Chapter 6 Literature Review

6.1.1 Vector and Array Processors
6.1.2 GPU Computing
6.1.3 Multiprocessor Systems
6.1.4 Multicomputer Systems

Justin Clark

Introduction

Chapter 6 of [1] covers topics related to parallel and high performance computer systems. In section [1§6.1], a computer system classification called Flynn's Taxonomy is introduced. There are four major classifications of computer systems: 1) uniprocessor systems, or *single instruction stream, single data stream (SISD)*, 2) array processor systems, or *single instruction stream, multiple data streams (SIMD)*, 3) *multiple instruction streams, single data stream (MISD)*, and 4) parallel systems or *multiple instruction streams, multiple data streams (MIMD)*. Section [1§6.1] focuses primarily on SIMD and MIMD systems since all previous chapters focused on SISD and because MISD systems are uncommon.

Section [1§6.1.1] is about vector and array processors. Computer systems based on vector processors have largely fallen out of style, but array processors are important building blocks for large, heterogeneous systems. In section [1§6.1.2], GPU computing is developed from the concepts of the previous section. One might think of GPUs as highly threaded variations of array processors. Then, in [1§6.1.3], the discussion continues on to multiprocessor systems. A simple example of these MIMD parallel systems is a computer with multiple CPUs. One of the biggest problems in this section is *cache coherence*, which is a way of synchronizing the cache memory of all the different processors. Finally, we wrap up with [1§6.1.4] on the topic of multicomputer systems (another variation of MIMD systems).

The topics of [1§6.1] are all extremely important developments as the amount of available data in the world continues to grow exponentially. How does one process and make sense of all the data? In previous computing eras, it would have been impossible. Now, with the advent of fast and highly parallel systems, computer scientists and data scientists are able to process an enormous amount of data relatively easily. This has led to major advancements in artificial intelligence and general data science. In the following sections, we will discuss the most important concepts in each of the corresponding sections of [1] and relate the topics to research currently (and previously) on the forefront of data science.

Vector and Array Processors

As mentioned in [1] and in the introduction, vector and array processors are a type of SIMD architecture. The SISD architectures covered in chapters 1 through 5 are all operating on single numbers (or scalars). Vector and array processors on the other hand can perform operations on data structures known as vectors, matrices and, more generally, tensors. A vector is a one-dimensional “list” or array of numbers (or a one-dimensional tensor). Further, a matrix is a two-dimensional grouping of numbers in rows and columns. One might think of it as a vector of vectors. Even further, a 3D tensor can be thought of as a vector of matrices or a vector of vectors of vectors. Ultimately, processing these data structures boils down to vector operations. These vectors can be processed with single instructions. For example, a very common operation is matrix multiplication. Architecturally, vector and array processors still have a single instruction set (vector multiplication, vector addition, etc.) and multiple data streams (vectors/arrays). The point is made in [1] that operations *could* be done with a sufficient number of loops at the software level, but designing a processor’s architecture to handle these operations at the hardware level is much more efficient.

While vector and array processors can perform the same sorts of instructions, they differ in their architectural implementation. Vector processors are typically parallelized temporally, meaning a small number of execution units are deeply pipelined. Conversely, array processors are spatially parallelized and have many execution units all processing a small part of a data array [1]. With many different units, there are two different ways to organize the memory of array processors: 1) with a global memory system and 2) with a local memory system. To conclude our discussion on [1§6.1.1], it is worth noting that the cost and niche market of vector processors have largely caused them to fall out of style, but array processors are still around because they are easier to integrate with other, general purpose processors.

Support vector machines (SVMs) were one of the earliest non-linear machine learning algorithms (though, they can also be linear with minor adjustments). SVMs are particularly

powerful because they were able to be trained on a relatively small sample size and still generalize well, in contrast to neural networks. This meant that earlier computer architectures could train SVM models effectively in a reasonable amount of time. However, SVMs ran into trouble when the training sample sizes increased. In fact, the time complexity of training SVMs on a popular C++/Java framework called LibSVM is $O(n^3)$ as discussed in [2]. This was obviously far from ideal as the amount of available data in the world grew (and, thus, training samples could grow to increase robustness of models). This is the motivation of the authors of [3] where they propose an architecture called *MatRISC* which takes advantage of the hardware in array processors to more efficiently train SVM models. Note that at the time of [3] being published, most SVM modeling was done on a single desktop computer with a single processor. To understand why MatRISC works, we must first understand, at a high level, how SVMs are trained. For brevity, it suffices to only know that several large matrices are added, multiplied, or transposed. It is in these operations that the array processor method of training SVMs really shines. Figures 1 and 2 of [3] show the proposed architecture of MatRISC. This paper was more theoretical, so concrete results are not shown. However, it is worth nothing that specialized architectures for machine learning have continued to evolve over the years and this evolution will likely continue into the future (more on this in the next section).

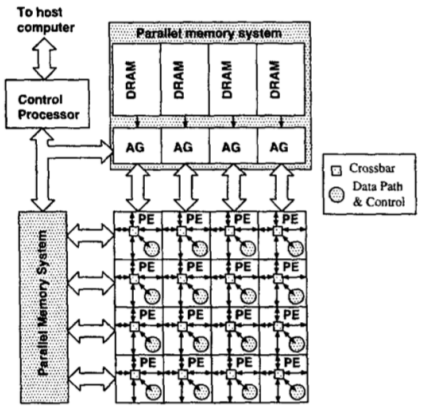


Figure 1: *Array architecture: Mesh topology of processing elements each with a crossbar switch.*

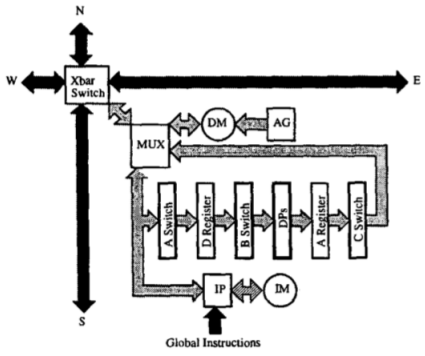


Figure 2: *Processing element data path: Data can route into or out of the data memory (DM), instruction memory (IM), or the data registers to external I/O through the crossbar switch.*

GPU Computing

As discussed in [1§6.1.2], GPUs are related to the SIMDs discussed in the previous section (and most closely related to array processors). The key distinction between a GPU and an array processor is that the array processor operates on a single thread whereas GPU is *highly* threaded. To dive deeper into the actual inner workings of GPUs, note that they contain several streaming multiprocessors (SMX). Each of which contain several stream processors (SP). Each SP can typically execute one instruction from a thread in a single clock cycle. Another major performance enhancement in GPUs is that threads are managed at the hardware level and are *much* faster than if they were handled at the software level. Given the highly threaded environment, many classify GPUs outside of Flynn's Taxonomy as *single instruction stream, multiple threads* (SIMT). It is worth noting that independent branching of threads is allowed, but for optimal performance, developers should use compilers that group together similar code wherever possible [1].

The highly threaded architecture of GPUs lends itself nicely to many problems in data science and artificial intelligence [4]. One such application of GPUs in practice is discussed in [5]. The paper focuses on multi-objective particle swarm optimization. Particle swarm optimizations are an effective tool, not only for maintaining information between particles with good solutions, but also for being able to search the entire solutions space while being resistant to converging on local-optima. The heart of [5] is in the data structure for coalescing parallelized particles on a GPU. Figure 3 below shows the method used:

Figure 3: There are N particles and each particle is represented by D dimensions.

With this data structure, a single memory transaction can be used after particles have searched the solution space, creating a much more efficient solution than what is done on CPUs.

GPUs have also been widely used in more traditional machine learning models such as artificial neural networks [4]. However, although GPUs can drastically reduce the training time associated with machine learning models when compared to CPUs, there have been continued efforts to produce specialized hardware for machine learning (as was discussed in the previous section). The most recent mainstream development came from Google where engineers have developed a processor called a TPU (*tensor processing unit*) [6]. An architectural overview and performance analysis of Google's TPU in a data center was discussed in [7]. A block diagram of a TPU is shown in figure 4.

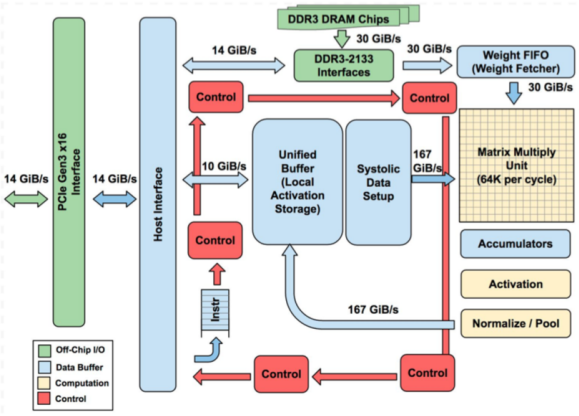


Figure 4 TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit in the upper right hand corner. Its inputs are the blue Weight FIFO and the blue Unified Buffer (UB) and its output is the blue Accumulators (Acc). The yellow Activation Unit performs the nonlinear functions on the Acc, which go to the UB.

The authors of [7] also point out that a TPU might be considered more of a floating-point unit (FPU) coprocessor than a GPU because of how instructions are dealt with. When compared to an Intel Haswell CPU and an Nvidia K80 GPU, the TPU was, on average, 15X-30X faster in terms of raw performance on neural network training. However, when accounting for energy savings, TPUs are about 30X-80X better (in terms of *TerraOps per second per Watt).*

Multiprocessor Systems

From a high level, the concept of multiprocessor systems seems pretty straightforward. Instead of a system containing a single CPU, it might contain two, eight, or even hundreds of

CPUs. Though, of course, there are many nuances. One of the largest performance considerations discussed in [1] for multiprocessor systems is that of *cache coherence*. Cache coherence has to do with the problem of making sure all the cache memories in a multiprocessor system reflect the most up-to-date memory. The idea of single cache strategies was discussed in [1§2.4], but the problem is much more difficult in multiprocessor systems. There are several protocols for dealing with cache coherence that cover a range of complexities in multiprocessor systems. The simplest approach is a *snoopy* protocol which essentially monitors a system's bus activity to determine which caches to update and when. There are also various *directory* approaches that hold information associated with cache memories. The most complete directory approach is called a *full-map directory*. However, this can sometimes be too large to implement and variations are needed. A couple variations described in [1] are *centralized directory* protocols and *distributed directory* protocols. Lastly, both directory-based and snoopy cache coherence calls require *serialization* (which is just a way of enforcing write operations behave correctly) if multiple writes are made to memory. The final consideration discussed in [1] is about ensuring synchronization of the different processors. The main idea here is that no processor should read (or do anything with) data that is being modified by another processor until the processor modifying the data is finished. The best way to deal with this issue is for a processor to *lock* a shared resource until it is safe for other processors to access the resource. Note that this is only one type of MIMD architecture and another type of MIMD architecture will be described in the next section.

By now, it should be clear that SIMD (or SIMT) systems have drastically improved the fields of artificial intelligence and data science for scientists and researchers. But how have MIMD architectures advanced these fields over the years? While A.I. was still relatively new and in its infancy (in terms of practical use cases), the authors of [8] discuss how to design multiprocessor systems specifically for A.I. Their survey breaks down the problem of A.I. computer architecture into three components: 1) the representation level or how to effectively acquire and learn new domain knowledge, 2) the control level deals with actually managing the

parallelism and control of the algorithms, and 3) the processor level actually deals with the hardware required to represent knowledge. The interesting part of this paper is when the authors concluded that the optimal computer system would be optimized for the Lisp language because of its effective knowledge representation. Obviously, since 1989, the view on this has changed as has the entire field of artificial intelligence. The point of reviewing this article is to illustrate the fact that not everything in computer architecture (or computer science in general) will remain relevant through time. Although there was much interest in parallel architectures to speed up the process of A.I. knowledge representation and learning, the underlying problem must first be properly formulated to really see breakthroughs. In [8], A.I. was thought of as a way of "hard-coding" domain expertise in hopes of making intelligent machines. It has since been realized that this is not an effective way of representing knowledge due to the extreme subtleties in regard to intelligence.

Thus far, we've reviewed general methods of artificial intelligence, such as specialized SVM architectures, optimized architectures for training neural networks (TPUs), and even a proposed A.I. architecture that would never come to pass. How could we use A.I. and machine learning to improve the system itself? This is the focus of [9] where machine learning is used to predict thread profiles of heterogeneous multiprocessor systems (HMPs) to improve thread scheduling. It turns out that this is a much more difficult problem in HMPs because the nature of threads in different processors can vary widely. The authors' approach was to test several machine learning models such as linear regression and *support vector regression* (SVR is a variation of SVM models). The first step in [9] was to extract relevant features of threads and also calculate features that may be predictive of a thread's profile using domain expertise. Next, a feature selection method called *principal component analysis* (PCA) was used to determine the most predictive features. Finally, these features were fed into various machine learning models for training and testing. Tables 3 and 4 of [9] show that both linear regression and SVR models were more predictive of thread profiles, but SVR outperformed linear

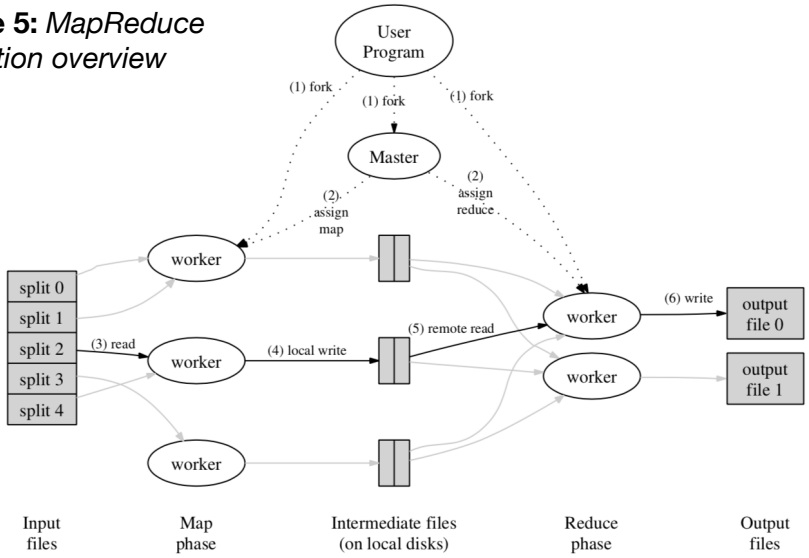
regression models. It is also noted in [9] that although SVR models take longer to train that linear regression models, the overhead of actually making a prediction with SVR is negligible.

Multicomputer Systems

The final MIMD architecture described in [1§6.1] is called a multicomputer system. There are many variations of multicomputer systems, but the most basic example is called a *cluster* where a group of computers with homogenous architectures are connected by some form of communications network. This is often in the form of Ethernet. A variation of this multicomputer system might be a *multi-multicomputer* system which refers to a system that contains multiple multiprocessor computers (discussed in the previous section).

Multicomputer systems have many advantages; primarily the ability to leverage the combined computing power of an arbitrary number of computers. In the context of machine learning, this, in theory, is extremely powerful. We have already mentioned how powerful GPUs and TPUs are, but what if we could use a distributed system with dozens of computers with GPUs or TPUs? The potential speedup in model training is huge [10]. In addition, there is likely not enough room in the memory of a single computer to hold all the available data in many research problems such as raw text, genomic data, or any other conceivable source of “big data”. The question is, how can we effectively distribute data processing between computers on a multicomputer system? The authors of [11] designed an algorithm called *MapReduce* which is perfectly suited to process large amounts of data in a distributed fashion. There are countless ways of processing distributed data with MapReduce (such as training machine learning models [12]), but simpler processes (such as sorting) are most common. The basic idea of MapReduce is to use a two-part algorithm: 1) map a list of key-value pairs from the raw data (which may be in JSON format), and 2) reduce the key-value pairs into similar objects. Figure 5 [11] below shows the conceptual framework of how MapReduce works on distributed systems. The exact process will change based on the use case, but MapReduce provides a powerful framework for distributed processing in data science applications.

Figure 5: *MapReduce execution overview*



Conclusion

We have discussed many topics related to parallel computing up to this point. First, we covered vector and array processors (SIMD). While vector processors have fallen out of style, array processors are used in conjunction with more general processors. It was in this section where we learned that the general architecture of array processors work well for many machine learning problems. In the second section, we moved to GPUs (SIMT), which are highly threaded processors that build from array processors. GPUs are used in many artificial intelligence applications today, despite being designed for graphics. We also learned that a more specialized A.I. processor called a TPU was designed by Google within the past couple of years. In the following section we moved to multiprocessor systems (MIMD) where we learned about a proposed A.I system that was never introduced. We also covered how machine learning can be used to improve a multiprocessor system. Finally, the last section reviewed multicomputer architectures (MIMD). We also covered how to leverage these architectures in distributed data processing with an important algorithm called MapReduce. Advances in artificial intelligence, machine learning, and data science are all heavily dependent on the advances in parallel computer architectures. We will likely see these fields grow together in the coming decades [13].

References

[1] Dumas, Joseph D. II. Computer Architecture: Fundamentals and Principles of Computer Design, Second Edition, CRC/Taylor & Francis, 2017. ISBN 978-1-4987-7271-6.

[2] Abdiansah, A., & Wardoyo, R. (2015). Time complexity analysis of support vector machines (SVM) in LibSVM. *International Journal of Computer Applications*, *128*(3).

[3] To, K., Lim, C., Beaumont-Smith, A., Liebelt, M., & Marwood, W. (1999). An array processor architecture for support vector learning. Knowledge-Based Intelligent Information Engineering Systems, 1999. Third International Conference (pp. 377–380). IEEE Publishing. doi:10.1109/KES.1999.820202

[4] Bleiweiss, A. (2008). GPU accelerated pathfinding. Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on graphics hardware (pp. 65–74). Eurographics Association.

[5] Zhou, Y., & Tan, Y. (2011). GPU-based parallel multi-objective particle swarm optimization. *International Journal of Artificial Intelligence*, *7*(A11), 125-141.

[6] Anonymous. (2016). Google unveil “AI” Chip. Linux Format, (214), 48–48. Retrieved from http://search.proquest.com/docview/1835604594/

[7] Jouppi, N., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., et al. (2017). In-Datacenter Performance Analysis of a Tensor Processing Unit. ACM SIGARCH Computer Architecture News, 45(2), 1–12. doi:10.1145/3140659.3080246

[8] Wah, B., & Li, G. (1989). A survey on the design of multiprocessing systems for artificial intelligence applications. Systems, Man and Cybernetics, IEEE Transactions on, 19(4), 667–692. doi:10.1109/21.35332

[9] Li, C., Petrucci, V., & Mosse, D. (2016). Predicting Thread Profiles across Core Types via Machine Learning on Heterogeneous Multiprocessors. Computing Systems Engineering (SBESC), 2016 VI Brazilian Symposium on (pp. 56–62). IEEE. doi:10.1109/SBESC.2016.017

[10] Wang, Lv, & Zheng, Z. (2014). CUDA on Hadoop: A Mixed Computing Framework for Massive Data Processing.

[11] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107–113. doi:10.1145/1327452.1327492

[12] Lakshmi, J., & Sheshasaayee, A. (2015). Machine learning approaches on map reduce for Big Data analytics. Green Computing and Internet of Things (ICGCIoT), 2015 International Conference on (pp. 480–484). IEEE. doi:10.1109/ICGCIoT.2015.7380512

[13] Bekkerman, R., Bilenko, M., & Langford, J. (Eds.). (2011). Scaling up machine learning: Parallel and distributed approaches. Cambridge University Press.