pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures

by Ankit Jain
$ankit@berkeley.edu$
Computer Science Division, University of California, Berkeley

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Katherine Yelick
Research Advisor

Date

* * * * * *

Professor James Demmel
Second Reader

Date

Contents

Abstract

We have developed pOSKI: the Parallel Optimized Sparse Kernel Interface – an autotuning framework to optimize Sparse Matrix Vector Multiply (SpMV) performance on emerging shared memory multicore architectures. Our autotuning methodology extends previous work done in the scientific computing community targeting serial architectures. In addition to previously explored parallel optimizations, we find that that load balanced data decomposition is extremely important to achieving good parallel performance on the new generation of parallel architectures. Our best parallel configurations perform up to 9x faster than optimized serial codes on the AMD Santa Rosa architecture, 11.3x faster on the AMD Barcelona architecture, and 7.2x faster on the Intel Clovertown architecture.

1 Introduction

A plethora of new multicore architectures have flooded the electronics industry in devices ranging from cell phones to supercomputers. In order to fully unleash the potential of this unprecedented scale of parallelism, it is essential that the scientific computing community develop multicore-specific optimization methodologies for important scientific computations in ways that are accessible to the greater community.

We have developed pOSKI: the Parallel Optimized Sparse Kernel Interface – an autotuning framework for the sparse matrix-vector multiply kernel on multicore architectures, which is an important kernel in many applications. For Krylov subspace methods such as Conjugate Gradient, SpMV often is one of the two major bottlenecks and accounts for the majority of the runtime. Historically, this kernel has run at 10% or less of peak performance and thus represents an opportunity to make significant improvements.

Our framework targets multicore architectures and includes many of the optimizations presented in previous work done at UC Berkeley [24]. We have designed our system hierarchically, much like the architectures we target. This enables us to implement optimizations at each layer that target different architectural features.

Although our system has been built to be adaptable to architectural parameters, we focus on four dual socket architectures for the rest of this paper:

- AMD Opteron 2214 (Santa Rosa) (dual socket x dual core)
- AMD Opteron 2356 (Barcelona) (dual socket x quad core)
- Intel Xeon E5345 (Clovertown) (dual socket x quad core)

This report makes the following contributions and conclusions:

- We have extended the Optimized Sparse Kernel Interface (OSKI) [20] framework with a set of serial optimizations that can benefit current users of OSKI. These users use OSKI to perform sparse linear algebra on serial architectures.
- We have incorporated the most important subset of the optimizations explored in [24] into the pOSKI layer.
- We demonstrate the value of a search over multiple data decompositions across the available threads. We show how a good decomposition can improve performance by more than 20% compared to a naive decomposition and justify an exhaustive search in this space.
- We present an offline parallel benchmark that enables us to optimize the performance of our run time heuristic-based framework up to 23% higher than previous implementations.
- We have developed an MPI layer on top of pOSKI in order to explore optimizations that are possible on distributed memory multinode multicore architectures.

After explaining SpMV in Section 2 and autotuning for SpMV in Section 3, we present pOSKI in Section 4. To complement pOSKI, which targets shared memory architectures, we explain our distributed memory implementation that is built on top of pOSKI using MPI in Section 5. We present our experimental setup in Section 6 and the results of our experiments in Section 7. Finally, we conclude in Section 8 and discuss future directions in Section 9.

2 SpMV Overview

SpMV lies at the heart of a diverse set of applications in many fields, such as scientific computing, engineering, economic modeling, and information retrieval. Sparse kernels are computational operations on matrices whose entries are mostly zero (often 99% or more), so that operations with and storage of these zero elements should be eliminated. The challenge in developing high-performance implementations of such kernels is choosing

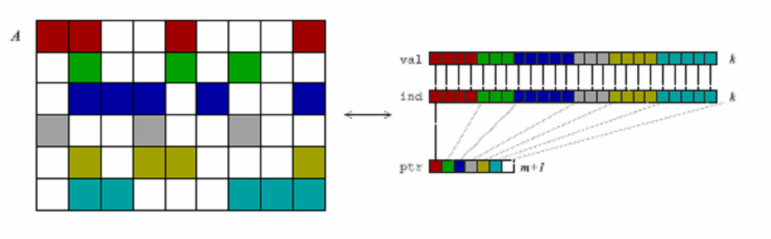


Figure 1: Compressed Sparse Row Data Structure Format

the data structure and code that best exploits the structural properties of the matrix (generally unknown until application run-time) for high-performance on the underlying machine architecture (e.g., memory hierarchy configuration and CPU pipeline structure). Given that conventional implementations of SpMV have historically run at 10% or less of peak machine speed on cache-based superscalar architectures as well as the success of autotuners in this domain [19], there is a need to develop autotuning frameworks targeting the current and next generations of multicore architectures.

Most sparse matrix storage data structures do not store the explicit 0's. While dense matrices are often stored in a single contiguous array, studies using the Sparsekit package [17] have shown that an efficient format to store sparse matrices is the Compressed Sparse Row (CSR) format. We use this format for our naive SpMV implementation and compare our optimized codes against kernels that operate on matrices stored in this format. We discuss the CSR Data Structure and Algorithm in the following subsection.

The Compressed Sparse Row (CSR) Storage

The CSR data structure stores only the entries of a matrix that are nonzero in a contiguous array, *val*, as well as two auxiliary indexing arrays that aid in the SpM×V calculation. This is illustrated in Figure 1. The *ind* array represents the index of the column (in the original matrix) that each corresponding *val* array entry is in. The length of *ind* and *value* arrays equals the number of non-zero elements in the matrix. The third array, *ptr*, points to the elements in the *ind* and *val* arrays that represent the start of a new row. Thus, length of the *ptr* array equals the number of rows in the matrix plus one(the last element stores the total number of nonzeros in the matrix).

Algorithm 1 traverses the CSR Data Structure described above in order to compute the SpMV. The code optimizations presented in Section 4 build upon this code.

Algorithm 1 Compressed Sparse Row Algorithm

Require: val: nonzero values in A
Require: ind: column indices of values in A
Require: ptr: pointers to row starts in A
Require: x: source vector array
Require: y: destination vector array
Ensure: $y_{final} = y_{initial} + A \times x$

```
for all row i do
    temp_i ← 0
    for j=ptr[i] to ptr[i + 1] - 1 do
        temp_i ← temp_i + val[j] × x[ind[j]]
    end for
    y[i] ← y[i] + temp_i
end for
```

3 Autotuning

Previous studies have shown that it is not always efficient to hand-tune kernels for each architecture/dataset combination because an optimization for one combination might hurt performance for another one [3, 5, 13, 16, 22]. Current generation compiler-optimized code is not always tuned for architecture or datasets. It is based on generic assumptions and therefore under specific architectures and datasets leads to suboptimal performance.

Over the last decade, researchers have found that performing an empirical search over the possible combinations of algorithms and data structures for a given kernel can be used along with heuristics to find optimal or nearly(within 90% of) optimal codes. Although the cost of tuning can be expensive, once chosen, a kernel may be used thousands of times. The class of frameworks that support such a search have come to be known as autotuners. Autotuners provide a portable and effective method for tuning over the plethora of optimizations available on today's architectures. Autotuners have a proven track record in the HPC community. Examples of successfully released autotuners include ATLAS [22], FFTW [5], OSKI [20] and Spiral [16].

The remainder of this section presents the state of the

autotuning space with respect to SpMV.

3.1 Serial SpMV Autotuners

The Optimized Sparse Kernel Interface (OSKI) [20] is a widely used autotuner for sparse linear algebra today. OSKI is a collection of low-level primitives that provides *automatically tuned* computational kernels on sparse matrices, for use by solver libraries and applications. The current implementation targets cache-based superscalar uniprocessor machines. Although OSKI provides optimized codes for Sparse Triangular Solve as well as SpMV, in our work, we only optimize SpMV execution.

The most important optimization within OSKI is register blocking. This converts the CSR format described in Section 2 into a blocked CSR (BCSR) format. This transformation is briefly described in Appendix A. A more detailed treatment on this format as well as its benefits can be found in [19].

OSKI was developed by Richard Vuduc [19]. We modify this serial codebase in order to enable a parallel layer on top of it that is aware of non-uniform memory access (NUMA) times that are common on many of today's multicore architectures. We also add algorithms to OSKI's search space, that explicitly prefetch data through architecture specific intrinsics. The motivation behind these optimizations as well their implementations are discussed in Section 4.

3.2 Parallel SpMV Autotuners

While there has been significant work with serial autotuners for SpMV, there is currently no parallel autotuner for SpMV that is optimized for modern multicore architectures. There has been work presenting optimized multithreaded SpMV codes [24] but no library that can be used by an application writer in order to efficiently use the underlying architecture.

The next section presents pOSKI – a parallel library that is built on top of OSKI to provide an optimized framework for performing SpMVs on various architectures.

4 pOSKI

pOSKI provides a serial interface to the user in order to provide the user with functions to break the matrix into blocks that can be run on multiple threads, cores, or sockets and subsequently tune kernels that are specific to the matrix and architecture. This interface implicitly exploits the parallelism present in the kernels. The primary aim of this interface is to hide the complexity of parallelizing and tuning operations to intelligently use the available sockets, cores and threads within one node to efficiently utilize the available memory bandwidth.

The interface is designed to hide the underlying parallelism. Each serial-looking function call that the user makes triggers a set of parallel events. pOSKI manages its own thread pool and manages the appropriate thread and data affinities in order to maximize performance.

pOSKI targets a shared memory architecture and uses a Pthreads execution model. Each thread has access to data from all other threads. Explicit synchronization, using barriers, is needed in order to guarantee correct execution.

Many of the optimizations incorporated within pOSKI are inspired by previous work done by S. Williams *et al.* [24] and R. Vuduc [19]. These optimizations include register blocking, cache blocking, software prefetching, software pipelining, and loop unrolling.

SpMV performance is dependent on many factors including loop overhead, memory bandwidth, memory latency and instruction level parallelism. These correspond to the nested loop structure, streaming nonzeros, the average cost for an indirect access and the flop-to-byte ratio. In the following sub-sections, we present optimizations through which we try to improve the use of the available memory bandwidth by masking the costs associated with bringing a value from the caches to the registers as well as from the main memory to the caches. The code segments provided in this section modify the CSR Algorithm shown in Algorithm 1. In our experiments, we apply these techniques to all the blocked version of the code as well.

Table 1 summarizes all the optimizations performed in our system and the software layer where they are implemented. In Table 1, OSKI 1.0.1h represents the latest release of OSKI while OSKI 1.1 represents the version of OSKI that we have developed (i.e. OSKI 1.0.1h with new optimizations). pOSKI refers to the Pthread layer built on top of OSKI 1.1, MPI-Layer refers to layer built on top of pOSKI. Appendix B details the pOSKI API. Finally, we present the optimizations presented in Williams, et al. [24] for comparison.

4.1 Serial Optimizations

The set of optimizations described in this section are applied at the serial OSKI layer. While register blocking and cache blocking were available in OSKI version 1.0.1h, we add explicit software prefetching to OSKI's search space. In addition, we extend OSKI's internal representation of each matrix to include matrix-specific allocators and deal-

No.	Optimization	OSKI 1.0.1h	OSKI 1.1	pOSKI	MPI-pOSKI	Williams $et.al.$ [24]
1.	Register Blocking	✓	✓			✓
2.	Cache Blocking	✓	✓			✓
3.	Software Prefetching		✓			✓
4.	Serial Benchmarking	✓	✓			
5.	Data Decomposition within Shared Memory			✓		✓
6.	NUMA Aware Allocation			✓	✓	✓
7.	Parallel Benchmarking			✓		
8.	Matrix Compression			(✓)		✓
9.	SIMDization			(✓)		✓
10.	Software Pipelining			(✓)		✓
11.	TLB Blocking			(✓)		✓
12.	Data Decomposition across Distributed Memory				✓	
13.	Overlap of Computation and Communication				(✓)	
14.	Array Padding			(✓)		✓

Table 1: Optimizations Implemented: optimizations checkmarked within parenthesis are not yet implemented. Optimizations 1-4 are serial (applied at OSKI layer: described in Section 4.1), optimizations 5-11 are shared memory parallel (applied at pOSKI layer:described in Section 4.2) and optimizations 12-14 are distributed memory parallel (applied at MPI-pOSKI layer:described in Section 5).

locators in order to enable NUMA-Aware allocation (Section 4.2.3). Register blocking and cache blocking were first presented in [7] but OSKI was the first distributed autotuning library that incorporated them.

Due to the time constraints for this project, we present, but do not implement explicit software pipelining, SIMDization, index compression and array padding. For the optimizations we do not implement, we present the performance gains as reported by [24]. We chose to implement the optimizations in order of decreasing performance improvement (as presented in [24]).

4.1.1 Register Blocking

The register blocking optimization is implemented through the Blocked CSR (BCSR) format and algorithm. BCSR is designed to exploit naturally occuring dense blocks by reorganizing the matrix data structure into a sequence of small (enough to fit in register) dense blocks. Only one set of indices is needed per block rather than per non-zero as is the case for the CSR format. This benefit does, however, come at the cost of adding explicit zeros to the matrix storage structures. Appendix A describes the BCSR format and algorithm in detail. An important point to note is that OSKI does not perform an exhaustive search over all register blockings on the runtime matrix. Instead, it benchmarks a fixed matrix (currently chosen to be dense) stored in sparse format at install time. At runtime, OSKI picks the register blocking based on a performance estimation heuristic that compares the fill ratios of the runtime matrix with that of the benchmarked matrix.

4.1.2 Cache Blocking

Cache blocking reorders memory accesses in an effort to keep the working set of the source vector in cache and thus increase temporal locality. In contrast, register blocking compresseses the data structure in an effort to reduce memory traffic. The increased locality comes at the cost of maintaining more complex data structures – an extra set of row pointers is needed to maintain the start of each cache block. Nishtala *et al.* [14] showed that this optimization helps performance significantly but only for a small class of matrices.

4.1.3 TLB Blocking

[15] highlighted the importance of TLB blocking showing how TLB misses can vary by an order of magnitude depending on the blocking strategy. [24] limited the number of source vector cache lines touched in a cache block by the number of unique pages touched by the source vector to 4KB pages on the three x86 architectures and to 256MB pages on the Niagara2 based architecture. We leave this optimization at the OSKI layer to future work.

4.1.4 Software Prefetching

The last serial optimization we present is explicit software prefeching using machine specific intrinsics. Intrinsics allow access to all operations normally available to assembly language programmers for the target architecture through a higher level interface.

We use these instructions to prefetch the values of

matrix-entries and column-indices that are needed next within the inner loop of the (B)CSR Algorithm. This optimization aims at masking the memory-to-cache latency that exists. While a compiler based prefetch is predictive and usually does not evict values out of the highest levels of cache, an intrinsic based prefetch is declarative and allows the programmer to prefetch values into the highest cache levels. The reason for this difference is because a programmer's instrinsic is not treated as a prediction; it is treated as a statement.

The intrinsics available on each machine are different and the options available vary as well. Williams, $et.al.$ [24], found that the optimal prefetch distance for the two arrays is matrix specific. We add this search to OSKI's tuning routine. Exhaustive prefetch distance search is quadratic in time since we are searching over all possible prefetch distances for the val array combined with all possible prefetch distances for the col_ind array. Thus, we limit our search space to bounds that allow the search to run fast while covering all the interesting values (powers of 2 up to the maximum number of threads studied). Future work can include a heuristic or benchmark that calculates these values more efficiently. Table 3 summarizes the architecture specific intrinsics used as well as the bounds set on prefetch distances.

If the prefetch distance for either or both arrays is 0, we eliminate the prefetch instructions for the appropriate arrays. Therefore, we write four versions of each kernel:

- kernel_VP0_CP0: No prefetch instructions in code.
- kernel_VP0_CPY: Prefetch instructions only for column index array with amount Y.
- kernel_VPX_CP0: Prefetch instructions only for val array with amount X.
- kernel_VPX_CPY: Prefetch instructions for val array with amount X and column index array with amount Y.

This methodology allows us to keep traffic to instruction caches to a minimum.

If available, we give the hardware a hint indicating that the prefetched values will not be used again and can be evicted directly out of the entire cache hierarchy instead of being relegated to a lower level.

It is important to note that the new search over prefetch distances adds a significant overhead to the tuning that takes part within OSKI. According to [20], OSKI's tuning (heuristics for register blocking, currently without prefetch distance tuning) takes the time it would take to perform approximately 40 matrix vector multiplies. The prefetch distance tuning performs an exhaustive search and performs 256 matrix vector multiplies and chooses the best combination. We leave the development of an efficient heuristic to calculate the prefetch distances to future work.

Algorithm 2
Software Prefetched CSR Algorithm
Here, we prefetch the val array by pref_v_amt and the ind arry by pref_i_amt

Require: val: nonzero values in A
Require: ind: column indices of values in A
Require: ptr: pointers to row starts in A
Require: x: source vector array
Require: y: destination vector array
Ensure: $y_{final} = y_{initial} + A \times x$

```
1: for all row i do
2:    temp_i ← 0
3:    for j=ptr[i] to ptr[i + 1] - 1 do
4:       temp_i ← temp_i + val[j] × x[ind[j]]
5:       pref_intrinsic(pref_v_amt+&val[j])
6:       pref_intrinsic(pref_i_amt+&ind[j])
7:    end for
8:    y[i] ← y[i] + temp_i
9: end for
```

4.1.5 Software Pipelining

We observe that the multiplication in the inner loop of the CSR algorithm requires two memory accesses: *val*[*j*] and *x*[*ind*[*j*]]. In addition, *x*[*ind*[*j*]] is an indirect access which costs the algorithm even more cycles per iteration. By explicitly software pipelining the fetching of these values across three iterations of the inner loop, it is possible to increase the L1 Cache to Register and Functional Unit throughput.

Algorithm 3 shows how the inner loop of the CSR Algorithm can be software pipelined. It is necessary that the arrays are padded with 0's at the end in order for the last iteration of the loop to execute correctly. Also, note the way in which the algorithm exploits the fact that the arrays are all continuous by pipelining the values across rows.

In the presence of explicit software prefetching, this optimization does not help SpMV performance significantly since the bottleneck on most architectures is the memory-to-cache bandwidth and not the cache-to-register bandwidth.

Algorithm 3
Software Pipelined CSR Algorithm
Here we pipeline across single iterations of the inner loop

Require: val: nonzero values in A
Require: ind: column indices of values in A
Require: ptr: pointers to row starts in A
Require: x: source vector array
Require: y: destination vector array
Ensure: $y_{final} = y_{initial} + A \times x$

```
1:  val_1 ← val[0]
2:  x_1 ← x[ind[0]]
3:  ind_2 ← ind[1]
4:  for all row i do
5:      temp_i ← 0
6:      for j=ptr[i] to ptr[i + 1] - 1 do
7:          temp_i ← temp_i + val_1 × x_1
8:          val_1 ← val[j + 1]
9:          x_1 ← x[ind_2]
10:         ind_2 ← ind[j + 2]
11:     end for
12:     y[i] ← y[i] + temp_i
13: end for
```

4.1.6 SIMDization

It is possible to use the 128b SIMD registers on most architectures to reduce the number of instructions for data parallel applications. By using SSE instructions, four integers (4×32b=128b) or two doubles (2×64b=128b) can be operated upon with a single instruction. [24] showed that implementations of SpMV that used the SSE intrinsics performed significantly better than straight C code on the Intel Clovertown but these codes did not improve performance on the Opteron based architectures.

4.1.7 Index Compression

For a matrix whose submatrix or cache block span fewer than 64K columns, the column indices can be represented using 16b integers rather than the default 32b integers. [24] shows that such an optimization provides upto a 20% reduction in memory traffic for some matrices, which translates up to a 20% increase in performance. [23] presents delta encoded compression (DCSR) and row-pattern based compression (RPCSR) to accelerate SpMV performance but we do not study these more general approaches.

4.1.8 Array Padding

This simple optimization makes sure that all matrix structures that are allocated so that the pointers are aligned to the L2 cache bank size. Doing this avoids cache and bank conflicts and increases performance due to aligned prefetches. [24] demonstrated this optimization to increase performance by as much as 10%.

4.2 Parallel Optimizations

The optimizations described in this section are applied at the pOSKI layer.

4.2.1 Data Decomposition

The first step in a parallel SpMV kernel is decomposing the matrix across the available threads where each thread is assigned a different subblock of the matrix. Finding the optimal layout of data across these threads turns out to be a non-trivial task because of the large number of possibilities. For a machine that supports 32 concurrent hardware threads, the possible divisions of the threads across the matrix are 1×32, 2×16, 4×8, 8×4, 16×2 and 32×1. We call the 1×32 layout column-blocked and the 32×1 layout row-blocked. Due to the significant time that an exhaustive search over all these choices would take, many current efforts completely ignore a search in this dimension and just divide the matrix across the rows or columns (i.e., either the 32×1 or the 1×32 option).

Initial studies have shown that having more software threads than the number of available hardware threads often yields higher performance. Thus, we expand our search space to include more software threads than the number of hardware threads. Since this space is extremely large, we search over all combinations of powers of 2 of up to 64 software threads on the AMD Opteron based and Intel Clovertown based architectures.

An important point to note is that a decomposition that includes more than one thread in the column dimension requires a reduction. All times reported for SpMV in the results section include the reduction time when applicable.

Figure 2 presents a 4×2 decomposition of the illustrated 7×8 matrix. Each 'x' in the figure represents a nonzero. Blank spaces represent zeros. Each colored section represents a separate submatrix. Each submatrix is operated upon by a separate software thread and therefore has its own instance of OSKI initialized. Since each matrix is tuned separately, each can have a different register blocking as well as prefetch distances (as explained in Section 4.1.4).

The decomposition takes place in a way to make sure that each thread is load balanced by number of nonzeros. pOSKI first blocks the matrix across available threads in

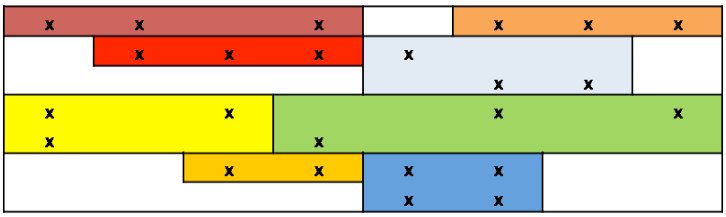


Figure 2: Example: A 4x2 Decomposition of a Matrix

row dimension (4 in this case). Since there are 24 total nonzeros, pOSKI assigns each of the row-blows 6 nonzeros. pOSKI then blocks across the available threads in the column dimension. Thus, each row block can have its own column block boundaries.

4.2.2 pBench: A Parallel Benchmark

pOSKI is built on top of OSKI: Each thread of pOSKI invokes an SpMV call to an OSKI kernel in parallel. At runtime, OSKI uses benchmark data that it collected at its install time along with heuristics in order to determine the best data structure and algorithm for the SpMV (Chapter 3, [19]).

OSKI's install-time benchmarking is single threaded. The installer gets all the machine's resources to benchmark different features of the underlying architecture as well as the performance of a single SpMV. Our hypothesis is that in architectures with multiple cores sharing the available bandwidth, this is not an accurate benchmark to mirror the runtime environment (where pOSKI will have multiple threads performing SpMV's in parallel and thus sharing the available bandwidth). Therefore, we propose the following parallel benchmark:

- At install time, we will run multiple instances of OSKI's benchmark – one in each thread. This will mirror the execution of pOSKI. We will run all combinations of powers of two threads up till the number of concurrent software threads we wish to study. E.g., on the AMD Santa Rosa, we will run 1, 2, ...,64 threads in parallel.

- OSKI's interface allows us to override the default benchmark data to use at runtime. We will load the parallel benchmark data that we have collected into each serial instance of OSKI. At runtime, OSKI will choose the appropriate data structure and algorithm for the submatrix in question based on this parallel benchmark data.

4.2.3 NUMA-Awareness

Sockets in many modern architectures are designed for cache coherent non-uniform memory accesses (ccNUMA). In ccNUMA, a locality domain is a set of processor cores together with locally connected memory which can be accessed without resorting to a network of any kind. This is the kind of clustering found in two and four socket AMD Opteron nodes. Modern operating systems are aware of the underlying hardware architectures and allocate space for data following a 'first touch' policy: they allocate space for the data closest to the thread which initializes them. Thus, binding a thread to a specific core signals the hardware that the data that is used by that thread should be kept in the DRAM closest to a given locality domain. This is referred to as memory affinity.

For such NUMA architectures, we apply a set of NUMA-aware optimizations in which we explicitly assign each submatrix block to a specific core and node. We followed the methodology in [24] to ensure that both the thread and its associated submatrix are mapped to a core (process affinity) and the DRAM (memory affinity) proximal to it. The optimal affinity routines varied by architecture and OS. [24] found the Linux scheduler [11] to perform well on the Opteron-based architectures.

However, the problem is not completely fixed by binding initial allocations. The problem lies within the implementation of malloc(). malloc() first looks for free pages on the heap before requesting the OS to allocate new pages. If available free pages reside on a different locality domain, malloc() still allocates them, violating the NUMA-aware invariants.

Our solution is to allocate one large chunk of memory per locality domain at the time pOSKI is initialized. All allocations and deallocations from this point come from this privately managed heap.

Our biggest challenge was to propagate the NUMA-aware optimizations to the OSKI layer. While the pOSKI layer respected the memory affinity of a given thread and submatrix, routines such as oski_TuneMat() are malloc() and free() intensive. In order to make them NUMA-aware,

we append the oski_matrix type with one required parameter, three optional functions and one optional parameter:

- **Required:** has_matspec_allocators: This parameter is 0 if there are no matrix specific allocators/deallocators. A value of 1 corresponds to the rest of the items in this list having non-null values.

- void* matspec_Malloc(size_t size, void* matspec_info): This optional function will be specific to the matrix it is a part of. Thus, each matrix can have a different allocator – depending on whether it was created from pOSKI or from another entrance point.

- void* matspec_Realloc(void* ptr, size_t size, void* matspec_info): Akin to the matspec_Malloc().

- void matspec_Free(void* ptr): Akin to the matspec_Malloc().

- void* matspec_info: This variable pointer can point to any auxiliary data that matspec_Malloc() might need. In pOSKI's case, this pointer will point to an integer that has the value corresponding to the Id of the thread that will execute the oski_MatMult() for the given matrix. Default Value: NULL.

After matrix creation, any allocation and deallocation is done using the allocators and deallocators associated with the matrix, if available (has_matspec_allocators == 1). If there is none available, OSKI defaults will be used.

5 MPI-pOSKI

Two common methods of programming parallel applications are using Pthreads and MPI tasks. While Pthreads target shared memory architectures, MPI applications target distributed memory architectures. The reason the Pthreads model is not used on distributed memory architectures is because it does not provide for a way to send and receive messages over the network – while MPI does. The reason for not using MPI for both classes of architectures is that each MPI task has a higher overhead than each Pthread. MPI libraries usually implement on-node task communication via shared memory, which involves at least one memory copy operation (process to process). On the other hand, there is no intermediate memory copy required because threads share the same address space within a single process. Therefore, Pthreads generally requires less memory bandwidth than MPI, and is thus normally faster. [2].

We built an MPI layer on top of pOSKI to serve two purposes: (1) to demonstrate how to use the pOSKI library and (2) to examine whether it is worth having multiple MPI tasks within a node. Future work can include examining the balance of MPI tasks and Pthreads on multinode multicore architectures. We decompose the matrix across MPI tasks in a way akin to the data decomposition for the pOSKI layer, explained in Section 4.2.1. A separate instance of pOSKI does the SpMV on each submatrix. The pOSKI layer then hierarchically decomposes its submatrix into further submatrices which are tuned for SpMV by the OSKI layer.

Section 7.2 presents the results of examining the effect of tuning the balance between Pthreads and MPI tasks.

6 Experimental Setup

6.1 Architectures Studied

Throughout this work, we target systems that are either single node multicore architectures or multinode architectures with each node containing multiple cores.

The following two subsections describe the architectures that we use in our study. Appendix C has Figures showing the layout of these architectures.

Before we explain the architecures, we define our terminology.

- Core: Smallest processing element.

- Socket: One or more cores on one chip. Defined by having a shared memory on chip, with uniform access time.

- Node: One or more sockets with locally connected memory that can be accessed without resorting to a network of any kind. E.g. Intel Clovertown uses a Front Side Bus and AMD Santa Rosa uses the Hypertransport. A core accessing memory on a remote socket might demonstrate Non Uniform Memory Access (NUMA) times.

- Full System: One or more Nodes. Nodes are connected by a high-bandwidth, low-latency switching network such as Infiniband or Myrinet.

6.1.1 AMD Santa Rosa (dual socket x dual core)

The first architecture we study is the Opteron 2214. Each core in this dual-core design operates at 2.2 GHz with a peak double-precision floating point performance of 4.4 GFlop/s per core or 8.8 GFlop/s per socket. Each socket

No.	Name	Dimensions	NNZ	NNZ/Row	Symmetric	Notes
1.	webbase-1M	1000005 x 1000005	3105536	3.11	No	Web connectivity matrix
2.	mc2depi	525825 x 525825	2100225	4.0	No	Ridler-Rowe Epidemic
3.	marca_tcomm	547824 x 547824	2733595	5.0	No	Telephone Exchange
4.	scircuit	170998 x 170998	958936	5.61	No	Motorola Circuit Simulation
5.	shipsec1	140874 x 140874	3977139	28.21	Yes	Ship section/detail
6.	qcd5_4	49152 x 49152	1916928	39	No	Computing Quark Propagators
7.	pdb1HYS	36417 x 36417	2190591	60.12	Yes	Protein Data Bank: 1HYS
8.	rail4284s	4284 x 1092610	11279748	2632.9	No	Railway Scheduling
9.	raefsky4	19779 x 19779	1328611	67.2	No	Buckling Problem
10.	ex11	16614 x 16614	1096948	66.0	No	3D Steady Flow Calculation
11.	bibd_22_8	231 x 319770	8953560	38760	No	Balanced incomplete block design
12.	dense2	2000 x 2000	4000000	2000	No	Dense Matrix in Sparse Format

Table 2: Matrix Suite Studied: Sorted in order of increasing density

Architecture	ISA	Compiler	Optimizations	Max (val, col_ind) prefetch distances	Intrinsic Used
AMD Santa Rosa	x86	gcc	-O4 -march=opteron -mtune=opteron -msse3 -m64 -funroll-loops	(1024, 256)	_mm_prefetch()
AMD Barcelona	x86	gcc	-O4 -march=opteron -mtune=opteron -msse3 -m64 -funroll-loops	(1024, 256)	_mm_prefetch()
Intel Clovertown	x86	gcc	-O4 -march=nocona -mtune=nocona -msse3 -m64 -funroll-loops	(1024, 256)	_mm_prefetch()

Table 3: Options Chosen on Architecture Suite

includes its own dual-channel DDR2-667 memory controller as well as a single cache coherent HyperTransport (HT) link to access the other sockets cache and memory. Each socket can thus deliver 10.6 GB/s, for an aggregate NUMA (non-uniform memory access) memory bandwidth of 21.3 GB/s for the dual-core, dual-socket SunFire X2200 M2 examined in our study.

6.1.2 AMD Barcelona (dual socket x quad core)

The newest of AMD's family of multicores, the AMD Barcelona, is made of cores that operate at 2.3 GHz with a peak double-precision floating point performance of 9.2 GFlop/s per core or 36.8 GFlop/s per socket. Each socket includes its own dual-channel DDR2-667 memory controller as well as a single cache coherent HyperTransport (HT) link to access the other sockets cache and memory. Each socket can thus deliver 10.6 GB/s, for an aggregate NUMA (non-uniform memory access) memory bandwidth of 21.3 GB/s for the quad-core, dual-socket machine examined in our study.

6.1.3 Intel Clovertown (dual socket x quad core)

The Clovertown is Intel's quad-core design. It consists of two dual-core Xeon chips paired onto a single multi-chip module (MCM). Each core is based on Intels Core2 microarchitecture (Woodcrest) and runs at 2.33 GHz. The peak double-precision performance per core is 9.3 GFlop/s.

Each socket has a single front side bus (FSB) running at 1.33 GHz (delivering 10.66 GB/s) connected to the Blackford chipset. In our study, we evaluate the Dell PowerEdge 1950 dual-socket platform, which contains two MCMs with dual independent busses. Blackford provides the interface to four fully buffered DDR2-667 DRAM channels that can deliver an aggregate read memory bandwidth of 21.3 GB/s. Unlike the AMD Santa Rosa, each core may activate all four channels, but will likely never attain the peak bandwidth. The full system has 16MB of L2 cache and 74.67 GFlop/s peak performance.

Compiler Options: Given that there are multiple compilers per architecture, each with many options, we summarize the options we chose for our study in Table 3.

6.2 Matrix Test Suite

In order to evaluate the performance of our autotuner on the architectures described above, we have chosen 12 matrices from Tim Davis' University of Florida Collection [4]. This diverse suite of matrices exhibits varying properties relevant to SpMV performance including matrix dimension, dimension ratios, non-zeros per row, the

existence of dense block substructure, symmetry, and degree of non-zero concentration near the diagonal. These properties enable us to better understand the advantages and disadvantages of the different algorithms and optimizations that we have described and applied and how they relate to the underlying architecture.

The properties of the 12 matrices that were chosen are referenced in Table 2. Spyplots for all the matrices can be found in Appendix D.

Previous work [6] has described a method for evaluating SpMV implementations efficiently while taking into account factors such as the data structure used for the sparse matrix, its density of nonzero entries, its dimensions, and even its specific pattern of nonzero entries (taking into account distance from the diagonal).

However, we choose to exhaustively search over the optimizations described in Section 4 and present our results in Section 7. While collecting data in this manner is more consuming, it provides many insights not easily seen otherwise.

6.3 Timing the Routines

We measure the total time it takes to calculate the Sparse Matrix Vector product in milliseconds using the built-in hardware performance counters of the system. Our results report the performance in MFlop/s for each combination of optimizations. Each result reported is the median value from 100 runs with a warm cache.

7 Results

We divide our results section into two subsections: pOSKI results and MPI-pOSKI results. The focus of this thesis is pOSKI and thus we will focus on the first subsection. The MPI-pOSKI layer was built mainly as a way to demonstrate the use of the pOSKI library as well as to justify future work in order to accelerate SpMV on distributed memory multinode, multicore architectures.

7.1 pOSKI

In this section, we present the SpMV performance for our matrix suite on the architectures described above. We compare our implementation to the implementation by Williams, $et.al.$ [24] since we have tried to include all of their important optimizations in this version of pOSKI.

Like [24], we present SpMV performance using a stacked bar format as shown in Figure 3. Each segment corresponds to a individual trial of all optimizations up to that point rather than to components of a trial with all the the optimizations. In addition, we present heatplots for each matrix showing the performance of the different decompositions in Appendix E.

Before presenting the performance results, we discuss the characteristics of three matrices that we choose to focus on. We discuss how we expect each matrix to perform given the structure of our SpMV kernels.

7.1.1 Performance Impact of Matrix Structure

While we present data from all 12 matrices in our suite, we will focus on three cases that make for interesting case studies.

- Matrix 1: This is the sparsest of matrices in our suite and is an interesting example to focus on because it has an average of only 3.11 nonzeros per row. OSKI uses the CSR algorithm and its BCSR variants to perform the SpMV. The inner loop length of the kernels is proportional to the number of nonzeros per row for a given matrix. For these matrices, the kernel is not able to amortize the loop startup overhead. Thus, we can expect this matrix to perform poorly even if its entire source vector fits in cache.

- Matrix 8: This matrix contrasts Matrix 1 in that it has over 2600 nonzeros per row. However, the aspect ratio for this matrix is extremely skewed. It has 4284 rows and 1092610 columns. Thus, the source vector for this matrix cannot fit into the cache of any of the machines we study. This matrix is thus very amenable to cache blocking. However, since cache blocking is not a default choice within OSKI1.0.1h, we expect the data decomposition to compensate for this by having multiple column threads.

- Matrix 12: This matrix is a dense matrix (100% nonzeros) stored in sparse format. Not only does this matrix exhibit a high nonzero per row ratio, but it is also the only matrix where the final flop:byte ratio is nearly 0.25 (two floating point operations for each eight bytes). It is also a matrix in which the vector has high reuse. Thus, we can safely assume that this matrix provides us with a performance upper bound for sustained memory bandwidth.

  In each of the following subsections (one per architecture studied), we will compare the bandwidth achieved for the dense matrix (Matrix 12) with the bandwidth achieved for a modified stream benchmark [12] that does a dot product. We convert the

GFlop/s metric to a GB/s metric by taking into account the flop:byte ratio of 0.25 for the dense matrix. This rationale is justified because we expect there to be near perfect register blocking in which case the bandwidth utilization due to the indices are in the noise. Additionally, for this matrix, the entire source and destination vector can fit in the cache of the architectures studied.

7.1.2 AMD Santa Rosa Results

Figure 3(a) shows SpMV performance on the AMD Santa Rosa. Each bar contains, at the bottom, a naive CSR SpMV implementation. Each additional segment shows the additional performance for adding the given optimization. We also present the performance of Williams, $et.al.$'s implementation for each matrix since we are trying to emulate that performance while extending the OSKI library.

What is clear from this figure is that the impact of each optimization corresponds closely to the matrix structure. By looking at our three focus matrices, the following results are clear:

- Register Blocking (delta between Naive CSR and OSKI1.0.1h) helps Matrix 12 more than it does Matrix 1 or Matrix 8 because it contains a clear dense subblock structure.
- An exhaustive search over all the data decompositions does not significantly improve performance over a simple parallelization because of the high level of source vector reuse and high presence of dense subblocks. In addition, the source vector for the SpMV on this matrix is only 16k in size and can therefore fit in the Santa Rosa's 1MB L2 cache without the presence of any decomposition in the column dimension.
- Data Decomposition helps the performance of Matrix 8 significantly because of the dramatic aspect ratio of this matrix. Since OSKI does not cache block the matrix by default, performance is improved significantly by a decomposition in the column dimension. The heatplot in Figure 8(h) shows that the most effective data decomposition for this matrix is 2×8. The performance of a fully optimized pOSKI is significantly higher than that of Williams, $et.al.$ because their search space does not include decomposition in the column dimension.
- The parallel benchmark helps Matrix 12 the most (out of our three focus matrices) because SpMV on this matrix saturates the available memory bandwidth (due to the high flop:byte ratio). The parallel benchmark helps OSKI's heuristic choose a block sizes for each submatrix that will efficiently split the available bandwidth across the multiple threads.

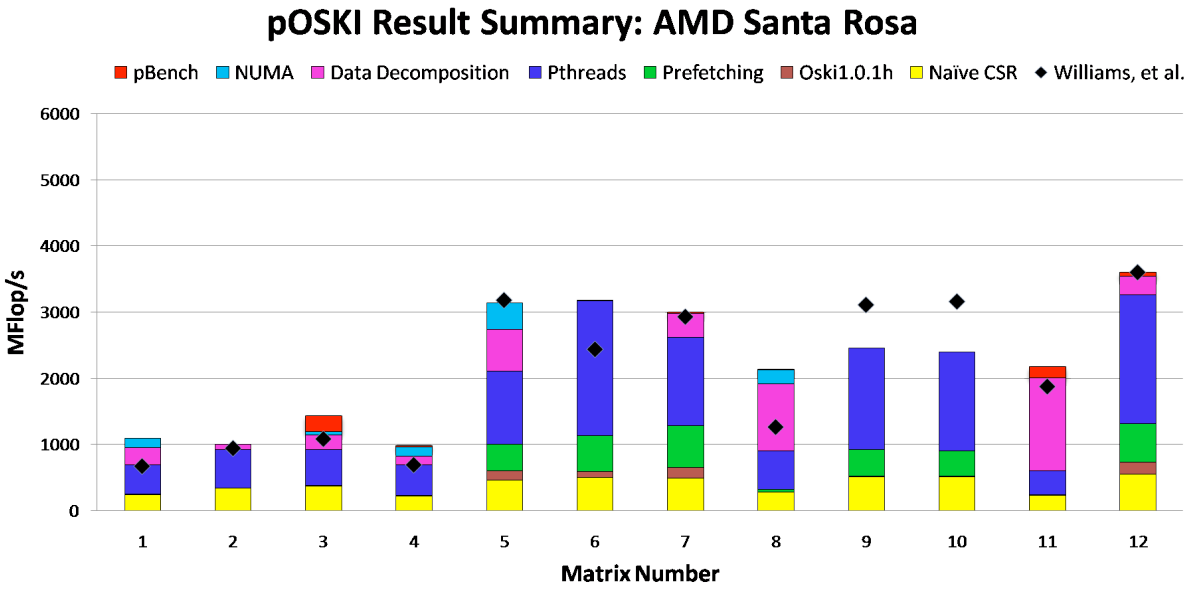
Examining the performance of some of the other matrices in our suite, we notice that prefetching helps the matrices that have a high concentration of nonzeros around the diagonal (Matrices 5, 6, 7, 9 and 10).

Matrices 1, 5 and 8 benefit significantly from the NUMA optimizations. By looking at their heatplots in Figure 8, we see that the optimal decomposition involves having more software threads than the available hardware threads. In addition, all of these matrices have source vectors that are larger than the 1MB L2 cache present on this architecture. Thus, many of the source vector accesses are serviced from memory making the NUMA optimizations extremely important.

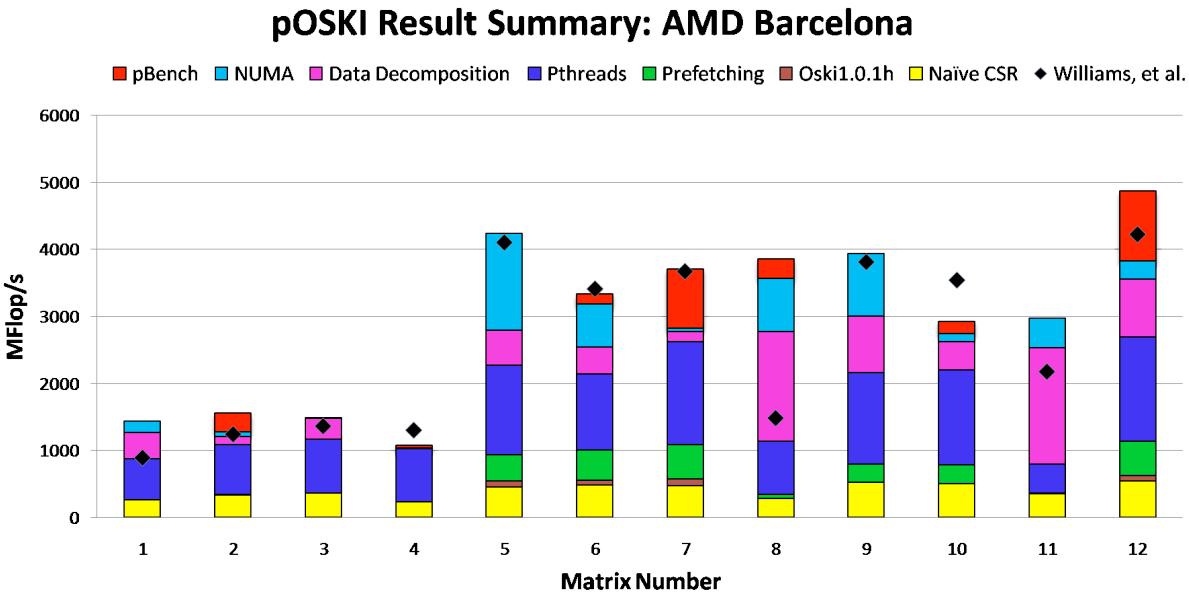
It is also important to understand why the performance of pOSKI with all the optimizations does not perform nearly as well as Williams, $et.al.$ for two of the 12 matrices (Matrices 9 and 10). The source vectors of both these matrices have fewer than 64K elements. Matrix 9 has a source vector that is 19779 elements long while Matrix 10 has a source vector that is 16614 elements long. Thus, they are good candidates for the column index compression described in Section 4. By storing the column indices as 16b integers rather than 32b integers, we can significantly reduce the memory bandwidth utilization and thus improve performance for this memory-bound kernel. This optimization is currently not present within pOSKI and we leave it as future work.

Looking at overall SpMV performance, the serial optimizations added improve performance by as much as $1.9\times$ over OSKI1.0.1h's best imlementation with a median improvement of $1.4\times$. pOSKI provides up to a $9\times$ further improvement over the optimized serial codes with a median improvement of $3.4\times$. This superlinear speedup can be attributed to the more efficient use of the memory subsystem (reduced latency due to prefetching) as well as more outstanding memory requests (due to more threads).

For the dense matrix, the best performing kernel only utilizes 67.6% of available memory bandwidth. The modified stream benchmark that we ran was able to sustain 56.8% of the published bandwidth. We expect our kernels to perform better than the benchmark because we have explicit software prefetching which has shown to significantly mask the memory-to-cache latency on this architecture.

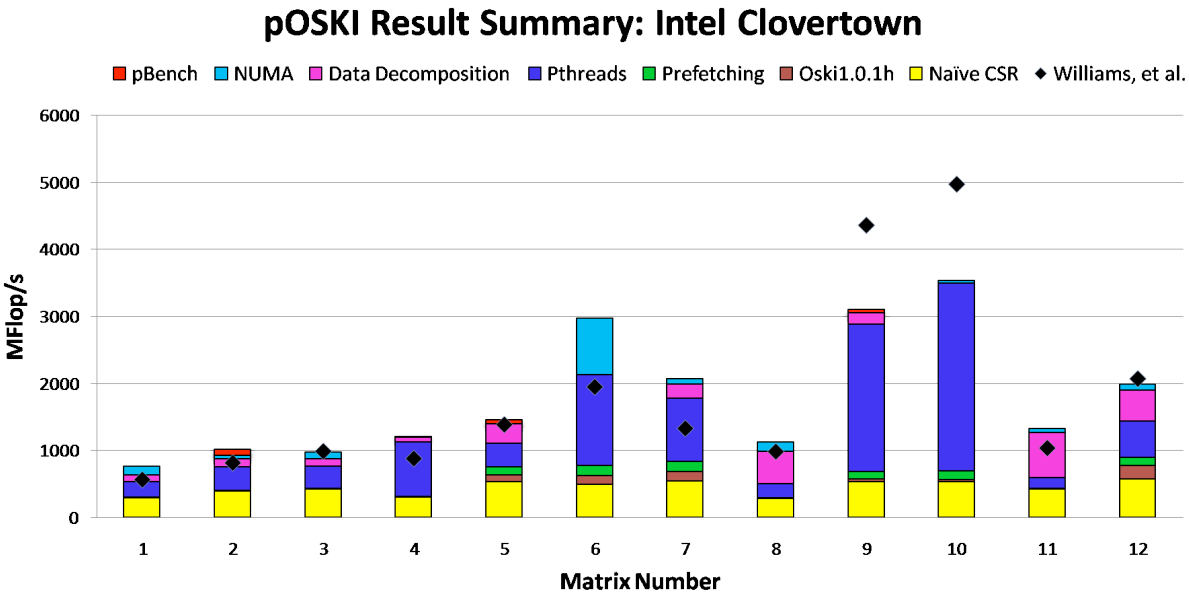


(a)



(b)

Figure 3: Effective pOSKI SpMV performance (not raw flop rate) on (a) AMD Santa Rosa and (b) AMD Barcelona showing increasing degrees of serial optimizations – OSKI1.0.1h and Prefetching – as well as performance as parallelism (Pthreads) and parallel optimizations are introduced – Data Decomposition, NUMA-Awareness and the parallel benchmark. Williams, et. al. are denoted using diamond for comparison based on codes written for Supercomputing 2007 submission. Note: Bars show the best performance for the current subset of optimizations parallelism.



(c)

Figure 3: Effective pOSKI SpMV performance (not raw flop rate) on (c) Intel Clovertown showing increasing degrees of serial optimizations – OSKI1.0.1h and Prefetching – as well as performance as parallelism (Pthreads) and parallel optimizations are introduced – Data Decomposition, NUMA-Awareness and the parallel benchmark. Williams, et. al. are denoted using diamond for comparison based on codes written for Supercomputing 2007 submission. Note: Bars show the best performance for the current subset of optimizations parallelism.

7.1.3 AMD Barcelona Results

Given the similarity of this architecture with the AMD Santa Rosa, we expect the performance characteristics to be extremely similar as well. Examining Figure 3(b), we notice two major differences: the impact of the NUMA optimizations and the parallel benchmark are more accentuated on this architecture.

The increased importance of the NUMA optimizations can be attributed to the smaller L2 caches per core. Additionally, the 8GB/s hypertransport in the Barcelona is shared among 8 cores while it was only shared among 4 cores on the Santa Rosa. Thus, the penalty for not being NUMA aware is higher.

This architecture has the same aggregate memory bandwidth as the Santa Rosa but twice the number of cores. This stresses the importance of managing memory-to-cache traffic efficiently. This is exactly what the parallel benchmark aims to target.

Looking at overall SpMV performance, the serial optimizations added improve performance by as much as 1.9× over OSKI1.0.1h's best imlementation with a median improvement of 1.4×. pOSKI provides upto a 11.3× further improvement over the optimized serial codes with a median improvement of 4.5×. This superlinear speedup can also be attributed to the more efficient use of the memory subsystem (reduced latency due to prefetching) as well as more outstanding memory requests (due to more threads).

For the dense matrix, the best performing kernel only utilizes 91.4% of available memory bandwidth. The modified stream benchmark that we ran was able to sustain 64.8% of the published bandwidth. Like the Santa Rosa, performance for optimized SpMV on this machine is higher than for the benchmark. It is also important to note that both the benchmark and SpMV perform at higher percentages of the published bandwidth on this architecture than on the Santa Rosa because the available bandwidth remains the same while the number of cores doubles.

7.1.4 Intel Clovertown Results

Looking at the results in Figure 3(c), we see that explicit software prefetching does not help much on this architecture. This can be attributed to the Xeon's superior hardware prefetching capabilities compared to the Opteron.

Unlike the two Opteron-based architectures, the heatplots in Figure 10 show that for ten out of the twelve matrices in our matrix suite decompositions in the column dimension do not provide optimal performance. This is because the Clovertown has 2MB of L2 cache per core and thus most of the source vectors can fit in this cache. Additionally, for two of the matrices, the optimal decomposition only has four threads – four less than the number of hardware available threads! Results from [24] show that this is due to the fact that using just two cores on a socket, rather than all four, attains a significant fraction of the sustainable Front Side Bus(FSB) bandwidth. Thus, there is little benefit that the additional computational power can add for our memory-bound kernel.

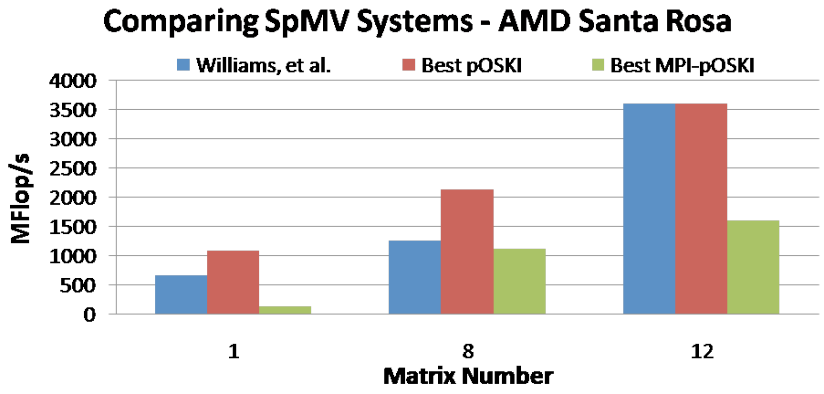
As expected, we do not see much benefit from the NUMA optimizations because the 8 cores share a unified memory. Since all communication between the sockets as well as between memory and each socket's caches goes through the same FSB, we expect performance to be extremely limited by the available bandwidth – and therefore expect that the parallel benchmark optimization to help a lot. The fact that it does not help on most of the matrices is surprising and will be the focus of further exploration.

Given the large caches as well as fast cores on the Clovertown, it is surprising that it performs significantly worse than the Opteron-based Santa Rosa architecture. The serial optimizations added improve performance by as much as 1.24× over OSKI1.0.1h's best imlementation with a median improvement of 1.1×. pOSKI provides upto a 7.2× further improvement over the optimized serial codes with a median improvement of *only* 2.8×. Unlike on the two Opteron-based architectures, we do not see superlinear speedups on this architecture because of the bandwidth limitations of the FSB on this architecture.

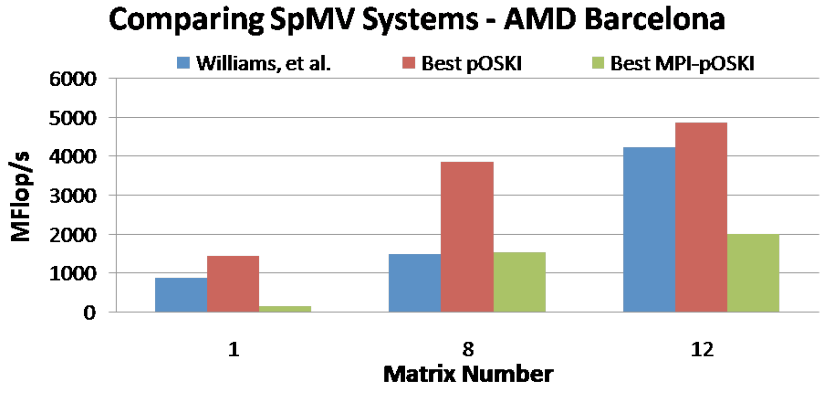
For the dense matrix, the best performing kernel only utilizes 38.8% of available memory bandwidth. The low bandwidth utilization is the reason SpMV on Matrix 12 on the Intel Clovertown performs significantly worse than on the Santa Rosa even though it has 4.2x the peak flop rate. The modified stream benchmark that we ran was able to sustain 27.6% of the published bandwidth. As in the case with the two Opteron-based architectures, the software prefetching is the main reason that SpMV is able to sustain a higher bandwidth than the benchmark.

7.2 MPI-pOSKI

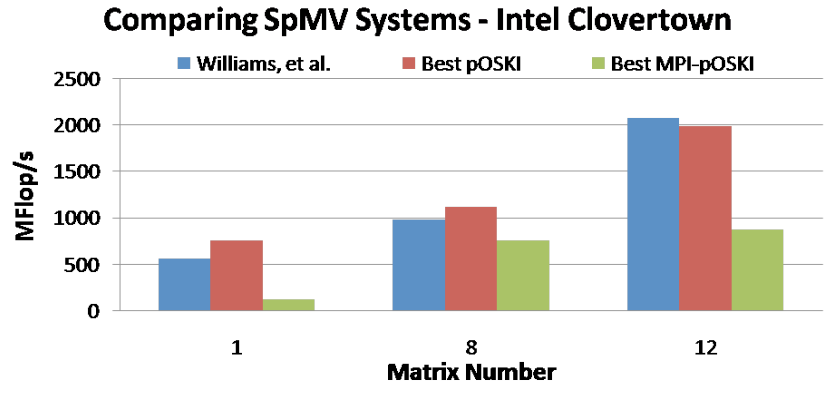
Appendix F contains heatplots representing the performance for different hierarchical decompositions for our three focus matrices on each of the architectures we are examining. There are two numbers on each axis in these heatplots. The outer number represents the decomposition at the MPI-layer while the inner number represents the decomposition at the pOSKI layer. Our hypothesis was that the best performing decomposition would consist of a single MPI task per node which controls one instance



(a)



(b)



(c)

Figure 4: A Comparison of the three SpMV systems discussed on our architecture suite.

of pOSKI. In addition, we hypothesized that using MPI within a node will add significant overhead and thus not perform as well as pOSKI or Williams, $et.al.$

Figure 4 compares the three SpMV systems that we have discussed: Williams, $et.al.$, pOSKI and MPI-pOSKI. Clearly, the performance of MPI-pOSKI is not at par with the other two systems. However, it is important to remember that MPI-pOSKI was not intended for use on single node, shared-memory systems.

Examining the heatplots in Appendix F, we can see that the value of this exercise is in the realization that the optimal number of MPI tasks per node is not always 1, as we had hypothesized. For Matrix 1 (extremely sparse), the optimal number of MPI tasks is greater than 2 on the AMD Santa Rosa, 4 on the AMD Barcelona and 8 on the Intel Clovertown.

The contribution from this study is justifying an exhaustive search over multiple MPI tasks per node for a system designed for multinode, distributed memory architectures.

8 Conclusions

We have developed and presented pOSKI and shown the impact of the many serial and parallel optimizations on a hierarchically decomposed SpMV. Our results show that significantly higher performance improvements can be attained through multicore parallelizations, rather than serial code or data structure transformations. This outcome is very heartening given the trend towards an increasing number of cores per chip with a relatively constant per core performance [1].

Compared to a naive implementation, the serial optimizations provide a 1.45x median speedup on the Opteron, 1.35x median speedup on the Barcelona and a 1.14x median speedup on the Clovertown. Compared to a naive implementation, the serial and parallel optimizations together provide a 5.40x median speedup on the Opteron, 7.20x median speedup on the Barcelona and a 3.66x median speedup on the Clovertown.

We have demonstrated the value of an exhaustive search over the possible data decompositions for a given number of threads and how it can boost performance by more than 20% for some matrices.

We have shown how the parallel benchmark helps OSKI's heuristic choose register block sizes for each submatrix that efficiently splits the available bandwidth across the multiple threads. This improves performance on matrices with high flop:byte ratios – matrices which tend to saturate an architecture's available memory bandwidth.

Finally, our study compares a Pthreads-only implementation to a hybrid MPI-Pthreads implementation within a shared memory machine. Although the Pthreads-only implementation clearly beats the hybrid implementation for the single node case, the lesson that it is beneficial to have multiple MPI tasks within a node will help focus future studies that target distributed memory multinode multicore architectures.

9 Future Work

In addition to adding the remaining optimizations studied in [24], the development of a heuristic to calculate optimal prefetch distances would increase tuning efficiency within OSKI1.1.

There are two studies that we leave to future work at the MPI layer:

- Study the effect of overlapping computation and communication between nodes. While each node computes a given part of the destination vector, we can start reducing a previously computed part of the destination vector using asynchronous sends. We hypothesize that this is an example of a situation where having more software threads than the number of available hardware threads can be used efficiently.

- Study whether there are potential performance gains by assigning some of the threads (or cores) to only handle sending/receiving of messages, and exploring if the communication can be handled in a coordinated asynchronous manner. We expect such an optimization to work well on architectures where there are a high number of hardware threads available per node (e.g. Sun T5140 T2+ [Victoria Falls]).

Finally, it would be of great use to the extended parallel computing community if libraries similar to OSKI and pOSKI were developed for other important computational kernels.

Acknowledgements: I would like to thank Professors Katherine Yelick and James Demmel for their guidance and support. They provided me with the opportunities to expand my computer science horizons. I would also like to express my deepest gratitudes to Professor Richard Vuduc (Georgia Tech) for all the hours he spent explaining OSKI and its architecture to me. I am grateful to the BeBOP research group, in particular Sam Williams and Mark Hoemmen, whose feedback was essential to the culmination of this work. Finally, I would like to thank Ra-

jesh Nishtala who has been a great mentor from the days when I was an undergraduate.

References

[1] K. Asanovic, R. Bodik, B. Catanzaro, and et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, December 2006.

[2] B. Barney. Posix threads programming. Livermore Computing. URL: http://computing.llnl.gov/tutorials/pthreads.

[3] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology, July 1997. ACM SIGARC.

[4] T. Davis. University of florida sparse matrix collection. URL: http://www.cise.ufl.edu/research/sparse/matrices.

[5] M. Frigo and S. G.Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[6] H. B. Gahvari. Benchmarking sparse matrix-vector multiply. Technical report, EECS Department, Berkeley, CA, USA, 2006.

[7] E.-J. Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, UC Berkeley, Berkeley, CA, USA, 2000.

[8] E.-J. Im, K. A. Yelick, and R. Vuduc. Sparsity: Framework for optimizing sparse matrix-vector multiply. In *International Journal of High Performance Computing Applications, Pages 135-158*, Berkeley, CA, USA, 2004.

[9] A. Jain. Load-balancing sparse matrix-vector multiply on multicore architectures, CS281A Class Project, Fall 2007.

[10] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of SIGPLAN '88, Conference on Programming Language Design and Implentation*, Atlanta, GA, USA, 1988.

[11] R. Love. Linux kernel development, September 2003. From Chapter 3: Scheduling. Part of the Developer's Library Series.

[12] J. McCalpin. Stream: Sustainable memory bandwidth in high performance computers, 1995. URL : http://www.cs.virginia.edu/stream/.

[13] D. Mirkovic, R. Mahasoom, and L. Johnsson. An adaptive software library for fast fourier transforms, May 2000. Pages 215-224.

[14] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. Performance modeling and analysis of cache blocking in sparse matrix vector multiply. Technical Report UCB/CSD-04-1335, Department of Computer Science, UC Berkeley, Berkeley, CA, USA, 2004.

[15] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking sparse matrix vector multiply works and why., March, 2007.

[16] M. Puschel, B. Singer, M. Veloso, and J. M. F. Moura. Fast automatic generation of dsp algorithms, May 2001. Pages 97-106.

[17] Y. Saad. Sparskit: A basic tool kit for sparse matrix computations. Technical Report 90-20, NASA Ames Research Center, Moffett Field, CA, 1990.

[18] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS, Boston, MA, 1996.

[19] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, UC Berkeley, Berkeley, CA, USA, 2003.

[20] R. Vuduc, J. Demmel, and K. Yelick. Oski: A library of automatically tuned sparse matrix kernels, 2005.

[21] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[22] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. URL : http://www.netlib.org/lapack/lawns/lawn131.ps.

[23] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of Supercomputing*, Cairns, Australia, June 2006.

[24] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of Supercomputing*, Reno, NV, USA, November 2007.

APPENDIX

A The Blocked Compressed Sparse Row Data Structure and Algorithm

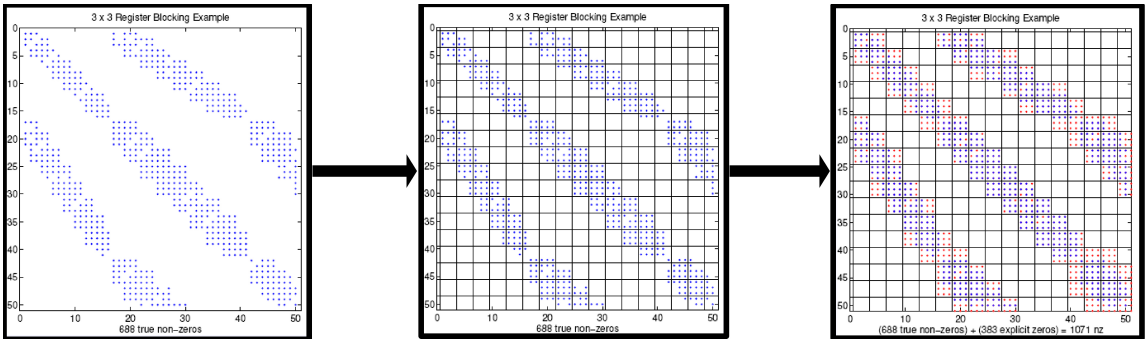


Figure 5: Register Blocking Example: After breaking a matrix into r×c blocks, we may need to fill the blocks with explicit 0's to store them as dense subblocks.

This extension of the CSR data structure and algorithm described in Section 2 aims at reducing memory traffic. Figure 5 shows how a matrix can be blocked at the register level. Given a matrix, we break it into $r \times c$ block submatrices. For each block, we only store one column-index rather than one per non-zero. The penalty paid is the inclusion of the explicit zeros that are illustrated in red in Figure 5. In addition to reducing memory traffic in the column-indices array, this optimization also increases temporal reuse of the source vector. This enables us to reduce the number of irregular source vector accesses that has been shown to be a bottleneck in previous studies [7, 19]. Finding the optimal r×c blocking efficiently can be done using a fill-ratio based heuristic. For an in-depth treatment of this, please refer to [19].

Sample code for a 2×3 is shown in Algorithm 4.

Algorithm 4 A 2×3 BCSR Algorithm

Require: val: nonzero values in A
Require: ind: column indices of blocks in A
Require: ptr: pointers to block row starts in A
Require: x: source vector array
Require: y: destination vector array
Ensure: $y_{final} = y_{initial} + A \times x$

```
for all block-row i do
    temp_1 ← 0
    temp_2 ← 0
    for j=ptr[i] to ptr[i + 1] - 1 do
        temp_1 ← temp_1 + val[j * 2 * 3 + 0] × x[ind[j] + 0]
        temp_1 ← temp_1 + val[j * 2 * 3 + 1] × x[ind[j] + 1]
        temp_1 ← temp_1 + val[j * 2 * 3 + 2] × x[ind[j] + 2]
        temp_2 ← temp_2 + val[j * 2 * 3 + 3] × x[ind[j] + 0]
        temp_2 ← temp_2 + val[j * 2 * 3 + 4] × x[ind[j] + 1]
        temp_2 ← temp_2 + val[j * 2 * 3 + 5] × x[ind[j] + 2]
    end for
    y[i * 2] ← y[i * 2] + temp_1
    y[i * 2 + 1] ← y[i * 2 + 1] + temp_2
end for
```

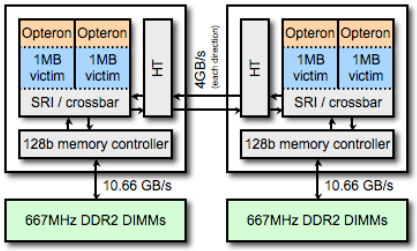
B pOSKI API

This section describes the pOSKI v1.0 API.

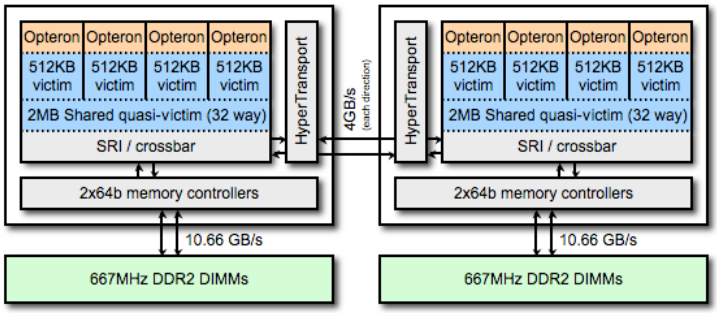
```
/*********************************************************
* poski_Init: This function initializes pOSKI as well as
*             OSKI.  It creates all the worker threads
*             and puts them to sleep.
* char* pbenchFN(in): If using parallel benchmark data,
*               initialize OSKI with a path to appropriate
*               data. NULL value will use OSKI's default
*               benchmark data.
*
* Note: the pbenchFN argument will be removed in
*       pOSKI v1.1. pOSKI will keep track of pbench data
*       files upon installation and will use the
*       appropriate one without user input.
*********************************************************/
void poski_Init(char* pbenchFN);


/*********************************************************
* poski_ParallelizeMatrixAndSourceVector: This function is
*       where the majority of pOSKI's value is.  This
*       function decomposes the matrix into
*       RThreads x CThreads submatrices.  Each submatrix
*       also has its associated partial copies of source
*       and destination vector allocated.
*       In v1.1, the exhaustive search will occur here.
*
* uint32_t *P, uint32_t *C, double* V, uint32_t NRows(in):
*       Matrix in CSR format to be parallelized.
* double* X: source vector
* double* Y: destination vector
* uint32_t NRows: Number of rows in matrix
* uint32_t NCols: Number of columns in matrix
* uint32_t RThreads: Row dimension for decomposition
* uint32_t CThreads: Column dimension for decomposition
*********************************************************/
POskiMatrix* poski_ParallelizeMatrixAndSourceVector
        (uint32_t *P, uint32_t *C, double* V,
        double* X, double* Y,
        uint32_t NRows, uint32_t NCols,
        uint32_t RThreads, uint32_t CThreads);
```

```
/*****************************************************
* poski_TuneMatrix: Tunes each submatrix within POskiSpA
*       separately using OSKI.
* POskiMatrix* POskiSpA(in): Matrix created using
*       poski_ParallelizeMatrixAndSourceVector().
*****************************************************/
void poski_TuneMatrix(POskiMatrix* POskiSpA);

/*****************************************************
* poski_MatMult: Performs SpMV given parallelized matrix,
*       source and destination vectors within POskiSpA
*       along with parameters alpha and beta.
* POskiMatrix* POskiSpA(in): Matrix created using
*       poski_ParallelizeMatrixAndSourceVector().
* double alpha(in)
* double beta(in)
*
* SpMV Operation: $y \leftarrow \alpha y + \beta x$
*
*****************************************************/
void poski_MatMult(POskiMatrix* POskiSpA,
                   double alpha, double beta);

/*****************************************************
* poski_DestroyMatrix: This function destroys a pOSKI
*       matrix by freeing the submatrices as well as any
*       temporary arrays that are allocated.
* POskiMatrix* POskiSpA(in): Matrix created using
*       poski_ParallelizeMatrixAndSourceVector().
* Post Condition: POskiSpA is NULL
*****************************************************/
void poski_DestroyMatrix(POskiMatrix* POskiSpA);

/*****************************************************
* poski_Close: Closes OSKI and makes sure all allocated
*       memory is freed.
*
* Requirement: All matrices of type POskiMatrix have been
*       destroyed by calling poskiDestroyMatrix()
*****************************************************/
void poski_Close();
```

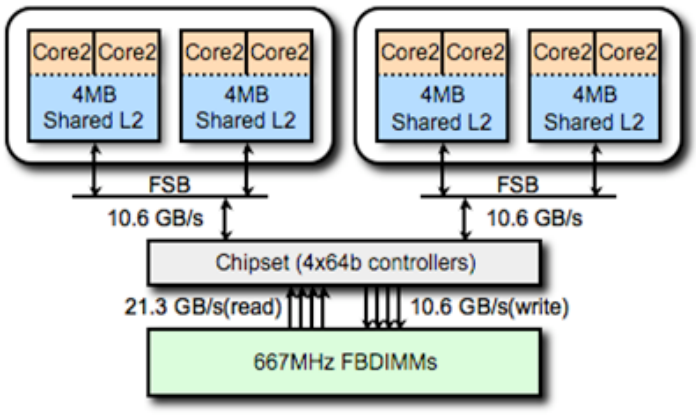
C Architecture Illustrations



(a)



(b)



(c)

Figure 6: (a) AMD Santa Rosa, (b) AMD Barcelona and (c) Intel Clovertown Architectures

D Matrix Spyplots

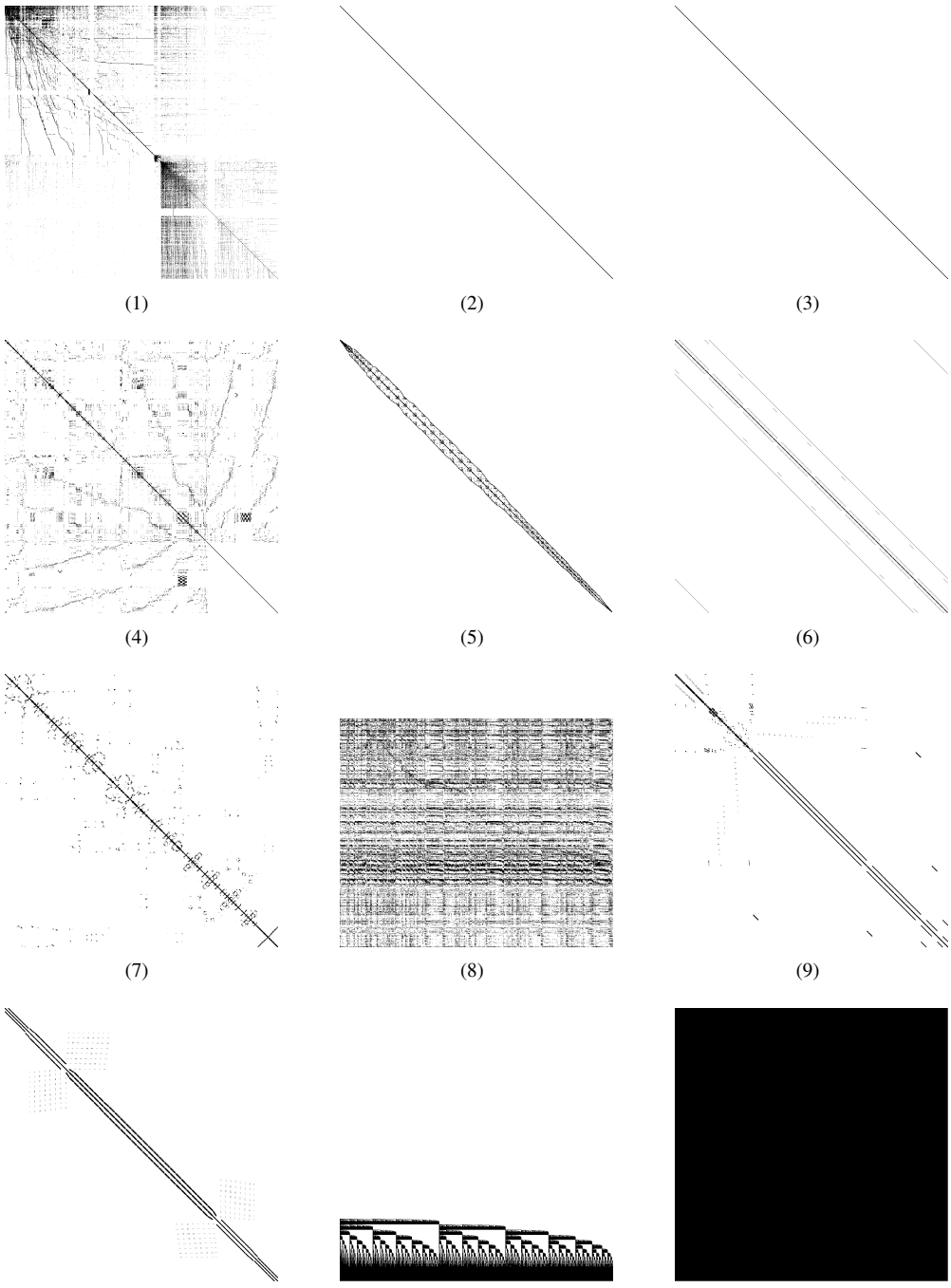
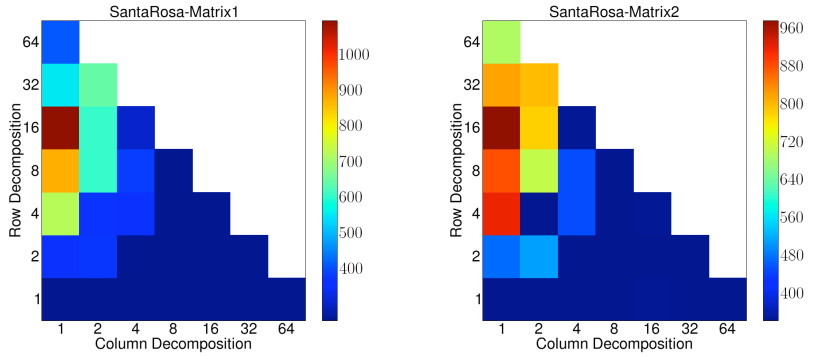


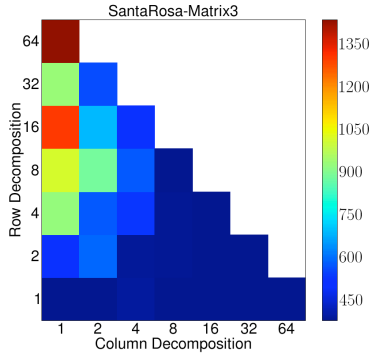
Figure 7: Spyplots for Matrix Text Suite

E pOSKI Data Decomposition Heatplots

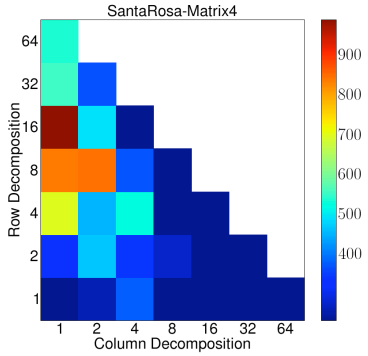


(1)

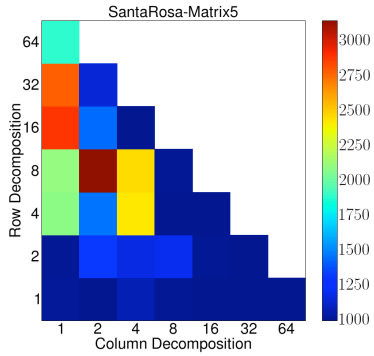
(2)



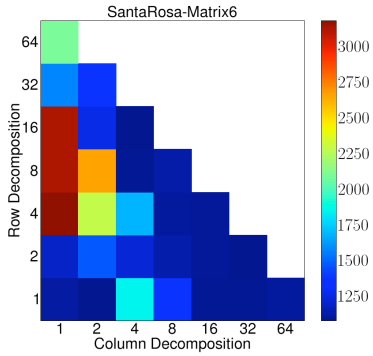
(3)



(4)

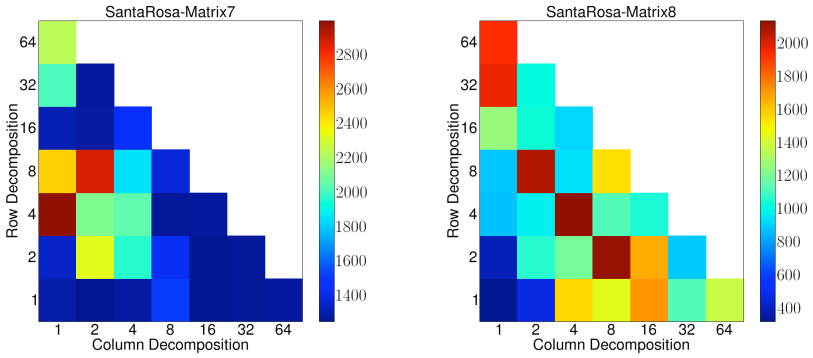


(5)



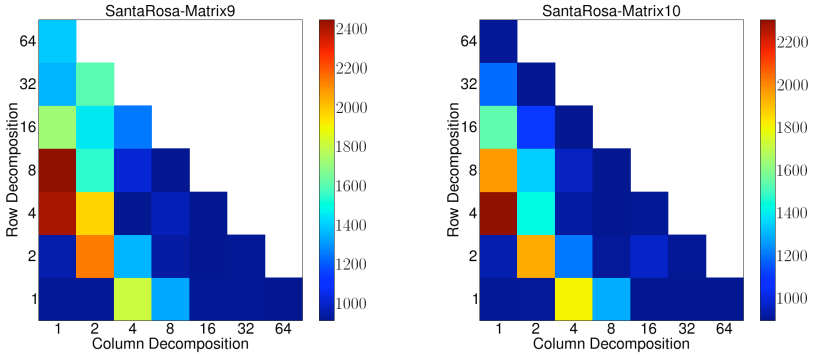
(6)

Figure 8: pOSKI Data Decomposition Heatplots for Matrix Suite on AMD Santa Rosa



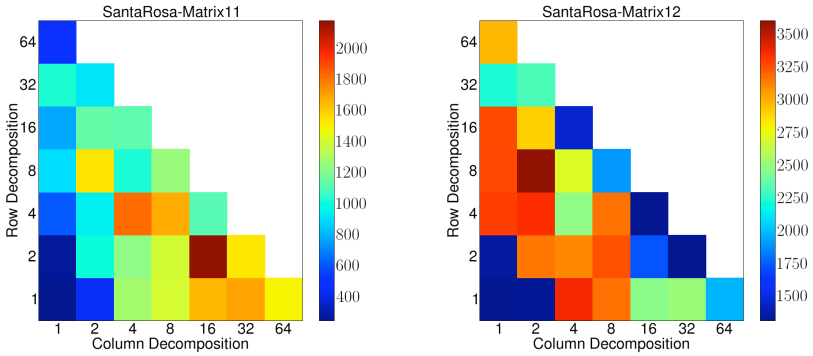
(7)

(8)



(9)

(10)



(11)

(12)

Figure 8: Continued pOSKI Data Decomposition Heatplots for Matrix Suite on AMD Santa Rosa

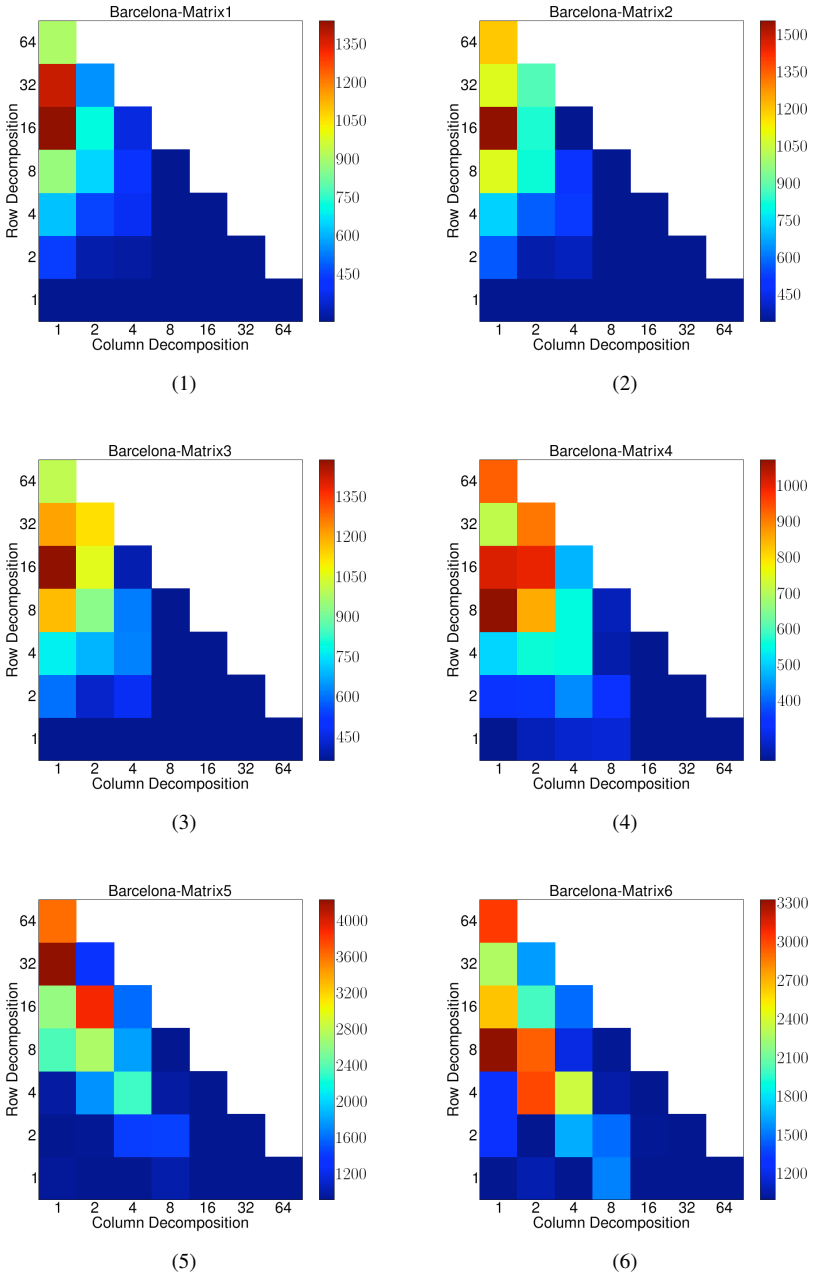
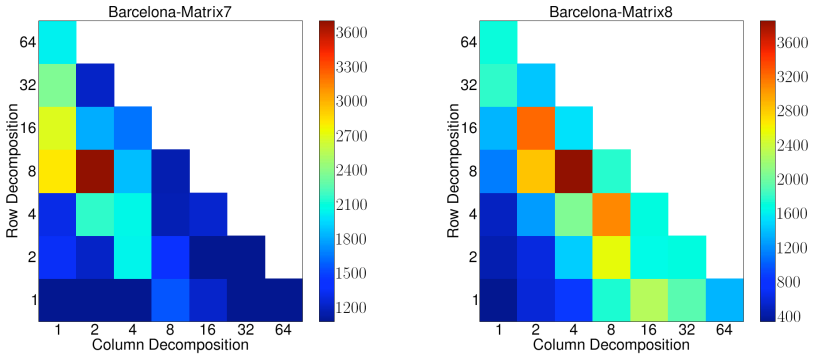
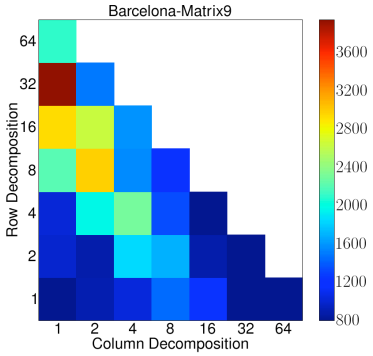


Figure 9: pOSKI Data Decomposition Heatplots for Matrix Suite on AMD Barcelona

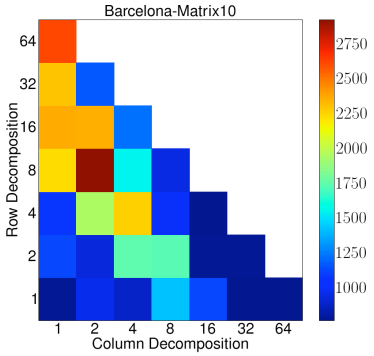


(7)

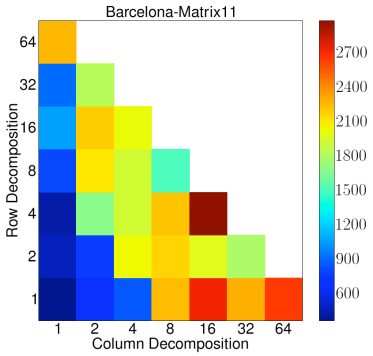
(8)



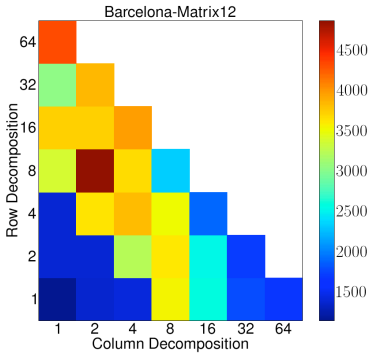
(9)



(10)



(11)



(12)

Figure 9: Continued pOSKI Data Decomposition Heatplots for Matrix Suite on AMD Barcelona

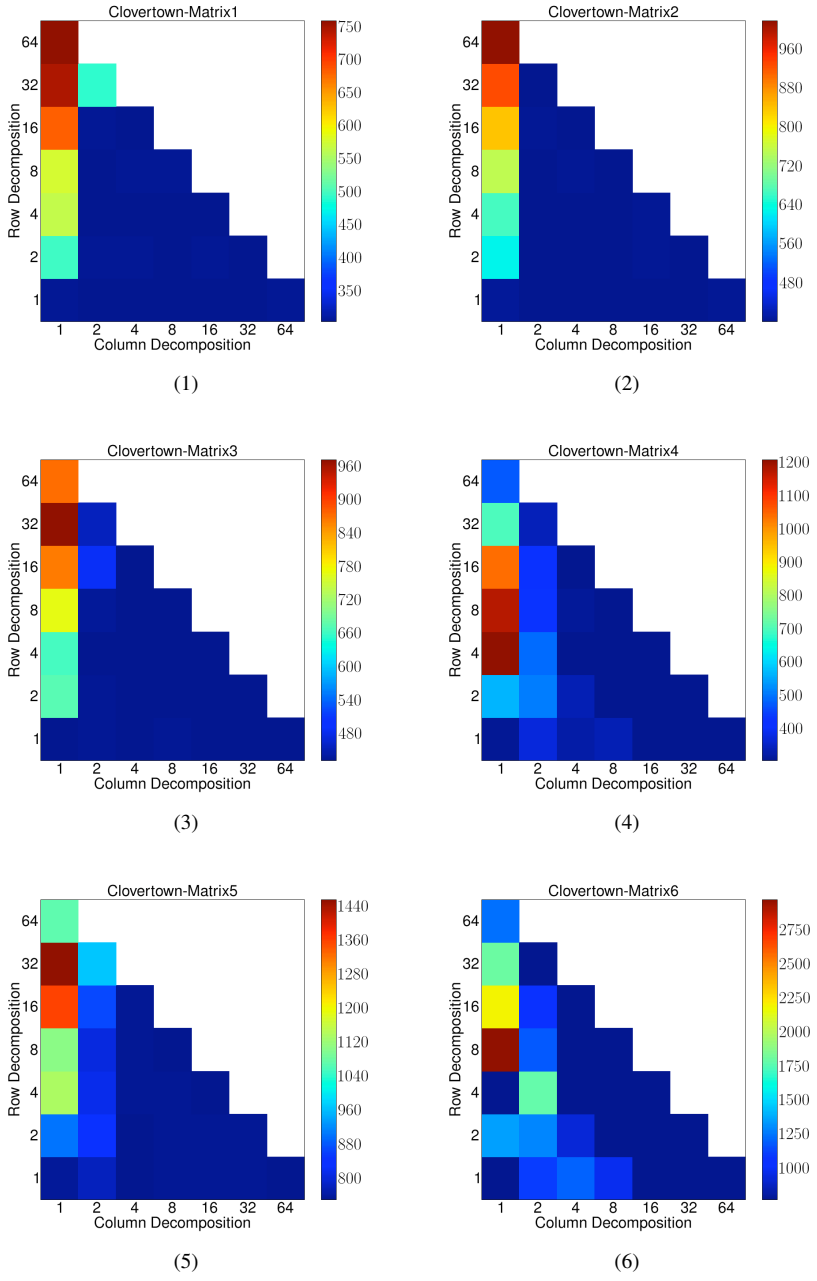
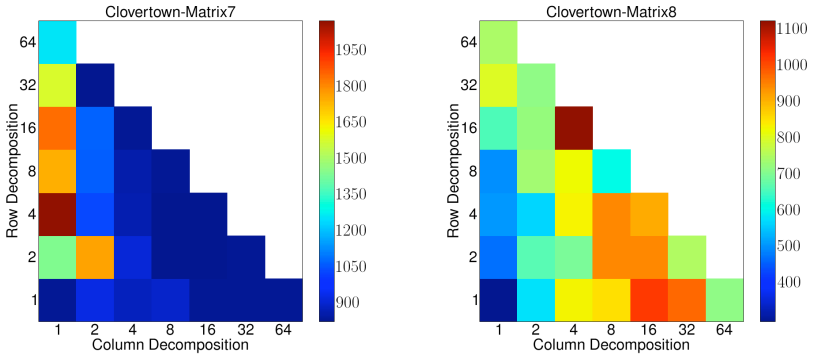
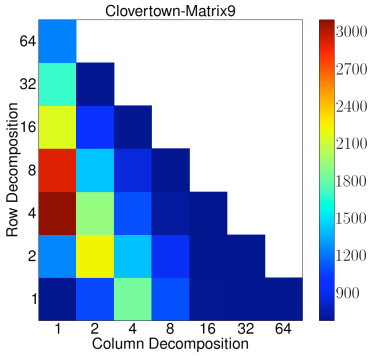


Figure 10: pOSKI Data Decomposition Heatplots for Matrix Suite on Intel Clovertown

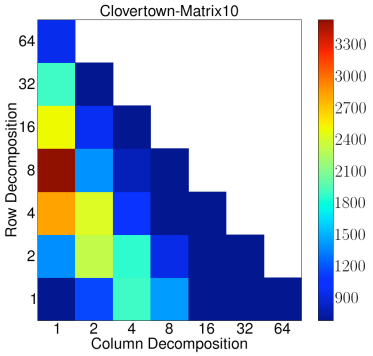


(7)

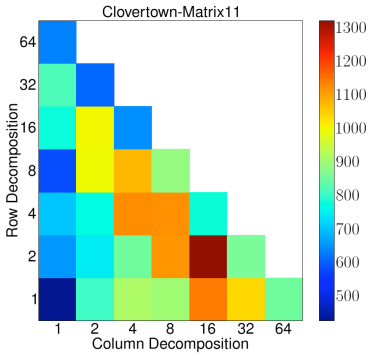
(8)



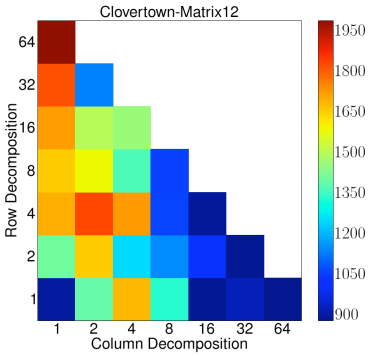
(9)



(10)



(11)

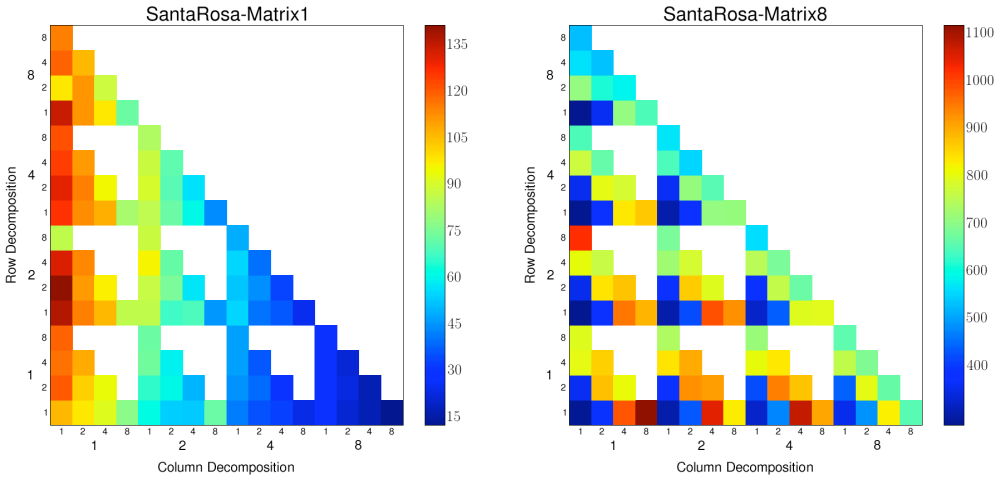


(12)

Figure 10: Continued pOSKI Data Decomposition Heatplots for Matrix Suite on Intel Clovertown

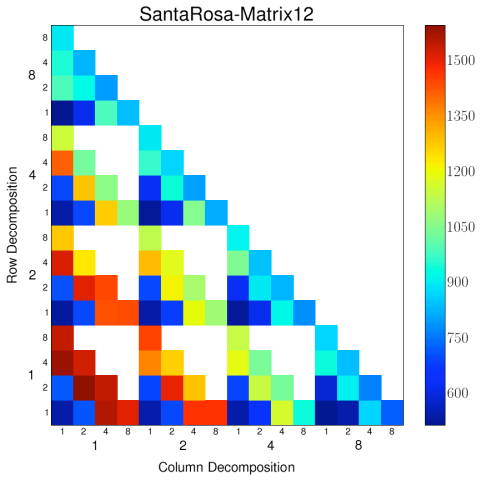
F MPI-pOSKI Data Decomposition Heatplots

There are two numbers on each axis in these heatplots. The outer number represents the decomposition at the MPI-layer while the inner number represents the decomposition at the pOSKI layer.



(1)

(2)



(3)

Figure 11: MPI-pOSKI Data Decomposition heatplots for three focus matrices on AMD Santa Rosa. The outer axis refers to the decomposition done at the MPI layer while the inner axis refers to the decomposition done at the pOSKI layer.

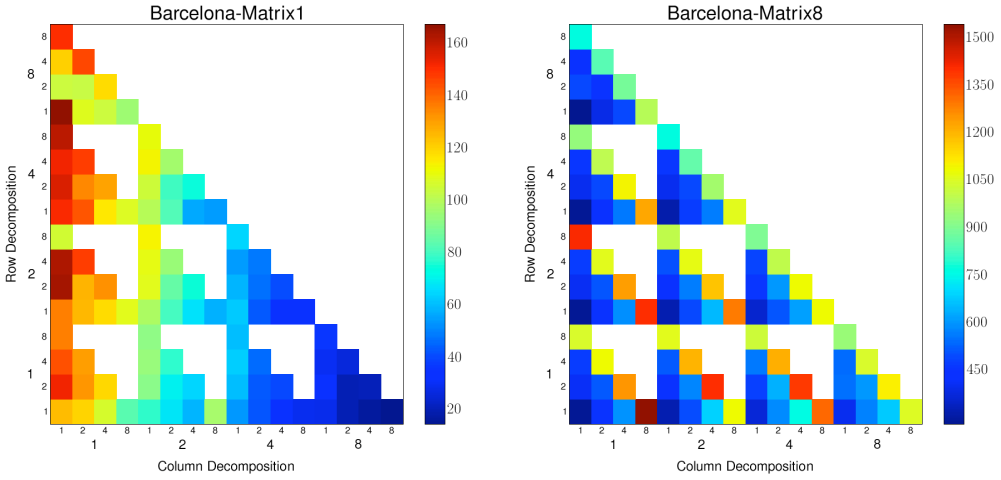
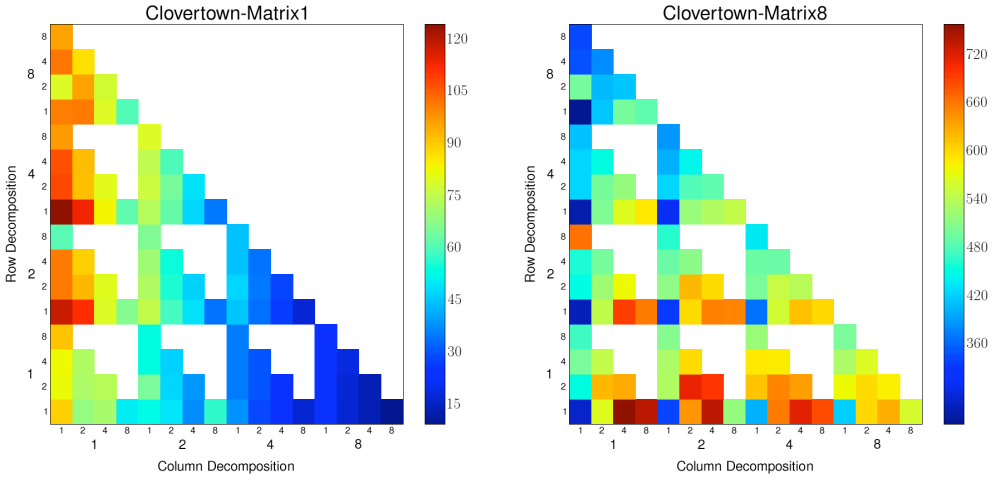
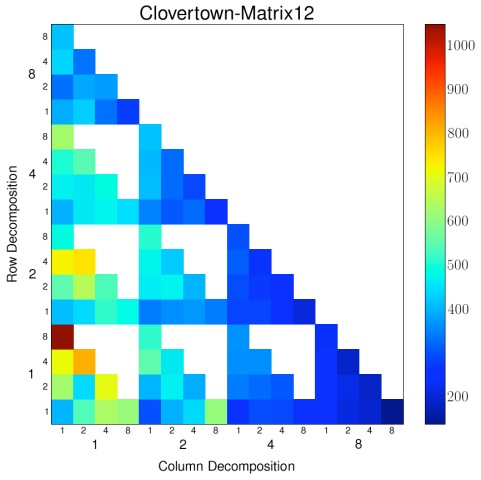


Figure 12: MPI-pOSKI Data Decomposition heatplots for three focus matrices on AMD Barcelona. The outer axis refers to the decomposition done at the MPI layer while the inner axis refers to the decomposition done at the pOSKI layer.



(1)

(2)



(3)

Figure 13: MPI-pOSKI Data Decomposition heatplots for three focus matrices on Intel Clovertown. The outer axis refers to the decomposition done at the MPI layer while the inner axis refers to the decomposition done at the pOSKI layer.