows a single binary executable to run on different hardware implementations. Thus, a low-cost ViVA design may have shorter hardware register lengths, while a higher-performance version may have longer registers. In our experiments, we study hardware registers lengths that vary from sixteen 64-bit words through 256 words, with most experiments using 64 words (typical of many traditional vector computers). Thus the ViVA buffer would require total storage of 16KB, approximately the same modest chip area as L1 data cache of the PowerPC G5 (used in the experiments of Section 4).

The ViVA buffer is logically positioned between the core and the L2 cache. In the system we model, no additional ports are added to the L2 cache. Instead, the cache arbiter is modified slightly to add ViVA requests to the collection of other types of requests that it prioritizes and presents to the L2 cache: demand and prefetch requests for both the L1 data and instruction caches. Figure 1(b) and (c) show the operation request and data flow for loads and stores performed with traditional scalar accesses, as well as with ViVA accesses; thin arrows correspond to requests alone (that don't need to transfer data), while thick arrows correspond to actual data flow. This flow diagram

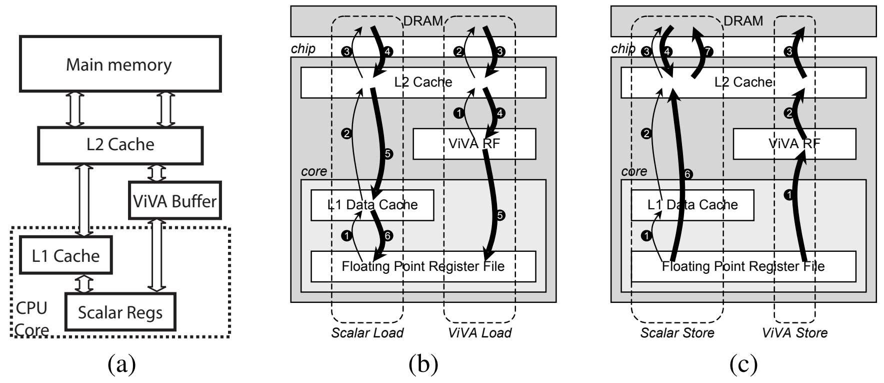


Fig. 1: ViVA overview showing (a) existing memory hierarchy with ViVA buffer, and data flow between DRAM and registers with and without ViVA for (b) loads and (c) stores. Requests without associated data are shown with thin arrows, and data transfer is shown with thick arrows.

clearly shows ViVA's potential to reduce memory traffic and requests compared with the default scalar processor.

In many ways, the ViVA buffer acts as a vector register file and is logically split into registers, which are used to identify the target or source for transfers. As a result, no coherence mechanisms are required in the buffer. An element loaded into the ViVA buffer will not be updated by scalar stores, much as an element that has been loaded into a scalar register will not be updated by scalar stores. Once values are stored from the ViVA buffer into the L2 cache, regular memory consistency models apply. No consistency orderings are guaranteed between ViVA and scalar stores; memory fences are required to enforce a particular order.

Because the ViVA buffer is treated as a vector register file, the lack of automatic coherence between values in the buffer and memory values is not problematic. Vector compiler technology can manage the coherence in the same manner as with traditional vector machines. In some cases, scalar and vector memory operations in the same program can be mixed without requiring the use of memory fences. As long as no single memory location is written by one type of operation (e.g., a non-ViVA scalar store) and then read by the other type (e.g., a vector load), a memory fence is not needed. This memory model tends to work well for vectorizable applications.

The out-of-order handling of the ViVA buffer works similarly to the handling of scalar out-of-order registers, with a few small differences. As with scalar registers, a number of extra physical registers are used in the system — in ViVA's case, that means that the physical buffer is larger than the visible state. The system keeps track of both committed state, as well as current state; in the case of an exception, the processor can recover the committed state and begin re-processing from there.

New instructions and control registers The second major component to ViVA is a small set of new instructions that are added to the processor's ISA. The first class of new instructions performs vector memory transfers between DRAM and the ViVA buffer; the second class performs scalar transfers between the ViVA buffer and scalar registers. The third and final class of new instructions contains only two operations: one that stores a value from a general-purpose register to a ViVA control register, and a complementary instruction that reads values from control registers.

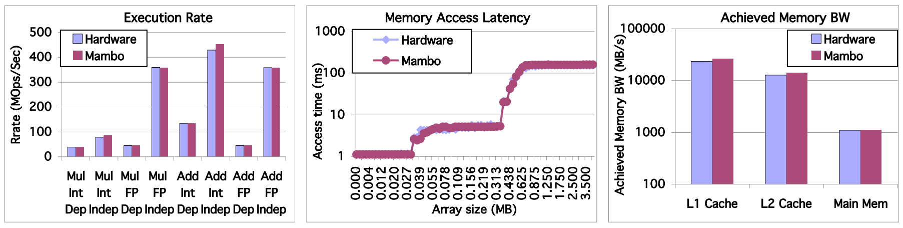
ViVA has a small number of control registers: two that control the lengths of vectors, and one for strides. The first length register is the maximum vector length (MVL) — this is a hardwired read-only control register, that contains the physical length of vector registers on the current machine. The MVL's complement is the vector length (VL). Programs write a value (up to MVL) to VL, and vector instructions are only processed to that length. By writing a smaller value to VL, a program can run instructions on vectors that are shorter than the physical vector registers. The stride register sets the distance between consecutive elements for strided memory operations.

3 Experimental Platform

The ViVA simulator is based on Mambo, a cycle-accurate full-system simulator developed and used by IBM [3]. We modified the original version of Mambo to allow us to model various configurations of ViVA. Approximately two years of graduate student effort was required to modify the simulator, and to test the ViVA extensions. A detailed description of the process is presented in [6].

The Mambo simulator used for our experiment was designed to model PowerPC systems, and it was carefully calibrated to simulate our Apple G5 hardware platform. This architecture contains a 2.7 GHz IBM PowerPC 970FX CPU, PC-32000 DDR memory with 6.4 GB/s of memory bandwidth, 5.4 GB/s of North Bridge bandwidth (each direction), and a load-to-use latency of approximately 156 ns. The overall structure of the modeled system is generally similar to platforms that include Intel or AMD processors; while the specific details may differ, the general out-of-order processing, memory configuration, and operation closely resemble our evaluated platform. Thus, we expect many of the insights gained in this study to be applicable to broad range of modern microprocessor technologies.

In order to calibrate the simulator, we compared the performance of the real and simulated systems running a variety of existing and targetted custom benchmarks. The first calibration benchmark uses a variety of arithmetic operations within small loops, including: fixed- and floating-point operations, adds and multiplies, independent (to test maximum throughput) and dependent (to test ALU latency) operations. Next, we examined memory system latency at various levels using a version of the lat_mem_rd code from the LMbench [11] benchmark suite. The benchmark creates a linked list of pointers in an array of a particular size, and then measures the average time it takes to read each element. Finally, we calibrate the memory bandwidth using a benchmark that streams data out of arrays, which are sized to fit within the various levels of the memory hierarchy. Figure 2(a–c) shows simulated performance compared with the G5 hardware for the arithmetic, latency, and bandwidth calibration benchmarks respectively. Observe that in general all configurations closely match the actual hardware performance, giving us high confidence in our simulation results.



(a) (b) (c)

Fig. 2: Calibration results comparing G5 hardware and Mambo for (a) various arithmetic operations, (b) memory latency times for pointer-chasing and (c) achieved memory bandwidth for varying array sizes.

ViVA Programming In a full system implementation, ViVA would be programmed much as vector computers are programmed today: a vector compiler would vectorize the loops and structure the code appropriately (using stripmining, etc). The ViVA compiler would require the additional straightforward step (included as the last part of a vector compilation phase) of adding a scalar loop, to replace traditional vector arithmetic instructions.

Since this effort focuses on ViVA's proof of concept, we use assembly language programming as the first step for conducting our experiments. Individual functions are written in assembly, and then called from the main C code. Note that no special assembly optimizations were used — in general, the code is a direct translation of the kernel into assembly, with the ViVA buffers being used in a double-buffered approach (values loaded into half of the registers, while the other half were used for calculations).

4 Performance Evaluation

We now examine performance results using ViVA within the Mambo simulator against the conventional hardware, using several microbenchmarks and two frequently used compact kernels. ViVA's target applications include memory-intensive programs that are common in traditional scientific computing. Such applications are also becoming increasingly important as drivers of desktop and handheld systems, in the form of recognition, synthesis, and other media processing programs. Because the simulator models a complete system, its slowdown prohibits the full execution of large applications in a timely manner, but the benchmark codes we use exercise the memory system in a manner that mirrors that of our target workload.

4.1 Microbenchmarks

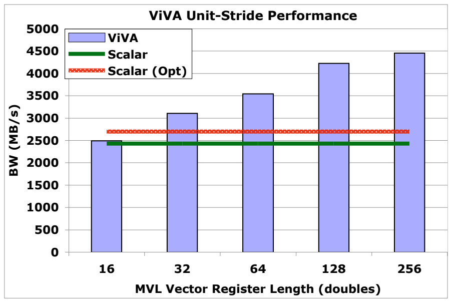
In this section, we examine four memory microbenchmarks that provide insights into performance of numerous applications. We start with the unit-stride stream triad benchmark, which takes one dense vector, scales it by a constant, adds it with a second, and stores to a third, executing the loop with the body: *z[i] = x[i]* $\times$ *factor* + *y[i]*. The parallelism expressed to the memory subsystem is equal to the array size. Since accesses are in unit stride, all data within each cache line are used. The next experiment explores strided memory accesses. The code is the same as the unit-stride benchmark, except that it only operates on elements spaced apart a constant displacement (called the stride). Once again, the memory level parallelism is as large as the number of elements accessed.

We then examine the microbenchmark: *z[i] = values[index[i]]*, which loops over indexed accesses. The indices point to random locations within the value array, where some values may be retrieved more than once, and some may not be retrieved at all. The cache is cleared before each run, and the index array is streamed through exactly once. Finally, we investigate a microbenchmark that mimics the memory access patterns for blocking optimizations, used in such computations as dense matrices, structured grids, and FFTs. This fourth kernel performs a *stanza triad*, where a given length stanza is accessed in a unit-stride fashion, followed by a jump to another memory address. Note that memory operations within and across stanzas are independent.

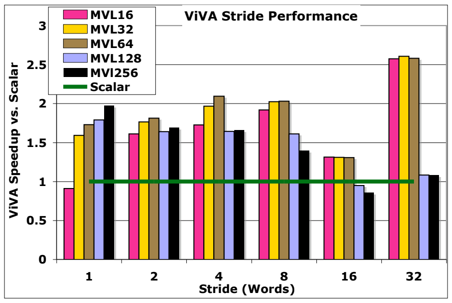
Unit-Stride Performance Results Figure 3(a) presents the unit-stride triad benchmark, showing performance of a straightforward scalar implementation, an optimized scalar implementation, and ViVA with a variety of MVLs. Observe that for a small MVL=16, performance is comparable to the scalar rate. This highlights that, unlike traditional vector platforms, we do not see significant penalties for small vector length accesses. Additionally, as MVL increases we see significant performance benefits — more than 1.8x at MVL=256.

It may seem counterintuitive that the ViVA approach can outperform the scalar core for long unit-stride accesses, as the G5's hardware prefetcher is designed to optimize memory access patterns of this kind. However, the G5 prefetcher works on physical memory addresses, which limits its ability to fetch across page boundaries. Thus, between page boundaries prefetching must ramp up streaming accesses before it can reach a steady state. ViVA, on the other hand, does not have these constraints, allowing it to request address across numerous pages. Additionally, a traditional hardware prefetcher accesses a fixed number of lines ahead, a decision that could not possibly account for the ultimate microprocessor's clock speed, memory type, etc. Finally, a hardware prefetcher in steady state is limited to prefetching lines from DRAM one at a time as the program submits new demand requests, to avoid cache pollution. However, the ViVA approach can submit multiple line requests at once, as it is known a priori that all of the requested values are actually needed.

Finally, ViVA has another advantage for the case of storing a full cache line at once. Typically, stores to lines that are not present in the L2 cache are required to first be filled. ViVA is able to express full cache line writes directly to the processor, thus not requiring a memory fetch for the fill. In principle a processor can perform the same optimization for scalar stores, but the cache organization makes this unlikely for most cases. Special instructions do exist on some architectures to avoid a cache fill, such as the PowerPC *dcbz* instruction that zeros an entire cache line. However, these result in relatively small performance improvements, as can be seen in the optimized scalar



(a)



(b)

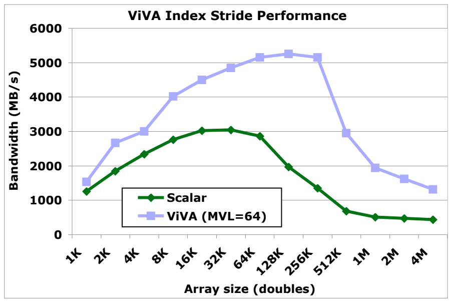
Fig. 3: Scalar vs. ViVA (a) bandwidth of unit-stride, for varying MVLs and optimized scalar code (*dcbz*, software prefetching, manual unrolling) and (b) speedup relative to scalar for varying strides and ViVA MVL (optimization of the scalar strided version did not improve performance).

behavior of Figure 3(a), which includes the use of the *dcbz* instructions, as well as software prefetching, manual unrolling, and a variety of compiler flags.

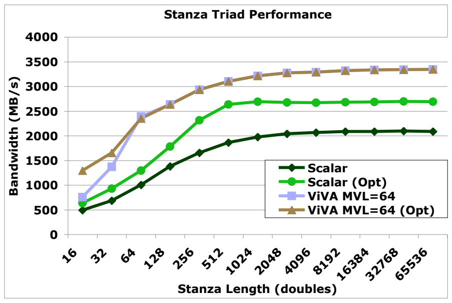
Strided Performance Results Figure 3(b) shows the speedup of ViVA compared to a default scalar version for varying strides; attempts to optimize the scalar code using *dcbz*, software prefetching, manual unrolling, and compiler flags did not improve performance. Observe that in almost every case, ViVA is able to deliver a significant improvement over the scalar implementations, attaining a 2.5x improvement for stride=32.

The results for longer MVLs shows the effects of two opposing trends. Longer MVLs allow more concurrency to be expressed to the memory subsystem, thereby increasing the memory bandwidth potential. However, long MVLs also increase the range of memory addresses touched by a single instruction, especially for operations with long strides. Thus, long MVLs can actually reduce performance, while consuming more register real estate and power. Due to these considerations we choose an MVL of 64 doubles as a "sweetspot" of these tradeoffs, and conduct the remainder of our experiments with this parameter.

Indexed Performance Results Figure 4(a) shows the scalar and ViVA bandwidth rate of the indexed microbenchmark for varying array sizes; attempts to optimize the scalar version did not improve performance. At the smallest array size of 1 KB (doubles) ViVA shows only a slight advantage compared with the default hardware. Both implementations improve in performance as the array size grows, but ViVA shows a clear advantage, achieving up to 4.3x speedup at 512KB arrays. In the scalar case, unit-stride data streams (for both stores, and loads of indices) become longer with larger arrays, allowing the hardware prefetcher to ramp up to its full potential – however, the hardware prefetcher is not able to provide any benefit for value elements. For ViVA, larger array sizes correspond to more available parallelism presented to the memory subsystem, for all loads and stores. As the array sizes become much larger than the cache there is a



(a)



(b)

Fig. 4: Performance comparison of scalar versus ViVA for (a) indexed microbenchmark (optimizing the scalar indexed version did not improve performance) and (b) stanza triad where "ViVA (Opt)" uses indexed accesses to transfer multiple short stanzas with a single vector instruction.

high probability that fetched cache lines will evict subsequently required data, resulting in reduced performance. Additionally, as the total data size becomes much larger than the amount of memory covered by the TLB, more page translation overhead is required.

Stanza Triad Performance Results Figure 4(b) presents the bandwidths of varying stanza lengths for the G5 scalar processor (default and optimized) and ViVA (default and optimized). For both scalar lines, performance continues to improve with increasing stanza lengths as the prefetcher ramps up and becomes fully engaged. The hardware prefetcher reaches its steady-state behavior after five consecutive cache lines are accessed, equal to 80 doubles. Performance continues to improve past stanzas of 80 doubles because various overheads (such as prefetch ramp-up, TLB misses and page translation) are amortized over the full length of the stream.

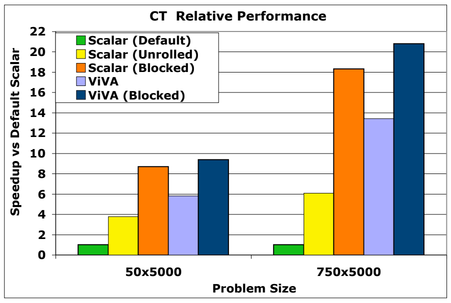
Figure 4(b) also shows that the ViVA implementation achieves better performance for all stanza lengths. At short stanzas, the vector benefits are limited since each instruction expresses less parallelism to the memory system. As stanza lengths increase so does the amount of parallelism expressed by a single ViVA instruction. As with scalar stanza triad, bandwidth continues to increase, as page translation costs are amortized over the number of elements accessed in the page.

4.2 Compact Kernel: Corner Turn

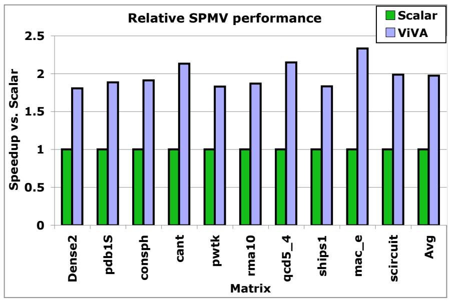
Having explored ViVA's microbenchmark behavior, we now examine the behavior of two compact kernels. The corner turn (CT) operation is frequently used in signal and image processing applications that operate on multi-dimensional data in multiple stages. An example where this is required is certain filtering operations followed by a beamforming computations [10]. CT's pseudocode is simply:

for(i = 0 to row_length) { *for(j = 0 to col_length)* { *out[j][i] = in[i][j];* } }

The idea behind this data-intensive kernel is to preserve data locality in the dimension



(a)



(b)

Fig. 5: Speedup over default scalar version for (a) CT for two (out of cache) problem sizes, showing unrolled scalar and ViVA performance and (b) SPMV for a variety of matrix structures.

being operated on by performing a matrix transpose, thus creating enormous pressure on the memory subsystem. This kernel depends on two memory access patterns — unit-stride and strided accesses — both of which were highlighted in our previous microbenchmarks. For larger sizes, the unit-stride reads can benefit from spatial locality and hardware prefetching, but the strided accesses will likely be in different cache lines and memory pages.

Figure 5(a) presents relative performance of the default and unrolled scalar codes as well as the ViVA version. The unrolled scalar implementation sees an improvement of approximately 4x and 6x for the small and large arrays (respectively). This speedup is seen because (after the initial fetch) strided loads access the same page (and potentially the same cache line). The ViVA results show even more impressive performance, achieving a 5x and 13x improvement for the small and large arrays. The ViVA strided accesses get the same benefit from page and cache-line reuse as the unrolled scalar implementation, however, overall delivered bandwidth is higher because ViVA is able to express more parallelism to the L2 cache. The relative advantage of ViVA compared to the unrolled scalar code, is consistent with the trends of the unit-stride and strided microbenchmarks (Figure 3).

Performance of CT using cache blocking, with and without ViVA, is also shown in Figure 5(a). Note that blocking is an algorithmic transformation that is normally not performed by a compiler, but instead requires hand optimization. Results show that (as expected) the blocked scalar version significantly outperforms the default CT. However, the blocked ViVA implementation delivers an additional performance improvements compared to the blocked scalar code (8% and 14% for the small and large test case). This demonstrates that ViVA can provide a performance boost either as a stand-alone approach or in conjunction with traditional algorithmic optimization techniques.

4.3 Compact Kernel: Sparse Matrix Vector Multiplication

We now examine the more complex memory access patterns associated with SpMV, an important kernel that dominates the performance of diverse applications in scientific and engineering computing, economic modeling and information retrieval. The sparse matrix structure is primarily filled with zero valued elements, which neither need to be stored nor computed on. As a result, significant instructions and meta-data are required to correctly index the vector per floating point operation [14]; thus, conventional implementations have historically delivered less than 10% of machine peak on single-core cache-based microprocessor systems [15]. Extensive research has been conducted to improve performance of this kernel, including code and data structure transformations and sophisticated auto-tuning libraries [15]. The standard scalar implementation utilizes the compressed sparse row (CSR) format. CSR stores nonzeros by encoding their values and columns, and an indexed operation is used to access the source vector. Additionally, an array is created to specify the first and last nonzero for each row.

The ViVA implementation follows the segmented scan algorithm [2], which strip-mines the matrix into vectors that may straddle multiple rows. For each of these vectors of nonzeros, ViVA uses unit-stride loads for the values and column indices, and an indexed load to access the source vector. Once the data is in the ViVA registers, it processes nonzeros using regular scalar FPU operations. Finally, at the end of a row, the sum is stored using scalar accesses. These two phases can be double buffered by loading a group of vectors while processing a previous group.

To evaluate SPMV performance, we examine a variety of matrix structures and non-zero patterns from actual physical simulations [15]. Figure 5(b) presents the ViVA speedup compared with the scalar version for each studied matrix. Results show that ViVA delivers an average of 2x performance (right-most values) compared with the scalar code. Although raw performance drops for poorly structured matrices with large vectors — which have trouble exploiting L2 temporal locality — ViVA consistently outperforms the scalar version for a wide variety of underlying matrix structures.

5 Related Work

A number of strategies have been employed to balance Little's Law — which states that the number of outstanding memory requests in progress must match the product of the memory latency and the available memory bandwidth [1] — by increasing the number of concurrent requests presented to the memory subsystem. Different approaches (detailed in [6]) explored in high-performance computing, microprocessors, and embedded computers [7], include software prefetching, hardware (stream) prefetching, out-of-order instruction processing, multithreaded and multicore processors, vector architectures, and software controlled memories. Nonetheless, the "memory wall" problem generally continues to be exacerbated between successive microprocessor generations.

We summarize leading approaches in Table 1, which compares a variety of hardware and software techniques to hiding memory latency (using the U.S. A–F grading system). In the first three columns we note whether the technique is effective in hiding DRAM latency for the specified memory access pattern. We also qualitatively note the VLSI design effort required to implement such a solution — ViVA being a memory external to the core requires relatively little work.

Perhaps one of the most important metrics for techniques which require software changes is the compiler technology: in terms of both the compiler complexity as well as

	Effectiveness			integration	compiler		user
	unit	strided	indexed	complexity[1]	complexity	maturity	programmability
Out-of-order	D	D	D	D	D	n/a	A
HW Prefetch	A	B	F	B	A	n/a	A
SW Prefetch	B	C	D	A	B	C	D
Multithreaded	B	B	B	C	D[2]	D	C
DMA/Local Store	A	B	B	B	D	F	D
Vector	A	A	B	C	D	B	B
ViVA	A	A	B	B	D	B[3]	B

Table 1: Qualitative comparison of DRAM latency hiding techniques using the A-F grading system. [1]Integration complexity compared versus unithreaded, in-order core. [2]For auto-parallelization. [3]Leveraging existing vector compilers.

the current maturity of this compilers technology. Vectorizing compilers, which ViVA leverages, are decades old, and well established technology. The maturity of many other compilation techniques is relatively poor. Finally we note the programmability, under an ideal programming model, for each techniques. For example, multithreading works well on multithreaded or parallelized codes; similarly, programmers are productive on Vector and ViVA architectures when implementing data parallel codes. Clearly, when compared to other techniques over a multidimensional analysis, ViVA provides a very attractive solution to hiding memory latency.

6 Summary and Conclusions

In this work we present ViVA, which incorporates the minimum set of hardware features required to see the benefits of vectorization without dramatically increasing the complexity of existing processor design or programmability. The ViVA infrastructure can exist within conventional cache hierarchies while employing familiar vector semantics that can take advantage of existing vectorizing compiler technology; resulting in memory accesses that are simpler and more power-efficient than the scalar approach.

ViVA has several important advantages over prefetching for tolerating memory latency. It allows programs to explicitly describe their memory access patterns, avoiding the need for power-hungry and error-prone stream detection hardware. A single ViVA instructions expresses many memory accesses, increasing the parallelism that can be presented to the memory systems. The use of a software-controlled memory buffer is a simple, efficient way to allow many memory accesses to proceed concurrently. Additionally, no coherence mechanism is needed for the ViVA buffer, since elements loaded into a scratchpad memory do not maintain automatic coherence with memory elements.

To validate our approach, we implemented ViVA on the Mambo cycle-accurate full system simulator, which was carefully calibrated to match the performance on our underlying G5 architecture. Results show that ViVA is able to deliver significant performance benefits over scalar techniques for a variety of memory access patterns as well as two important memory-bound compact kernels, CT and SpMV —- achieving 2x–13x improvement compared the scalar version. Overall, our preliminary ViVA exploration points to a promising approach for improving application performance on leading microprocessors with minimal design and complexity costs, in a power efficient manner.

In future work, we plan to consider a broader array of application kernels, while studying additional extensions of the ViVA architecture that relax the strict semantics of vectorization. We also plan to further compare ViVA to related technologies such as DMA transfers. Additionally, ViVA could be extended to support more complex latency-bound load patterns such as pointer-chasing that are common in database and data mining applications. Finally, we plan to investigate the integration of ViVA to help organize memory access patterns for chip multiprocessors.

7 Acknowledgments

All authors from LBNL were supported by the Office of Advanced Scientific Computing Research in the DOE Office of Science under contract number DE-AC02-05CH11231.

References

1. D. Bailey. Little's law and high performance computing. In *RNR Technical Report*, 1997.
2. G. E. Blelloch, M. Heroux, and M. Zagha. Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. Technical Report CMU-CS-93-173, Aug 93.
3. P. Bohrer, J. Peterson, M. Ozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo: a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, 2004.
4. Creating science-driven computer architecture:a new path to scientific leadership. `http://www.nersc.gov/news/reports/blueplanet.php`.
5. R. Espasa, M. Valero, and J. E. Smith. Vector architectures: past, present and future. In *Proceedings of the 12th international Conference on Supercomputing*, 1998.
6. Joseph Gebis. *Low-complexity Vector Microprocessor Extensions*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, May 2008.
7. P. Grun, A. Nicolau, and N. Dutt. *Memory Architecture Exploration for Programmable Embedded Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.
8. M. Gschwind. Chip multiprocessing and the cell broadband engine. In *Proceedings of 3rd Conference on Computing Frontiers*, pages 1–8, New York, NY, USA, 2006.
9. Y. Guo, S. Chheda, I. Koren, C. M. Krishna, and C. A. Moritz. Energy characterization of hardware-based data prefetching. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design*, pages 518–523, Washington, DC, USA, 2004. IEEE Computer Society.
10. HPEC Challenge Benchmark Suite. `http://www.ll.mit.edu/HPECchallenge`.
11. Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.
12. K. Natarajan, H. Hanson, S.W. Keckler, C.R. Moore, and D. Burger. Microprocessor pipeline energy analysis. pages 282–287, Aug. 2003.
13. David A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.
14. O. Temam and W. Jalby. Characterizing sparse algorithms on caches. In *Proc. Supercomputing*, 1992.
15. R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC 2005*, Journal of Physics, 2005.