Professional report

Machine Learning
In-Memory

MIAI Label

Sylvan Brocard

le campus numérique
in the ALPS

Campus Numérique in the Alps

Professional report

Machine Learning In-Memory

by

Sylvan Brocard

Sylvan	Data+5
Brocard	2021–2022

Company Tutor:	Julien Legriel
School Tutor:	Marie Thieulin
Project Duration:	June, 2021 - May, 2022
Company:	UPMEM SAS
School:	Campus Numérique in the Alps

Cover:	UPMEM DIMM (artistic render)
Style:	EPFL Report Style, with modifications by Batuhan Faik Derinbay

le campus numérique in the ALPS

Preface

The author would like to thank the following people for their support:

- Julien Legriel for his mentorship and support during the apprenticeship.
- Denis Makoshenko for his supervision and support during the apprenticeship.
- Marie Thieulin for her concern and support during the apprenticeship.
- Fabrice Devaux for inventing a heck of a technology.
- My colleagues at UPMEM for the stimulating coffee breaks talks.
- Juan Gómez Luna and Yuxin Guo of ETH Zürich for their open mind and collaboration.
- The Campus Numérique in the Alps staff for their expertise and teachings.
- My friend Tom for his discussions and the Lex Fridman podcast recommandation.
- And finally my wife for her love and support.

The source code of this report is available at `https://github.com/SylvanBrocard/MIAI_report`. It also includes a dev container to compile the report.

Sylvan Brocard
Grenoble, May 2022

Contents

Preface i

Nomenclature iii

1 Introduction 1

2 UPMEM 2
2.1 The company . . . 2
2.2 The product . . . 2
2.3 The team . . . 2
2.3.1 Work environment . . . 3

3 Project 4
3.1 The technological context . . . 4
3.1.1 Motivation . . . 4
3.1.2 Architecture . . . 4
3.1.3 Limitations and Constraints . . . 4
3.2 Algorithms . . . 5
3.2.1 K-Means . . . 5
3.2.2 Decision Trees . . . 6
3.3 Collaboration . . . 6
3.4 Work Organization . . . 7
3.5 Deliverables . . . 7
3.5.1 Python Libraries . . . 7
3.5.2 Benchmarks . . . 7
3.5.3 Compiler Simulation . . . 7

4 Results 8
4.1 K-Means . . . 8
4.1.1 Implementation . . . 8
4.1.2 Results . . . 9
4.2 Decision Trees . . . 11
4.2.1 Implementation . . . 11
4.2.2 Results . . . 12

5 Conclusion 14
5.1 Lessons and future works . . . 14
5.2 Personal assessment . . . 14

References 15

A A deep dive into the compiler 20
A.1 Comparison . . . 20
A.2 Operands auto-promotion . . . 20

Nomenclature

If a nomenclature is required, a simple template can be found below for convenience. Feel free to use, adapt or completely remove.

Abbreviations

Abbreviation	Definition
Memory	
PIM	Processing-in-Memory
DIMM	Dual In-line Memory Module (a.k.a. a memory stick)
RAM	Random-Access Memory
SDRAM	Synchronous Dynamic Random-Access Memory
MRAM	Main RAM
WRAM	Work RAM
DDR	Double Data Rate
DMA	Direct Memory Access
Processing	
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit (a.k.a. a processor)
GPU	Graphics Processing Unit (a.k.a. a graphics card)
DPU	Data Processing Unit (UPMEM technology)
MKL	Math Kernel Library
Miscellaneous	
SDK	Software Development Kit
CI	Continuous Integration
LAN	Local Area Network
R&D	Research and Development
SSH	Secure Shell protocol
AWS	Amazon Web Services
EC2	Elastic Compute Cloud

1

Introduction

“You cannot answer a question that you cannot ask, and you cannot ask a question that you have no words for.”

Judea Pearl, pioneer of Bayesian Networks

The following is a report on the work I conducted at UPMEM during my apprenticeship with the Campus Numérique In The Alps.

A point has been reached in Machine Learning where the ubiquitousness of graphics cards as hardware accelerators has become a problem. An environmental problem, as the focus has been mostly on performances rather than power savings ; and a supply problem, as the hardware shortage in the past years has crippled entire industries. A few contenders have emerged as alternatives to graphics cards, and UPMEM technology is one of those.

In this report, I will explain the UPMEM technology and demonstrate that it undeniably has great potential for Machine Learning applications.

2

UPMEM

2.1. The company

UPMEM is a fabless[1] semiconductor company. It was founded in 2015 by electronics engineers Gilles Hamou and Fabrice Devaux.

At the time of this writing it counts nineteen employees, mostly located at the Grenoble office. There is a secondary office at Station F in Paris, and a couple of employees work remotely.

2.2. The product

UPMEM's main product is a memory-centric hardware accelerator for data-intensive applications, generally referred to as PIM, for Processing-in-Memory. It is sold either as individual memory sticks (DIMM), or preinstalled in an Intel server. The company also offers PIM-enabled cloud computing services for customers and researchers.

UPMEM PIM boasts the following features:

- **General-purpose processor:** A wide range of instruction set for various types of calculation.
- **Massively parallel:** Up to 2560 PIM units can be combined in a single socket server with 256GB PIM DRAM.
- **Larger bandwidth:** 2,5 Tera bytes per second of memory bandwidth.
- **Extra computing power:** Roughly equivalent to 15 additional x86, with the main CPU used for orchestration.

So far some of the most mature and successful applications are:

- **UPVC:** A genomic application for mapping and variant calling.
- **Index Search:** A database application.

Most of the applications so far are however at the prototype stage.

2.3. The team

I was attached to the Application and SDK team for my apprenticeship, under the supervision of tech lead Julien Legriel. The team is part of the software department, headed by director Denis Makoshenko.

There are five members in the Application and SDK team: the tech lead, two software engineers, one CIFRE PhD candidate and myself.

[1] Meaning it designs microchips but contracts out their production.

2.3.1. Work environment

- **OS:** All software employees work in a Linux environment (Ubuntu Budgie in my case).
- **Machines:** There are several PIM-enabled cloud servers reserved for development and internal R&D, accessible via a SSH connection. There are also regular servers for development and CI accessible only through the company LAN. Server access and security is provided by JumpCloud services. All servers and cloud machines run on Debian.
- **Sources:** The UPMEM toolchain[2] source code is hosted on private GitHub repositories. It is tested nightly on a local server with Jenkins. The applications codes are also hosted on the company GitHub account, with visibility settings depending on the project confidentiality.
- **Cloud:** When the need arises for extra computing power or for GPU calculations, the company subscription to AWS allows me to use EC2 instances. For long-term storage, the company uses a Google Drive.
- **Project tracking:** The company uses Jira for project tracking, although this is a recent addition, and most workflows are not yet integrated.

[2]The set of programming tools to develop for UPMEM PIM.

3

Project

3.1. The technological context

The idea of processing in memory is an old one and has been considered since the mid 1990s [1–100]. Despite being the first company to reach the stage of a commercially available PIM product, UPMEM is in need of more technological demonstrators to encourage widespread adoption of its system.

My role during this apprenticeship was to develop machine-learning applications running on PIM that display an enticing performance gain (be it speed or energy consumption) when compared to existing CPU or GPU implementations.

3.1.1. Motivation

Data transfer on SDRAM is limited by a communication bus. Currently, the transfer rate on DDR5-6400 is 51.2 GB/s per channel (409.6 GB/s on an 8-channel system). Although this speed has been steadily increasing, there are popular algorithms today that end up with memory-bound performances (such as pooling operations in neural networks [101]).

The principle of PIM is to execute as many operations as possible in memory, and to minimize the number of operations that are executed in the CPU, thus limiting the flow of data through the memory bus.

3.1.2. Architecture

UPMEM DIMM is based on DDR4 and attaches an onboard RISC-V processor to each 64 MB memory bank, these processors are called DPU for Data Processing Units. A schematic representation of an UPMEM DIMM can be seen in Figure 3.1. The DPUs access their MRAM via a DMA engine. The transfer rate between each DPU and its associated memory bank (called MRAM) is 1 GB/s. This means that on a server with the maximum number of DPUs (2560), the total transfer rate is 2.5 TB/s. Each DPU can execute up to 24 tasklets in parallel.

Each DIMM has two ranks with 64 DPUs each. This amounts to 8.2 GB of memory per DIMM. Each DPU also has an internal Work RAM (WRAM) of 64 KB.

In reality, at the hardware level, DPUs come in pairs, and each pair is connected to a memory bank. However, this is entirely transparent to the programmer. Therefore, we will ignore this fact.

3.1.3. Limitations and Constraints

A number of engineering choices had to be made to make UPMEM PIM a viable product. We will now list the ones that are most relevant to developers:

- **Floating point arithmetic:** Due to the limited space available on the chip, the DPUs do not have a floating point unit, and therefore do not support floating point operations at the hardware level. It is still possible to perform floating point operations on the DPUs, but these are simulated operations, and therefore much slower than hardware supported operations. For example, a single 32-bit floating point multiplication on a DPU will take upwards of 120 instructions.

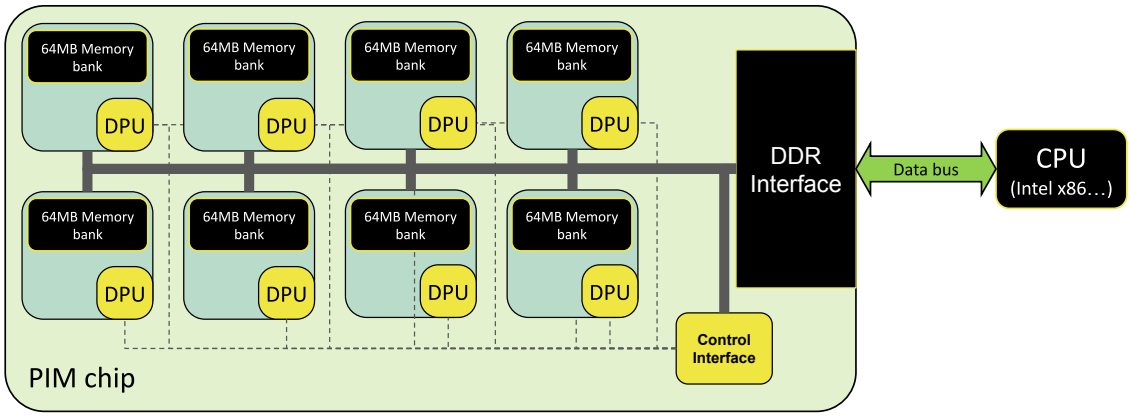


Figure 3.1: UPMEM PIM Architecture

- **Discrete arithmetic:** At the hardware level, the DPUs support 32 bits integer additions/subtractions, and 8 bits integer multiplications.
- **CPU-DPU concurrency:** The memory banks cannot be accessed by the DPUs and host CPU at the same time. Therefore, the DPUs cannot be running while the host CPU is reading results from the memory banks. This prevents any kind of streaming programming model. Instead, iterations must be performed between host and PIM.
- **Data parallelism:** The DPUs cannot communicate with each other. The orchestration must be performed by the host CPU. Since every CPU-DPU communication is relatively costly, implemented algorithms must be data-parallel.
- **Thread parallelism:** Being RISC processors, the DPUs need to execute instructions in parallel to take full advantage of the available resources. Instructions in a DPU go through a pipeline of 11 stages when they execute. Thus, we need to use at least 11 tasklets to reach peak performance. In practice, we generally use 16 tasklets because it's a power of two, and because the extra tasklets can sometimes hide the latency of DMA accesses. Tasklets can synchronize via mutexes or semaphores, but this has a performance cost and should be kept to a minimum.
- **Stack size:** Each DPU has to share its WRAM between its tasklets, and generally keep some space for global variables. This usually leaves 1 KB or 2 KB of stack for each tasklet.
- **DMA memory accesses:** Accessing the MRAM is slower than the WRAM, and it incurs some overhead. Therefore, MRAM accesses should be done in large blocks, or streamed with a sequential reader. Performing too many random accesses impacts performance negatively. Also, there are alignment constraints: each memory access has to be aligned on 8 bytes. Forgetting that fact can lead to concurrency issues if two tasklets try to modify two values in the same 8-byte aligned memory location.
- **Compilation:** The DPU kernels are compiled with a modified version of the LLVM compiler. While it does generally perform as expected, the C standard was mostly created with x86 and x86_64 processors in mind. This means that sometimes the compiler will generate code that is not optimized for the target architecture. An example of that behavior can be seen in section A.2.

3.2. Algorithms

Here we will present the machine learning algorithms that we chose to implement in UPMEM PIM. Both those algorithms are implemented in Scikit-learn [102] and RAPIDS [103], allowing us to run benchmarks against CPU and GPU.

3.2.1. K-Means

K-means [104] is a popular clustering algorithm, due to its simplicity and efficiency. It is used to identify unlabeled groups in a dataset, by grouping them in clusters around a so-called centroid. The algorithm is divided into an E (for Evaluation) phase, where the points in the dataset are assigned to the closest

centroid, and an M (for Move) phase, where the centroids are recalculated as the average of the points in their cluster. A simplified flowchart of the algorithm can be seen in Figure 3.2.

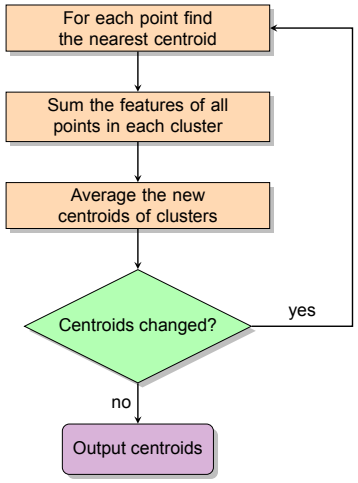


Figure 3.2: The K-Means algorithm on CPU

This natural division of the K-Means suits the UPMEM architecture, as the E step can be entirely performed in memory, and the M step can be performed when the CPU and DPUs synchronize.

It should be noted that there exists several variants of the K-Means algorithm, such as the Elkan variant [105] which uses the triangle inequality to reduce the amount of distances calculations that have to be performed. For our implementation and for comparisons, we stuck with the original Lloyd variant. There are two reasons for that:

- It is the simplest to implement.
- When it comes to the Scikit-learn implementation, it turns out in most benchmarks that the Elkan variant is slower. This boils down to implementation. The core of the Lloyd algorithm is essentially a large matrix multiplication. The Scikit-learn implementation takes full advantage of that fact by making a direct call to the Intel MKL [106] to perform that multiplication. Being developed by Intel, the MKL uses the vectorization capabilities built in x86 processors to speed up the computation. By contrast, the Elkan variant is implemented with naive loops, and puts the onus of optimization on the compiler, which cannot reach the same performances. It should be noted that Intel also only implemented the Lloyd algorithm in their own Data Analytics Library.

Since we are trying to compare hardware capabilities, it is better to stick to the Lloyd variant.

A point should be made about the norm used to compute distances. Since performing multiplications on DPUs is slow, it could be tempting to use the L1 norm instead of the Euclidean L2 norm. However, a naive approach consisting of simply replacing the L2 distances with L1 distances in K-Means would not converge in the general case [107]. The reason is that K-Means is a variance-minimizing algorithm, and the mean is the minimizer of the variance. The minimizer of the L1-variance is the median. Thus, if we wanted to avoid multiplications altogether, we should implement the K-Medians [108] algorithm. While it is possible in principle, it would require using a median-of-median approach [109], and implementing it on PIM while limiting CPU-DPU communication would be much more challenging.

3.2.2. Decision Trees

Decision trees [110] are tree-based algorithms used for classification and regression, commonly known as CART (for Classification And Regression Trees). They successively partition the dataset based on thresholds. A simplified representation of a tree-building algorithm can be seen in Figure 3.3. Trees are a good candidate for PIM because the amount of calculations is limited, and most of the workload is about memory reads and writes.

In this work, we only implemented classification trees, as this allows us to avoid any sort of floating-point operations on the DPUs, other than comparisons. Furthermore, we only implemented the so-called *extremely randomized tree* [111], referred to as *extra-tree* in Scikit-learn. Unlike the regular trees, this variant of the algorithm only selects thresholds at random, and does not use any heuristic to select the best threshold. This allows us to make the CPU ”forget” about the dataset once it has been loaded into the DIMMs. Extremely randomized tree present a larger bias and lower variance than regular trees, and are commonly used as the base unit for forests of randomized trees [112].

3.3. Collaboration

Once the potential of machine learning algorithms in PIM started to become apparent, we decided to start a collaboration with Juan Gómez Luna of ETH Zürich and his student Yuxin Guo.

Dr. Gómez Luna provided us with the academic insights on what the scope of a publication on machine learning in-memory should be. He also wrote the bulk of our joint article. Meanwhile, Yuxin Guo implemented the Linear Regression and Logistic Regression algorithms in UPMEM PIM. I won't expand on the result of his work in his report, but they can be found in the soon-to-be-published article.

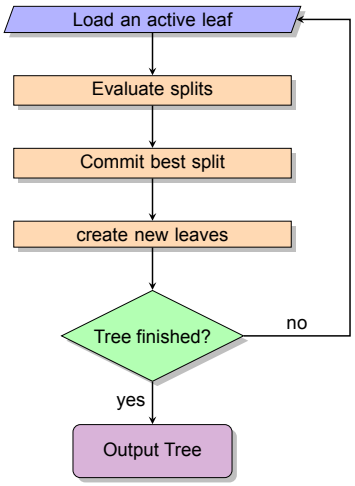


Figure 3.3: A tree algorithm on CPU

3.4. Work Organization

Internally, the software team has two weekly stand-ups on Monday and Wednesday, and one software point on Friday. The stand-ups are meant to update the rest of the team on the current advancement of projects, and the point is to discuss the projects at more length. There is also a company-wide stand-up on Tuesday, where department heads update everyone on their advancement. This project also had weekly synchronization meetings with the ETH Zürich team, where we discussed necessary additions to the article.

For most of the work, the experiment results were shared with the ETH team in Google Sheets, however I've now set up a data version control system with DVC [113] to track the experiments.

3.5. Deliverables

3.5.1. Python Libraries

My first task at my arrival at UPMEM was to work on an existing C code for K-Means that had been provided by an academic partner, and suffered from extremely poor optimization (about 1000 times slower than a CPU code). As part of my work on the code, I decided to reimplement it as a Python library. The main motivation here being that if we want to garner interest among data scientists for UPMEM PIM, there should be a Python API. The library is available on UPMEM GitHub repository. It uses C code for the backend, and Pybind-11 for the Python bindings.

Decision trees are also available as a Python library. In an effort of fairness, most of the backend code is written in Cython, to mimic the Scikit-learn implementation.

For now the two libraries are separate, but they will be merged, along with Linear Regression and Logistic Regression, in a yet-to-be-named PIM ML library.

3.5.2. Benchmarks

With the co-publication in mind, a lot of effort was put into benchmarking the algorithms. This required implementing various performance counter, as well as modifying parts of Scikit-learn to add those same performance counters. We decided not to use native Python profiling tools for this task, because they have a significant overhead and shouldn't be used for benchmarking, as stated in the Python documentation [114].

3.5.3. Compiler Simulation

Apart from generating interest, the applications team role is also to inform the hardware team about which improvements to the future versions of the DPUs would result in the best performance gains. In this optic, I tested different versions of the K-Means on the DPU software simulator, with simulated assembly instructions, and compared the resulting instructions counts.

4

Results

4.1. K-Means

4.1.1. Implementation

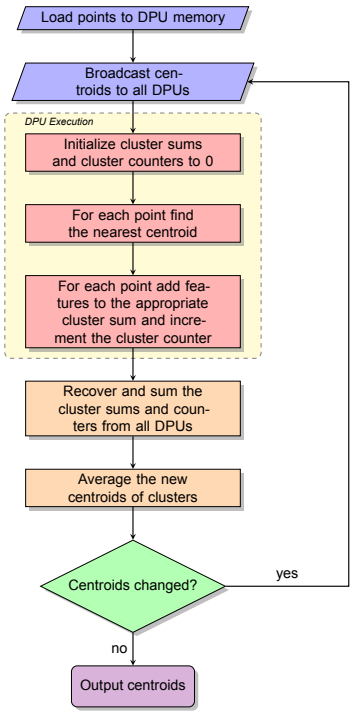


Figure 4.1: The K-Means algorithm on DPU

The main workload in K-Means is to measure pairwise distances between each point-centroid pair. Obviously this would be way too slow in floating point arithmetic with DPUs. My approach is instead to do a linear quantization of the data on 15 bits. That is to say the features are mapped linearly to the range [-16385,16384] and encoded as 16-bit integers. The extra bit of leeway is there so that we don't overflow when subtracting features. The entire algorithm is then run in this format, and only at the end the quantized data is converted back to floating point.

This brings two questions. The first one is why choose 16 bits and not 8, since the hardware supports only 8-bit multiplications? Wouldn't that mean that we would end up being 4 times slower than in 8 bits, since performing a 16 bits multiplication with 8 bits hardware needs 4 multiplications? To understand this choice, remember that the DPUs are RISC processors. This means that for each cycle they only execute a very simple instruction. For example, a multiplication needs two load instructions (one for each operand), one multiplication instruction, and one store instruction.

Consider the following simple function to compute the squared euclidean distance between two vectors:

```
#include <stdint.h>

int64_t euclid(
    int16_t* a,
    int16_t* b,
    int size
) {
    int64_t accumulate;
    for(int i=0; i<size; i++) {
        volatile int16_t diff = a[i] - b[i];
        accumulate += diff * diff;
    }
    return accumulate;
}
```

Once compiled for DPUs, this function ends up being 26 instructions long. Only 12 instructions are spent on the actual multiplication, and the rest is spent on the subtraction, the loop and the return. If we replace the inputs and the difference with 8-bits integers, the function is now 17 instructions long. All

in all, that's only a 60% performance degradation for more than double the numerical precision, which is quite the trade-off.

The other question is can we trust the algorithm once we've quantized the data? Surely, it is possible to construct a counter-example where the quantization would lead to an incorrect result. However, and allow me to emphasize this, **floating point K-Means has similar issues, especially in 32 bits** [115]. Working in floating point isn't a guarantee against rounding errors, quite the opposite. Working with integers, we can at least be sure that small values don't get rounded to zero when we perform long addition loops.

The important question isn't theoretical, but empirical: does the algorithm generally work on real datasets? We will see that it does.

The implementation is shown in Figure 4.1. First, the data on the host is quantized, split (without redundancy) and loaded to the DPUs. Then, the initial random centroids are broadcasted to the DPUs. The DPUs keep a registry of cluster coordinate sums and cluster point count. The DPUs compute the nearest centroid for each point, and adds this point's coordinates to the appropriate cluster sum and increments the cluster point count. Once all the DPUs have returned, the host reads the cluster sums and counts from all DPUs, and aggregates them to compute the new centroid coordinates. The host then compares the new centroid coordinates to the old ones, and if they are different, the centroids are broadcasted and the process is repeated. The process is repeated until the centroid coordinates don't change anymore.

There is one final step omitted for simplicity: the DPUs run one more time to compute the total inertia. This is necessary to select the best clustering when we restart the K-Means algorithm with different initialization.

Do note that, unlike on the CPU implementation, at no point do we keep a list of cluster assignments for the data points, neither in the host nor in the DPUs. The final output is a list of centroids, and not the cluster assignments for the data points. This choice was made because we only set out to study training performances and not inference. By contrast, on the Scikit-learn version, the cluster assignments are returned as well, but that is because the optimized CPU version already needs to compute that list during the training.

4.1.2. Results

Hardware

The tests for DPU and CPU are run on a server with an Intel Xeon Silver 4215 processor with 251 GB of RAM. The tests for GPU are run at ETH Zürich on an A100 GPU.

Datasets

For benchmarking, we used a mixture of synthetic and real datasets. Synthetic datasets are convenient for scaling up or down the size and dimensionality for benchmarking and profiling purposes. The real dataset offers a guarantee of real-world applicability.

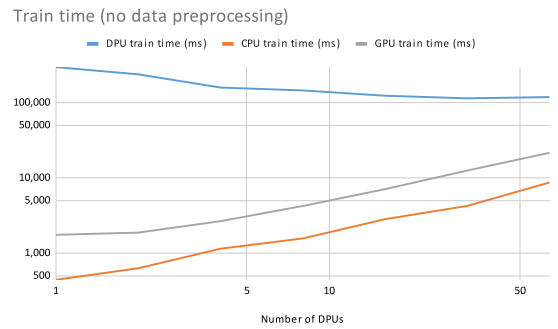
The synthetic datasets are generated using the `make_blobs` function from Scikit-learn. For the weak scaling tests, the datasets have 100,000 samples per DPU, with 16 features grouped into 16 clusters, for a pre-quantization size of 6.4 MB per DPU. For the strong scaling tests, the dataset has 25,600,000 samples total, with 16 features grouped into 16 clusters, for a pre-quantization size of 1.6 GB (halved after 16-bits quantization).

The real dataset is the HIGGS Boson dataset [116] available at [117]. It has 11 million points, 28 features, and one binary label column that we drop for clustering.

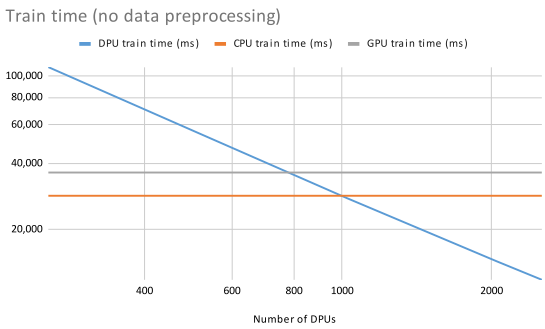
Weak scaling

A weak scaling test means that we scale the number of DPUs, and we also scale the dataset size so that the number of points per DPU remains constant.

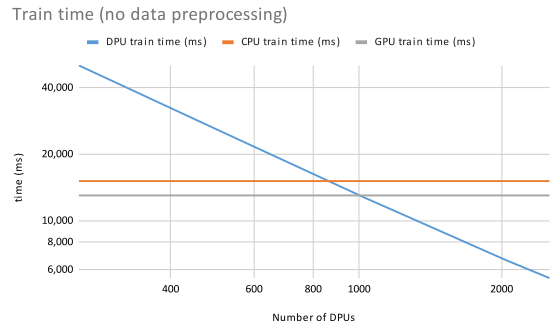
The results compared to CPU and GPU can be seen on Figure 4.2a. We can see that the processing time remains almost constant with the number of DPUs. It even reduces a little, but that's because K-Means converges in few iterations on smaller datasets. The time per iteration (not shown here) remains perfectly constant.



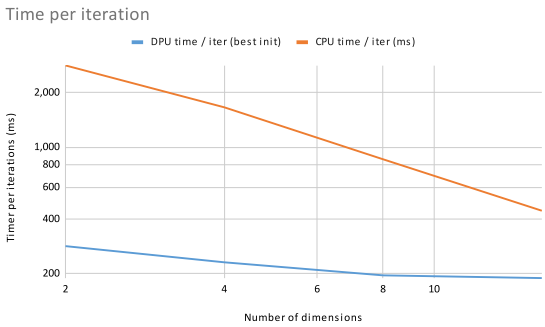
(a) Weak scaling run times for 1 to 64 DPUs



(b) Strong scaling run times for 256 to 2524 DPUs



(c) Higgs run times for 256 to 2524 DPUs



(d) Dimension run times for 2 to 16 dimensions.

Figure 4.2: K-Means benchmark results

Strong scaling

A strong scaling test means that we scale the number of DPUs, but keep the size of the dataset constant.

The results can be seen on Figure 4.2b. We can see that we get a perfectly linear scaling of the performance with the number of used DPUs, surpassing the CPU at 1000 DPUs. We can also notice that the GPU is slower than the CPU. It happens on some datasets, depending on their geometry. This is due to the fact that the GPU implementation of K-Means in RAPIDS is batched for internal memory reasons, and thus less numerically stable.

Higgs

We run the K-Means algorithm on the Higgs Boson dataset. The performance results can be seen on Figure 4.2c. This time the GPU does perform slightly faster than a CPU. UPMEM PIM still outperforms both at 1000 DPUs. The best speedup is achieved at 2524 DPUs with a 2.37x speedup against the GPU.

Dimensionality

The dimensionality of the dataset is also a factor in the performance. The results can be seen on Figure 4.2d. We generate datasets of constant size in memory, but with a varying number of features. The best speedup vs CPU is achieved at the lowest dimensionality. We hypothesize that the CPU is more able to take advantage of its vectorization capabilities at higher dimensions, hence the difference. The DPU/CPU performance ratio goes from 10 at 2 dimensions to 1.54 at 16 dimensions.

Quality

In section 4.1.1, we discussed the potential precision issues coming from quantization. To assess the quality of the produced results, we used the `Calinski-Harabasz` function from Scikit-learn. The Calinski-Harabasz index [118] is a measure of how well the points are clustered. Unlike the silhouette score, it has a reasonable complexity, so we can use it for large datasets. We also use the adjusted Rand index [119] to assess the similarity of the CPU and DPU clusterings.

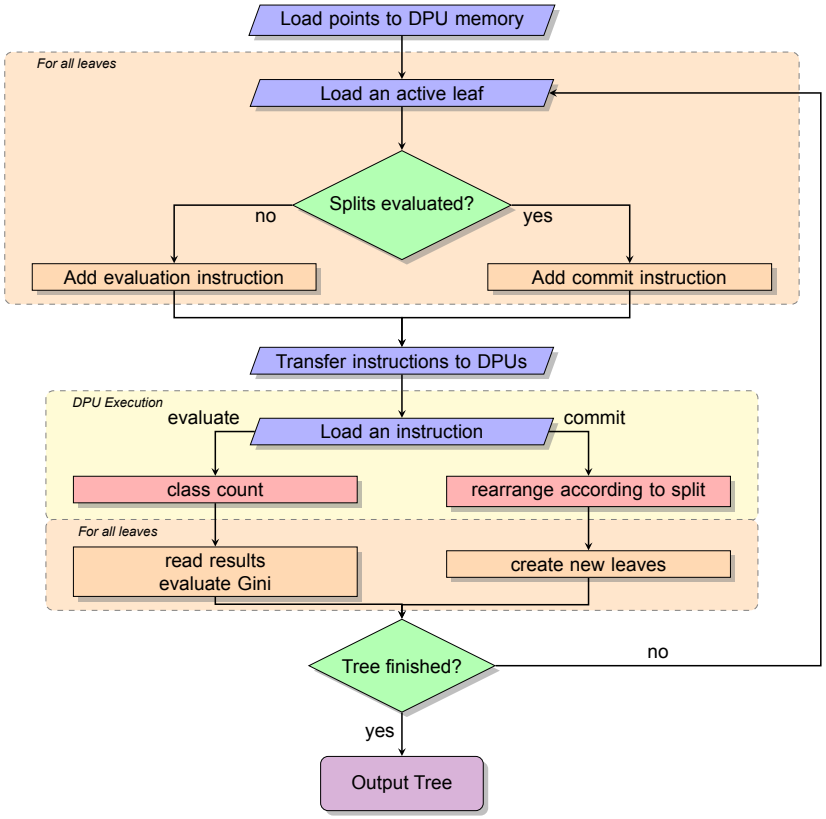


Figure 4.3: The Decision Tree algorithm on DPU

We find that, for every data set, synthetic and natural, both algorithms always converge to the same minimum. The Calinski-Harabasz are identical within at least 5 significant digits, and the lowest adjusted Rand index in all the tests is 0.99889. This shows that the rounding errors don't cause the DPU implementation to diverge.

Pipeline occupancy
We also measured the pipeline occupancy of the DPU kernels. The pipeline occupancy is defined as the number of instructions divided by the number of cycles. It is above 95%. Along with some profiling on the number of used tasklets, this shows that we are not limited by memory accesses.

4.2. Decision Trees

4.2.1. Implementation

Unlike in K-Means, we don't need to do any sort of quantization for decision trees. The only operation on features are comparisons, and those can be run very efficiently even on fixed-point hardware (with a pointer trick described in annex A.1).

Where we diverge from the Scikit-learn implementation is that sklearn builds decision trees in a depth-first fashion. That is, assuming we do not provide a stopping condition on the total number of leaves in the tree, in which case it switches to a best-first approach. The great thing about this is that if we do not use said condition, there is no interdependency between the leaves. Which means the tree build order does not matter. We can still use stopping criteria like maximum tree depth, minimum number of samples to split, minimum number of samples in a leaf, and minimum impurity decrease.

This allows us to build the tree in a breadth-first fashion. At every iteration, the host looks at every active leaf and decides to do one of three things with it:

- Pick a not-yet-evaluated candidate feature for the split, and ask the DPUs "What are the minimum and maximum values of this feature in this leaf?"

- Pick a random threshold between the minimum and maximum, and ask the DPUs “What are the class counts if we split this leaf with that threshold?”
- Decide which split is the most discriminant, add leaves to the current node, and tell the DPUs “Commit to this split by reorganizing the data belonging to this leaf.”

The list of instructions on what to do for each leaf is then sent to the DPUs, which execute them. After they return, the host reads the results and does the appropriate post-processing:

- Aggregate the min and max from each DPUs to find the global min and max.
- Read the class counts and use it to compute an impurity score.
- Decide if the new leaves can be split, or if this branch is finished.

This process is illustrated in Figure 4.3. The algorithm ensures that the data in the DPU memory is always organized according to the tree diagram (i.e. the points belonging to a left node are before the points belong to its sister right node).

The kernel code for the DPUs is quite a bit more complex than in K-Means, and my tutor Julien Legriel wrote the bulk of it when we were pressed for time. It uses streaming memory access to reorganize the data efficiently during a split commit.

4.2.2. Results

Hardware

The hardware used for benchmarking is the same as for K-Means.

Datasets

As in K-Means, we use both synthetic and real datasets. The synthetic datasets are generated using the `make_classification` function from Scikit-learn. For the weak scaling tests, the datasets have 600,000 samples per DPU, with 16 features in 2 classes, for a size of 38.4 MB per DPU. For the strong scaling tests, the dataset has 153,600,000 samples total, with 16 features in 2 classes, for a size of 9.83 GB.

We also use the Higgs dataset, this time using the target labels for classification, and the last 500,000 samples are reserved for validation.

Weak scaling

We can see in Figure 4.4a that the DPU execution time is perfectly constant.

Strong scaling

We can see in Figure 4.4b that the performance gain with the number of CPUs is roughly linear. It does start to saturate after 1024 DPUs. Do note that there is no GPU comparison for this benchmark. The GPU can load the dataset in memory, but doesn’t have enough space left to perform the computation.

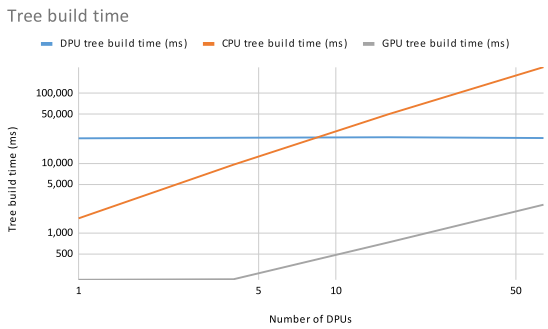
Higgs

Unfortunately, we see in Figure 4.4c that our DPU system reaches saturation before outperforming the GPU. The Higgs dataset is just too small for us to take advantage of the architecture. Nevertheless, we won’t go cherry-picking[1] here just to find a dataset that shows the results we want.

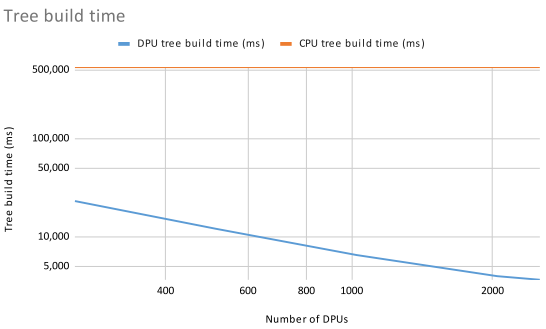
Dimensionality

Figure 4.4d shows that there is also a dimensionality effect at play here. We go from a 191 speedup factor in dimension 2 to 40 in dimension 32. This is harder to explain, we hypothesize that it is due to the fact that a higher dimension means we have to make more, shorter, back-and-forth iterations between the host and DPUs.

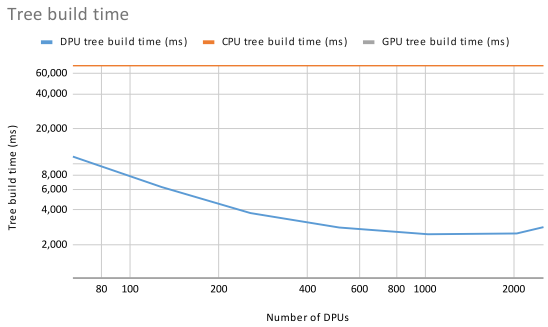
[1] The practice of pointing only to data that seems to confirm a particular hypothesis.



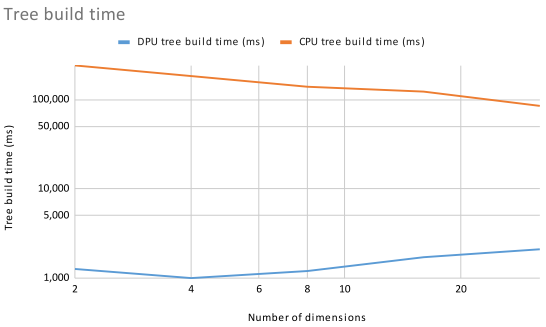
(a) Weak scaling run times for 1 to 64 DPUs



(b) Strong scaling run times for 256 to 2524 DPUs



(c) Higgs run times for 256 to 2524 DPUs



(d) Dimension run times for 2 to 24 dimensions.

Figure 4.4: Decision Trees benchmark results

Quality

Although our implementation is functionally identical with the Scikit-learn version, there is one difference coming from the fact that we don't build tree leaves in the same order. Because this is a probabilistic algorithm, the random number generation ends up being different. So to compare the two, we run the algorithms several times with different seeds and compare the accuracies. This time, this is simply a sanity check on our implementation.

On ten runs on one DPU with 600,000 samples, the mean accuracies for DPU and GPU are 0.90008 and 0.90175 respectively, and the covariance between the two series is 1.39E-4. This confirms our implementation is correct outside of extreme edge cases.

Pipeline occupancy

Perhaps more surprisingly this time, pipeline occupancy and tasklets profiling reveals that we are still not memory-bound, despite this algorithm being much less computationally intensive than K-Means. This is good news for the scalability of the algorithm.

5

Conclusion

5.1. Lessons and future works

One thing stuck out during the benchmark process of the algorithms: there aren't many datasets big enough to fully take advantage of them. Most datasets upwards of 5 GB are not tabular data, but rather in the form of time series or string, so not suitable for those algorithms. Rather than keeping up a search for a use case, my effort is now going to focus on using job-level parallelism to use the full capacities of the PIM on average-sized datasets. Obvious follow-ups would be:

- **K-Means:** Replicate the data on every rank, and run different runs with different random initialization in parallel.
- **Decision Trees:** Build independent trees on every rank, and create a random forest estimator.

Such parallelism is supported by the hardware, but will need some serious implementation effort to set the parallelism at the Python level.

We'll also need to up the antes by expanding the comparison benchmarks to other, more performance-oriented libraries, such as XGBoost and Intel DAL.

5.2. Personal assessment

This apprenticeship has been a period of tremendous personal growth, in terms of technical skills and intellectual maturity and ownership of my work. The development of my library was an opportunity to dive deep into the code of Scikit-learn, as well as the SDK. I have learned a lot about software architecture, and how to use it to my advantage. Being in a start-up environment, this was also the occasion to set up my own continuous integration and data management pipelines.

I feel privileged to be standing on the work of current and previous employees at UPMEM. The hardware, the SDK, the network technological stacks, the profiling tools, all feel like a cathedral built by a small team of brilliant individuals, with hidden gems hidden around every corner.

Of course, I cannot forget the excellent classes provided by the Campus Numérique. They expanded my horizons in digital technology and data science, without them, I would still be unaware of the existence of entire fields of knowledge.

References

[1] Harold S Stone. “A Logic-in-Memory Computer”. In: *IEEE TC* (1970).

[2] W. H. Kautz. “Cellular Logic-in-Memory Arrays”. In: *IEEE TC* (1969).

[3] David Elliot Shaw et al. “The NON-VON Database Machine: A Brief Overview”. In: *IEEE Database Eng. Bull.* (1981).

[4] P. M. Kogge. “EXECUBE - A New Architecture for Scaleable MPPs”. In: *ICPP*. 1994.

[5] Maya Gokhale, Bill Holmes, and Ken Iobst. “Processing in Memory: The Terasys Massively Parallel PIM Array”. In: *IEEE Computer* (1995).

[6] David Patterson et al. “A Case for Intelligent RAM”. In: *IEEE Micro* (1997).

[7] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. “Active Pages: A Computation Model for Intelligent Memory”. In: *ISCA*. 1998.

[8] Yi Kang et al. “FlexRAM: Toward an Advanced Intelligent Memory System”. In: *ICCD*. 1999.

[9] Ken Mai et al. “Smart Memories: A Modular Reconfigurable Architecture”. In: *ISCA*. 2000.

[10] Jeff Draper et al. “The Architecture of the DIVA Processing-in-Memory Chip”. In: *SC*. 2002.

[11] Shaizeen Aga et al. “Compute Caches”. In: *HPCA*. 2017.

[12] Charles Eckert et al. “Neural Cache: Bit-serial In-cache Acceleration of Deep Neural Networks”. In: *ISCA*. 2018.

[13] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. “Duality Cache for Data Parallel Acceleration”. In: *ISCA*. 2019.

[14] Mingu Kang et al. “An Energy-Efficient VLSI Architecture for Pattern Recognition via Deep Embedding of Computation in SRAM”. In: *ICASSP*. 2014.

[15] V. Seshadri et al. “Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology”. In: *MICRO*. 2017.

[16] Vivek Seshadri et al. “RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization”. In: *MICRO*. 2013.

[17] Shaahin Angizi and Deliang Fan. “Graphide: A Graph Processing Accelerator Leveraging In-dram-computing”. In: *GLSVLSI*. 2019.

[18] J. Kim et al. “The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern DRAM Devices”. In: *HPCA*. 2018.

[19] J. Kim et al. “D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput”. In: *HPCA*. 2019.

[20] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. “ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs”. In: *MICRO*. 2019.

[21] Kevin K. Chang et al. “Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM”. In: *HPCA*. 2016.

[22] Xin Xin, Youtao Zhang, and Jun Yang. “ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM”. In: *HPCA*. 2020.

[23] S. Li et al. “DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator”. In: *MICRO*. 2017.

[24] Q. Deng et al. “DrAcc: a DRAM Based Accelerator for Accurate CNN Inference”. In: *DAC*. 2018.

[25] Nastaran Hajinazar et al. “SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM”. In: *ASPLOS*. 2021.

[26] S. H. S. Rezaei et al. “NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories”. In: *CAL* (2020).

[27] Yaohua Wang et al. “FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching”. In: *MICRO*. 2020.

[28] Mustafa F Ali, Akhilesh Jaiswal, and Kaushik Roy. “In-Memory Low-Cost Bit-Serial Addition Using Commodity DRAM Technology”. In: *TCAS-I*. 2019.

[29] Shuangchen Li et al. “Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories”. In: *DAC*. 2016.

[30] S. Angizi, Z. He, and D. Fan. “PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-efficient Logic Computation”. In: *DAC*. 2018.

[31] S. Angizi, A. S. Rakin, and D. Fan. “CMP-PIM: An Energy-efficient Comparator-based Processing-in-Memory Neural Network Accelerator”. In: *DAC*. 2018.

[32] S. Angizi et al. “AlignS: A Processing-in-Memory Accelerator for DNA Short Read Alignment Leveraging SOT-MRAM”. In: *DAC*. 2019.

[33] Yifat Levy et al. “Logic Operations in Memory Using a Memristive Akers Array”. In: *Microelectronics Journal* (2014).

[34] S. Kvatinsky et al. “MAGIC—Memristor-Aided Logic”. In: *IEEE TCAS II: Express Briefs* (2014).

[35] A. Shafiee et al. “ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars”. In: *ISCA*. 2016.

[36] S. Kvatinsky et al. “Memristor-Based IMPLY Logic Design Procedure”. In: *ICCD*. 2011.

[37] S. Kvatinsky et al. “Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies”. In: *TVLSI* (2014).

[38] P.-E. Gaillardon et al. “The Programmable Logic-in-Memory (PLiM) Computer”. In: *DATE*. 2016.

[39] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay. “ReVAMP: ReRAM based VLIW Architecture for In-memory Computing”. In: *DATE*. 2017.

[40] S. Hamdioui et al. “Memristor Based Computation-in-Memory Architecture for Data-intensive Applications”. In: *DATE*. 2015.

[41] L. Xie et al. “Fast Boolean Logic Papped on Memristor Crossbar”. In: *ICCD*. 2015.

[42] S. Hamdioui, S. Kvatinsky, and et al. G. Cauwenberghs. “Memristor for Computing: Myth or Reality?” In: *DATE*. 2017.

[43] J. Yu et al. “Memristive Devices for Computation-in-Memory”. In: *DATE*. 2018.

[44] Christina Giannoula et al. “SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures”. In: *HPCA*. 2021.

[45] Ivan Fernandez et al. “NATSA: A Near-Data Processing Accelerator for Time Series Analysis”. In: *ICCD*. 2020.

[46] Damla Senol Cali et al. “GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis”. In: *MICRO*. 2020.

[47] J. S. Kim et al. “GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies”. In: *BMC Genomics* (2018).

[48] Junwhan Ahn et al. “PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture”. In: *ISCA*. 2015.

[49] Junwhan Ahn et al. “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing”. In: *ISCA*. 2015.

[50] A. Boroumand et al. “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks”. In: *ASPLOS*. 2018.

[51] Amirali Boroumand et al. “CoNDA: Efficient Cache Coherence Support for near-Data Accelerators”. In: *ISCA*. 2019.

[52] Gagandeep Singh et al. “NAPEL: Near-memory Computing Application Performance Prediction via Ensemble Learning”. In: *DAC*. 2019.

[53] Hadi Asghari-Moghaddam et al. “Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems”. In: *MICRO*. 2016.

[54] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. “JAFAR: Near-Data Processing for Databases”. In: *SIGMOD*. 2015.

[55] P. Chi et al. “PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation In ReRAM-Based Main Memory”. In: *ISCA*. 2016.

[56] Amin Farmahini-Farahani et al. “NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules”. In: *HPCA*. 2015.

[57] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. “Practical Near-Data Processing for In-Memory Analytics Frameworks”. In: *PACT*. 2015.

[58] Mingyu Gao and Christos Kozyrakis. “HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing”. In: *HPCA*. 2016.

[59] Boncheol Gu et al. “Biscuit: A Framework for Near-Data Processing of Big Data Workloads”. In: *ISCA*. 2016.

[60] Q. Guo et al. “3D-Stacked Memory-Side Acceleration: Accelerator and System Design”. In: *WoNDP*. 2014.

[61] Milad Hashemi et al. “Accelerating Dependent Cache Misses with an Enhanced Memory Controller”. In: *ISCA*. 2016.

[62] M. Hashemi, O. Mutlu, and Y. N. Patt. “Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads”. In: *MICRO*. 2016.

[63] Kevin Hsieh et al. “Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems”. In: *ISCA*. 2016.

[64] Duckhwan Kim et al. “Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory”. In: *ISCA*. 2016.

[65] G. Kim et al. “Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs”. In: *SC*. 2017.

[66] Joo Hwan Lee, Jaewoong Sim, and Hyesoon Kim. “BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models”. In: *PACT*. 2015.

[67] Zhiyu Liu et al. “Concurrent Data Structures for Near-Memory Computing”. In: *SPAA*. 2017.

[68] Amir Morad, Leonid Yavits, and Ran Ginosar. “GP-SIMD Processing-in-Memory”. In: *ACM TACO* (2015).

[69] Lifeng Nai et al. “GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks”. In: *HPCA*. 2017.

[70] Ashutosh Pattnaik et al. “Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities”. In: *PACT*. 2016.

[71] Seth H. Pugsley et al. “NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads”. In: *ISPASS*. 2014.

[72] D. P. Zhang et al. “TOP-PIM: Throughput-Oriented Programmable Processing in Memory”. In: *HPDC*. 2014.

[73] Qiuling Zhu et al. “Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware”. In: *HPEC*. 2013.

[74] Berkin Akin, Franz Franchetti, and James C. Hoe. “Data Reorganization in Memory Using 3D-Stacked DRAM”. In: *ISCA*. 2015.

[75] Mingyu Gao et al. “Tetris: Scalable and Efficient Neural Network Acceleration with 3D Memory”. In: *ASPLOS*. 2017.

[76] Mario Drumond et al. “The Mondrian Data Engine”. In: *ISCA*. 2017.

[77] Guohao Dai et al. “GraphH: A Processing-in-Memory Architecture for Large-scale Graph Processing”. In: *IEEE TCAD* (2018).

[78] Mingxing Zhang et al. “GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition”. In: *HPCA*. 2018.

[79] Yu Huang et al. “A Heterogeneous PIM Hardware-Software Co-Design for Energy-Efficient Graph Processing”. In: *IPDPS*. 2020.

[80] Youwei Zhuo et al. “GraphQ: Scalable PIM-based Graph Processing”. In: *MICRO*. 2019.

[81] Paulo C Santos et al. “Operand Size Reconfiguration for Big Data Processing in Memory”. In: *DATE*. 2017.

[82] Saugata Ghose et al. “Processing-in-Memory: A Workload-Driven Perspective”. In: *IBM JRD* (2019).

[83] Wen-Mei Hwu et al. “Rebooting the Data Access Hierarchy of Computing Systems”. In: *ICRC*. 2017.

[84] Maciej Besta et al. “SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems”. In: *arXiv:2104.07582 [cs.AR]* (2021).

[85] João Dinis Ferreira et al. “pLUTo: In-DRAM Lookup Tables to Enable Massively Parallel General-Purpose Computation”. In: *arXiv:2104.07699 [cs.AR]* (2021).

[86] Ataberk Olgun et al. “QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAMs”. In: *ISCA*. 2021.

[87] Scott Lloyd and Maya Gokhale. “In-memory Data Rearrangement for Irregular, Data-intensive Computing”. In: *Computer* (2015).

[88] Duncan G Elliott et al. “Computational RAM: Implementing Processors in Memory”. In: *IEEE Design & Test of Computers* (1999).

[89] Le Zheng et al. “RRAM-based TCAMs for pattern search”. In: *ISCAS*. 2016.

[90] Joshua Landgraf, Scott Lloyd, and Maya Gokhale. *Combining Emulation and Simulation to Evaluate a Near Memory Key/Value Lookup Accelerator*. 2021. arXiv: `2105.06594 [cs.AR]`.

[91] Arun Rodrigues, Maya Gokhale, and Gwendolyn Voskuilen. “Towards a Scatter-Gather Architecture: Hardware and Software Issues”. In: *MEMSYS*. 2019.

[92] Scott Lloyd and Maya Gokhale. “Design Space Exploration of Near Memory Accelerators”. In: *MEMSYS*. 2018.

[93] Scott Lloyd and Maya Gokhale. “Near Memory Key/Value Lookup Acceleration”. In: *MEMSYS*. 2017.

[94] Maya Gokhale, Scott Lloyd, and Chris Hajas. “Near Memory Data Structure Rearrangement”. In: *MEMSYS*. 2015.

[95] Ravi Nair et al. “Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems”. In: *IBM JRD* (2015).

[96] Arpith C Jacob et al. *Compiling for the Active Memory Cube*. Tech. rep. Tech. rep. RC25644 (WAT1612-008). IBM Research Division, 2016.

[97] Zehra Sura et al. “Data Access Optimization in a Processing-in-Memory System”. In: *CF*. 2015.

[98] Ravi Nair. “Evolution of Memory Architecture”. In: *Proceedings of the IEEE* (2015).

[99] Rajeev Balasubramonian et al. “Near-Data Processing: Insights from a MICRO-46 Workshop”. In: *IEEE Micro* (2014).

[100] Yue Xi et al. “In-Memory Learning With Analog Resistive Switching Memory: A Review and Perspective”. In: *Proceedings of the IEEE* (2020).

[101] *Memory-Limited Layers User’s Guide*. Tech. rep. NVIDIA Corporation, 2020.

[102] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *JMLR* (2011).

[103] RAPIDS Development Team. *RAPIDS: Collection of Libraries for End to End GPU Data Science*. 2018. URL: `https://rapids.ai`.

[104] Stuart P. Lloyd. “Least Squares Quantization in PCM”. In: *IEEE Transactions on Information Theory* (1982).

[105] Charles Elkan. “Using the triangle inequality to accelerate k-means”. In: *Proceedings of the 20th international conference on Machine Learning (ICML-03)*. 2003, pp. 147–153.

[106] Intel. *Intel oneAPI Math Kernel Library (MKL)*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. 2022.

[107] Paul Bradley, Olvi Mangasarian, and W Street. “Clustering via concave minimization”. In: *Advances in neural information processing systems* 9 (1996).

[108] Anil K Jain and Richard C Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[109] Manuel Blum et al. “Time bounds for selection”. In: *J. Comput. Syst. Sci.* 7.4 (1973), pp. 448–461.

[110] Shan Suthaharan. “Decision Tree Learning”. In: *Machine Learning Models and Algorithms for Big Data Classification*. 2016.

[111] Pierre Geurts, Damien Ernst, and Louis Wehenkel. “Extremely Randomized Trees”. In: *Machine learning* (2006).

[112] Leo Breiman. “Random forests”. In: *Machine Learning* (2001).

[113] Ruslan Kuprieiev et al. *DVC: Data Version Control - Git for Data & Models*. Version 2.10.2. Apr. 2022. DOI: 10.5281/zenodo.6501662. URL: https://doi.org/10.5281/zenodo.6501662.

[114] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[115] Fabienne Jézéquel et al. “Can we avoid rounding-error estimation in HPC codes and still get trustful results?” working paper or preprint. Feb. 2020. URL: https://hal.archives-ouvertes.fr/hal-02486753.

[116] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. “Searching for Exotic Particles in High-energy Physics with Deep Learning”. In: *Nature Communications* (2014).

[117] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[118] Tadeusz Caliński and Jerzy Harabasz. “A Dendrite Method for Cluster Analysis”. In: *Communications in Statistics-theory and Methods* (1974).

[119] Lawrence Hubert and Phipps Arabie. “Comparing Partitions”. In: *Journal of Classification* (1985).

A

A deep dive into the compiler

For this section, rather than show every listing for the source and compile code, I will provide links to the excellent tool `dpu.dev` to follow along. This is a web-based tool that allows us to interactively explore how snippets of codes are compiled for DPUs.

A.1. Comparison

We mentioned in section 4.2.1 a pointer trick. To follow along, check out bit.ly/3a3s3fK.

Consider the following code for a float comparison:

```
1 int leq(float a, float b) {
2     return a <= b;
3 }
```

We can see in line 4 of the compiler that this results in long, simulated float comparison.

Now let's bit-cast out inputs to integers.

```
1 int leq(float aa, float bb) {
2     int a =*(int*)&aa;
3     int b =*(int*)&bb;
4     if (a < 0 && b < 0)
5         return b <= a;
6     else
7         return a <= b;
8 }
```

We're now doing the comparison in 3 instructions! This works because an IEEE floating point number is encoded as

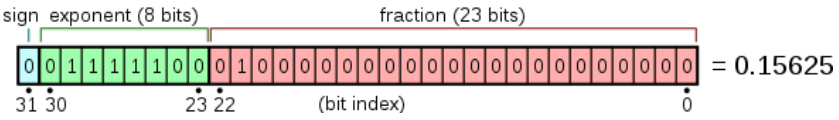


Figure A.1: Float encoding.[1]

So the conversion between a floating point and integer representation is monotonous (except for the inversion for negative 32 bits integers).

A.2. Operands auto-promotion

In section 3.1.3, we mentioned the limitations of the compiler when it comes to optimizing for our target architecture. We're now going to expose one such occurrence and how to deal with it. Follow along in bit.ly/39o9PFx.

Consider the following code to compute a square euclidean distance between vectors `a` and `b`:

[1] By Vectorization: Stannered - Own work based on: Float example.PNG, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=3357169`

```
1  #include <stdint.h>
2
3  int64_t euclid(int16_t* a, int16_t* b, uint32_t size) {
4      int64_t accumulate;
5      for(uint32_t i=0; i<size; i++) {
6          int16_t diff = a[i] - b[i];
7          accumulate += diff * diff;
8      }
9      return accumulate;
10 }
```

What are we expecting to do here?

- On line 6, we expect to compute the difference between two 16-bits coordinates and store it in a 16-bits integer. We know we can do that because we made sure our quantized coordinates fit in 15 bits.
- On line 7, we expect to compute the square of the difference ($16 \times 16 \rightarrow 32$) and accumulate it in a 64-bits integer.

But look at the compiler. There is a simulated 32-bits multiplication on line 17! The compiler coalesced lines 6 and 7. Since the compiler doesn't know that the subtraction won't overflow, it automatically promoted the subtraction result into a 32-bits integer. So now we're doing a $32 \times 32 \rightarrow 64$ multiplication.

This can be solved either by manually writing that part of the code in the assembler, or adding a `volatile` qualifier to the `diff` variable, preventing the compiler from optimizing it out. Now we have the $16 \times 16 \rightarrow 32$ multiplication we expected.

Do note that the C compiler does NOT perform such auto promotions between 32 and 64 bits variables, because the writers of the C standard were aware that it would lead programmers to make such mistakes on 32-bits processors. It's a simple fix, but the issue is so, so easy to miss.