EasyChair Preprint

№ 7471

Exploring Strategies to Improve Locality Across Many-Core Affinities

Neil Butcher and Peter Kogge

February 15, 2022

Exploring Strategies to Improve Locality Across Many-core Affinities

Neil Butcher[1] nbutcher@nd.edu and Peter Kogge[1] pkogge@nd.edu

University of Notre Dame, Notre Dame, IN, USA

Abstract. Several recent rank one systems in the Top500 include many-core chips with complex memory systems, including intermediate levels of memory, multiple memory channels, and explicit affinity of specific memory channels to specific sub-blocks of cores. Creating codes to utilize these features efficiently is thus a significant challenge. This paper uses Intel's Knights Landing (KNL) processor as a testbed, as it includes both intermediate memory and multiple architectural knobs to adjust affinity. This paper also uses a 2D Fast Fourier Transform (FFT) as a test case to explore what combination of architectural and algorithmic techniques are of most benefit. Several codes are used, including state-of-the-art FFT codes FFTW and MKL, along with two additional simple parallel 2D FFT codes exploring explicit options. The conclusions are that intermediate memory does provide a significant boost, that there are architectural modes in the memory subsystem that are better suited to FFT than others.

Keywords: Multilevel Memory · FFT · Cache-Oblivious · Buffering · Affinity

1 Introduction

Processor chips are trending towards increased numbers of cores to increase the achievable flops/s. For many applications, increasing the number of cores creates a higher demand for memory bandwidth. Modern chips increase bandwidth by adding more channels to conventional memory and, in many cases, by adding a new class of memory that may lie in between main memory and the highest cache level. Today, this **Intermediate Memory** (**IM**) is frequently in the form of 3D stacks of DRAM chips, each with multiple channels to the processor chip.

Creating a cache coherent memory system that scales to a number of cores and memory channels is a challenging task. When a core initiates a memory access, the processor has to determine if the data is resident in another cache. If so, the processor routes the value from the core currently holding the data to the core requesting the data. When accessing data not already in a cache, the processor has to determine which memory channel contains the data and then routes the data to the requesting core. Researchers have studied ways to maintain cache coherence in manycore systems. [1,8,7].

The standard cache coherency protocols are a snoopy protocol and directory protocol. Snoopy protocols operate communicate a cache modifying value to all cores. The cores then snoop to see if the change is relevant to their data. Directory protocols rely on a "directory" structure. A directory keeps track of which cache currently has the data. There can be multiple directories or "home agents," each one responsible for maintaining coherence for a different portion of memory.

In many applications, the key to effective use of the memory hierarchy is efficient *memory affinity*: the mapping of data to memory controllers. Linux decides the affinity of memory using a *first-touch* policy, meaning Linux places pages on memory channels based on the first core to access it. A first-touch policy thus enforces an affinity of cores to memory channels. A typical memory affinity groups cores and memory channels based on proximity. Creating groups of cores and assigning them to memory channels that limit the distance accesses need go, given they are on the group's designated memory controller. These groups are referred to as Non-Uniform Memory Access (*NUMA*) domains. Application designers often create codes that minimize accesses across NUMA domains.

There are at least two types of affinity: logical memory affinity and physical memory affinity. *Logical affinity* is the relationship between home agents and the logical address spaces - the mapping between cache directory and logical memory. *Physical affinity* is the relationship between cache directories and the physical address spaces - the mapping between home agents and memory controllers. The KNL makes an interesting test-bed because users can adjust both affinities via setting "clustering modes."

This work explores how the memory affinities in the KNL affect the performance of Fast Fourier Transform (FFT). We run four different FFT codes: one written to be architecture-agnostic but self-tuning (FFTW), one designed explicitly for the KNL architecture (MKL), one that performs thread-buffering, and one designed to be "cache-oblivious". **Cache-oblivious** means it is agnostic to the cache parameters but still has an asymptotic optimal number of cache misses. We adapt the last two to use IM.

The goal of this paper is thus three-fold: first, demonstrate how functional is IM to a real problem (FFT), second, understand how codes can utilize IM efficiently without introducing architecture-specific optimizations, and finally give insight into which of the cluster modes offered by the KNL are of most value. We also analyze the impact that size and shape have on FFT strategies' effectiveness to utilize IM. Together such information should be helpful in architecting future systems that can use IM effectively with relatively simple codes.

The organization of this paper is as follows. Section 2 discusses the baseline architecture we assumed and some details of the implementation. Section 3 reviews the nature of an FFT, and two popular FFT codes used as reference points. Section 4 describes what a cache oblivious algorithm is and our implementation of a cache oblivious algorithm. Section 5 describes our thread buffering algorithm. Section 6 discusses the experimental setup. Section 7 describes the results. Section 8 discusses related work. Section 9 concludes.

2 Architectural Background

Many-core machines are beginning to include IM layers to the memory hierarchy to match the increasing bandwidth demand. Often high-bandwidth IMs have limited capacity, resulting in them being implemented alongside a larger main memory.

Implementing an IM is most beneficial when the problem size is larger then the IM. If the problem fits entirely in the IM, the main memory is obsolete and all data can be stored on IM. IM acting as a cache allows programs to utilize IM without redesigning the code. It also requires data evictions and replacements to be communicated between the memory banks. However, on the KNL there are many cases where configuring IM as a cache actually lowers performance.

Architechtures with IM are still emerging, and the most effective memory structure is uncertain. Memory systems have used various strategies to map different sections of logical address space to a processor's physical memory ports and how to distribute directory information about such mappings among the cores. Comparing these strategies is complex, especially across chips.

Our baseline system for testing uses Intel's PHI 7250 (a.k.a. "Knight's Landing" (KNL)) as its processor. This chip has 34 "tiles" each of which contains two physical cores, a 1MB shared L2, and a "Caching Home Agent" (CHA) that serves as a directory for coherency traffic in some part of the address space [13]. Each of the physical cores is capable of supporting four hyperthreads. A 2D mesh connects the 68 cores, with a subdivision of tiles into four "quadrants." The cores can also be divided into two hemispheres. There are two memory controllers and six conventional DDR4 ports (three per controller) to provide roughly 96GB of storage with 90GB/s of bandwidth.

The KNL also has a configurable IM called MCDRAM implemented with eight 3D stacks of DRAM chips, each with 2GB capacity and a separate port into the processor chip. Two such ports are physically close to each quadrant of the chip. The behavior of the MCDRAM is adjusted in the BIOS at boot time and can act as an extension to the main memory ("flat mode"), a large L3 cache ("cache mode"), or a hybrid. The aggregate MCDRAM has roughly 16GB of storage and provides approximately 400GB/s bandwidth.

Given a large number of cores and memory ports, how does a load/store that misses all on-chip caches make its way to the appropriate physical memory port while also performing cache coherency checks? First, the KNL allows at boot time several different "clustering modes" for defining how CHAs are associated with physical MCDRAM ports. Coupled with this are options as to how to associate different logical memory pages to a particular CHA.

Fig. 1 demonstrates the available clustering modes on the KNL. Fig. 2 illustrates how the clustering mode affects how logical memory maps to memory controllers, as well as CHAs to memory controllers. In Fig. 2, CHAs and core groups with the same color mean the core group sends requests directly to those CHAs. Memory controllers and portions of the logical address space that are the same color mean that the memory controller is responsible for that portion of the logical address space. In all modes, the CHAs communicate to ensure cache

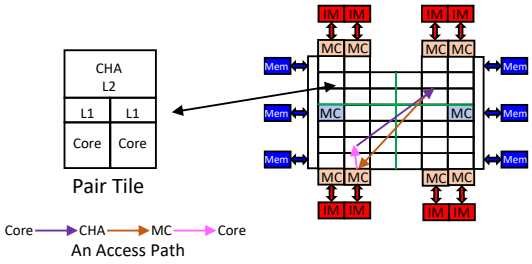

Fig. 1. KNL Access Modes.

coherency. There is a potential for collision when CHAs are passing messages. A CHA can only pass so many messages in a single cycle, creating the possibility of network congestion.

The simplest mode is "all-to-all" (A2A), where the physical CHAs mapping different partitions have no relationship to the nearest physical IM port connected to the associated physical memory. It appears that in this mode, the relationship between logical pages and controlling CHAs is one-to-one but unconstrained. Thus a reference to two sequential pages of IM may end up going to two different CHAs almost anywhere on the chip. In this case, an L2 cache miss, tag directory probe, and data access could be on opposite sides of the chip. This option has the downside of memory requests having to traverse many CHAs before reaching the memory controller. The advantage is that it reduces the probability of collisions when many cores are accessing memory simultaneously.

In contrast, "Quadrant mode" ensures data is in the same quadrant as the CHA managing it. Sequential pages in logical memory are striped across the quadrants, increasing the memory controllers' utilization. Each page is stored inside a quadrant, with the following pages being stored in a different quadrant. The downside is a strided access pattern could have all accesses go to the same memory controller, resulting in contention.

The "Hemisphere mode" is similar but divides the memory ports in half rather than quarters. However, it appears that two sequential pages in logical space may still be in two different quadrants/hemispheres of the chip.

"Subnuma Clustering Four (SNC-4)" is similar to Quadrant mode in that a CHA only handles requests to memory controllers in its quadrant. The distinction from Quadrant mode is that in SNC-4 mode each quadrant acts as a separate NUMA node. SNC-4 aggregates logical memory in each quadrant to act as a contiguous partition of the address space. Thus if all cores in a quadrant work on the same data, no memory reference to the shared partition will cross a quadrant boundary. For example, this would allow four MPI ranks to run essentially independent of each other, at least in memory access traffic. In SNC-4, the memory controllers in each quadrant operate on a contiguous block of memory. Localized memory operations are more efficient, but accesses across the quadrants are expensive. For example, streaming a large block of data in

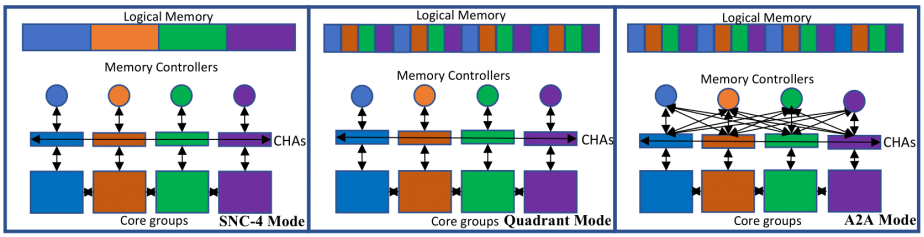

Fig. 2. Examples of directory and logical memory layout in three clustering modes

another quadrant creates a high amount of traffic and increases the chance for accesses to interfere with each other. SNC-4 is effective if the programmer is aware of memory affinity in the program and localizes data accesses.

"Subnuma Clustering Two (SNC-2)" is similar but where the CHAs controls half the logical memory space in the physical half closest to the memory ports holding the associated physical memory. [17].

Two other many-core chips used in TOP500 systems with similar properties are the A64FX and SW26010. Both have the equivalent of 4 quadrants. The A64FX has 12 cores and eight channels to a separate HBM per quadrant. Each quadrant in the A64FX has a local memory channel but maintains cache coherency. A64FX has not adjustable clustering mode and most closely resembles the KNL in SNC-4 mode. The SW26010 has a scratchpad IM for each of its 256 compute cores and a DDR3 channel for each quadrant of 64.

3 Fast Fourier Transforms

Fast Fourier Transforms (FFT) have a non-unit stride access pattern inherent in the algorithm [16]. The access pattern of FFT computations are challenging for cache prefetchers to predict. Thus the memory system plays an important role of FFT-based codes. Our work focuses on 2D FFTs, which take an NxM complex matrix as the input and produces an NxM complex matrix as the output. A typical algorithm uses a *pencil decomposition* i.e., it computes a series of 1D FFTs or "pencils." We first compute a 1D FFTs of each column, and then 1D FFTs of each row, i.e., M pencils of size N, then N pencils of size M. The matrices are stored in row-major order and we transpose the input matrix to keep pencils contiguous when necessary.

A 1D FFT of size N has a complexity of $O(Nlog_2(N))$. Likewise, the complexity of a 2D FFT is $O(NMlog_2(NM))$. Note that NM is the size of the matrix in complex points, i.e., the problem size. For this work, we put the dimensions in terms of a ratio r such that M=N*r making the computational complexity $O(N^2r\ log_2(N^2r))$. By keeping the problem size constant, the number of flops is unchanged, but by changing r we see a change in the overall performance, which we attribute to the memory access patterns. We examine ratios ranging from $r = 1/512$ (tall/skinny) to $r = 512$ (short/fat).

For the KNL, we use as a baseline two respected multi-dimensional FFT codes, MKL and FFTW. The FFTW3 library adapts to the hardware on which it is running [11]. During a preliminary phase, FFTW looks at the problem size, vector capabilities, and the number of available threads. Using this information FFTW measures the performance of a variety of FFT algorithms before choosing the most efficient strategy represented by what they refer to as ''plans." Plans can be precomputed and saved for reuse, or calculated at run-time. FFTW plans are hardware agnostic but can make use of vector units if compiled correctly. The planning phase is often time-consuming, but can provide speedup if multiple FFTs of the same size reuse the same plan. Using IM is not an option the planner explores. Experiments with FFTW on the KNL have to either treat MCDRAM as main memory (with the matrix preloaded into it and the result going back to it), or another layer of cache.

The FFTW library is memory affinity unaware, meaning when invoking the planner with a first-touch policy, it does not attempt different memory affinities. The planner uses as input a specified memory allocation, allowing the user to provide a memory affinity before invoking the planner. We have seen no information suggesting FFTW is NUMA-aware.

In contrast Intel developed the MKL library to specifically target Intel processors and utilize all available features. The MKL FFT routines support the same interface as FFTW, allowing FFT applications to be easily ported to MKL. Options exist in the MKL library to leverage MCDRAM but based on the performance we have observed, we do not believe MKL manages the IM at runtime.

4 Cache Oblivious Programming

For the third of our codes, we use the established technique of cache-oblivious algorithms. Frigo et al. introduced such algorithms [12] and others have implemented cache-oblivious algorithms on real systems [6,18]. Cache-oblivious algorithms often use a divide-and-conquer strategy. Most cache-oblivious algorithms have recursive calls breaking the problem into parts and solve in a depth-first manner. The base case is typically a problem so small that solving it is trivial. The recursion ensures effective use of cache because two trivial solutions are produced at the bottom level and then reused in subsequent recombination steps. As computation gets higher in the recursion tree, the size of the working set grows. The lower levels of recursion make efficient use of the cache without awareness of the cache size. Once the recursion reaches a point where the problem no longer fits in the cache, the cache is no longer used efficiently.

To prove the algorithms are optimal a **ideal cache model** is used which assumes the cache will make optimal decisions on replacement choices. The ideal cache is assumed to be tall (large capacity with short cache lines) and fully associative with an optimal offline replacement strategy that always makes an optimal choice, i.e., the data accessed furthest in the future is chosen to be replaced. A typical L1 cache found in a processor will often approximate “ideal” behavior in many applications.

Researchers have updated the cache-oblivious model to more closely resemble modern architectures. They extended the sequential cache-oblivious work with multicore cache-oblivious algorithms [6,9]. Cache oblivious research was later extended to include dynamically sized caches in **cache adaptive algorithms** [2,5,4,3]. The multithreaded CO work assumes caches are non-interfering, or more precisely, data accesses from one processor does not force data to be evicted from another cache. When threads operate independently on separate regions of memory, this assumption is realistic. When moving to manycore machines, coherency and data sharing has to be considered to represent performance accurately. Manycore processors with a large shared IM cache results in each core having a dynamically sized local cache. Using IM as a cache leads to contention among threads, permitting the amount of cache available to each thread dynamic. In our work, creating cache-adaptive algorithms benefits performance because cache size will naturally vary.

The original cache-oblivious FFT code was introduced by Frigo in [12]. The algorithm computed a 1D FFT of size n by viewing the 1D input into a matrix of size n_1 and n_2 such that $n_1 * n_2 = n$. Frigo's strategy chooses $n_1 = 2^{\lceil log_2(n) \rceil}$ and $n_2 = 2^{\lfloor log_2(n) \rfloor}$. The 1D input is transposed into a $n_2 * n_1$ matrix. Then n_2 FFTs of size n_1, are computed, and each element is multiplied by a twiddle factor. The matrix is transposed again, then we compute n_1 FFTs of size n_2. The FFT is solved through recursion until the subproblem is just a single element. Finally the matrix is transposed again to put the output in correct order.

In our work, we focus on solving 2D FFTs. We solve pencils of the columns, then rows. We perform transposes to make the pencil contiguous in memory. We use a similar approach to Frigo's cache-oblivious algorithm to solve the 1D FFT; with the slight modification, we choose $n_1 = 64$ and solve the subproblems invoking MKL. By choosing n_1 to be small, we avoid a great deal of the overhead of recursion and still increase cache locality.

5 Thread-Level Buffering

Thread buffering assumes two memories, a large, slower "main memory" and smaller IM memory that provides a bandwidth improvement. The idea is based on conventional buffering techniques where the input is broken into chunks approximately the size of the fast memory. A single chunk is moved into fast memory at a time. The subproblem in fast memory is computed and then moved back to main memory. Once all the chunks have been computed, the chunked solutions are merged to produce a final result. A similar strategy has been implemented successful on multicore processors in the work of Popovici et. al. [14]. Popovici acheives speedup of 1.2-3x speedup over FFTW and MKL by utilizing some of the threads to perform 'soft DMA' operations.

Our fourth FFT code is specially written to use explicit thread level buffering to compute a FFT. To do this we divide threads into compute threads and copy threads. Each compute thread is given three buffers that are the size of the largest dimension of the 2D FFT. There are three buffers per thread to overlap

computation with data movement. While one buffer is being computed on, one of the others is copied out, and the other has data copied in. Each copy thread is assigned to specific compute threads. The copy threads are placed in the same tile as the compute thread they manage, but on a separate physical core. In this work we chose a one-to-one ratio of copy threads to compute threads. The copy threads perform non-temporal accesses so they do not interfere with the compute threads. While a compute thread is operating on a block, the copy thread is able to move out the last block and bring in the next block. If the copy threads do not transport data quickly enough the compute thread idles, waiting for the copy to finish. We chose a one-to-one ratio because one thread the copy operations could be completed before the computation.

We use both implicit and explicit copying of data into IM. Implicit copying refers to putting IM in cache mode and having copy threads act as soft DMA devices by accessing the data they want to move into IM. In explicit copying, the copy threads directly move data into IM. The disadvantage of explicit copying is that it requires two memory accesses to perform the copy (a load and a store for each data element). Implicit copying requires loading a single address of memory and relies on the hardware/OS to efficiently move data into IM, but performs poorly if the accessed data set is too large, resulting in thrashing in the cache.

6 Methodology

Our experiments compare four codes: MKL, FFTW, Cache Oblivious (CO), and thread buffering (TB). We run all codes with 64 threads, except the thread-buffering code, which has 128 threads. We run in 5 cluster modes (SNC-2, SNC-4, A2A, Quad, Hemi) with the MCDRAM in flat and cache mode. The CO, MKL, and FFTW codes do not use the MCDRAM in flat mode, so those options are not run. We run 5 ratios (1/512,1/8,1,8,512) with 4 problem sizes for each ratio. All problems use dimensions that are a power of two, and for each ratio we ran problem sizes with (6.71E7, 2.68E8, 1.07E9, 4.29E9) complex numbers, except the ratio 1 which used (3.35E7,1.34E8,5.37E8,2.15E9). The 1x ratio used different problem sizes due to the limitations of powers of two.

We run the thread buffering code in three forms, flat mode with buffers in IM, flat mode with buffer in main memory, and cache mode with the buffers also in memory. The TB code runs with the `KMP_AFFINITY` variable set to `balanced` with 64 compute and 64 copy threads. Copy threads are placed on the same core as the compute threads they manage. We run MKL version 2020.0.166 using the FFTW interface. We create plans using the `FFTW-ESTIMATE` flag. We do not include the planning time in our measurement of the run time analysis.

7 Results

We compare the execution time of the different FFT algorithms in quadrant mode in Fig. 3. To keep the chart readable, we only include the largest problem sizes we ran at each ratio. The “thread-buffering IM” strategy runs in flat mode

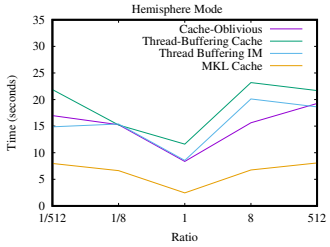

Fig. 3. Cache Quadrant mode

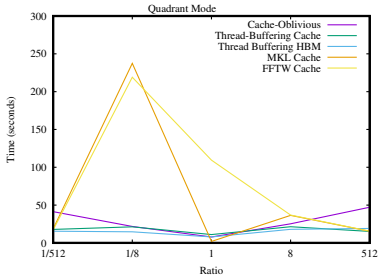

Fig. 4. Cache Hemisphere mode

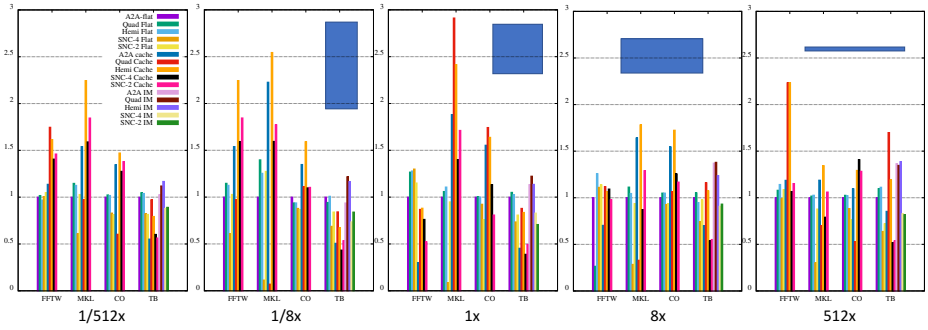

Fig. 5. Speedup of FFT codes relative to A2A flat. Each chart represents a separate ratio. The rectangle in the top left approximates the shape of the input matrix

with the buffers allocated to MCDRAM. The rest of the lines are runs with MCDRAM in cache mode. MKL performs best in quadrant mode on square problems, with a relative speedup of roughly 4x over the cache-oblivious algorithm. These speedups vary in magnitude with smaller problem sizes of the same ratio, but the general trend shows the CO algorithm performs at its best at large problem sizes. The cache-oblivious strategy appears to perform slightly better than the thread buffering, with a performance difference of 1-1.5x.

We show a similar chart for FFT in hemisphere mode in Fig. 4. Performance in this mode is more consistent than for quadrant mode, but quadrant mode has better performance in many cases. MKL computes a square problem in hemisphere mode up to four times faster than the cache-oblivious algorithm. MKL is roughly 2x faster than cache-oblivious FFT code for the 512 ratio cache mode. The square ratio has a smaller problem size, hence the lower execution time.

Fig. 5 shows the speedup of each code relative to A2A flat mode at five different ratios. The figure shows the variance of the codes across different clustering modes and ratios. The biggest performance gain MCDRAM gives our FFT computations is quadrant mode ran on a square problem. Quadrant mode exhibit

generally the best performance of all the clustering modes. In a few cases and algorithms, the hemisphere algorithm performs the best.

For square FFTs, MKL gains roughly 2.5x speedup from utilizing the IM in quadrant mode and 2.3x in hemisphere mode. At nonsquare ratios, IM is consistently beneficial in hemisphere mode, with MKL having speedups of from 1.3x-2.5x. MKL in SNC-4 cache and SNC-2 cache has improved performance square ratios with a steady decrease in performance at larger ratios, particularly SNC-4 cache.

The cache-oblivious algorithm benefits from MCDRAM acting as a cache in almost all of the clustering modes. MCDRAM does not improve the performance of the CO FFT on the 512 and 1/512 ratios in quadrant mode and square ratio in SNC-2. The cache-oblivious does not vary widely between clustering modes, suggesting the code is portable to other architectures with IM. In hemisphere mode, utilizing the IM cache improved the performance of the CO algorithm by 1.4x-1.6x across all the ratios.

The thread-level buffering algorithm consistently gains performance managing the IM memory explicitly, particularly in A2A, quadrant, and hemisphere mode. Copy threads moving data into IM buffers increases the overall bandwidth utilization and improves performance. The effectiveness of copy threads demonstrates that FFT is a bandwidth-limited problem. The thread-level buffering strategy does not appear effective in SNC-2 or SNC-4 mode, even with the buffers allocated to IM. We ensure the buffers are in the same quadrant as the threads utilizing them, but the incoming data copied could reside in any quadrant. The thread-level buffering algorithm in hemisphere mode gains performance of roughly 1.2x-1.4x. Since most of the increased bandwidth utilization comes from the copy threads, thread-level buffering does not exhibit the same performance gains the other strategies do.

8 Related Work

Popovici et al. [15] improves 2D and 3D FFTs by repurposing cores/threads as soft direct memory access (DMA) engines, with an improvement of up to 1.3x over MKL and FFTW. They utilize a highly efficient 1D algorithm to perform the 'pencil' computations, and focus on improving performance on FFTs on nodes with significant main memories. They note that for the processors they tested, FFTW/MKL achieved at most 47% of the peak FLOPs. In comparison, the highest performance we have seen from the KNL is roughly 13% of peak.

The Locality Aware Roofline Model [10] builds a roofline model that applies to NUMA machines, especially the KNL. They identify three main bottlenecks in memory accesses in a NUMA memory system: congestion, contention, and remote access. Congestion is when many data requests simultaneously go through a single CHA, delaying the data requests. Congestion occurs when many cores are accessing a single memory bank. The memory bank can only process a certain number of requests simultaneously, resulting in delayed responses. Each of the bottlenecks delays the overall memory requests of a system. They also discuss a

method to create additional roofs based on the KNL hardware and the memory access patterns of the problem. They use a series of simple kernels to explore the variations in performance on the KNL. Often in manycore memory hierarchies, the available bandwidth fluctuates with problem size and algorithm. We relate this to our work because varying the problem size and ratio of an FFT naturally leads to these bottlenecks. It is often difficult to tell if/which of these bottlenecks is happening without carefully observing performance.

Cache adaptive algorithms are not aware of the size of the cache just like cache oblivious algorithms, with the distinction they remain optimal even if the cache size changes dynamically. This distinction comes up in a variety of situations, but in our work it occurs when multiple threads are contesting a shared cache. Shared caches have to be considered very differently than private caches, and being cache oblivious is not sufficient to state an algorithm is cache adaptive [5].

9 Conclusion

We provided a detailed explanation of the different clustering modes available to the KNL. We demonstrate how the clustering mode impacts the performance of a 2D FFT using many different sizes, shapes, and algorithms. All of the algorithms gain performance when utilizing IM in one mode or another. Whether or not an algorithm effectively utilizes the IM varies with the clustering mode. Many of the algorithms we ran have excellent speedup with the IM in quadrant mode on square ratios but perform poorly on non-square ratios. Hemisphere consistently benefits from the IM configured as a cache across all ratios.

The exact reason hemisphere mostly outperforms quadrant mode is unclear, although we have some ideas. Often FFTs have powers-of-two strided memory accesses, which causes makes accessing memory problematic. One typical example is every access mapping to the same cache set, reducing the effective cache size. We suspect a large enough stride will result in subsequent accesses to map to the same CHA. Note that the KNL has 34 CHAs, so two quadrants will have eight CHAs, the other two will have nine. The quadrants with eight CHAs may have implications for strides that are a power of two. In future work, we plan to do a more investigative analysis of how striding affects CHAs.

We implemented thread-level buffering and cache-oblivious FFT codes that have competitive performance with highly optimized codes. Both our FFT codes contain no architecture-specific optimizations. Both of these algorithms benefit from the IM across all the ratios. Our results show the most effective clustering mode to compute a square FFTs is quadrant, and hemisphere for non-square problems. When computing both square and non-square FFTs, the best strategy is to run in the hemisphere as the quadrant tends to be more erratic in performance. Our thread buffering algorithm is an effective strategy to use the IM in flat mode, which no other existing FFT codes does, to our knowledge.

References

1. Al-Hothali, S.: Snoopy and directory based cache coherence protocols: A critical analysis. J. of Information & Communication Technology (JICT) **4**(1), 11 (2010)
2. Barve, R.D., Vitter, J.S.: A theoretical framework for memory-adaptive algorithms. In: 40th Symp. on Foundations of Comp. Sci. (Cat. No. 99CB37039). pp. 273–284. IEEE (1999)
3. Bender, M.A., Chowdhury, R.A., Das, R., et al: Closing the gap between cache-oblivious and cache-adaptive analysis. In: Proc. 32nd ACM Symp. on Parallelism in Algorithms and Architectures. pp. 63–73 (2020)
4. Bender, M.A., Demaine, E.D., Ebrahimi, R., et al: Cache-adaptive analysis. In: Proc. 28th ACM Symp. on Parallelism in Algorithms and Architectures. pp. 135–144 (2016)
5. Bender, M.A., Ebrahimi, R., Fineman, J.T., Ghasemiesfeh, G., Johnson, R., McCauley, S.: Cache-adaptive algorithms. In: Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms. pp. 958–971. SIAM (2014)
6. Blelloch, G.E., Gibbons, P.B., Simhadri, H.V.: Low depth cache-oblivious algorithms. In: Proc. 22nd ACM Symp. on Parallelism in algorithms and architectures. pp. 189–199 (2010)
7. Caheny, P., Casas, M., Moretó, M., et al: Reducing cache coherence traffic with hierarchical directory cache and numa-aware runtime scheduling. In: 2016 Int. Conf. on Parallel Arch. and Compilation Techniques (PACT). pp. 275–286. IEEE (2016)
8. Chaiken, D., Fields, C., Kurihara, K., Agarwal, A.: Directory-based cache coherence in large-scale multiprocessors. Computer **23**(6), 49–58 (1990)
9. Chowdhury, R.A., Ramachandran, V., Silvestri, F., Blakeley, B.: Oblivious algorithms for multicores and networks of processors. J. of Parallel and Distributed Computing **73**(7), 911–925 (2013)
10. Denoyelle, N., Goglin, B., Ilic, A., et al: Modeling large compute nodes with heterogeneous memories with cache-aware roofline model. In: Int. Workshop on Perf. Modeling, Benchmarking and Simulation of High Perf. Computer Systems. pp. 91–113. Springer (2017)
11. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005)
12. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proceedings of the 40th Annual Symposium on Foundations of Computer Science. p. 285 (1999)
13. León, E.A., Hautreux, M.: Achieving transparency mapping parallel applications: A memory hierarchy affair. In: Proc. Int. Symp. on Memory Systems. pp. 185–189 (2018)
14. Popovici, D.T., Low, T.M., Franchetti, F.: Large bandwidth-efficient FFTs on multicore and multi-socket systems. In: 2018 IEEE Int. Parallel and Distributed Processing Symp. (IPDPS). pp. 379–388. IEEE (2018)
15. Popovici, D.T., Low, T.M., Franchetti, F.: Large bandwidth-efficient FFTs on multicore and multi-socket systems. In: 2018 IEEE Int. Parallel and Distributed Processing Symp. (IPDPS). pp. 379–388. IEEE (2018)
16. Rockmore, D.N.: The fft: an algorithm the whole family can use. Computing in Science and Engineering **2**(1), 60–64 (2000)
17. Weinberg, V.: Prace autumn school 2016-intel xeon phi programming (2016)
18. Yotov, K., Roeder, T., Pingali, K., et al: An experimental comparison of cache-oblivious and cache-conscious programs. In: Proc. 19th ACM Symp. on Parallel algorithms and architectures. pp. 93–104 (2007)