Exploring the Parallel Programming Design Space of Proximate, a Multi-Tile Programmable Accelerator

(CS 758 Project Report)

Vinay Gangadhar and Kai Zhao
University of Wisconsin-Madison
{vinay, kzhao32}@cs.wisc.edu

Abstract

The slowing of Moores law and Dennard scaling is limiting the performance improvements of single core processors. Increasing clock frequency any farther will lead to high leakage current and infeasible power consumption. Over the past decade, focus has been shifted to multi-core processors to increase throughput by having multiple cores to target different types of parallelism - instruction (ILP), data (DLP) and thread (TLP). However, even the multi-core processors are not scalable for parallelization beyond a point due to Amdahl's law. To address these challenges of rising dark silicon and the end of Dennard scaling, in recent years architects have turned to heterogeneous architectures with special purpose domain specific accelerators (DSAs), for higher performance and energy efficiency. While providing huge benefits, DSAs are prone to obsoletion due to domain volatility, have recurring design and verification costs, and have large area footprints when multiple DSAs are required in a single device to reap out the benefits of different application acceleration. To attack such problems of DSAs and multi-core processors, while still retaining programmability of general purpose processors and efficiency of DSAs, there is an on-going research in Vertical Research Group, University of Wisconsin-Madison aiming to build a general purpose multi-tile programmable accelerator called Proximate.

This project focuses on exploring the parallel programming design space of Proximate, a multi-tile programmable hardware accelerator. The aim is to investigate the programming interface of Proximate, the parallel hardware improvements and in general explore the parallel programs run on proximate. We have tried to explore a new type of the multi-tile programmable architecture, its scalability and speedup of such accelerator architecture compared to a traditional server class multi-core processors for parallel programs. In summary, we explored different programming design points in the project and proximate is able to achieve speedups of 50-100x over a traditional server class multi-core processor. We try to analyze this trend over the coarse of this document by explaining the architecture, programming model and the performance results of proximate.

1 Introduction

The end of classical device scaling means that the power per unit area on chip is rising with each technology generation. This implies that architectures for future technology nodes will not be able to power-on all components of the chip simultaneously, with some estimates being 50% “dark silicon” by 8nm [2], which is less than 10 years away. This trend, the utilization wall, will curtail expected performance improvements. Interestingly, much of the core’s energy is not expended in the functional units, but rather in the power-hungry structures needed for attaining reasonable performance on general purpose workloads. To exploit this, architects have in part turned to hardware specialization and accelerators, which sacrifice generality for efficiency in executing either specific computations or computations for certain application domains.

The slowing of Moores law and Dennard scaling has also led a trend towards domain specific accelerators (DSAs) for performance and energy consumption [4]. DSAs are high performance computation engines for a specific domain. In essence, DSAs trade obsoletion for performance and energy efficiency. DSAs employ common specialization principles for concurrency, computation, communication, data reuse, and coordination, but sacrifice programmability completely. [11] shows that a specialized architecture can be developed for general purpose programmability and be competitive against DSAs by trading only up to 3.8x in area and 4.1x in power.

This project tries to explore one such general purpose programmable accelerator architecture which tries to target different possible program regions. The idea behind a scalable, programmable and parallel execution hardware is that programmer can explicitly shard the data into multiple partitions and the computation can happen in parallel across these partitions which enable orders of magnitude of performance. Figure 1 shows the ideal programmer needed abstraction to map the computation kernels to the partitioned data without any conflicts or explicit synchronization primitives. So, our goal in the project is to explore this parallel programming design space and see what are some lessons we can learn from this exercise w.r.t programming model and the parallel hardware itself.

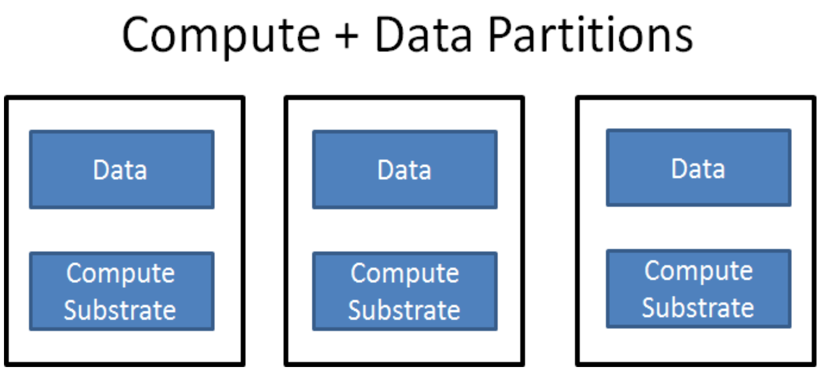


Figure 1: Data and Task Locality for Parallelization

Document Overview Section 2 describes the prior work on which the project is based on, and how it is resurfaced and remodeled for the project purposes. Section 3 motivates the need of architectures like proximate and describes the insights and mechanisms of parallelization on proximate. Section 4 explains the high level architecture and programming overview of proximate. Section 5 details over the parallel programming model of proximate and its API calls. Section 6 explains the methodology used for our project and Section 7 explains the results obtained and the corresponding analysis. We finally end with conclusion in Section 9 and future work needed.

2 Prior Work and Project Proposal

We build our project on the prior work done on LSSD [11] (Work done in Vertical Research Group), which includes identification of specialization principles and proposal of the high-level architecture of the high throughput compute engine (called Softbrain now). Softbrain builds upon the LSSD principles and implements the general micro-architectural mechanisms identified in prior work. Before explaining the high level organization of Proximate architecture, the five specialization principles employed to consider this style of accelerator architecture are explained. The primary insight on coming to the architectural substrate is based on well-understood mechanisms of specialization used in DSAs.

2.1 Specialization Principles

Broadly, these principles are seen in as a counterpart to the insights from Hameed et al. [4], in that they describe the sources of inefficiency in a general purpose processor, whereas our findings are oriented around elucidating the sources of potential efficiency gain from specialization.

Concurrency Specialization The concurrency of a workload is the degree to which its operations can be performed in parallel. This concurrency can be derived from data or thread level parallelism found in the workloads. Examples of specialization strategies include employing many independent processing elements with their own controllers, or using a wide vector model with a single controller. The former one is chosen as baseline architecture having many processing elements with a low-power controller.

Computation Specialization Computations are individual units of work in an algorithm executed by functional units (FUs). Specializing computation means creating problem-specific FUs. For instance, a sin FU would much more efficiently compute the sine function than iterative methods on a general purpose processor. Specializing computation improves performance and energy by reducing the total work. Most of the neural network applications employ some commonality in FU types.

Communication Specialization Communication is the means of transmission of transient values between the storage and functional units. Specialized communication is simply the instantiation of communication channels, and potentially buffering, between FUs to ultimately facilitate a faster operand throughput to the FUs. This reduces power by lessening access to intermediate storage, and potentially area if the alternative is a general communication network.

Data Reuse Specialization Data reuse is an algorithmic property where intermediate computed values are consumed multiple times. The specialization of data reuse means using custom storage structures or reuse buffers for these temporaries. Specializing reuse benefits performance and power by avoiding the more expensive access to a larger global memory or register files.

Coordination Specialization Hardware coordination is the management of multiple hardware units and their timing to perform work. Instruction sequencing, control flow, interrupts handling and address generation are all examples of coordination tasks. Specializing it usually involves the creation of small finite state machines to perform each task. A low-power in-order core or a micro-controller could be used for this coordination specialization. Based on the this principle, to aim at high concurrency workloads and provide the task/data locality to programmers, Proximate has lot of such small in-order cores acting both as coordination unit and computation engine to execute irregular workloads. This is based heavily on the principle of having many small in-order cores near memory to do the high-throughput computation [10].

Based on extensions to all the prior work explained above, we currently have built a vector dataflow programmable accelerator called Softbrain that is general purpose programmable using a high level hardware programming interface (HPI). Regular streaming workloads that are mainly single threaded showed comparable performance to the fixed function implementation of these workloads. The current research also aims at having a general purpose programmable tiled organization of in-order cores which run single threaded fine-grained parallel tasks, connected to the programmable accelerator (Softbrain) briefed above. Together this entire hardware organization targets both irregular and regular workloads, and we call this combined organization of hardware as Proximate. Proximate combines this multi-core tiled architecture with programmable/specialized accelerator (Softbrain) together into a single external PCIE device. Figure 2 shows high-level overview of the current Proximate offload engine.

Proximate has a 16 compute tiles, each compute tile with 8 (RISC-V) in-order cores, a programmable/specialized accelerator (softbrain fabric) for a total of 128 RISC-V inorder cores and 16 softbrain fabric instances

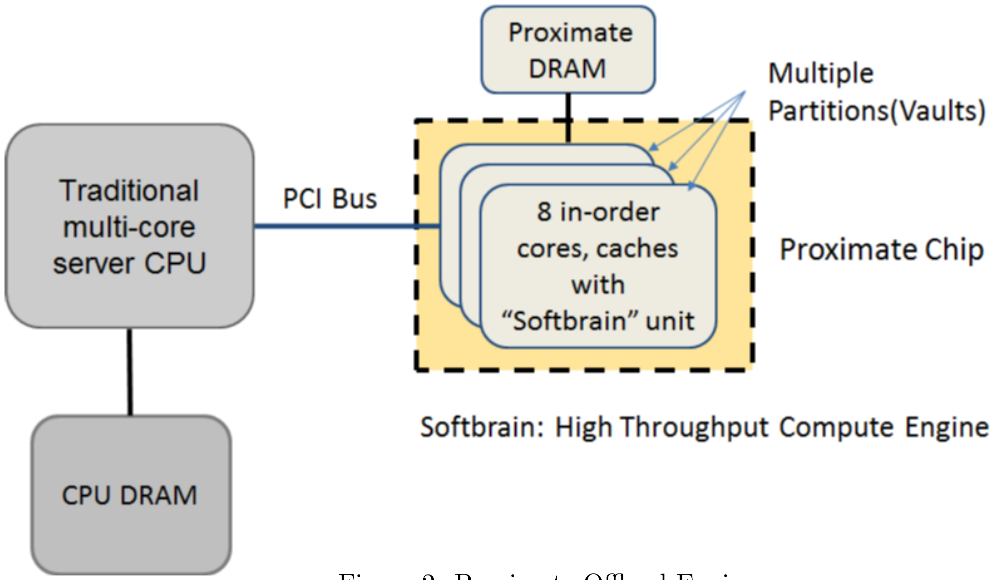

Proximate DRAM
Multiple Partitions(Vaults)
Traditional multi-core server CPU
PCI Bus
8 in-order cores, caches with "Softbrain" unit
Proximate Chip
Softbrain: High Throughput Compute Engine
CPU DRAM

Figure 2: Proximate Offload Engine

connected to a common memory hierarchy. It is programmable using a high-level offload API to offload the data, allocate memory on device and copy the data to/from the host processor.

2.2 Project Proposed

Since, we exactly cannot have the same hardware model setting as Figure 2, for our project, we have to slightly modify the API as well as the hardware model to suit our experiments. Figure 3 shows the organization of the entire device for our project purposes. We assume both the host core and the proximate device to be located in the same shared address space and proximate operates in a memory mapped region of the shared address space. The host core also runs the Proximate parallel programming API and the runtime is responsible for offloading the tasks at coarser-granularity to a device-specific task scheduler. This task scheduler is connected to the proximate tiles, with each tile having group of in-order cores and the high-throughput compute engine, Softbrain.

This project aims to parallelize some of the single-threaded regular HPC workloads and irregular workloads on Proximate architecture and evaluate performance and bandwidth limitations compared to a server class chip like 64-core Xeon Phi Knights landing. We start this by converting all the single-threaded workloads to the parallel versions of pthreads and OpenMP. Th idea behind this is to get the data from real hardware as well as see the scalability of these parallel workloads. We then implement the proximate versions of the parallel programs and evaluate the performance w.r.t to the best configuration of pthread run

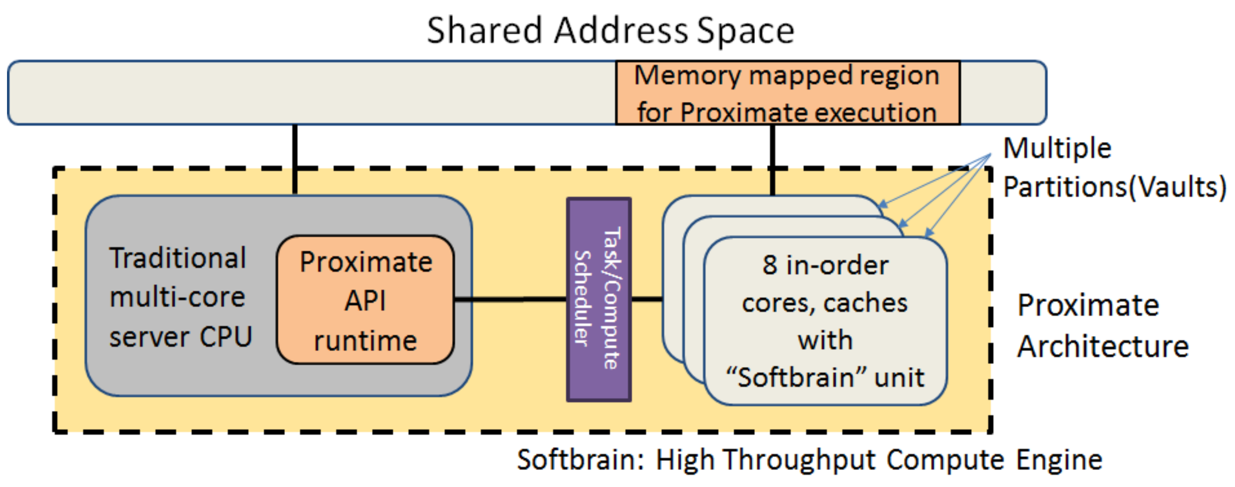

Shared Address Space
Memory mapped region for Proximate execution
Multiple Partitions(Vaults)
Traditional multi-core server CPU
Proximate API runtime
Task/Compute Scheduler
8 in-order cores, caches with "Softbrain" unit
Proximate Architecture
Softbrain: High Throughput Compute Engine

Figure 3: Proximate Architecture in Our Project Setting

on Xeon-Phi.

3 Motivation

As we mentioned earlier, general purpose cores are highly inefficient and are optimized for single thread workloads only. They have very low energy efficiency for server workloads mainly due to high-frequency, deep pipelines, speculation and explicit memory/register based communication [5]. Even the new heterogeneous offload engines like GPUs,FPGAs and ASICs face either programming or power-area challenges. So, this motivates us to invent new heterogeneous offload engines which - i) achieves better scalability and higher performance per watt, ii) have flexible programmer abstraction and iii) able to target different program behaviors in the core regions – both regular and irregular.

With such requirements, this section tries to motivate why this type of architecture and programming abstraction is needed for Proximate, and what are some of the insights and mechanisms needed to arrive at such architecture for current parallel workloads.

3.1 Insights

The main insight for parallelization in Proximate is that - it is reasonable to presume that programmer can reason about the data/task locality of the kernel. Programmer managed data sharding and task/data co-location is fairly straight forward for several server class workloads. By co-locating the data and task you would reduce the access latency and achieve higher memory or cache bandwidth.

The other insight we had was, each program or application can have multiple phases during the program

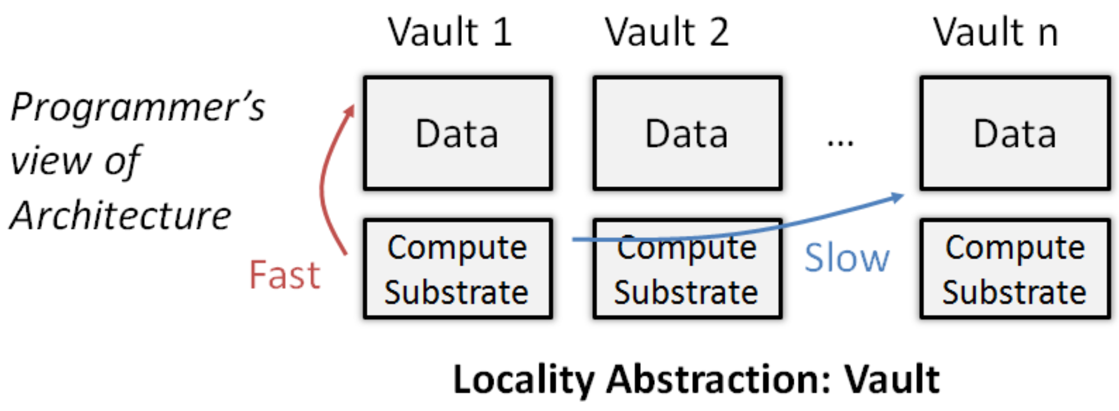


Figure 4: Programmer Abstraction of the Architecture

execution [12]. So, your parallel hardware must be able to recognize such phases and offload the kernel to corresponding hardware in a heterogeneous environment. Program regions generally tend to exhibit two main behaviors - i) Irregular data access pattern with low ILP and ii) Regular data access patterns with high DLP. So, the architecture executing such programs need to accelerate both these regions and the insight is to execute both these phases in proximate. The final insight is that, since regular workloads are memory streaming workloads, the architecture needs to support high bandwidth memory hierarchy and should be able to provide sufficient memory bytes in less number of cycles.

3.2 Mechanisms to Exploit Insights

We now explain some of the mechanisms we have come up with to exploit the insights described above.

Programmer Control Data/Task Locality The high level question one has to ask in how to give the programmers control of data and task locality within proximate cores. We answer this in Proximate by providing a flexible programming model to enable the programmer managed data sharding and task/data co-location. Figure 6 shows the programmer's view of the architecture, and the vault abstraction proximate provides. Each vault is group of cores and a predetermined slice of virtual address spaces, exposed during task offloading and memory allocation.

The locality abstraction for programmers is given by the vault. Each vault, has an address partition (data) and the compute substrate to work or offload task on that address partition. There is a latency penalty if each compute substrate does an out-of-vault access to access data not in its address partition. By giving this abstraction to programmer, he/she can manage the data sharding explicitly and reason about the data/task locality.

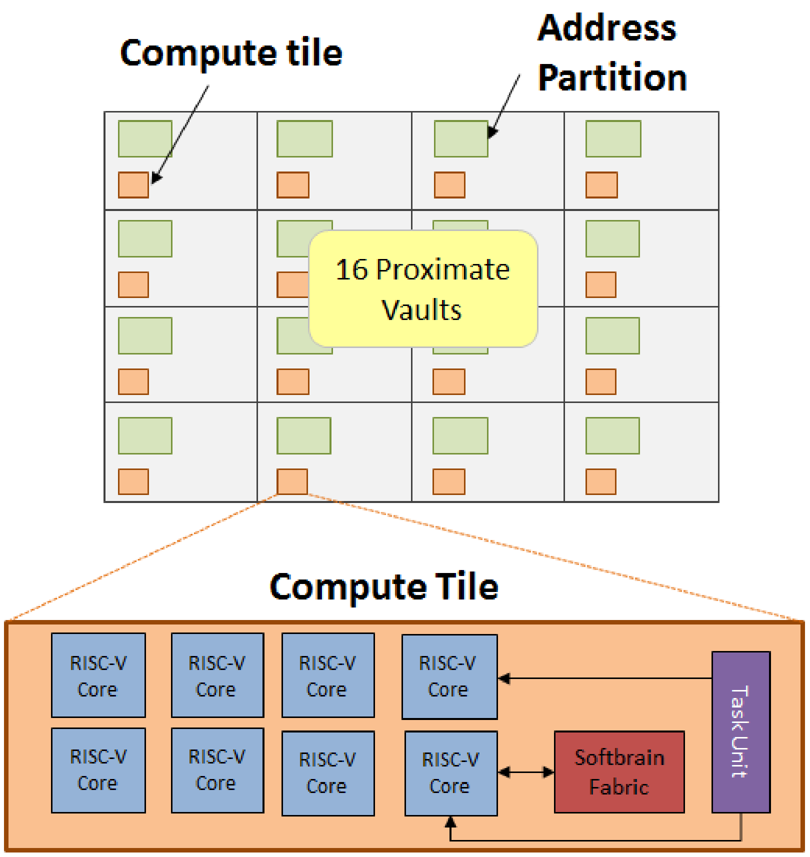


Figure 5: Proximate Architecture Overview

Program Regions and Memory Bandwidth Requirement To satisfy the high bandwidth demand and memory requirement, Proximate has address partitioned per-vault caches. Proximate has simple in-order cores to target program regions exhibiting irregular data access patterns with low ILP. It has a high throughput compute engine (Softbrain) to target program regions exhibiting regular data access patterns with high DLP.

4 Proximate Architecture and Programming Overview

This section first gives the architecture overview of proximate. Then it details over how would one program proximate. And finally delve into the micro-arch details of proximate.

4.1 Architecture Overview

Figure 5 shows the high level architecture organization of proximate. As mentioned in Section 3, there is a vault abstraction in proximate with a compute substrate and an address partition. Proximate has 16 vaults, and this decision is based on the bandwidth requirement of each vault and the interface to memory itself.

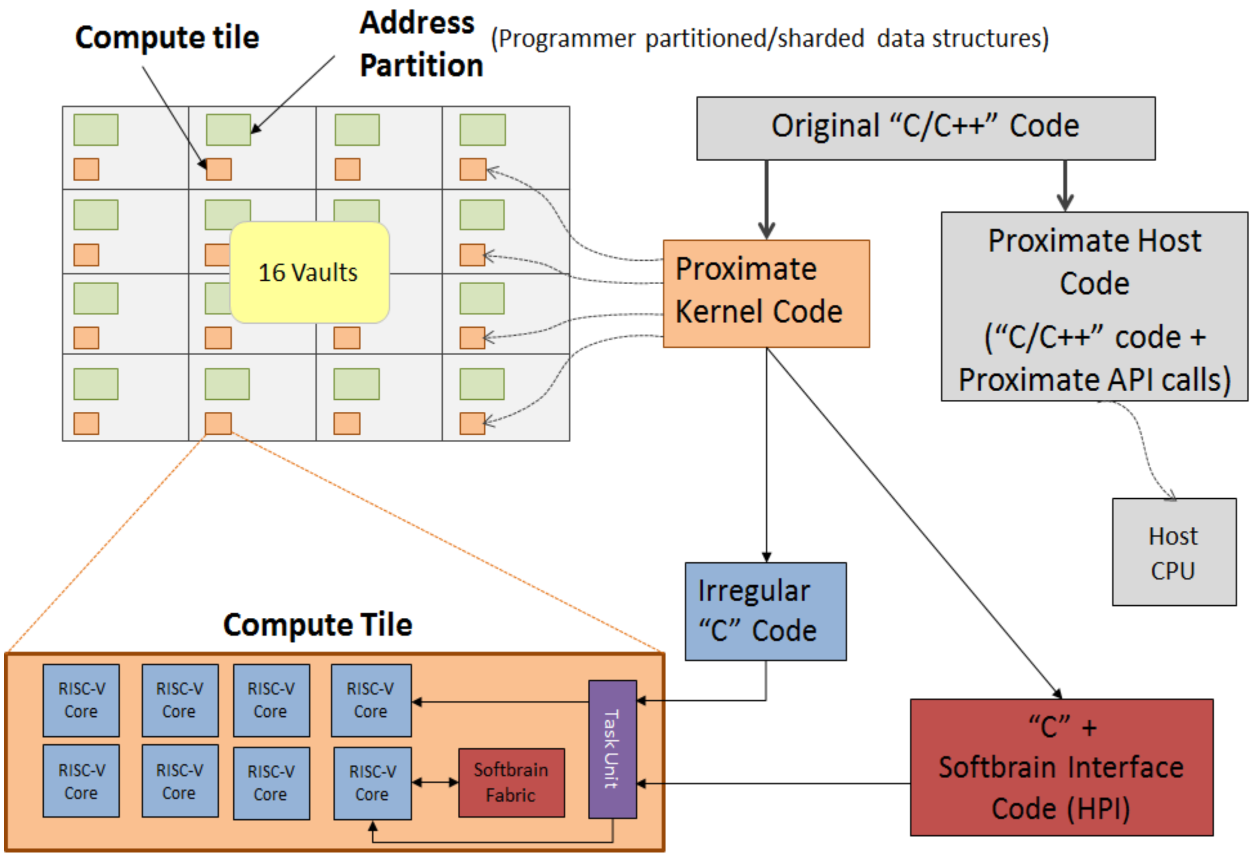


Figure 6: Proximate Programming Overview

The compute tile as shown in the detailed figures, further has 8 RISC-V [16] cores for irregular computation and one Softbrain unit for regular workload acceleration. There is also a per-vault task unit responsible for offload the kernels and tracking their status throughput the coarse of execution.

4.2 Programming Overview

We now explain how proximate would be programmed with the architecture overview described above. Figure 6 shows how the original C/C++ program gets mapped to the proximate hardware components. The original C/C++ code has 2 parts - i) The host code which has the proximate API calls (*explained in detail in* Section 5.) to set up the memory, copy the memory and en-queue the tasks, and ii) the second part is the actual computation kernel which gets offloaded to the compute tile (*shown in orange in figure*). The kernel portion of the program can have two parts again. If the programmer identifies the computation as irregular workload, then that can be mapped to the in-order for energy efficient high concurrent execution. Whereas, if the kernel is a regular streaming workload, them it can be mapped to the high-throughput engine *Softbrain*. The task unit inside each compute unit is responsible for doing this mapping.

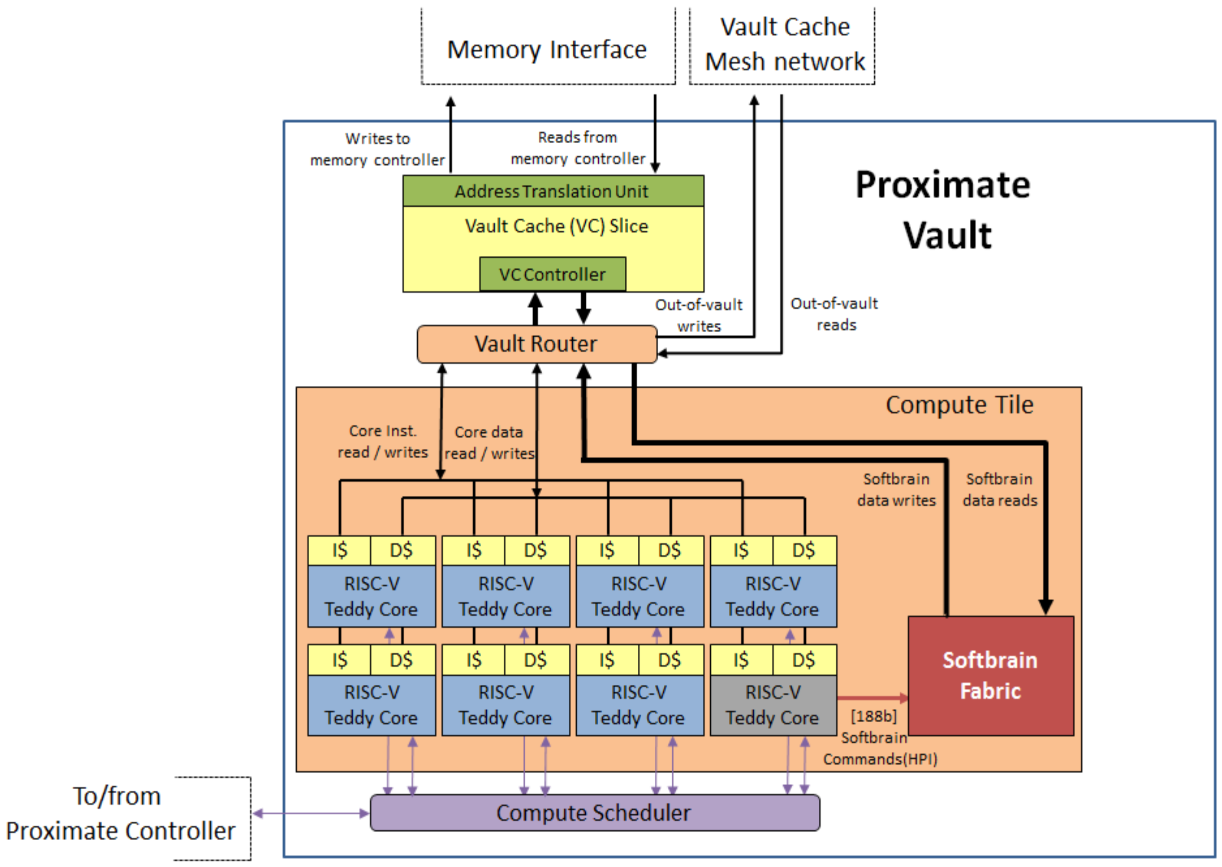


Figure 7: Proximate Vault Micro-Architecture

4.3 Proximate Detailed Micro-Architecture

Now, that we have understood both the architecture and programming overview of proximate, this section details over the micro-architecture elements of proximate.

Figure 7 shows the micro-architecture of each vault of proximate. The compute tile part is same as the one described above in the arch. overview. The extra element is the interface between one of the RISC-V cores and the Softbrain - *command interface*. Outside the compute tile is an address partitioned L2 cache or Vault Cache (VC) slice which is responsible for handling all the memory requests from cores and softbrain. The vault router sitting in between the cache interface and compute tile is responsible for arbitration of many requests as well as sending any out-of-vault access requests to global vault cache mesh network based on the address indexing the request is for. Each vault also has a vault specific compute scheduler which is responsible for taking task requests from a global scheduler and offloading the tasks in each compute tile. The detailed design of this task based queuing model is described in Section 5.2.

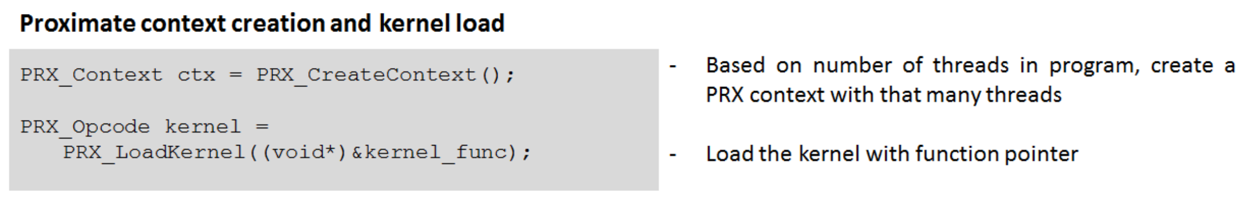
The high-throughput compute engine - Softbrain, is itself a big dataflow engine, and the details of its micro-architecture are not described here as it is itself a big piece of design space to explore. Right now, one can assume each Softbrain unit has a Coarse-Grained Reconfigurable Architecture (CGRA) similar to Dyser [3]. This CGRA has required mix of functional units to execute the kernel portion of regular workload.

5 Proximate Programming Interface and API

This section gives an overview of Proximate programming interface or API and how one would program the entire application using proximate API. And then we will describe the task based queuing model we have implemented for proximate to scheduled tasks for in-order cores as well as Softbrain.

5.1 Proximate API

At a high-level Proximate API is very much similar to any task based API like Cilk [8] or TBB [13]. The only difference is the way memory gets allocated in each of the vault and the data sharding part where the programmer has to explicitly do a copy of the data structures to each of these vault allocated structures.



Proximate context creation and kernel load

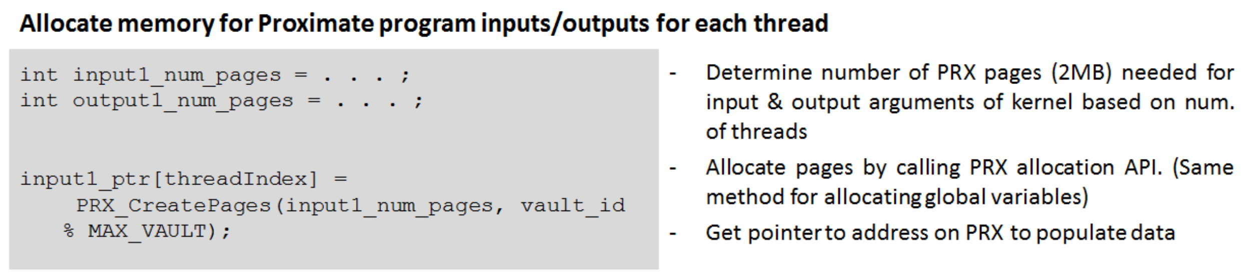
```
PRX_Context ctx = PRX_CreateContext();

PRX_Opcode kernel =
   PRX_LoadKernel((void*)&kernel_func);
```

- Based on number of threads in program, create a PRX context with that many threads
- Load the kernel with function pointer

Figure 8: API for Context Creation and Kernel Loading

Kernel Context and Loading Figure 8 shows the code snippet for Kernel creation and loading the kernel to device memory location. This API can be used by the programmer to register a function/kernel, located in the host address space, with the Proximate library. The input argument is the function pointer to the kernel in the host virtual address space. The return value is the index of the page in the Proximate code Region.

Memory Allocation The next API is for the memory management on Proximate memory mapped region.

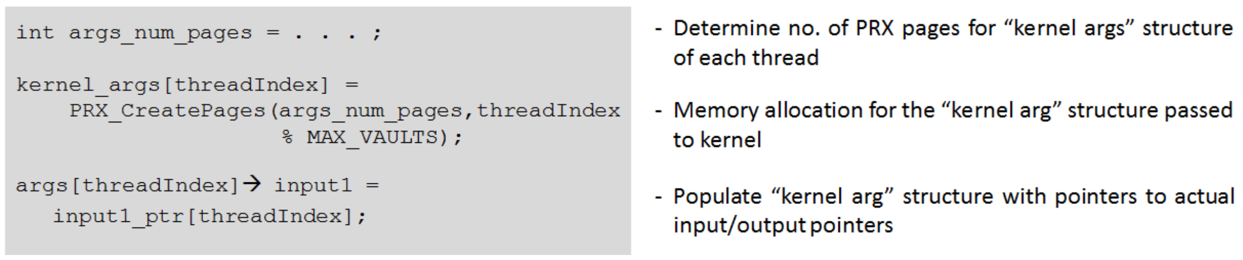


Allocate memory for Proximate program inputs/outputs for each thread

```
int input1_num_pages = . . . ;
int output1_num_pages = . . . ;


input1_ptr[threadIndex] =
    PRX_CreatePages(input1_num_pages, vault_id
   % MAX_VAULT);
```

- Determine number of PRX pages (2MB) needed for input & output arguments of kernel based on num. of threads
- Allocate pages by calling PRX allocation API. (Same method for allocating global variables)
- Get pointer to address on PRX to populate data

Figure 9: API for Input Arguments Memory Allocation



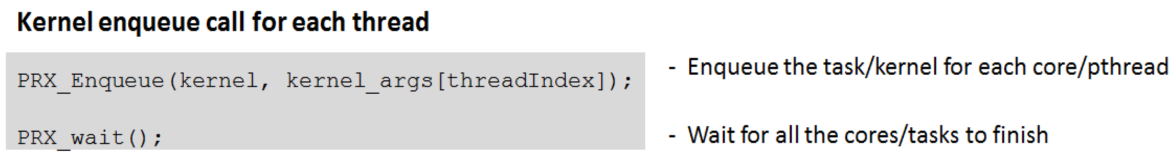
```
int args_num_pages = . . . ;

kernel_args[threadIndex] =
    PRX_CreatePages(args_num_pages,threadIndex
                     % MAX_VAULTS);

args[threadIndex]→ input1 =
   input1_ptr[threadIndex];
```

- Determine no. of PRX pages for "kernel args" structure of each thread
- Memory allocation for the "kernel arg" structure passed to kernel
- Populate "kernel arg" structure with pointers to actual input/output pointers

Figure 10: API for Kernel Argument Structure Allocation

Figure 9 shows the code snippet for memory allocation of input and output arguments required for the computation. This API can used by the programmer to allocate required number of pages on the Proximate side. The input argument is the number of pages required. The size of each page is 1 MB, chosen based on our analysis of the per-kernel data set size that are suitable for the target workloads. The return value is the virtual address of the next free page in the host mmaped heap space.

Since, *pthread* based kernels only take a single argument, we need to pack all the input/output arguments into a global structure and pass the address of that structure to kernel. So, to achieve this Proximate uses the same memory allocation based *PRX_CreatePages* API for a global "args" structure. Figure 10 also hows how a global "args" structure is created and memory is allocated for the same.



Kernel enqueue call for each thread

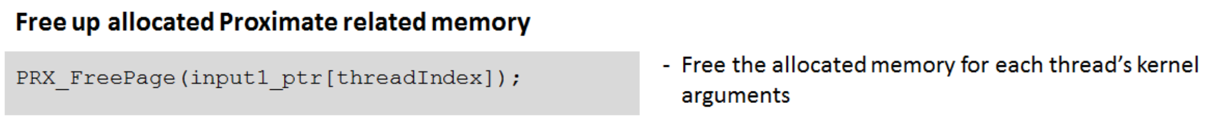
```
PRX_Enqueue(kernel, kernel_args[threadIndex]);

PRX_wait();
```

- Enqueue the task/kernel for each core/pthread
- Wait for all the cores/tasks to finish

Figure 11: API for Kernel Enqueuing and Synchronization

Kernel Enqueuing and Waiting This API can be used by the programmer to offload a kernel for computation on the Proximate compute tile. The input arguments are: Kernel page index in the Proximate space (return value of *PRX_LoadKernel*), and virtual address (in the host mmaped space) of the argument structure which is the only allowed argument to the kernel. The argument structure might look like the one shown in Figure 10. Note that Proximate physical memory space for the argument structure and for each of input array, output array and global variable pointers within that structure, must be allocated using *PRX_CreatePages* before calling *PRX_Enqueue*. The programmer also uses *PRX_Wait* API to wait for all Enqueueed kernels to finish execution. There are no input arguments or return value.

Memory Free The programmer can use the API shown in Figure 12 to update the memory to indicate that the set of pages associated with this allocation are now free.



Free up allocated Proximate related memory

```
PRX_FreePage(input1_ptr[threadIndex]);
```

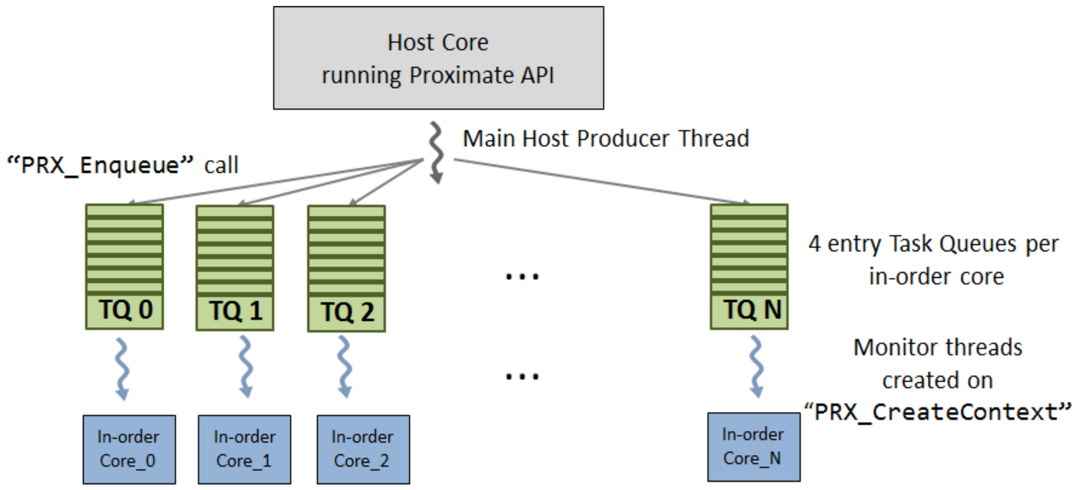
- Free the allocated memory for each thread's kernel arguments

Figure 12: API for Memory Freeing

There are some explicit API calls for memory copying to the sharded structures, but not indicating them here because of the space constraint. One can imagine them to be very much similar to *CUDA_MemCpy* based API calls.

5.2 Proximate Task Based Queuing Model

In order to facilitate an efficient task offloading scheme, we need a queuing model in order to take in the kernel offload requests and schedule them efficiently on the cores or softbrain. So, we extended the kernel scheduler of proximate to support a simple queuing model which is based on the MIT Swarm based task model [7, 6].



Host Core running Proximate API
Main Host Producer Thread
"PRX_Enqueue" call
4 entry Task Queues per in-order core
TQ 0
TQ 1
TQ 2
TQ N
Monitor threads created on "PRX_CreateContext"
In-order Core_0
In-order Core_1
In-order Core_2
In-order Core_N

Figure 13: Proximate Task Based Queuing Model

Figure 13 shows the high-level overview of the Proximate queuing model which is implemented in ZSim simulator. In this task based model, each in-order core has a 4 entry task-queue (TQ) able to store the offloaded tasks/kernels. When the runtime on host core calls the *PRX_CreateContext* with the number of pthread instances to be launched, that manu *monitor threads* are created for each in-order to monitor the per-core task queue. These monitor threads are always running threads monitoring the task-queue for new work, and destroyed only when the *PRX_Free* API is called.

Everytime, the host core running the proximate runtime offloads a task using, *PRX_Enqueue* the task is scheduled on one of the task-queues on a round-robin fashion. And when the task-queue gets full, the task scheduler will notify the host core not to issue anymore kernel calls. We do not have a sophisticate load balancer one would need when handling with many kernels, but we consider that as a part of future work.

6 Methodology

This section describes our evaluation methodology for evaluating proximate and running the proximate parallel programs.

6.1 Workloads

We first describe the workloads we have targeted and then the methodology to evaluate them. Since, proximate is aimed at high performance and throughput for both regular and irregular workloads, we have considered both class of workloads and their parallel implementations.

Regular Workloads Regular workloads are generally the high-performance computing applications which have following properties: i) regular streaming memory access patterns, ii) Computationally intensive and iii) ample amount of data-level parallelism. We consider deep learning neural networks kernels similar to Diannao accelerator engine [1] and the convolution based image processing workloads from Convolution Engine [14]. Some other micro-benchmarks for functional evaluation considered are: Ocean, summation and reduction, but we don't consider in our performance evaluation.

Irregular Workloads Irregular workloads generally have i) irregular memory access patterns with low ILP, ii) Very small computations and iii) Data dependent control. For such workloads, we consider the graph based workloads like histogram, shortest path and breadth-first-search.

The workloads are further explained in detail here: The deep neural networks (DianNao [1]) workload suite is regular workloads with 3 benchmarks: Convolution, Pooling, and Classifier. Convolution applies local filters to the input data, which is used to find the synaptic weight between feature maps. Therefore, convolution is a application with nest matrix multiplication using a sliding window. Pooling aggregates data among a set of neighboring input data, which is used for extracting and scaling key features. Therefore, pooling is a tiling matrix reduction workload. Classifier aggregates all feature maps and classify the object. Classifier is therefore a matrix-vector multiplication workload.

The image processing (Conv. engine [14]) workload suite is regular workloads with 3 benchmarks: IME-SAD, SIFT-Blur, and SIFT-DoG. Sum of Absolute Difference (SAD) is a 2D kernel to take the difference between 2 images for integer motion estimation (IME). IME computes the motion vector by searching for an image blocks closest match to the reference. IME can be used in video compression by encoding image blocks and their motions, as opposed to encoding every pixel of every frame. Blur is a 1D convolution used for blurring images for Scale Invariant Feature Transform (SIFT). SIFT looks for distinctive features for object detection in images. Difference of Gaussian (DoG) is a 2D matrix subtraction operation that can also be used for SIFT. The extremas of the DoG pyramid shows the features of interest.

The irregular workloads explored are breadth first search (bfs), sssp (single source shortest path), and histogram. Breadth first search is an algorithm for searching a graph data structure. Bfs starts at the root node and explores all neighboring nodes before the next level neighbors, which matches the application of a task queue. Sssp finds the minimum cost path from source s to every other vertex. Sssp works on the node with the shortest path, and then add all its neighbors to the work queue. Multiple tasks can be executed concurrently with some scheduling to ensure that the task that gives each vertex the shortest path. Histogram reads every pixel of an image, and sums up the number of pixels with red/green/blue with values of 0/1/2/.../254/255. Histogram shows the color (RGB) value (0 through 255) frequency. Histogram is an irregular workload because it accesses/increments a color value count.

6.2 Simulation Methodology

For proximate multi-core riscv inorder core simulation, we consider a multi-core simulator called ZSim [15], which is a fast x86-65 simulator. It is based on dynamic binary translation and PIN [9] model which natively runs the pthread instances of the kernel on the real hardware.

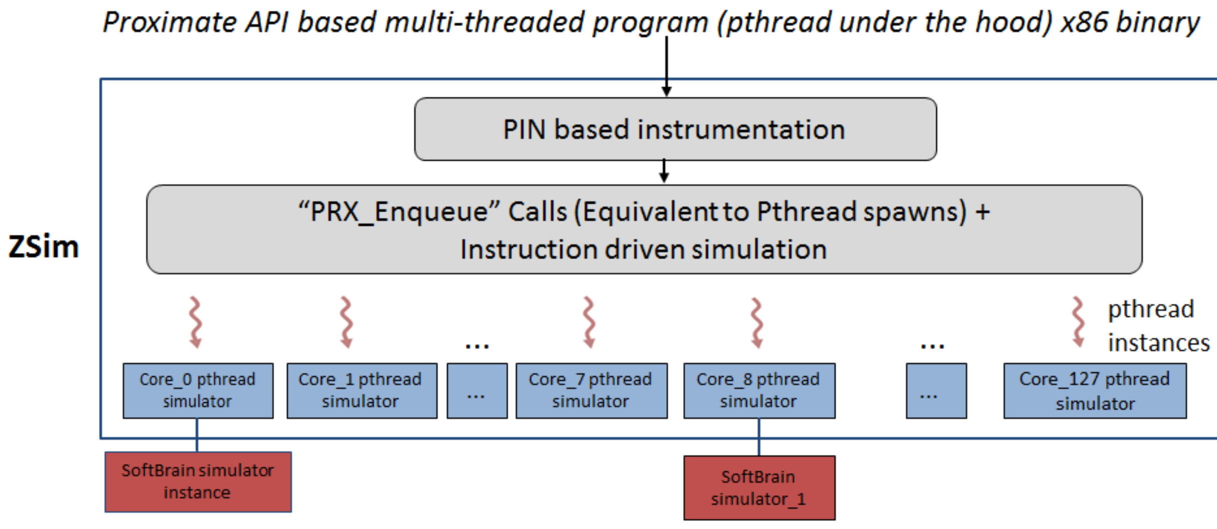

Proximate API based multi-threaded program (pthread under the hood) x86 binary
ZSim
PIN based instrumentation
"PRX_Enqueue" Calls (Equivalent to Pthread spawns) + Instruction driven simulation
pthread instances
Core_0 pthread simulator
Core_1 pthread simulator
...
Core_7 pthread simulator
Core_8 pthread simulator
...
Core_127 pthread simulator
SoftBrain simulator instance
SoftBrain simulator_1

Figure 14: ZSim based Proximate Simulation

Figure 14 shows our simulation methodology, and each compiled proximate API based pthread program ins instrumented for multi-core simulation initially. Then, the proximate API based hooks would identify the proximate kernel and whenever ZSim main thread encounters a *PRX_Enqueue* API call, it would offload the kernel to a) multiple in-order core pthread instances - if the kernel is an irregular type, or b) Softbrain simualtor instance which runs the softbrain portion of the kernel, if the kernel is a regular type. Currently, we don't have simulator support to run, multiple instances of Softbrain kernels, and hence all the softbrain based parallel kernels need to be serialized to the same softbrain simulator instance. We aim to support this in our infrastructure in coming days.

6.3 Proximate Configurations

In this section, we explain different configurations of proximate we have simulated, to compare against the real hardware.

Baseline Configuration	**Explanation**
pthread_xeon	Pthread program run on a 64-core Xeon-Phi processor
omp_xeon	OpenMP program run on a 64-core Xeon-Phi processor

Table 1: Baseline Xeon-Phi configurations used for comparison

For comparison of parallel program execution on proximate with a server class processor, we first the baseline versions of parallel programs implemented in pthreads and OpenMP running on r a 64-core 4way SMT Xeon-Phi processor. Table 1 explains the 2 baseline versions of parallel programs we run on Xeon-Phi machine. We vary the number of threads for these 2 baselines, to find out the best design space w.r.t number of threads and choose that as the comparison point for proximate parallel program.

We now explain, the actual Proximate configurations simulated using the methodology explained above. As we know that current proximate hardware has *1 host-core + 128 in-order cores + 1 Softbrain*, we try to support the same hardware configuration in our simulations. Although, ideally you would be needing 8 Softbrain instances running on each vault, our infrastructure cannot support that for now and hence only 1 softbrain.

Based on kernel type, each proximate program can be simulated only on in-order cores without the queueing model explained previously here in Section 5.2. Or, if more kernel instances are present in the program, then they can be dynamically spawned on the queuing model of proximate. When you have softbrain portions in the program, you would want to run on softbrain only simulation. So, there are different ways of simulating proximate based on the kernel type and Table **??** summarize the possible simulated configurations for proximate. We would be using these configurations for comparison in our evaluation.

Proximate Configuration	**Explanation**
prx_inorder	Proximate API pthread program w/o queuing model
prx_inorder_q	Proximate API pthread program with queuing model
prx_sb_only	Proximate API + single SoftBrain only program
prx_multi_sb	Proximate API + multi-threaded SoftBrain program – Not simulated

Table 2: Proximate Simulated Configurations

7 Evaluation and Results

Before explaining the results and analyzing them , we first show the workloads we could post successfully to Proximate and other workloads not able to port to proximate are mainly because of bugs in the proximate queuing model.

Workload Suite	**Benchmark**	**CPU**	**Pthread**	**OpenMP**	**prx_inorder**	**prx_inorder_q**	**Softbrain**
Deep Neural Regular Workloads	Classifier	yes	yes	yes	yes	yes	yes
	Convolution	yes	yes	yes	yes	no	yes
	Pooling	yes	yes	yes	yes	no	yes
Image Processing Regular Workloads	ime-sad	yes	yes	yes	yes	no	yes
	sift-blur	yes	yes	yes	yes	no	yes
	sift-dog	yes	yes	yes	yes	no	yes
Graph and Irregular Workloads	histogram	yes	yes	yes	yes	yes	Not supported
	bfs	yes	no	no	no	yes	
	sssp	yes	no	no	no	yes	

Table 3: Parallel Workloads Implementation Progress

Table 3 shows the workload progress state for each, and show what has been successfully simulated w.r.t their design space.

7.1 Scalability Analysis

This section first explores the scalability of each workload w.r.t pthread and OpenMP implementation on 64-core Xeon-phi machine.

Figure 15 shows speedups relative to a single thread of neural network applications using pthreads on Xeon Phi. Classifier (red) exhibits linear speedups up to 32 threads from sharding the workload evenly. There is slowdowns after 32 threads due to shared cache capacity, and then slowdown after-wards. Pooling (green) demonstrated poor scalability because it aggregates information among a set of neighboring inputs, which causes more cache conflicts as the number of threads increases. Convolution (blue) is a filtering application, which scales linearly for medium sized kernels. However, larger kernels will reach cache capacity bottlenecks and demonstrate slowdowns after a certain point.

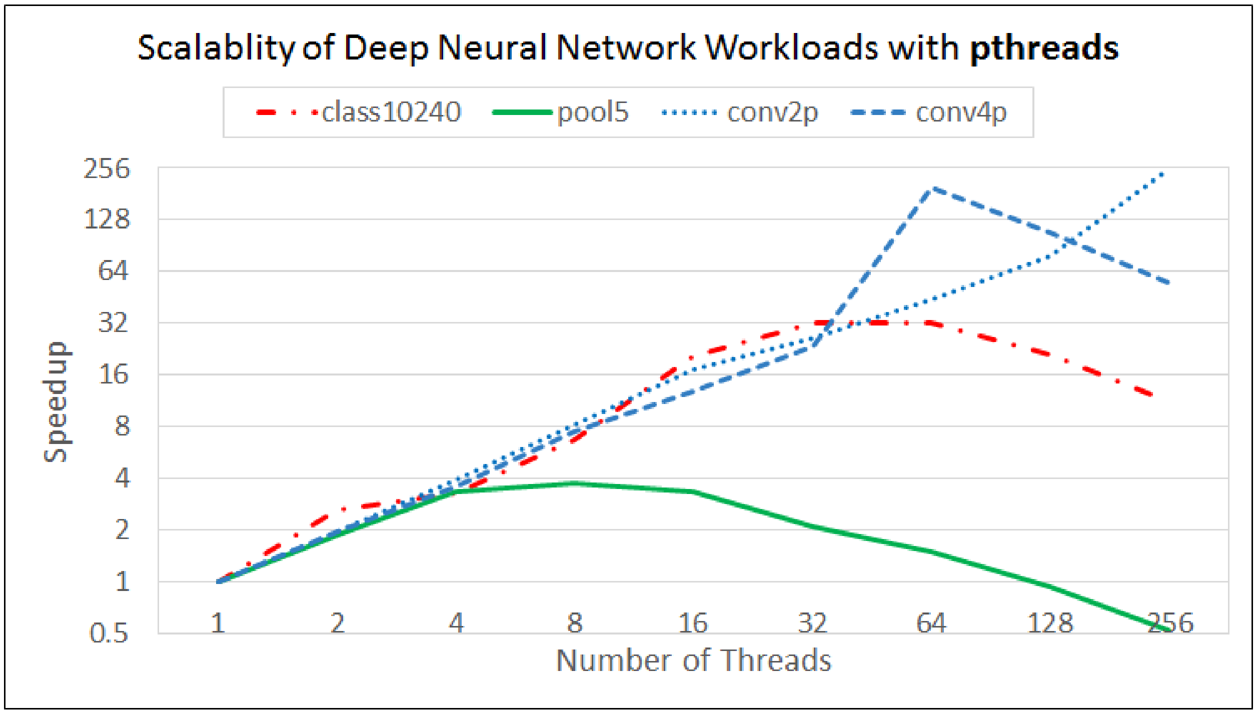


Figure 15: Scalability of Deep Neural Network workload suite using pthreads

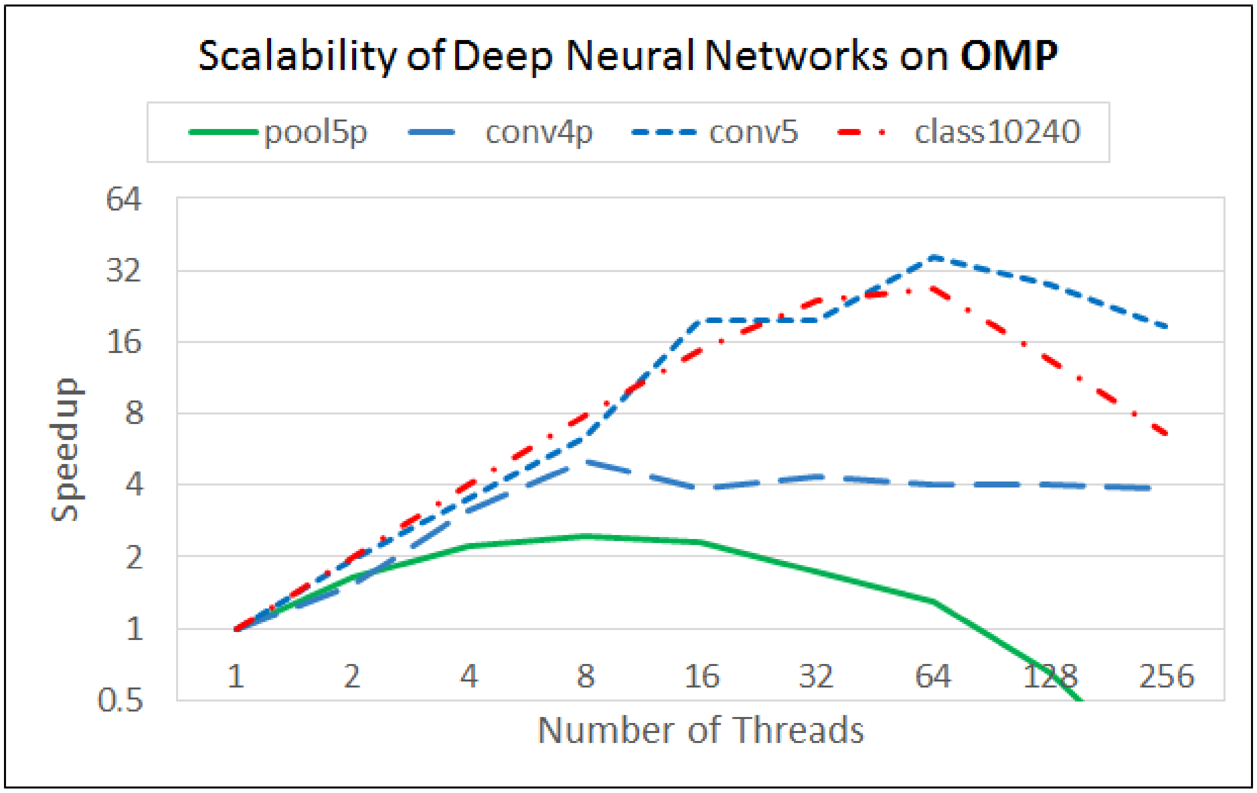


Figure 16: Scalability of Deep Neural Network workload suite using OpenMP

Figure 16 shows speedups relative to a single thread of neural network applications using OpenMP on Xeon Phi. In general, pthreads requires the programmer to do more work and partition data evenly among

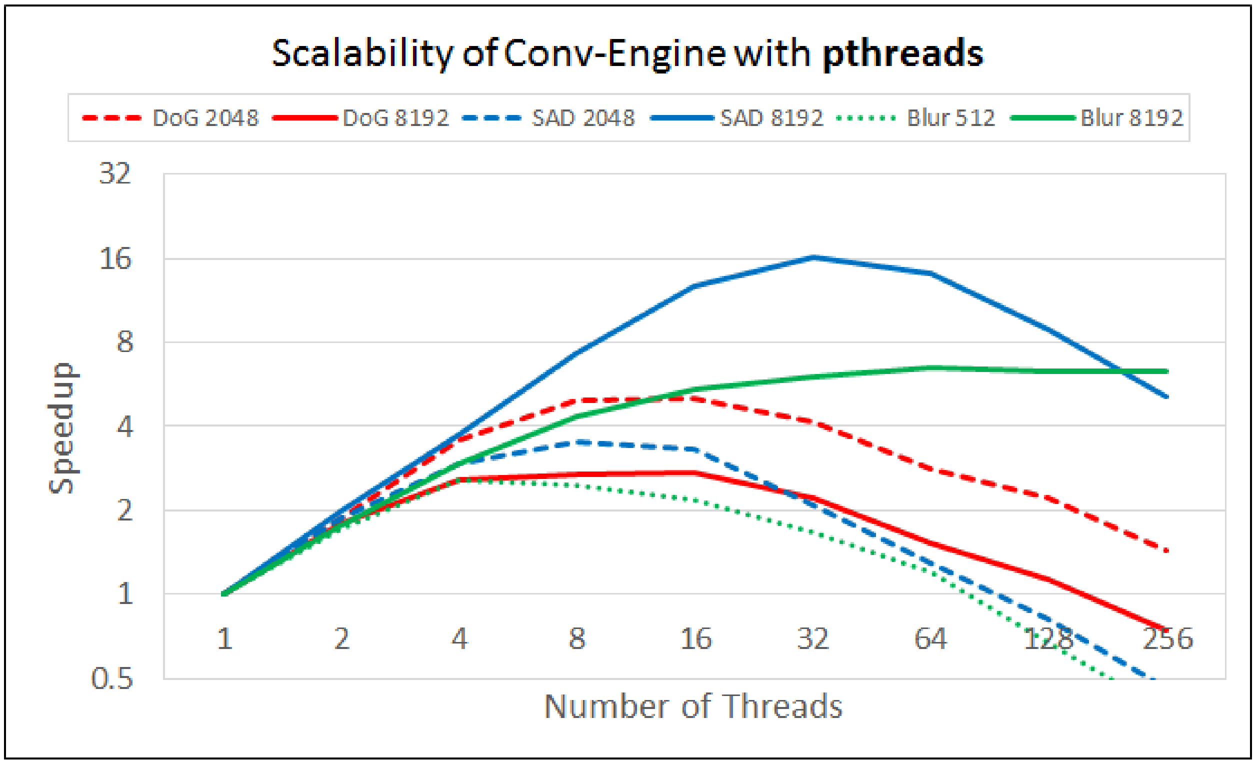


Figure 17: Scalability of Convolution Engine workload suite using pthreads

threads. On the other hand, OpenMP is easier to program simply by adding a pragma *omp* parallel. Therefore, pthreads usually requires more work to get better speedups and OpenMP requires almost no work to get decent speedups.

Figure 17 shows speedups relative to a single thread of image processing convolution engine applications using pthreads on Xeon Phi. DoG (red) exhibits linear speedups up to 4 threads from sharding the workload evenly. There is sublinear speedups up to 16 threads, and slowdowns after wards because of false sharing as each thread is trying to write to the DoG pyramid/image. SAD (blue) shows sublinear speedups, and peaks at different points based on kernel size because larger kernels allows more parallelism. Blur (green) mixes neighboring pixels with each other, so increasing the number of threads on small kernels are quick to cause cache conflicts and slowdowns.

Figure 18 shows speedups relative to a single thread of image processing convolution engine applications using OpenMP on Xeon Phi. Note that DoG OpenMP outperforms pthreads, probably because of dynamic scheduling. Therefore, comparing proximate to both pthreads and OpenMP is important for performance evaluation.

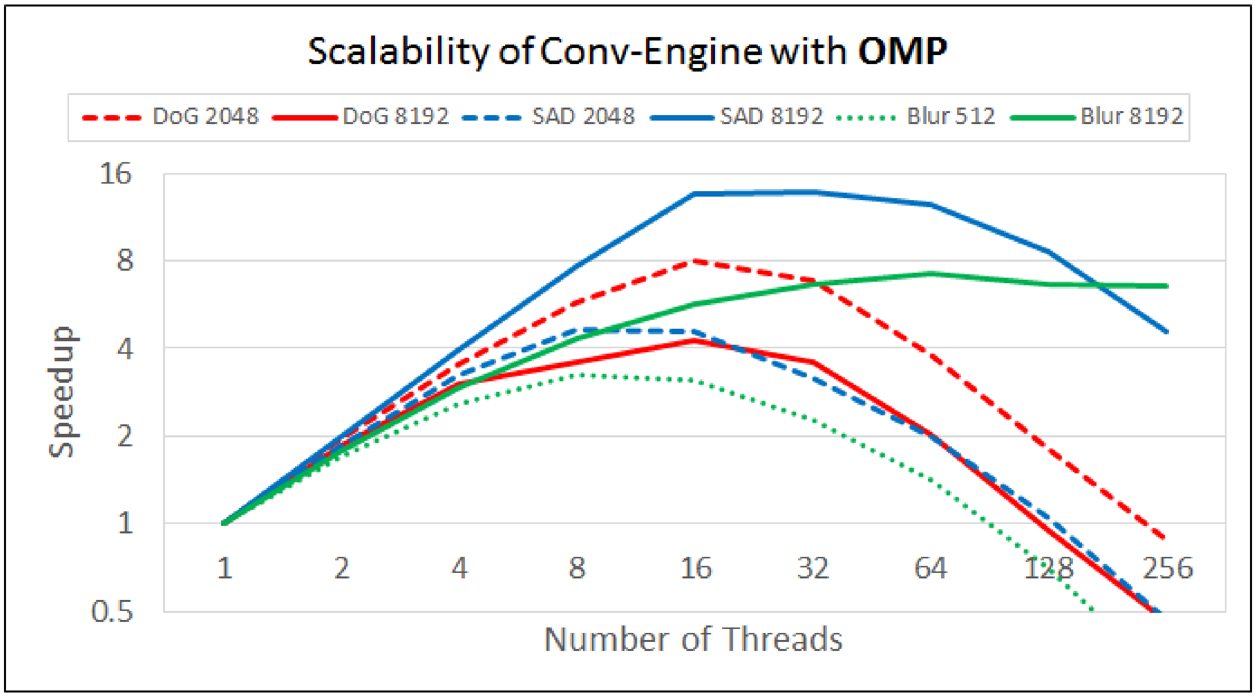


Figure 18: Scalability of Convolution Engine workload suite using OpenMP

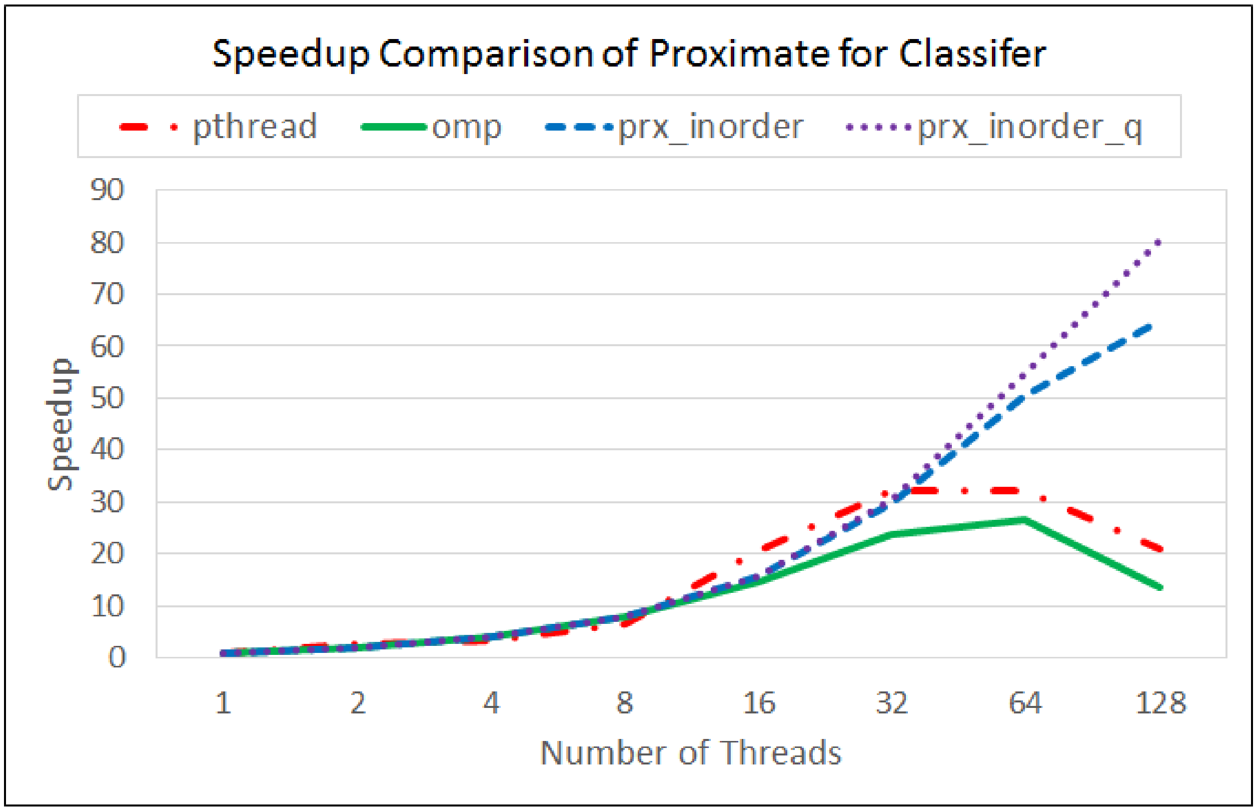


Figure 19: Speedup comparison of Proximate for Classifier with input size of 10240x10240

7.2 Proximate Comparison Analysis

Figure 19 shows speedup analysis of proximate for classifier. First we plotted pthreads (red) and OpenMP (green) as baseline speedup results on CPU using Xeon Phi. Next, we ran the workload on proximate

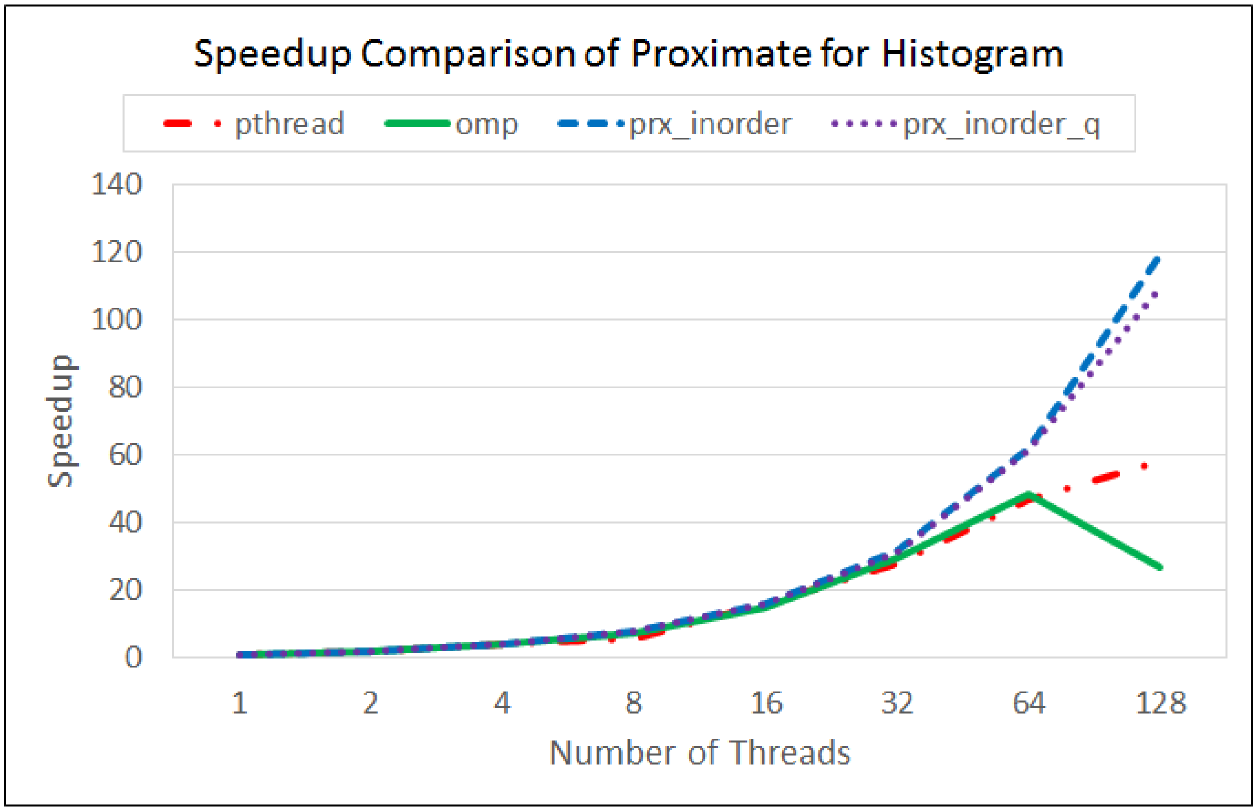


Figure 20: Speedup comparison of Proximate for Histogram with input size of 470 million pixels

without queuing model, and its similarity to pthreads is used to validate the simulator. The differences can be attributed to errors in hardware modeling and hardware limitation. The preliminary results shows that proximate (with queueing model) will given the highest speedup.

Figure 20 shows the same speedup analysis of proximate for histogram. Although histogram is an irregular workload, 256 values of 3 colors were small enough to fit in the thread level cache and result in good speedups. Proximate again gives the highest performance, which informs us to design hardware exactly the way the simulator works to outperform Xeon Phi.

7.3 Summary of Proximate Results

This section shows the proximate speedup summary of both one regular and one irregular workload comparison to a baseline 1-core machine. We don't have all the workloads analyzed this way, mainly because of proximate queueing model not working for those workloads. Also, irregular workloads perform very poorly on Softbrain and that is the reason we dont run them on Softbrain.

Figure 21 shows summarizes how different programming paradigms compare on different workloads to compare OpenMP with the optimal number of threads on Xeon Phi vs pthreads with optimal number of threads on Xeon Phi vs proximate with the optimal number of threads versus a single softbrain instance.

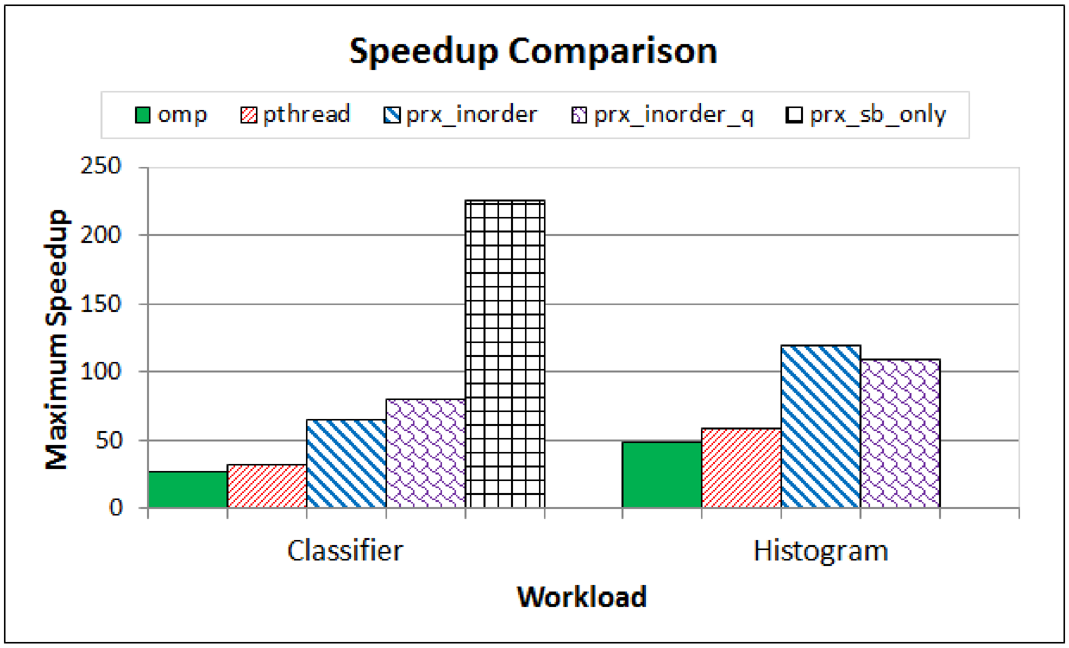


Figure 21: Speedup comparison between multiple proximate configurations

Softbrain does not support irregular workloads such as histogram. In general, proximate is about twice as fast as pthreads, and pthreads is about 20% faster than OpenMP.

8 Challenges

The challenges we faced is implementing a queueing model on Zsim. The queueing model should mimic the Swarm style execution model to schedule and execute multiple irregular workloads concurrently. The challenge was debugging and getting to working in a short amount of time.

Another challenge we faced is choosing the right base line for all experiments. Small workloads results in slowdown for pthreads and OpenMP while large workloads requires unaffordably slow runtimes on simulators. Furthermore, there were differences between real hardware and Zsim, each of which shows peak speedup at different kernel size or different number of threads. Therefore, we had to run preliminary results to get a estimate of which workload configurations works well on which parallel programming paradigms.

Yet another challenge that we faced is the large design space of programming paradigms for a large set of workload. The programming paradigms includes sequential, OpenMP, pthreads, proximate_inorder, proximate_inorder_with_queueing, proximate_softbrain_only. The workload suites include neural networks DianNao, image processing convolution engine, and irregular graph workloads, each of which has multiple benchmarks.

9 Conclusion and Future Work

This project focused on exploring the parallel programming design space of Proximate, a multi-tile programmable hardware accelerator. The aim is to investigate the programming interface of Proximate, the parallel hardware improvements and in general explore the parallel programs run on proximate. We have tried to explore a new type of the multi-tile programmable architecture, its scalability and speedup of such accelerator architecture compared to a traditional server class multi-core processors for parallel programs. In summary, we explored different programming design points in the project and proximate is able to achieve speedups of 50-100x over a traditional server class multi-core processor.

Proximate is a programmable accelerator that targets high efficiency and programmability. It is a compute engine for both regular and irregular workloads.

Proximate exposes a flexible programming model with a task-based programming API. The programmer manages data sharding to achieve task and data locality for performance and scalability. The simple queueing model supports more concurrency with locality as each worker thread will only execute tasks in their own queue. Preliminary results shows that Proximate outperforms current multiprocessor programming paradigms by a factor of 10x for regular workloads and a factor of 2x for irregular workloads. Although multi-threaded SoftBrain may cause new issues such as synchronization, schedule, and saturate memory bandwidth, it could uncover new bounds of performance. We leave multi-threaded SoftBrain to future work.

10 Appendix

This section acts as an appendix and explain an example Classifier program in Proximate space.

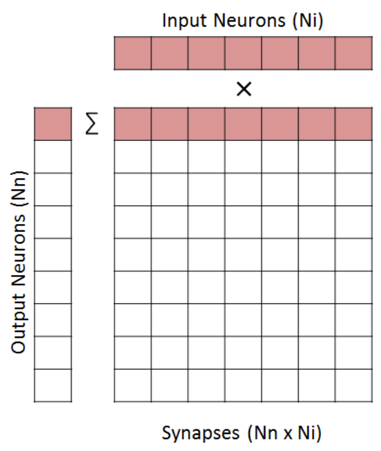
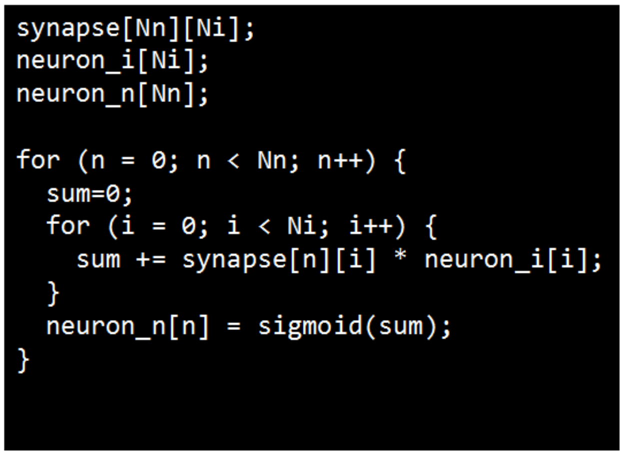


Figure 22: Classifier workload diagram

Figure 22 shows the classifier workload. Classifier requires multiplying the input neurons (a vector) by

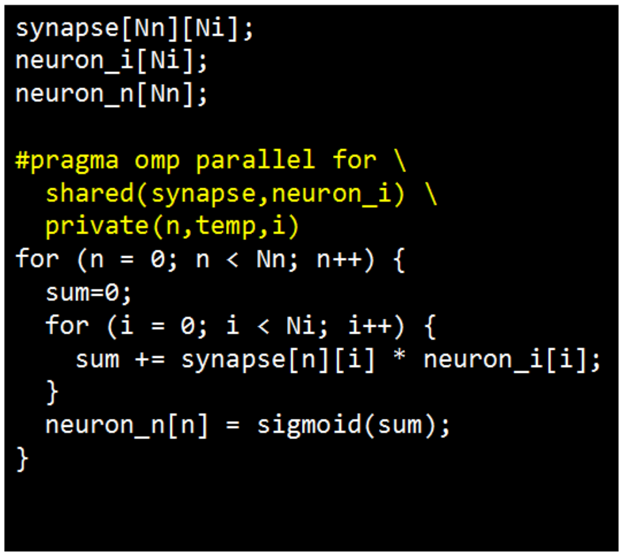


```
synapse[Nn][Ni];
neuron_i[Ni];
neuron_n[Nn];

for (n = 0; n < Nn; n++) {
  sum=0;
  for (i = 0; i < Ni; i++) {
    sum += synapse[n][i] * neuron_i[i];
  }
  neuron_n[n] = sigmoid(sum);
}
```

Figure 23: Classifier implementation using sequential CPU

synapses (a matrix) and then doing the summation reduction.

Figure 23 shows the sequential implementation of classifier. Classifier can be done with a nested for loop to take the summation of the product between the input neurons and synapses.



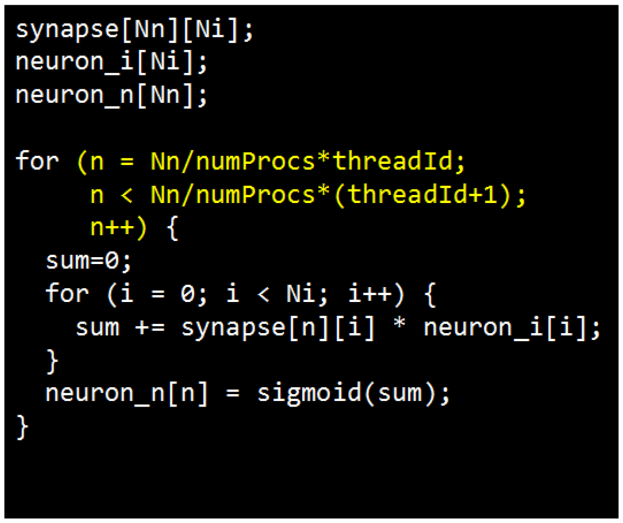
```
synapse[Nn][Ni];
neuron_i[Ni];
neuron_n[Nn];

#pragma omp parallel for \
  shared(synapse,neuron_i) \
  private(n,temp,i)
for (n = 0; n < Nn; n++) {
  sum=0;
  for (i = 0; i < Ni; i++) {
    sum += synapse[n][i] * neuron_i[i];
  }
  neuron_n[n] = sigmoid(sum);
}
```

Figure 24: Classifier implementation using OpenMP

Figure 24 shows the OpenMP implementation of classifier. The OpenMP implementation of classifier is using a *#pragma omp parallel* on the outer loop.

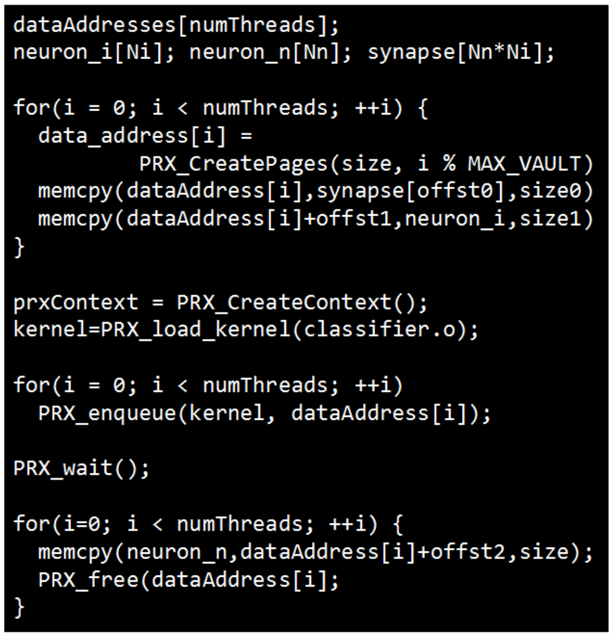
Figure 25 shows the pthread implementation of classifier. The pthreads implementation of classifier shards the workload among pthreads.

Figure 26 shows the proximate implementation with inorder cores and queuing model of classifier. The proximate queuing implementation of classifier shards the workload to different vaults. Calling create context will spawn threads to monitor work queues. Then, load the kernel so that each thread know which kernel to



```
synapse[Nn][Ni];
neuron_i[Ni];
neuron_n[Nn];

for (n = Nn/numProcs*threadId;
     n < Nn/numProcs*(threadId+1);
     n++) {
  sum=0;
  for (i = 0; i < Ni; i++) {
    sum += synapse[n][i] * neuron_i[i];
  }
  neuron_n[n] = sigmoid(sum);
}
```

Figure 25: Classifier implementation using pthreads

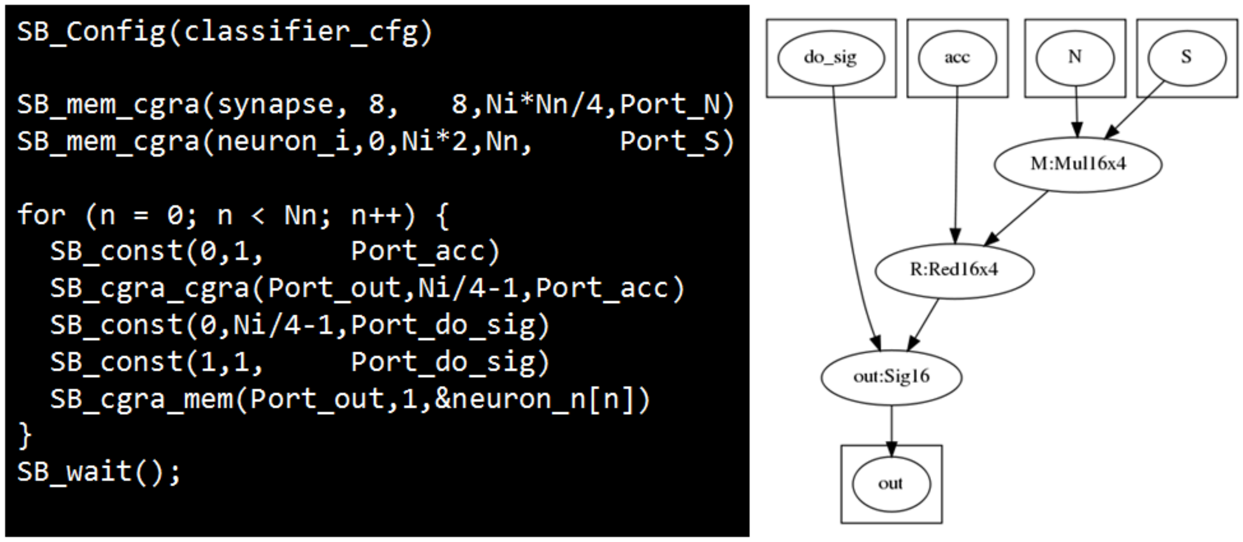


```
dataAddresses[numThreads];
neuron_i[Ni]; neuron_n[Nn]; synapse[Nn*Ni];

for(i = 0; i < numThreads; ++i) {
  data_address[i] =
          PRX_CreatePages(size, i % MAX_VAULT)
  memcpy(dataAddress[i],synapse[offst0],size0)
  memcpy(dataAddress[i]+offst1,neuron_i,size1)
}

prxContext = PRX_CreateContext();
kernel=PRX_load_kernel(classifier.o);

for(i = 0; i < numThreads; ++i)
  PRX_enqueue(kernel, dataAddress[i]);

PRX_wait();

for(i=0; i < numThreads; ++i) {
  memcpy(neuron_n,dataAddress[i]+offst2,size);
  PRX_free(dataAddress[i];
}
```

Figure 26: Classifier implementation using proximate with inorder cores and queuing model

execute upon receiving their thread arguments. Next, enqueue the workloads and wait for them to finish. Finally, gather the results back into a single array.

Figure 27 shows the proximate implementation with softbrain only of classifier. The softbrain implementation first configures the hardware to handle classifier. Then, pass inputs and synapses into softbrain. Loop through the softbrain fabric to do multiple iterations of the computation, one for each output. Finally, wait for the softbrain computational fabric to finish.



```
SB_Config(classifier_cfg)

SB_mem_cgra(synapse, 8,   8,Ni*Nn/4,Port_N)
SB_mem_cgra(neuron_i,0,Ni*2,Nn,     Port_S)

for (n = 0; n < Nn; n++) {
  SB_const(0,1,     Port_acc)
  SB_cgra_cgra(Port_out,Ni/4-1,Port_acc)
  SB_const(0,Ni/4-1,Port_do_sig)
  SB_const(1,1,     Port_do_sig)
  SB_cgra_mem(Port_out,1,&neuron_n[n])
}
SB_wait();
```

Figure 27: Classifier implementation using proximate with softbrain only

References

[1] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA '11*.

[3] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy efficient computing. *IEEE Micro*, 33(5), 2012.

[4] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA '10*.

[5] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

[6] M. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. Swarm: A scalable architecture for ordered parallelism.

[7] M. C. Jeffrey, S. Subramanian, C. Yan, J. S. Emer, and D. Sanchez. Unlocking ordered parallelism with the swarm architecture. *IEEE Micro*, 36(3):105–117, 2016.

[8] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[9] Chi-Keung Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, PLDI, 2005, pp. 190-200.

[10] J. Menon, L. De Carli, V. Thiruvengadam, K. Sankaralingam, and C. Estan. Memory processing units. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pages 1–1. IEEE, 2014.

[11] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright. Pushing the limits of accelerator efficiency while retaining programmability. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 27–39. IEEE, 2016.

[12] T. Nowatzki and K. Sankaralingam. Analyzing behavior specialized acceleration. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 697–711. ACM, 2016.

[13] C. Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[14] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: Balancing efficiency & flexibility in specialized computing. In *ISCA*, 2013.

[15] D. Sanchez and C. Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer Architecture News*, 41(3):475–486, 2013.

[16] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic. The risc-v instruction set manual volume ii: Privileged architecture version 1.7. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-49*, 2015.