CHA and Core Discovery on Intel Chips and Generating Optimized Thread Bindings

by

Aydın Özcan

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

January 26, 2024

CHA and Core Discovery on Intel Chips and Generating Optimized Thread Bindings

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Aydın Özcan

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Dr. Didem Unat (Advisor)

_____

Asst. Prof. Dr. M. Emre Gürsoy

_____

Asst. Prof. Dr. Ayşe Yılmazer

Date: _____

ABSTRACT

CHA and Core Discovery on Intel Chips and Generating Optimized Thread Bindings

Aydın Özcan

Master of Science in Computer Science and Engineering

January 26, 2024

In modern multi-core architectures with distributed directory-based cache coherence, each memory address is overseen by a distributed directory unit, known as a Caching/Home Agent (CHA), that monitors cache line state and location. Neither the CHA nor core locations in a processor are directly exposed to the programmer. In this work, we firstly analyze and compare the methodologies for uncovering both the CHA and core topology of Intel Xeon Scalable processors, as well as the methods to reveal the mapping of memory addresses to CHAs. Leveraging the topology and the address mapping information, we investigate the impact of spatial proximity between communicating cores and CHAs on application performance, and propose a thread mapping heuristic that assigns threads to cores by considering cache coherence traffic. We expect our heuristic to achieve significant performance gains on applications with high amount of on-chip cache coherence traffic due to high percentage of shared written data. We evaluated our heuristic on applications that exhibit high amount of on-chip communication traffic. The heuristic achieves up to 5.6% speedup over compact placement on merge-based SpMV application, up to 8% with an average of around 4.4% on Barnes application, around 25% for Fluidanimate application to simulate 60 frame per second, and lastly approximately 6% for LU across different matrices. We also prove the improved performance is in fact related to reduced on-chip traffic on the mesh.

ÖZETÇE

CHA ve Çekirdek Topolojisiyle Uyumlu İş Parçacığı Haritalaması
Aydın Özcan
Bilgisayar Bilimleri ve Mühendisliği, Yüksek Lisans
26 Ocak 2024

Dağıtık dizin temelli önbellek tutarlılığına sahip modern çok çekirdekli bilgisayar mimarilerinde her bir bellek adresi, ona atanmış olan bir dağıtık dizin birimi tarafından yönetilir. Bu birime Önbellekleme/Merkez Aracısı (ÖMA) ismi verilir ve birim, önbellek satırını gözlemler. ÖMA ve çekirdeklerin fiziksel konumları programcılar tarafından bilinmez. Bu çalışmada öncelikle, Intel Xeon işlemciler için ÖMA ve çekirdeklerin konumlarını açığa çıkaran farklı yöntemlerin analizi ve kıyaslaması yapılmıştır. Bununla birlikte bellek adreslerinin ÖMA birimlerine haritalamasını yapan yöntemlerin de analiz ve kıyaslaması yapılmıştır. Topoloji ve adres haritalaması bilgisi kullanılarak, birbiriyle haberleşen çekirdekler ve ÖMA'ların arasındaki fiziksel mesafenin uygulama performansı üzerindeki etkileri üzerine araştırma yapılmıştır. Bu araştırmadan yola çıkarak önbellek tutarlılığını sağlayan trafiği azaltmayı hedefleyen ve iş parçacıklarının çekirdeklere atanmasıyla görevli bir iş parçacığı haritalama algoritması geliştirilmiştir. Geliştirdiğimiz algoritmanın, iş parçacıkları arasında paylaşımlı yazılabilir bilginin yüksek oranda mevcut olduğu uygulamalarda performansı geliştirmesini bekliyoruz. Bu algoritma, yüksek oranda yonga trafiğine sebep olan uygulamalar üzerinde test edilmiştir. Ardışık iş parçacığı haritalamasına kıyasla Seyrek matris-vektör çarpımında %5.6'ya, Barnes'ta %8'e, sıvı akışkanlığı simülasyon uygulamasında %25'e, LU ayrıştırmasında %6'ya varan hızlanmalar sağladığı gözlemlenmiştir.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ccNUMA	Cache-Coherent NUMA
CHA	Caching-Home Agent
CLK	Cascade Lake Microarchitecture
CPU	Central Processing Unit
CSR	Compressed Sparse Row
DRAM	Dynamic Random-Access Memory
GNU	GNU's Not Unix
IMC	Integrated Memory Controller
KNL	Knight's Landing Microarchitecture
LLC	Last Level Cache
LU	Lower-Upper Decomposition
MCDRAM	Multi-Channel DRAM
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
PA	Physical Address
PARSEC	Princeton Application Repository for Shared-Memory Computers
PCI	Peripheral Component Interconnect
PMA	Power Management Agent
POSIX	Portable Operating System Interface for Unix
QPI	QuickPath Interconnect
SIMD	Single Instruction Multiple Data
SKX	Skylake Microarchitecture
SPLASH	Stanford Parallel Applications for Shared-Memory
SpMV	Sparse Matrix–Vector Multiplication
UPI	Ultra Path Interconnect
VA	Virtual Address

Acknowledgements

I extend my deepest gratitude to Assoc. Prof. Dr. Didem Unat, my advisor, for her unwavering guidance, invaluable insights, and constant support throughout the journey of this research. Her expertise and encouragement have been instrumental in shaping the trajectory of my work. A special thanks to Dr. Muhammad Aditya Sasongko, my esteemed project partner and colleague. His collaboration, enthusiasm, and shared dedication have enriched the depth of this research, making it a truly collaborative effort. I am immensely grateful to my wife for her boundless patience, understanding, and unwavering support. Her encouragement and belief in me have been my pillars of strength, providing the motivation to persevere through challenges. To my former work colleague, Ali Volkan Atli, I extend sincere appreciation for planting the seed of ambition within me. His encouragement to embark on this master's journey was a pivotal moment that set me on this path of academic pursuit. This thesis stands as a testament to the collective support and belief of these individuals, without whom this journey would not have been as rewarding. Thank you for being the guiding lights on this journey.

This thesis was supported in part by the European High-Performance Computing Joint Undertaking under grant agreement No 956213, the Royal Society-Newton Advanced Fellowship, by the Turkish Science and Technology Research Centre Grant No 120E492, 120N003 and from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949587).

Chapter 1

INTRODUCTION

Cache coherence traffic poses a significant performance burden in parallel programs that utilize shared memory [Kommrusch et al., 2021]. This traffic encompasses not only the transfer of cache lines between cores but also the exchange of other message types across core tiles to maintain cache coherence protocol [Caheny et al., 2018]. The responsibility of handling these messages lies with the CHAs located alongside CPU cores in each tile [Horro et al., 2019a]. To reduce latency in on-chip communication, it is not enough for communicating threads to be physically close to each other; they must also be in close proximity to the corresponding CHAs that manage the coherence of the cache lines used for inter-thread communication. Therefore, optimizing performance by reducing cache coherence traffic in these multicores must consider both the physical arrangement of cores and CHAs.

Topology information is not readily provided by vendors, requiring reverse engineering methods to reveal it. For instance, in Intel processors, CHA locations can be disclosed through a PCI configuration space register [McCalpin, 2021b], while core locations can be unveiled using uncore performance monitoring counters [McCalpin, 2021b,Wan et al., 2021]. Additionally, the mapping of addresses to CHAs can be deduced by reverse engineering the hash function that links physical addresses to their corresponding CHAs, which manage the coherence of these addresses [Kommrusch et al., 2021]. This mapping can also be derived using uncore performance counters using LLC (last level cache) lookups [McCalpin, 2021a].

In this study, we first revisit the techniques used to disclose core and CHA topology, as well as the address-to-CHA mapping in a profiled application. We delve into a comprehensive comparison, discussing the relative strengths and weaknesses of these techniques. Based on the distributed directory topology and address-to-CHA mapping derived from these methods, we developed a synthetic multithreaded

UPI	PCIe	PCIe	RLink	UPI2	PCIe
CHA-0 Core-0	CHA-4 Core-4	CHA-9 Core-36	CHA-14 Core-26	CHA-19 Core-50	CHA-24 Core-2
IMC	CHA-5 Core-32	CHA-10 Core-24	CHA-15 Core-54	CHA-20 Core-6	IMC
CHA-1 Core-28	CHA-6 Core-20	CHA-11 Core-52	CHA-16 Core-10	CHA-21 Core-34	CHA-25 Core-30
CHA-2 Core-16	CHA-7 Core-48	CHA-12 Core-12	CHA-17 Core-38	CHA-22 Core-18	CHA-26 Core-14
CHA-3 Core-44	CHA-8 Core-8	CHA-13 Core-40	CHA-18 Core-22	CHA-23 Core-46	CHA-27 Core-42

Figure 1.1: Topology of two socket 56 core Intel Cascade Lake, showing only one socket

benchmark that allows for a configurable volume of inter-thread communication and address-to-CHA mapping. Through this benchmark, we scrutinize the impact of proximity among communicating cores and CHAs on application performance. Leveraging the insights gained from the benchmark, we design a thread mapping heuristic that considers distributed directory topology and address-to-CHA mapping on CPU sockets with a primary aim of cache coherence traffic.

Here are the highlights of our findings and contributions:

- We evaluate methods to uncover CHA and core topology for Intel Skylake/-Cascade Lake microarchitectures, showcasing the Cascade Lake topology in Figure 1.1.

- To unveil core topology, one of the two methods (power monitoring) is easier to implement and reason about its results. The second method (traffic monitoring) incurs significantly higher memory overhead.

- Both core toplogy discovery methods are accurate, but the power monitoring method requires repeated measurements for stability, increasing its runtime overhead.

- We explore two methods to discover the mapping of addresses to CHAs: LLC lookup and hash-function-based methods. While both methods can achieve the same accuracy, the hash function approach incurs significantly lower overhead.

- We create a synthetic multi-threaded benchmark with configurable inter-core communication, allowing users to control thread communication volume and map threads and data to physical cores and CHAs.

- This benchmark reveals that there is a positive correlation between execution time and the spatial proximity between communicating cores and the CHAs managing accessed cache lines, impacting application performance.

- We introduce a CHA and core topology-aware thread mapping heuristic and evaluate its performance and overhead. Our heuristic yields performance improvements of up to 5.6%, 8%, 25%, 6% and %1 for merge-based SpMV, Barnes, Fluidanimate, LU and Jacobi-2D stencil applications, respectively.

- We prove that the performance improvement can in fact be attributed to the reduced cache coherence traffic by using performance counters to observe the traffic across the mesh before and after the optimization and compare them for all data, invalidation, acknowledge and address traffic flows.

To our knowledge, the closest work to ours is [Kommrusch et al., 2021], which leverages a distributed directory topology to enhance application performance. However, their technique is restricted to SpMV only, and it differs from ours as it necessitates the copying of data to enable each core to access data managed by a nearby CHA. In contrast, in our work, we perform thread placement rather than data placement. Lastly, although our heuristic incurs overhead in mapping data addresses to CHAs, in the context of iterative solvers, this overhead can often be amortized over thousands of iterations. Regardless, our overhead is significantly lower than the overheads of the runtime and static approaches highlighted in [Kommrusch et al., 2021]. A notable drawback of their approach is its underperformance relative to the baseline when the hardware prefetcher is enabled, which is supposed to be enabled

by default. To our knowledge, there has not been any study on Fluidanimate, LU, Barnes or Jacobi-2d to improve their performance by reducing the cache coherence it incurs.

Chapter 2

BACKGROUND

2.1 Mesh-based On-chip Network in Intel Microarchitectures

This paper explores the Cascade Lake [Arafa et al., 2019] microarchitecture from Intel, which employs a mesh-based network-on-chip design. However, this work is also relevant for the Skylake [Tam et al., 2018] and Ice Lake microarchitectures due to their similarity with Cascade Lake. Figure 1.1 showcases an Intel Xeon Gold 6258R CPU and provides a visual representation of the mesh topology inherent in the microarchitecture.

The socket comprises 30 tiles organized in a grid-like structure. Each tile houses a core, a CHA, and a snoop filter except for one tile on left and right side is designated for the integration of memory controllers (IMCs), leaving up to 28 tiles containing cores. Nevertheless, certain versions feature fewer cores than the maximum capacity, as some of the tiles are disabled. The quantity of disabled tiles remains constant across both halves of the socket for all configurations, potentially attributable to the requirements of heat dissipation. Despite being disabled, these tiles can still facilitate transaction traffic. Commencing with the Knight's Landing microarchitecture (KNL) [Sodani, 2015], the last-level cache (LLC) underwent a significant change to become non-inclusive. As a result, data residing in private caches is no longer mandated to be stored in the LLC. Consequently, this change substantially reduces the amount of redundant data across the mesh. The non-inclusive nature of the caches imply that a snooping-based cache coherence protocol would be ineffective, given that the tags contained within the LLC do not correspond to the content of the private caches. To address this, the snoop filter was integrated into the microarchitecture.

Transactions on the mesh network are routed across the mesh respecting the YX

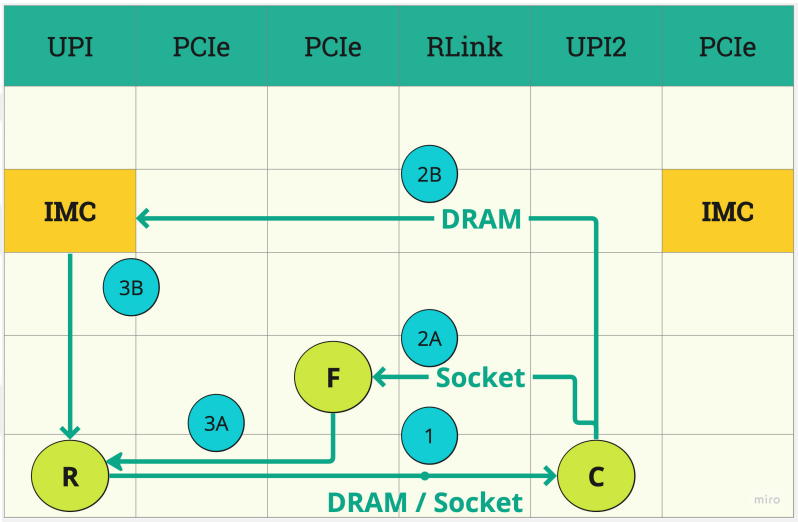

Figure 2.1: Possible scenarios of cache coherence traffic: (1) As the requester core R cannot find a cache line *cl* that it needs to read from/write to in its local cache, a query request is sent to CHA C that manages the coherence info of *cl*. (2A) Through its snoop filter, CHA C finds out that *cl* is present in the local cache of the core F in *F* state, and CHA C sends a request to the core F to forward *cl* to R. (3A) The core F forwards *cl* to R. (2B) In case the snoop filter in CHA C cannot find *cl* in all local caches and in the LLC of the socket, (assuming a single socket machine) CHA C sends a request to one of the IMCs to retrieve *cl* from DRAM. (3B) The IMC that handles the request retrieves *cl* and forwards it to the core R.

protocol [Tam et al., 2018]. Routing always favors vertical sections in the mesh. When a transaction arrives at the destination row, it will travel horizontally until it arrives at the target tile. This behavior serves to reduce latency as the horizontal hop cost is twice the cost of a vertical hop, taking 2 and 1 clock cycles, respectively.

The Skylake and Cascade Lake microarchitectures adopt the MESIF cache coherence protocol [Goodman and Hum, 2004] [Horro et al., 2019a]. As shown in Figure 2.1, this protocol departs from its predecessor, MESI, through the introduction of the "forward" state, designated as F. Unlike earlier microarchitectures, where multiple cache lines would serve a request when stored in the S state, resulting in redundant traffic and bandwidth consumption, MESIF ensures that only the cache line in the F state is forwarded to the requester core. The recipient cache will then store the line in the F state and is assigned the responsibility of responding to requests for that line.

2.2 Cache Coherence Traffic

CHA enforces the MESIF protocol by managing a portion of the distributed directory to ensure coherence of the addresses assigned to it. Each address is assigned to a single CHA using a proprietary pseudorandom hash function, which is not disclosed by Intel but has been exposed by Kommrusch et al [Kommrusch et al., 2021] for KNL processors and by McCalpin [McCalpin, 2021b] for Skylake and Cascade Lake processors.

When a core attempts to access an address that is not present in its local caches, the CHA responsible for managing the coherence of that address is queried. The CHA checks its own buffer queues to detect if the address is currently involved in any active transactions. Additionally, the address is checked against the LLC co-located with the CHA, and a snoop filter processes the address to determine its whereabouts. If a private cache in a tile currently holds the cache line in F state, the request is forwarded to it, and that tile sends the address to the requesting core.

If the system consists of multiple sockets, the CHA determines which socket owns the address at the moment. If the address is not present on the requesting socket,

the local memory controller responsible for the address fetches it from the local main memory within the NUMA node. Depending on some undocumented system activity parameters, the CHA may also send a snoop request to other sockets immediately or wait until it determines the snoop is necessary. Assuming there is available bandwidth on UPI, the snoop request may not need to wait. If the requested cache line is found in a dirty state in one of the other sockets, the socket will return the cache line after invalidating it in all cache levels. The CHA assigned to the requested address on that socket handles this process. If no data is found, the requesting socket is notified and the data will be fetched from memory.

Figure 2.1 illustrates possible scenarios of cache coherence traffic of CHAs, where requester, coherency manager and forwarder cores are different. Thus, even if the tile that stores the requested address might be a neighbor to the requester core, the overall path might not be optimal in terms of distance as the CHA managing the coherence of the requested variable might be located physically remote to the requester core.

Chapter 3

CHA & CORE TOPOLOGY

The allocation of CHAs across a chip die follows a logical, straightforward pattern. Each CHA is unambiguously mapped to a singular core tile, following a systematic numbering increment from the top to the bottom of the die, and subsequently from left to right, as depicted in Figure 1.1. However, this enumeration skips over any disabled tiles present on the die. Thus, to accurately determinate the layout of CHAs on the die, it is important to have the information about the count and exact positions of the disabled tiles.

The assignment of numbers to physical cores on a die is more irregular than the numbering of CHAs, as illustrated in Figure 1.1. Cores are not consistently numbered in the same manner as CHAs. Considering that both the count and locations of disabled tiles can be influenced by manufacturing process variations [Horro et al., 2019b, Horro et al., 2019a, Kommrusch et al., 2021], and the mapping between physical cores and CHAs may vary among different processor models and system manufacturers [McCalpin, 2021b], it is essential to gather all this information for CHA-aware performance optimization on a specific multicore machine.

While determining the CHA and core topology is a one-time task for a specific machine, it remains necessary because not all vendors disclose the mapping of distributed directory units to their corresponding cores located on the machine. Software methods that streamline the process of discovering this topology would prove invaluable for programmers aiming to explore new machines prior to running their computational workloads.

UPI	PCIe	PCIe	RLink	UPI2	PCIe
A	E	J	O	T	Y
IMC	F	K	P	U	IMC
B	G	L	Q	V	Z
C	H	M	R	W	@
D	I	N	S	X	#

CAPID6 Register

				#	@	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Figure 3.1: Bits in a CAPID6 register can be interpreted to expose enabled tiles.

3.1 Unveiling CHA Topology

The location of CHAs can be determined by accessing the CAPID6 PCI configuration space register [Corporation, 2017]. Each socket within Xeon scalable processors has a unique CAPID6 register, specifically employed to reveal enabled tiles. This register is composed of 32 bits, where the 28 least significant bits represent tile information and the remaining 4 most significant bits remain unused. A single bit corresponds to a particular tile, when set to 1, signifies that the respective tile is active otherwise it is disabled. For example, the least significant bit within the register signifies whether the tile positioned at the top-left corner is disabled or not. Mapping of tiles to 28 bits is shown in Figure 3.1.

3.2 Unveiling Core Topology

Finding out the core topology is not as straightforward as CHA topology. There are two methods that we are aware of to display this information, both of which utilize performance counters programmed to monitor different events. The first method, introduced in [McCalpin, 2021c], monitors data traffic from memory while triggering data flow across the mesh. For brevity, we refer to this method as the *traffic monitoring method*. The second method, introduced in [Wan et al., 2021],

analyzes power consumption of the cores while keeping the core under test busy. We refer to this method as the *power monitoring method*. In the KNL microarchitecture, one can also execute the CPUID instruction with certain parameters to learn about the core topology [Kommrusch et al., 2021]. However, this method is not applicable to SKX/CLK microarchitecture to our knowledge.

3.2.1 Traffic Monitoring Method

This method leverages uncore performance counters to quantify the data traffic traversing the mesh, triggered by accessing a large block of data from memory. We programmed performance counters to monitor HORZ_RING_BL_IN_USE and VERT_RING_BL_IN_USE events. To monitor, we crafted a synthetic benchmark that first binds the thread to a specific core, then allocates a large memory array that surpasses the total cache size within a single socket. Four uncore performance monitoring counters are programmed in each tile, tasked with monitoring the data traffic as it traverses the tile in all four directions.

Following this, all elements within the array are retrieved. Fetching of the array elements will trigger a data flow across the mesh and this will be captured by the uncore counters. Upon completion of the data fetching, we read the values from all the uncore counters for comparative analysis with their initial readings. We expect that the tiles exposing the most pronounced disparity between the initial and final counter readings likely fall along the path stretching from the core -to which the thread is bound- to one of the integrated memory controllers (IMCs).

3.2.2 Power Monitoring Method

This method measures the power consumption across all core tiles to identify the relationship between a core and its colocated CHA. The setup echoes that of the previous method. Initially, we capture the values of the uncore performance monitoring counters to monitor Power Management Agent (PMA) events from all the core tiles. Following this, we bind the thread to a core, the colocated CHA of which we aim to identify. The thread then increases a variable from the main memory

an artificially high number of times. This process ensures the core remains active, leading to elevated power consumption. Consequently, the core tile experiences an increased number of PMA events.

Ultimately, we again record the values of the PMA counters to contrast with their initial readings. We expect the tile that exhibits the most substantial difference in the PMA counter value to house the core to which the thread is bound. Therefore, its CHA is presumed to be colocated with the core.

3.2.3 Comparison of Methods

Both methods produce the same result, although the traffic monitoring method requires more effort due to the necessity to know the topology of CHAs and IMCs on the die in order to discern the map of the captured traffic. Furthermore, there is a confusing design decision by Intel that causes the meaning of "left" and "right" of the tiles to change in alternating columns in the mesh. This is an artifact of alternating columns facing the opposite direction. As a result, one has to take this unorthodox design into consideration before drawing conclusions from the traffic monitoring method. The latter method is easier to interpret and requires much fewer code lines. Finally, one might get inaccurate results if the iteration count in both methods is not high enough. In Section 5.2, we evaluate both methods quantitatively in terms of accuracy and overheads.

3.3 Discovering Memory Address to CHA Mapping

We explored two methods to determine the relationship between a memory address and its associated CHA. The first, described in [Maurice et al., 2015], utilizes uncore performance monitoring counters. The second method, outlined in [McCalpin, 2021b], leverages the pseudo-random hash function of Intel Xeon Scalable machines. We refer to the first method as the *LLC lookup method* and the second as the *hash function method*.

3.3.1 LLC Lookup Method

We devised a benchmark that generates a significant number of cache misses and measures the number of LLC lookup events occurring throughout the mesh. Initially, the LLC lookup values of all cores are recorded. The thread is then bound to a core, and a previously allocated variable is fetched from memory sufficiently high number of times. After each fetch operation, we write to and flush the fetched address, ensuring that it must be retrieved from the main memory again in the subsequent iteration. In order to fetch an address that is not present in local caches in an appropriate state, i.e. *modified (M)*, *exclusive (E)*, *shared (S)*, or *forward (F)* state, the CHA responsible for managing the coherence of the address is queried. This query emits an LLC lookup event on the corresponding CHA [McCalpin, 2021a]. Upon completion of the fetch/flush benchmark, we record the LLC lookup values again to determine which tiles have exhibited the most significant increase in emitted LLC lookup values. The CHA that indicates this change is presumed to be the one that manages the fetched and flushed address.

3.3.2 Hash Function Method

The hash function method relies on a hash function that maps physical addresses to CHAs. The formulation of the hash function is disclosed in [Kommrusch et al., 2021] for Intel KNL processors and in [McCalpin, 2021a] for Intel Xeon Scalable processors. In this method, initially, a variable is allocated, and its VA is converted into its PA. To convert the VA to its physical counterpart, the pagemap of the process is read. Pagemap is accessed from the `/proc/self/pagemap` file, and the page number of the VA is found by dividing the VA by the current page size of the system. The pagemap file is then read at an offset (pagemap entry size $\times$ page number). It should be noted that the pagemap entry is 64 bits in size, with the first 55 bits exposing the corresponding physical frame number. The frame number is then multiplied by the page size, and the resulting value is shifted with the modulo of the VA with the page size as an offset to obtain the corresponding PA.

Once the PA is identified, the base sequence of CHA IDs specific to the target

microarchitecture and a set of selector masks are employed to generate the CHA ID mapped to the PA. A binary permutation number is generated for the PA using the selector masks. The PA's index is extracted from the address, with index bits located in bits 6 to $(X + 5)$, where X is the logarithm base 2 of the base sequence length. Consequently, the index number is within bits 6 to 17. XOR'ing the binary permutation number and the index number produces another number. The CHA ID associated with the PA is the element in the base sequence of CHA IDs indexed by this result. The work in [McCalpin, 2021a] provides the base sequences for all available core counts in the Skylake microarchitecture, and the base sequences for Cascade Lake are identical to their Skylake counterparts.

3.3.3 Comparison of Methods

Comparing both methods, it is evident that the hash function approach is more convenient, providing accurate results consistently. However, it is important to note its limitation due to Linux's lazy mapping strategy: it necessitates a read from or write to the address before revealing its physical address. Should the page for the address remain unallocated, the page frame number for the virtual address will register as zero, indicating it has not yet been mapped. Contrastingly, the LLC lookup method may not consistently yield accurate results, especially in busy systems. If another process triggers more cache misses than the process under observation, it can lead to discrepancies in the results. Additionally, the LLC lookup method introduces a considerably higher overhead, contributing to its potential limitations.

3.4 CHA-Aware Communication Benchmark

In order to evaluate the performance implication of the distance between communicating cores and CHA, we devised a synthetic benchmark in which two threads simultaneously write to a variable atomically. Figure 3.2 illustrates this situation where two communicating threads are mapped next to the CHA that handles the cache line, while it also illustrates another situation where the CHA that manages

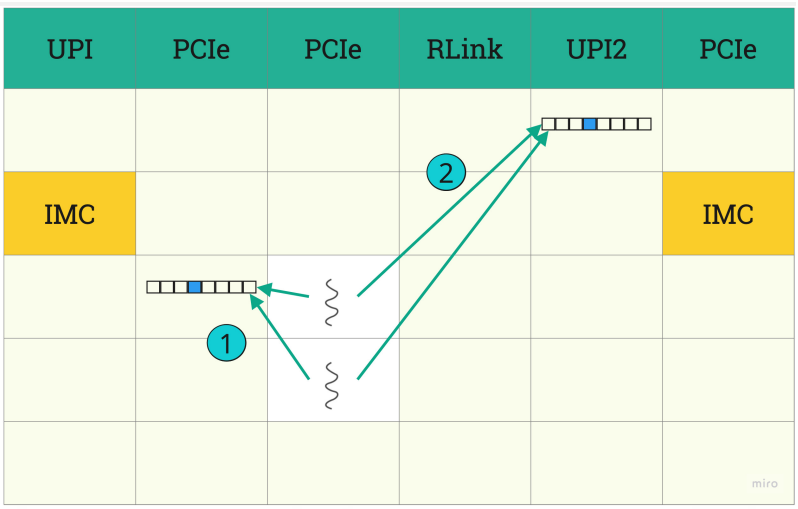

Figure 3.2: Exchanged data is managed by (1) a nearby CHA, (2) a far CHA

Algorithm 1 CHA-Aware Ping-Pong Benchmark

```
1: procedure PingPongBench(CoreID C1, CoreID C2, CHAID CH1)
2:     addr A= GetCacheLineManagedByCHA (CH1)
3:     #pragma omp parallel num_threads(2)
4:     Map thread 0 to C1 and thread 1 to C2
5:     for counter = 0 to N do
6:         Atomically increment the value in address A
7:     end for
8: end procedure
```

the cache line is located more remotely from the communicating cores.

The pseudocode of this benchmark is presented in Algorithm 1. Since all of the cache lines in the memory block are already associated with their assigned CHAs, a cache line whose offset address is managed by a given CHA CH_1 can be picked from the memory block using the function call in Line 2. To evaluate the impact of the distance between the CHA managing the cache line coherence and the two communicating threads, we map the two threads on two fixed neighboring physical cores in Line 4. Since both threads update the same cache line in Line 6, we expect a ping-pong exchange of the cache line between the two threads. We repeat this benchmark while varying the CHA that handles the coherence of the cache line.

Figure 3.3 plots the results of this experiment, where the y-axis shows the distance between the pair of communicating cores and the CHA that manages the

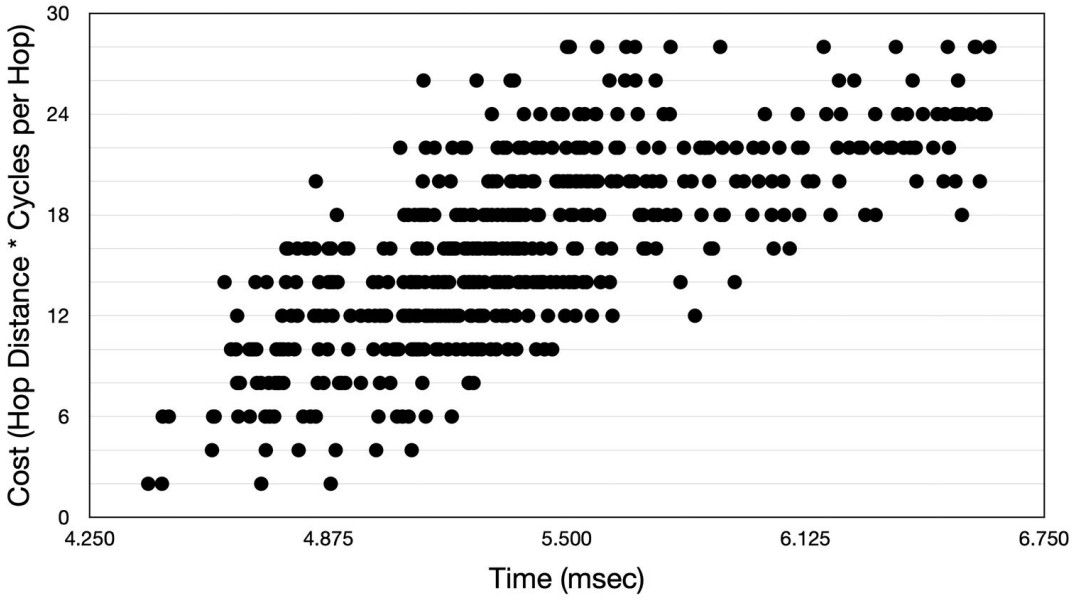

Figure 3.3: Execution time of two threads performing ping-pong communication across different hop distances.

coherence of the cache line. The figure shows that there is a positive correlation between the execution time and the distance.

Chapter 4

THREAD TO CORE MAPPING

This section introduces the thread-to-core binding algorithm using CHA and core topology. To achieve notable performance improvement through CHA-aware placement of threads to cores, the application should feature shared data modified by multiple threads. Thus, our first step involves identifying shared data across threads, with a specific focus on those involving at least one writer thread to trigger MESIF protocol.

The binding algorithm follows a systematic approach. We construct data structures to facilitate communication between thread pairs, starting with those exhibiting the highest inter-thread communication. For each pair, we identify the most frequently used CHAs, as they govern MESIF traffic. Threads are then bound to cores around the centroid of the most accessed CHAs if those cores are not already assigned to other threads. This process continues until all threads are successfully bound to cores.

Algorithm 2 shows the pseudocode of our thread mapping algorithm. As shown in Line 2, our algorithm, firstly, tracks the CHA mapping of the physical address of shared objects and records the CHA numberings in a separate array, i.e. the `CHAMap`. Then it runs the main computation in one iteration in the `AnalyzeTraffic` to generate the communication matrix of the computation (Line 3). Different from the communication matrix in previous works [Diener et al., 2015,Diener et al., 2016, Sasongko et al., 2019], our algorithm records not only the communication volume between each pair of threads but also the access count to each CHA by each pair of threads in the generated communication matrix. The captured communication volumes of all pairs of threads are then ranked in descending order. The algorithm, then, greedily picks pair of threads iteratively in a loop starting from the threads that have the highest communication volume (Line 6), and maps them to CPU

Algorithm 2 Thread To Core Assignment Algorithm

```
1: procedure MapThreadstoCores(Objects shared_objs)
2:     CHAMap = GetCHAMap(shared_objs)
3:     totalCommCountsPerPair = AnalyzeTraffic(CHAMap)
4:     mappedThreadCount = 0
5:     while mappedThreadCount < ThreadCount do
6:         threadA, threadB = popMostCommunicatingPair(totalCommCountsPerPair)
7:         cha = getCentroidOfMostAccessedCHAs(threadA, threadB)
8:         if threadA is not bound yet then
9:             core1 = getClosestAvailableCoretoCHA(cha)
10:            assign threadA to core1
11:            mappedThreadCount++
12:        end if
13:        if threadB is not bound yet then
14:            core2 = getClosestAvailableCoretoCHA(cha)
15:            assign threadB to core2
16:            mappedThreadCount++
17:        end if
18:    end while
19: end procedure
```

cores on a single socket that are the closest to the centroid point of the CHAs most frequently accessed by the pair of threads (Line 10 and Line 15). This iterative process of mapping generates a complete one-to-one mapping of threads to CPU cores in the end of the iteration. Finally, each thread binds itself to a CPU core by following the resulting mapping and executes its partition of the computation. We only use cores in a single socket in order not to account for inter-socket traffic which could tamper with the results.

It is crucial to note that comprehensive understanding of the target application is not necessary for our approach. Our primary emphasis revolves around identifying shared data writes and optimizing them accordingly. The size of the read-only data also has effect on the performance speedup since that data will occupy some space on core caches, however the primary focus in our algorithm is the shared writes. This guiding principle forms the foundation of our thread-to-core binding algorithm, allowing us to effectively enhance multithreaded applications.

4.1 Reducing Overhead of CHA Discovery

In the thread assignment algorithm, we used the hash function method in order to discover CHA mappings. Even though this method is faster by a large margin than the other proposed method, it is not a panacea. As there can be a huge number of addresses whose CHAs need to be queried in large data sets, the hash function method can become quite expensive and might even dwarf the overhead introduced by the mapping algorithm. To address this issue, we implemented a standalone preprocessing application that allocates a large block of memory in DRAM as a POSIX shared memory object that can be used for memory allocation by other applications. This large memory block consists of two parts; a data buffer for storing data and a CHA buffer that records the CHAs that handle the addresses of all cache lines in the data buffer.

The profiled application allocates memory in the shared memory object by mapping its virtual address space to the shared memory object using Linux `mmap` system call. Consequently, the profiled application can access the data and CHA buffers in the shared memory block. The application can then use the preallocated data buffer instead of allocating data by itself. In addition to this, it can also use the CHA buffer to retrieve info on the CHAs of the addresses in data buffer without the need for calculating by itself. We were able to accomplish this simply by leveraging the inherent nature of CHA computation, which is grounded in the utilization of physical addresses.

When allocating N cache lines in the data buffer, we also allocate space for the CHA buffer to store N CHA values. The CHA for the i^{th} cache line in the data buffer will map to the i^{th} index in the buffer. It is noteworthy to mention that it is sufficient to map the data buffer's addresses to CHAs in cache line granularity as the shared memory object is cache line-aligned and all addresses residing in the same cache line are assigned to the same CHA. It is also important to ensure that the allocated buffers are sufficiently large so that all input data for the benchmark can take advantage of this optimization. Currently, this precomputation is designed for use by a single benchmark application at a time, eliminating the need for synchro-

nization when accessing the addresses or tracking the available index ranges within the data and CHA arrays.

4.2 Application Use-Cases

We selected *merge-based SpMV, Barnes, Fluidanimate* and LU as the use case irregular applications to evaluate our thread mapping heuristic. These irregular applications have a distinct access frequency to all CHAs, with a pronounced inter-thread communication. We also studied on Jacobi-2D with regards to cache coherence traffic optimization. The inherent regularity of Jacobi-2D leads to uniform communication patterns across the mesh, presenting limited opportunities for our optimization strategies to make an impact.

4.2.1 Merge-based SpMV

Sparse matrix-vector multiplication (SpMV) is a fundamental operation in many scientific and engineering applications. SpMV involves multiplying a sparse matrix by a dense vector, resulting in another dense vector. In this paper, we propose a thread mapping heuristic algorithm for merge-based SpMV computation [Merrill and Garland, 2016]. Our algorithm leverages SpMV computation of the form $x = Ax$. Compared to the form $y = Ax$, the form $x = Ax$ incurs more inter-core cache line transfers since the x vector is subject to both read and write accesses, and therefore, triggers higher number of inter-core cache line invalidations and transfers.

In Algorithm 2, the object list passed to the algorithm includes the x vector, nonzeros array, column indices array and row offsets array of the CSR format of the input matrix. The `AnalyzeTraffic` function runs the SpMV computation for one iteration to generate its communication matrix. After binding all threads to cores, each thread executes its part of the merge-based SpMV computation.

4.2.2 Barnes

Barnes from SPLASH-2 [Woo et al., 1995] is an application that simulates interactions in a system of bodies, e.g. particles, over multiple time steps using the

Barnes-Hut hierarchical N-body method. In each time step, a number of traversals occur on an octree data structure that contains information on the bodies in the system, such that one traversal is performed for each body.

Using the communication analyzer in [Sasongko et al., 2019], we identified that the data structures involved most frequently in inter-thread communications are the *ctab* field of the *struct local_memory* data structure, which is an array of cells for the octree, and the *struct CellLockType* data structure, which contains an array of locks that ensure exclusive accesses of the cells. For this reason, we include the two data structures in the object list passed to Algorithm 2, and use the generated thread-to-core mapping in running the application.

4.2.3 Fluidanimate

Fluidanimate from the PARSEC benchmark suite [Bienia et al., 2008] simulates the intricate behavior of fluids, such as air or liquid in a medium and calculates the movements of individual particles while taking into account the complex interactions that occur between these discrete particles. Fluidanimate represents particles using dense matrix formats and comes with different input files tailored to various particle counts, which are read as binary files into the application. Using this data, cells representing the fluid are created within a grid. For each frame, the simulation first clears particles from the previous grid, rebuilds the grid and its cells, calculates densities and forces across the grid, handles collisions, and finally renders the cells in the current frame. Throughout these steps, multiple threads regularly update the array of cells, resulting in on-chip traffic. Therefore, in the thread mapping algorithm, this array of cells serve as the shared object input.

4.2.4 LU Factorization

The LU kernel from the SPLASH-2 benchmark suite [Woo et al., 1995] decomposes a dense matrix into a combination of a lower triangular matrix and an upper triangular matrix. In order to utilize temporal locality, this n by n matrix is broken down into N by N sized blocks. Each block is processed by the processor that owns it. For

enhancing spatial locality, elements within each block are placed consecutively in the memory. Computing the triangular matrices require multiple accesses to the indices on the original matrix, therefore this matrix is passed as the shared object input in our algorithm.

4.2.5 Jacobi-2D Stencil

The basic idea behind the Jacobi 2D stencil is to update the value of each grid point based on the average of the current point and its four neighboring points. The algorithm iterates through the grid, updating the value of each point until convergence is reached. Grid is partitioned in a way that all threads share a equally sized area of the grid. Each thread does the computation for the partition assigned to them independently. Dependency is only relevant for north and south boundaries for each partition where two threads need to read values from the bounds to do the computation.

The aim is to reduce the coherence traffic, reducing the overall latency. Aligned with this purpose, CHAs of the addresses in each partition are recorded. Following that, threads are assigned to partitions greedily. Partition that has the CHA with the maximum frequency is first assigned to the thread that is co-located with that CHA. The assigned CHA and cores are then ignored for the succeeding assignments. The remaining CHAs and cores are taken into account for the next steps with the exact same algorithm till all of the CHAs and cores are paired.

Chapter 5

EVALUATION

This section evaluates the different methods to expose CHA-to-core mapping and address-to-CHA mapping. In addition, we present the performance of the distributed directory-aware thread mapping algorithm. Our evaluation system is a 2-socket Intel Xeon 6258R Cascade Lake CPU. There are 28 cores per socket without simultaneous multi-threading being activated. Each core has its own local 32 KB L1i, 32 KB L1d, and 1MB L2 caches, while all cores in a socket share a common 38.5 MB L3 cache. Even though this is a 2-socket server, we only used cores on the same socket in order not to get QPI interfere with our on-chip optimization. We use Linux 5.10.121 and GNU-10.3.0 toolchain.

5.1 CHA Topology Experiment

To map the topology of CHAs on the chip dies of the Cascade Lake machine, we read the content of the CAPID6 register on each die. Since there are two sockets in the machine, there are two dies in it. The value of each CAPID6 register in both dies is `0x0FFFFFFF`, which means that none of the 28 core tiles on each die is disabled. Using the value of each register, we map the topology of the CHAs as illustrated in Figure 1.1. The positions of the IMC tiles in the topology are in the leftmost and rightmost columns of the second row as described in [Tam et al., 2018].

5.2 Core Topology Experiment

We evaluate the traffic monitoring and power monitoring methods to discover core-to-CHA mapping by developing two synthetic benchmarks that implement these methods. These benchmarks run an outer loop in a single thread such that, in each iteration of the loop, the thread is bound to a different core in the machine

Table 5.1: The comparison of accuracy and overheads between the traffic and power monitoring methods.

Method	**Iteration Count**	**Accuracy**	**Runtime (msec)**	**Memory (Kbytes)**
Traffic Monitoring	100 M	100%	212.2	783452.8
Power Monitoring	100 M	92.9%	182.8	2201.6
Power Monitoring	200 M	92.9%	358	2200.8
Power Monitoring	400 M	100%	693.2	2189.6

and executes one of the mapping discovery methods to identify the CHA that is co-located with the core.

For the benchmark that implements the traffic monitoring method, the size of the large array allocated in the main memory is 100 MBytes. We chose this size as it is larger than the size of the L3 cache in the machine. There is a nested loop in this benchmark. While the outer loop tests the mapping of the main thread to each core, the inner loop accesses the elements of the large array iteratively in order to trigger data traffic from an IMC to the current core. The second benchmark that implements the power monitoring method also has a nested loop with the same outer loop. In the inner loop, the benchmark keeps the current core busy by incrementing a variable to increase the power consumption of that core. Table 5.1 compares the accuracy, time, and memory overheads of both benchmarks on the Cascade Lake machine.

From the experiment, we discover that the traffic monitoring method is accurate in identifying the co-located CHAs of all cores. From Table 5.1, we can also observe that the traffic monitoring method incurs much higher memory overhead than the power monitoring method. The reason for this huge memory overhead is the accesses to the large array that are needed to trigger data traffic.

In Table 5.1, we can also observe that the accuracy of the power monitoring method depends on the iteration count of its inner loop. Furthermore, while the runtime of the benchmark becomes longer as the iteration count increases, the mem-

Table 5.2: The comparison of accuracy and overheads between the LLC lookup and hash function methods.

Method	**Iteration Count**	**Accuracy**	**Runtime (msec)**	**Memory (Kbytes)**
LLC lookup	250	54.6%	2268	3652
LLC lookup	500	83.1%	2539	3652
LLC lookup	750	97.4%	2777	3652
LLC lookup	1000	99.9%	2952	3652
LLC lookup	1250	100%	3186	3652
Hash function	N/A	100%	12	3652

ory consumption stays relatively constant since this method involves only repeated accesses to a single variable.

5.3 Address to CHA Mapping

We evaluate the LLC lookup and hash function methods to discover address-to-CHA mapping by devising two synthetic benchmarks that implement these methods. While the hash function method is computationally more complex as it requires translation from virtual to physical address and transforming the resulting physical address through a set of mathematical operations, the LLC lookup method has more latency as it requires a block of memory to be fetched and flushed repeatedly in a loop for the purpose of counting LLC lookup events.

As shown in Table 5.2, the hash function method is faster and accurate. Accuracy comes from the fact that its output is deterministic as it relies on a non-probabilistic mathematical operation. The LLC lookup method lacks accuracy as it involves a loop to count an uncore event, however the number of iterations in the loop plays a key role in strengthening the accuracy of the method. If the number of iterations is lower than 1000 on an idle system, then the output starts diverging from the output we get from the hash function as depicted in Table 5.2, which we consider as the ground truth due to its absolute accuracy. If we increase the iteration count, the

accuracy increases but the method suffers from increased overhead. Furthermore, the hash function method also does not need root privileges whereas the LLC lookup method needs them in order to program the uncore counters and read them.

The LLC lookup method is more portable as it does not depend on any base sequence that is specific to certain microarchitecture and certain core count. The base sequence is needed for the hash function to be computed. Besides that, the hash function method is not future-proof as the emerging microarchitectures will most probably have newer base sequences. The LLC lookup method is portable as long as Intel keeps maintaining LLC lookup events and continues providing an interface to monitor those events.

5.4 Distributed Directory-Aware Thread Mapping

To evaluate the performance improvement and overhead of our thread mapping heuristic, we implemented and tested it on merge-based SpMV, Barnes, fluidanimate, and LU. In all evaluations, the baselines use compact mapping of threads to physical cores on a single socket.

5.4.1 Merge-based SpMV

We evaluated the performance of our algorithm on merge-based SpMV by running it on 100 matrices with varying numbers of nonzero values and row counts. The nonzero counts of these matrices range from 100K to 28M. The results indicates that our algorithm outperforms the baseline by up to 5.6% and with an average of 0.67% when the non-zero count of the input matrix is less than approximately 5M as presented by the scatter plot in Figure 5.1. This performance improvement is higher than the improvement achieved by ChoiceMap [Soomro et al., 2018], which is up to 1.26% and with an average of 0.17%. Notably, the performance improvement achieved by our heuristic is observed given that the memory footprint of these matrices is comparable to the size of the last-level cache in a single socket.

However, as the non-zero count surpasses this range, the algorithm's memory footprint exceeds the cache capacity, resulting in cache evictions and increased data

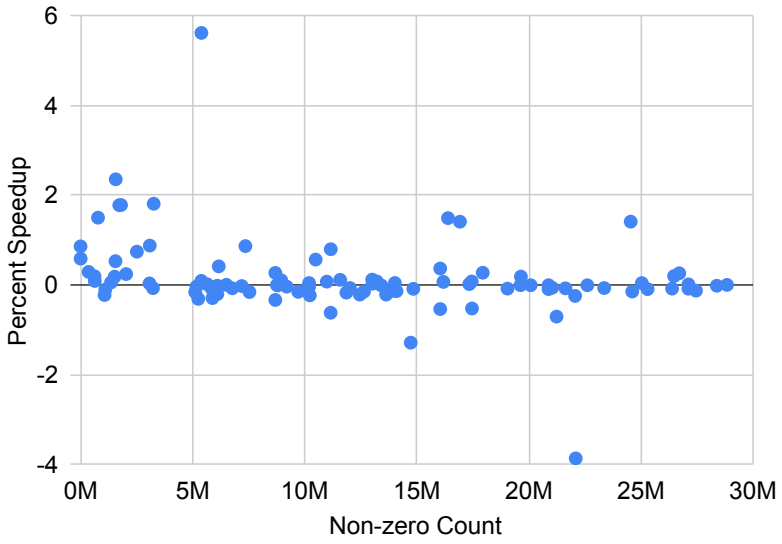

Figure 5.1: Correlation between the performance improvement in merge-based SpMV over compact mapping and the nonzero count of each matrix

fetches from DRAM. Consequently, the algorithm's effectiveness is hindered due to the latency introduced by DRAM accesses.

When taking the overhead into account, the performance gain of the algorithm becomes apparent only after certain number of iterations. The overhead is equivalent to around 8100 iterations of SpMV computation on each matrix. The primary contributor to the overhead is the thread-to-core mapping algorithm, which traverses the virtual grid to track the addresses accessed by threads and make informed decisions regarding the mapping. The process of finding out CHAs mapped to the accessed addresses by using the POSIX shared memory also contributes to the overhead, although its impact is minimal compared to that of the thread mapping algorithm as it typically accounts for around 1% of the entire preprocessing overhead.

Figure 5.2 displays cache coherence traffic reduction for merge based SpMV. The traffic that we measure here is data/cache line transfer and invalidation traffic. This result confirms that our heuristic manages to reduce coherence traffic during application runs. There seems to be correlation between traffic reduction values and the speedups. However, the fluctuation across input sizes in the graph does not correlate with speedup because we only take on-chip traffic into account without

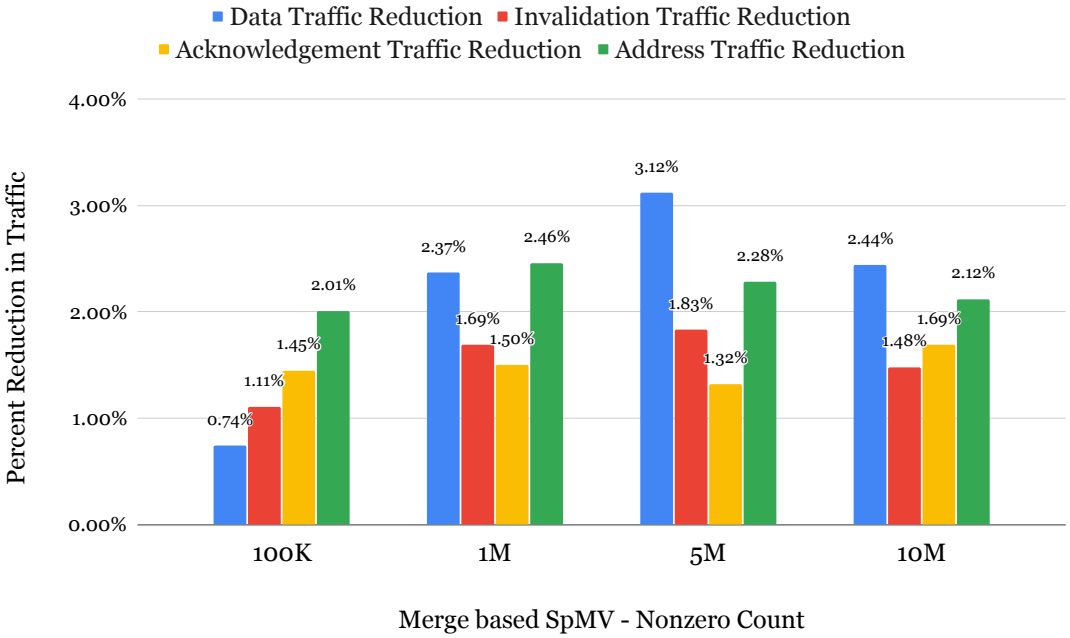

Figure 5.2: Cache coherence traffic reduction in Merge based SpMV with thread mapping as a function of different nonzero counts

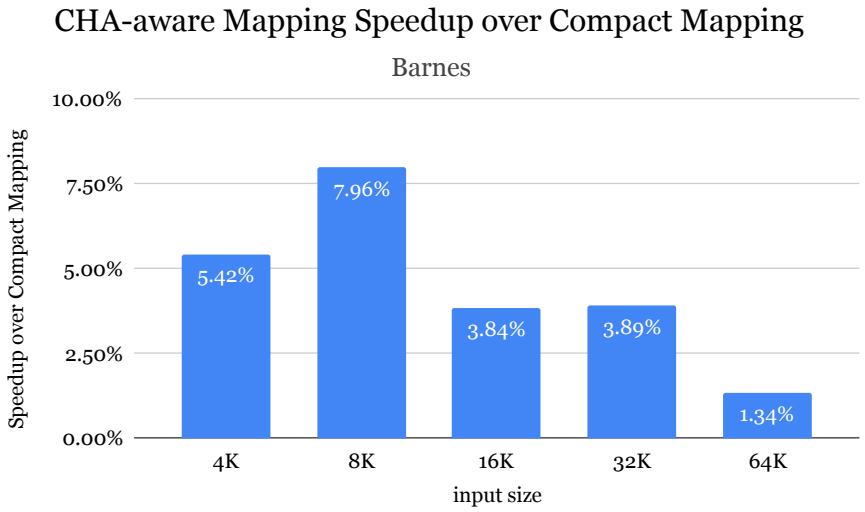

Figure 5.3: Performance improvement in Barnes with thread mapping as a function of different input sizes (*nbody*)

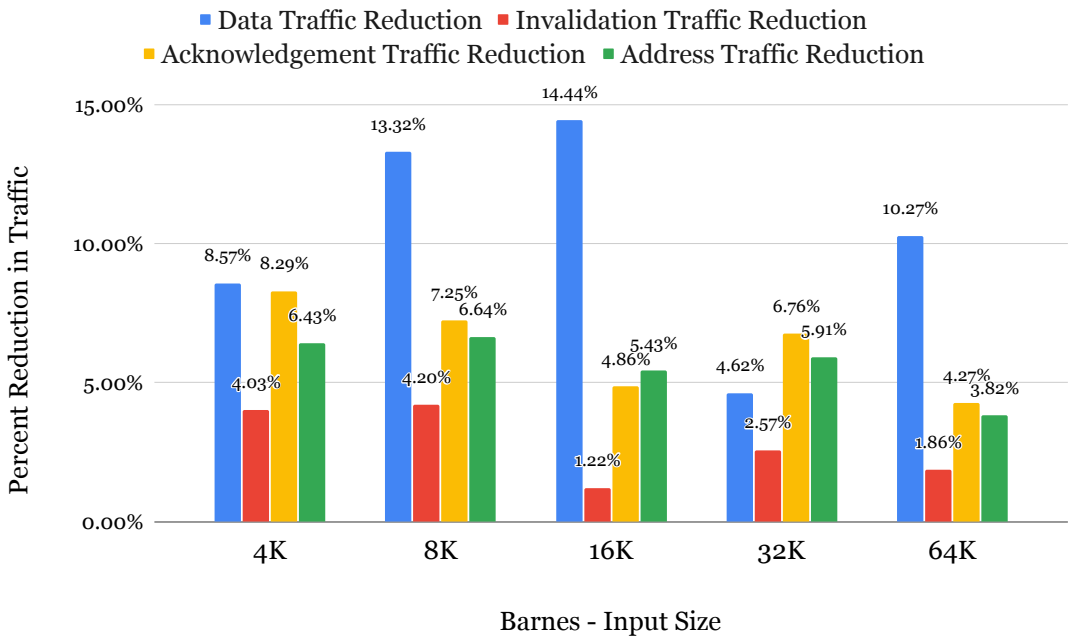

Figure 5.4: Cache coherence traffic reduction in Barnes with thread mapping as a function of different input sizes (*nbody*)

considering accesses to DRAM.

5.4.2 Barnes

We ran Barnes under different input sizes, i.e. the *nbody* parameter. As shown in Figure 5.3, our thread mapping optimization achieves a speedup of up to ∼8% and with an average of 4.4% on Barnes over all of the parameter values. We observe lower performance gains as *nbody* increases, which is expected, as *nbody* defines the size of the octree data structure. Larger *nbody* values make the size of the entire data structure too large to fit in the L3 cache, and therefore, require the application to access data from DRAM more frequently.

Figure 5.4 displays cache coherence traffic reduction for merge based SpMV for invalidation and data traffic. Traffic reduction gets progressively larger till 5M nonzero count. After this point, we observe a decline in traffic reduction. This can be explained by the data not fitting into the socket cache space, and therefore needing to fetch data from DRAM, disrupting the effectiveness of the algorithm.

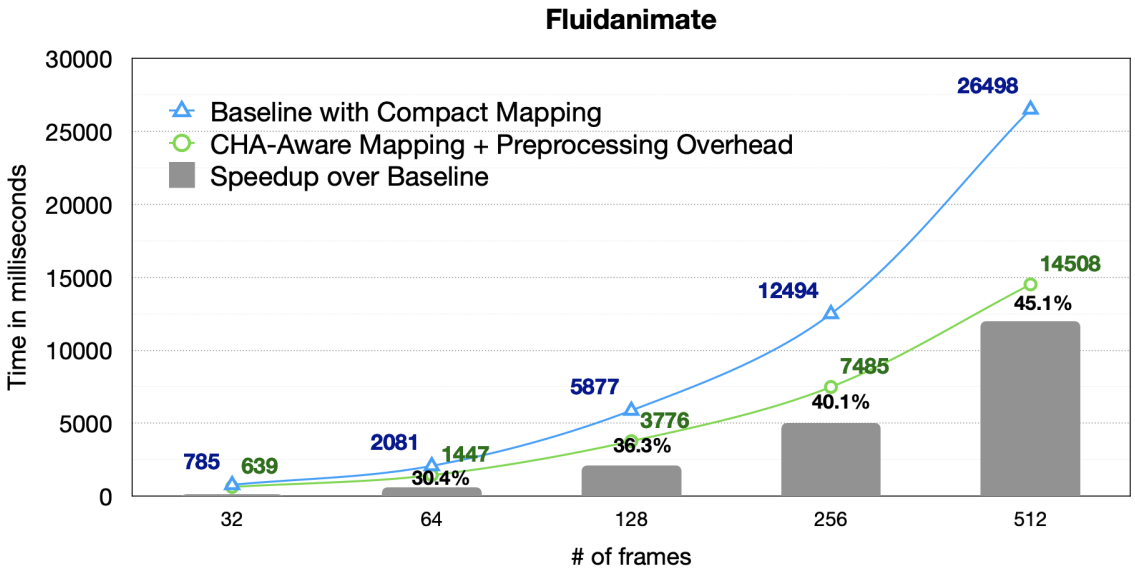

Figure 5.5: Comparing the elapsed time in milliseconds for the baseline using compact mapping versus CHA-aware thread mapping (including the pre-processing overhead) on a 500K particle count using 16 threads for frame counts of 32, 64, 128, 256 and 512.

Overhead imposed by the preprocessing step is compensated when the input size is around 32K. From this point onwards, we are able to get net performance gains despite the overhead introduced by the preprocessing.

5.4.3 Fluidanimate

We evaluated the performance of the thread pinning algorithm across various parameters, including particle counts at several gradations (5K, 15K, 35K, 100K, 300K, and 500K), thread numbers (4, 8, and 16), and frame numbers (32-512). The results clearly indicate an improvement in performance proportional to the increases in particle counts and frame numbers.

This improvement can be attributed to the fact that higher particle counts provide a richer dataset for the algorithm to process, offering greater opportunities for refinement and efficiency gains. Similarly, an increased number of frames corresponds to a rise in the iteration count of the core algorithm. This increase significantly outweighs the initial preprocessing time, especially for high frame number

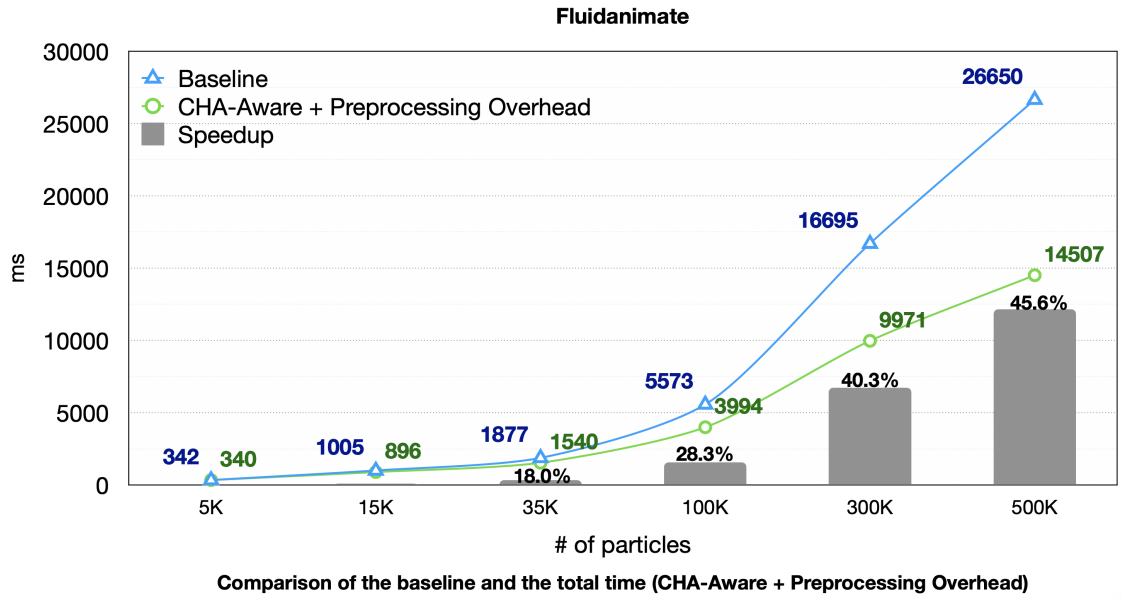

Figure 5.6: Comparing the elapsed time in milliseconds for the baseline using compact mapping versus CHA-aware thread mapping (including the pre-processing overhead) using 16 threads and 512 frame numbers for particle counts of 5K, 15K, 35K, 100K, 300K and 500K.

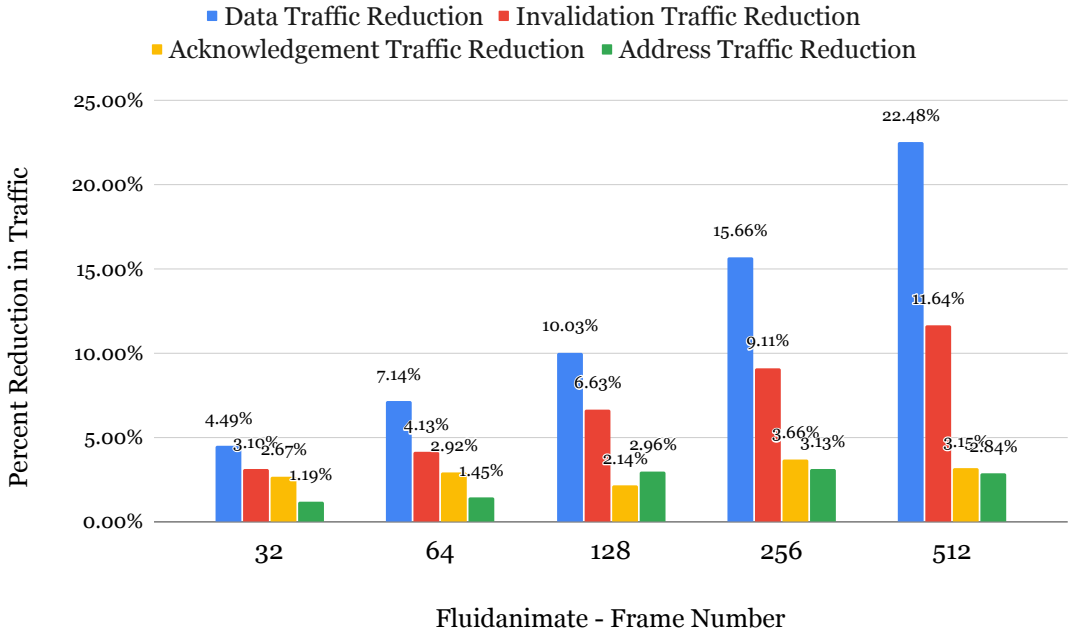

Figure 5.7: Cache coherence traffic reduction in fluidanimate with thread mapping as a function of different frame numbers

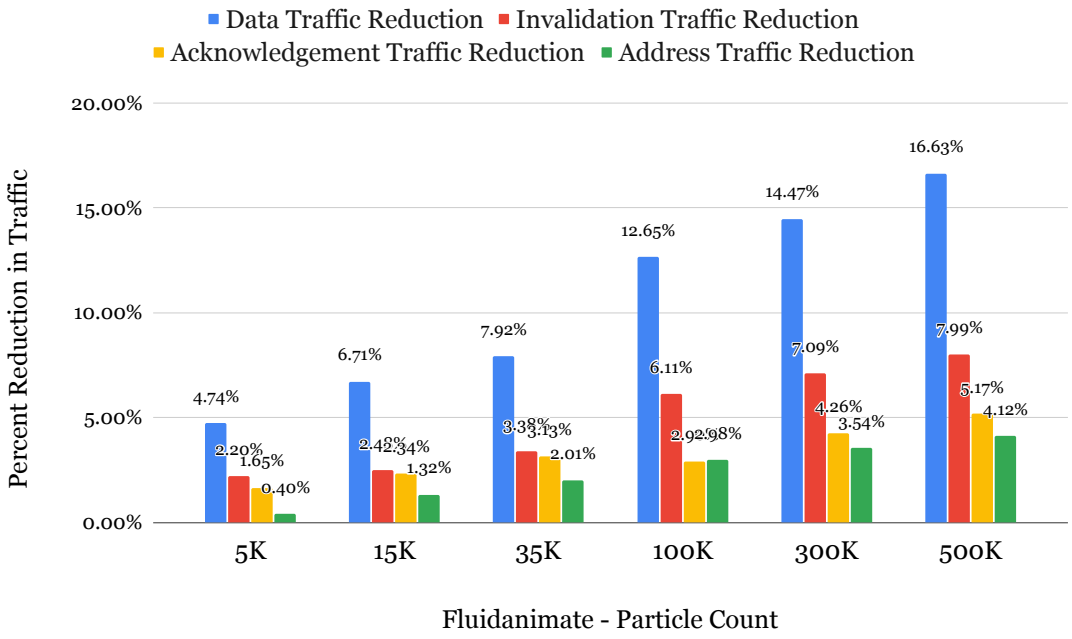

Figure 5.8: Cache coherence traffic reduction in fluidanimate with thread mapping as a function of different particle counts

runs, as evident from Figure 5.5. In addition to this, Figure 5.6 also demonstrate that with the increasing particle counts, our algorithm perform better with respect to the baseline. Consequently, it becomes apparent that when the initial setup or preprocessing is mitigated by the gains from optimization, the algorithm demonstrates a significant performance uptick relative to the baseline. We also tried only varying thread counts (2, 4, 8, 16) and keeping frame number and particle count constant. Results for these runs ended up being similar for all cases.

Figure 5.7 and Figure 5.8 both display cache coherence traffic reduction for fluidanimate application. As the particle count increases, we observe almost a linear pattern since we did not yet exhaust the available cache space with the memory allocated by the studied particle counts. Linearity is less pronounced as the frame number increases. There will be more thread communication between the threads with the increasing frame numbers. Since we optimized for this communication, we tend to see more reduction for larger frame numbers.

As it can be observed from Figure 5.5, in order to simulate a 500K particle

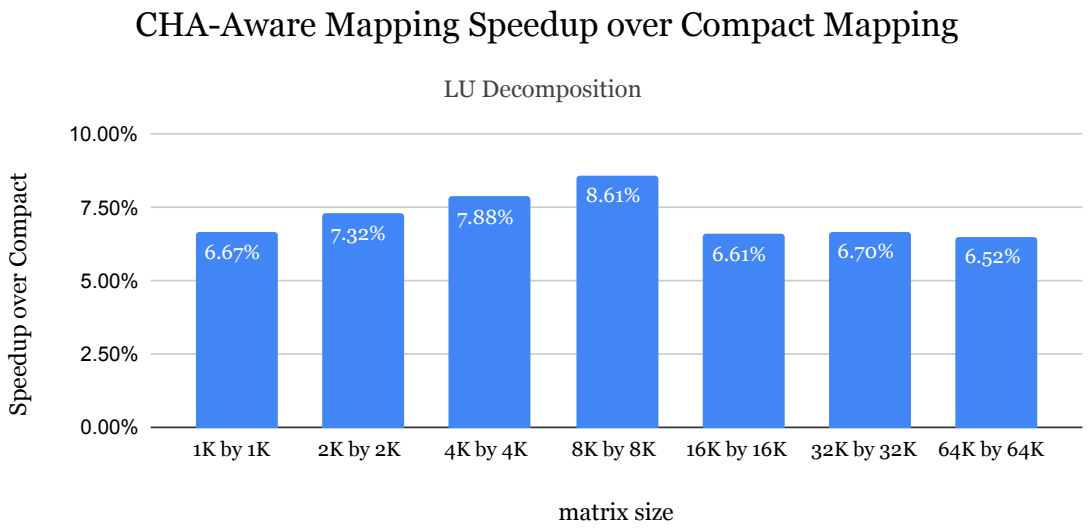

Figure 5.9: Comparing performance speedup on LU decomposition for matrices of various sizes.

fluid, we compensate for the overhead starting from 64 frame number. Figure 5.6 demonstrates that when the particle count exceeds 35K, overhead is compensated if the frame number is 512 with 16 threads.

5.4.4 LU

We executed the optimized algorithm on matrices of various sizes (from 1K by 1K to 64K by 64K). Across all these matrices, we observed an improvement in the LU operation latency, with larger matrices benefiting slightly more than the smaller ones. However, after a point for which the available cache capacity is not able to accomodate the working set, speedup is hindered as evident in Figure 5.9. The results took into account the preprocessing overhead incurred by our algorithm. The performance speedup is around 6% for the 1K by 1K matrix. This base algorithm aims to reduce inter-thread communication by employing a 2-D scatter decomposition to allocate matrix blocks to processors and having each processor handle the multiplication of the blocks it owns. We noted a slight enhancement in performance improvement when using larger block sizes, primarily because they result in more cache misses. Our algorithm is designed to optimize for these cache misses, con-

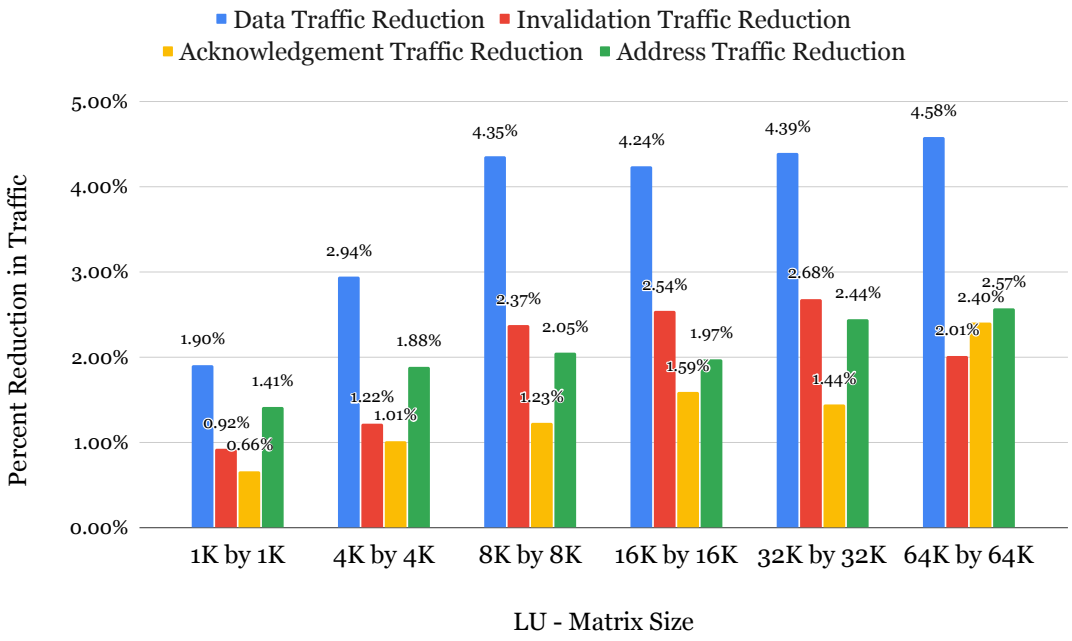

Figure 5.10: Cache coherence traffic reduction in LU with thread mapping as a function of different matrix sizes

tributing to the observed improvements. Figure 5.10 demonstrates traffic reduction values for varying sized matrices. Our algorithm proves to incur less traffic overall across all sizes. Overhead is completely compensated and we start seeing net performance gains starting from matrix size of 32K by 32K.

5.4.5 Jacobi-2D Stencil

We were able to improve the performance of Jacobi-2D stencil by approximately 1% for all matrix sizes we executed, ignoring the overhead. This underwhelming performance is expected as this application is inherently regular. As CHAs of neighboring addresses are pseudorandom and each thread modify addresses at contiguous locations, CHA frequencies of the addresses modified by each thread for each partition will be similar. This is not an ideal case for our algorithm to function properly as the algorithm shines when there are irregularities between CHA frequencies for each work unit assigned to threads. Thus, performance improvement does not justify the preprocessing overhead incurred by finding out CHAs of the addresses and the

greedy assignment algorithm. Incurred data and invalidation traffic compared to the baseline are similar and only differ by around %0.5, optimized version being the lower. Overhead is mitigated when the matrix size is more than around 50K by 50K, if we consider iteration needed to converge as 100. It is harder to compensate for the overhead for this application as the optimization did not prove to be as effective due to regularity.

5.5 Compensating Overhead

While our developed algorithm has demonstrated enhanced performance, it does introduce a preprocessing overhead comprising two main components: CHA finding and the thread-to-core mapping algorithm. To ensure the practical utility of the algorithm, mitigating this overhead is essential.

In the context of merge-based SpMV, the overhead averages around 8100 iterations. Notably, the Barnes application begins offsetting this overhead when the input size exceeds approximately 32K particles. In the case of the Fluidanimate application, compensation for the overhead occurs when simulating a particle system consisting of 500K particles at a frame rate of around 64 frames per second. An alternative perspective reveals that to achieve a net gain over the baseline while simulating an object with 512 frames per second on 16 threads, the particle count needs to surpass approximately 35K. In the LU application, net gains relative to the baseline are observed when the matrix size reaches 32K by 32K. These insights highlight the impact of the algorithm's overhead on various applications and underscore the importance of tailoring it based on the specific computational requirements of each application in order to optimize performance.

Chapter 6

RELATED WORK

6.1 Reduction of Cache Coherence Traffic

The runtime optimization approach presented in [Kommrusch et al., 2021] is tailored towards KNL microarchitecture and optimizes SpMV by using the raw representation of the matrix and parallelizing the serial code with OpenMP and SIMD intrinsics. It makes use of MCDRAM and requires the processor chips to work on quadrant mode. The approach ensures that each core computes data with coherence information resident on its quadrant only, so it works in quadrant granularity as opposed to tile granularity which we are aiming for.

The optimization approach proposed in [Horro et al., 2019a] transforms the data layout of an application in order to set up the CHAs managing the coherence of the data accessed by each CPU core to be very close to the core. However, this approach has substantial overhead. This method involves a memory-intensive stage that involves copying the data to some targeted memory locations. Following this stage, arrays need to be recomputed since the locations of the data have just changed.

Caheny et al. [Caheny et al., 2018] characterized cache coherence traffic in cc-NUMA machines and developed NUMA-aware work scheduling and data allocation that reduce the coherence traffic. Another work by Scolari et al. [Scolari et al., 2016] leverages the hash functions that map memory blocks to LLC slices in Intel Sandy Bridge microarchitectures to implement page coloring that partitions the shared LLC to achieve performance isolation. Horro et al. [Horro et al., 2019b] studied the coherence traffic in Intel KNL machines by modeling these structures on a simulator. By characterizing the behavior of the coherence traffic, they proposed an Intel Mesh-focused locality optimization that reduces the coherence traffic. Two works by Xiao et al [Xiao et al.,] and Liu et al [Liu et al., 2015] propose compiler tech-

niques that cluster computational and memory operations, and map these clusters to physical cores in a way that reduces on-chip communication traffic.

6.2 Communication-Aware Thread Mapping

Jeannot et al [Jeannot et al., 2014] proposed *TreeMatch*, a thread mapping algorithm that greedily generates a set of process/task pairs from a graph of incompatibilities. The graph covers all possible task pairs, with each vertex representing a task pair and edges connecting vertices with common tasks, such as (1, 3) and (1, 5). Utilizing incompatibility information from the edges, *TreeMatch* extracts a final set of task pairs without redundant tasks. The algorithm selects pairs with the fewest external communication counts with other tasks in the application. Finally, *TreeMatch* maps the task pairs to the computing units of the underlying machine, leveraging the machine's topology information.

EagerMap, a thread mapping algorithm proposed by Cruz et al. [Cruz et al., 2015], greedily pairs a given task with the task that it communicates the most. The algorithm also performs the same pairing process iteratively on task pairs/groups in higher levels of granularity to decide the placement of the paired task groups on the sockets and nodes of the machine, while each pair of tasks at the lowest level of granularity are placed on the same physical core sharing the same local caches. Another thread mapping algorithm, *ChoiceMap* [Soomro et al., 2018], pairs the tasks in an application across different levels of hierarchy in a multicore machine by matching each task/task group with another task/task group that it communicates the most by considering the priority lists of both sides.

Different from these previous works, our thread mapping does not consider only the communication volume among application threads, but also the topology of cores and CHAs on the computer chip and the mapping of communicated data addresses to the CHAs on the chip.

Chapter 7

LIMITATIONS AND FUTURE WORK

Our research was conducted on a 2nd generation Intel Xeon Scalable servers with Cascade Lake and Skylake microarchitectures. However, the methodology presented in this paper is applicable seamlessly to the latest Intel Xeon Scalable machines, given their use of the MESIF protocol and mesh-interconnected distributed directory-based cache coherence.

It's important to note that our CHA-aware approach may not be compatible with AMD machines, which employ bus snooping for cache coherence, differing from our focus on distributed directory-based cache coherence. Nevertheless, we anticipate that our CHA-aware framework can be extended to recent ARM CPUs using mesh interconnects and distributed system cache, such as the Arm Neoverse cores on NVIDIA Grace.

Chapter 8

CONCLUSION

In this work, we compared different methods that uncover the topology and investigated the impact of distance between communicating cores and the CHAs involved in the communication on application performance. Leveraging the knowledge on the uncovered topology, we developed a thread mapping heuristic that reduces cache coherence traffic, resulting in better overall latency.

We studied on 5 applications in detail. For merge-based SpMV, we present the algorithm of the mapping heuristic and evaluated its performance and overhead on a wide range of sparse matrices. Our heuristic achieves a speedup of up to 5.6%. In the evaluation of our algorithm using the Fluidanimate, it became evident that its performance consistently surpassed the baseline for rendered frame count per second being higher than 30. This improvement can be attributed to the increased inter-thread communication inherent in rendering more frames. Notably, rendering larger particle counts helps us optimize better as there is more room for improving to reduce communication latency, as there is more data at hand that would deliver our algorithm more to get insights from. Shifting our focus to the LU benchmark, where we experimented with matrices of varying sizes up to 256x256, we observed a consistent performance gain of approximately 6.5% across the tested matrices. The application's lower percentage of shared written data led to a limited scalability with larger matrices. Consequently, the performance gain remained relatively constant regardless of the matrix size. Barnes application achieved up to 8% performance improvement with an average of 4.4% over all of the parameter values. Jacobi-2D was the only application for which we could not achieve impressive performance improvements for none of the parameters we tried. This can be attributed to the fact that the memory accesses in this application is regular and there is no major communication between threads, the data transfer between threads only occur at

halo boundaries.

Due to the increased use of distributed directory-based cache coherence in recent CPU chips and the extensive use of the applications we investigated in various domains, we believe that our heuristic and the findings of our investigation can benefit domain experts and programmers in tuning the performance of their applications. Our optimization strategy, focused on reducing inter-thread communication latency by reducing cache coherence overhead related to shared writable data, proved effective.

BIBLIOGRAPHY

[Arafa et al., 2019] Arafa, M., Fahim, B., Kottapalli, S., Kumar, A., Looi, L. P., Mandava, S., Rudoff, A., Steiner, I. M., Valentine, B., Vedaraman, G., and Vora, S. (2019). Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro*, 39(2):29–36.

[Bienia et al., 2008] Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th Int'l Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA. Association for Computing Machinery.

[Caheny et al., 2018] Caheny, P., Alvarez, L., Derradji, S., Valero, M., Moretó, M., and Casas, M. (2018). Reducing cache coherence traffic with a numa-aware runtime approach. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):1174–1187.

[Corporation, 2017] Corporation, I. (2017). *Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring*. Intel.

[Cruz et al., 2015] Cruz, E. H., Diener, M., Pilla, L. L., and Navaux, P. O. (2015). An efficient algorithm for communication-based task mapping. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 207–214.

[Diener et al., 2015] Diener, M., Cruz, E. H., Pilla, L. L., Dupros, F., and Navaux, P. O. (2015). Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation*, 88-89:18–36.

[Diener et al., 2016] Diener, M., Cruz, E. H. M., Alves, M. A. Z., and Navaux, P. O. A. (2016). Communication in shared memory: Concepts, definitions, and efficient detection. In *2016 24th Euromicro Int'l Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 151–158.

[Goodman and Hum, 2004] Goodman, J. R. and Hum, H. (2004). Mesif: A two-hop cache coherency protocol for point-to-point interconnects (2004).

[Horro et al., 2019a] Horro, M., Kandemir, M. T., Pouchet, L.-N., Rodríguez, G., and Touriño, J. (2019a). Effect of distributed directories in mesh interconnects. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6.

[Horro et al., 2019b] Horro, M., Rodríguez, G., and Touriño, J. (2019b). Simulating the network activity of modern manycores. *IEEE Access*, 7:81195–81210.

[Jeannot et al., 2014] Jeannot, E., Mercier, G., and Tessier, F. (2014). Process placement in multicore clusters:algorithmic issues and practical techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002.

[Kommrusch et al., 2021] Kommrusch, S., Horro, M., Pouchet, L.-N., Rodríguez, G., and Touriño, J. (2021). Optimizing coherence traffic in manycore processors using closed-form caching/home agent mappings. *IEEE Access*, 9:28930–28945.

[Liu et al., 2015] Liu, J., Kotra, J., Ding, W., and Kandemir, M. (2015). Network footprint reduction through data access and computation placement in noc-based manycores. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC'15. ACM.

[Maurice et al., 2015] Maurice, C., Scouarnec, N., Neumann, C., Heen, O., and Francillon, A. (2015). Reverse engineering intel last-level cache complex addressing using performance counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, page 48–65, Berlin, Heidelberg. Springer-Verlag.

[McCalpin, 2021a] McCalpin, J. (2021a). Mapping Addresses to L3/CHA Slices in Intel Processors. Technical report, The University of Texas at Austin.

[McCalpin, 2021b] McCalpin, J. (2021b). Mapping core and l3 slice numbering to die location in intel xeon scalable processors.

[McCalpin, 2021c] McCalpin, J. (2021c). Mapping Core, CHA, and Memory Controller Numbers to Die Locations in Intel Xeon Phi x200 ("Knights Landing", "KNL") Processors. Technical report, The University of Texas at Austin.

[Merrill and Garland, 2016] Merrill, D. and Garland, M. (2016). Merge-based parallel sparse matrix-vector multiplication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 678–689.

[Sasongko et al., 2019] Sasongko, M. A., Chabbi, M., Akhtar, P., and Unat, D. (2019). Comdetective: A lightweight communication detection tool for threads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19. ACM.

[Scolari et al., 2016] Scolari, A., Bartolini, D. B., and Santambrogio, M. D. (2016). A software cache partitioning system for hash-based caches. *ACM Trans. Archit. Code Optim.*, 13(4).

[Sodani, 2015] Sodani, A. (2015). Knights landing (knl): 2nd generation intel® xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24.

[Soomro et al., 2018] Soomro, P. N., Sasongko, M. A., and Unat, D. (2018). Bindme: A thread binding library with advanced mapping algorithms. *Concurrency and Computation: Practice and Experience*, 30(21):e4692. e4692 cpe.4692.

[Tam et al., 2018] Tam, S. M., Muljono, H., Huang, M., Iyer, S., Royneogi, K., Satti, N., Qureshi, R., Chen, W., Wang, T., Hsieh, H., Vora, S., and Wang, E. (2018).

Skylake-sp: A 14nm 28-core xeon® processor. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 34–36.

[Wan et al., 2021] Wan, J., Bi, Y., Zhou, Z., and Li, Z. (2021). Volcano: Stateless cache side-channel attack by exploiting mesh interconnect.

[Woo et al., 1995] Woo, S., Ohara, M., Torrie, E., Singh, J., and Gupta, A. (1995). The splash-2 programs: characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 24–36.

[Xiao et al.,] Xiao, Y., Xue, Y., Nazarian, S., and Bogdan, P. A load balancing inspired optimization framework for exascale multicore systems: A complex networks approach. In *Proceedings of the 36th Int'l Conference on Computer-Aided Design*, ICCAD '17, page 217–224. IEEE Press.