Data Parallel Architectures

Lecture #2: Thursday, April 3rd, 2003
Discussion Leaders: Nuwan Jayasena, Suzy Rivoire
Notetakers: Paul Wang Lee, Wajahat Qadeer

1. Introduction

Some applications, especially many multimedia applications, have a high degree of inherent data level parallelism (DLP). The two papers studied in this lecture describe two processor architectures that exploit this DLP to efficiently run these classes of applications.
Both the VIRAM and Imagine papers point out that superscalar architectures are poorly suited to the demands of multimedia applications. These applications tend to be computationally intensive and highly data parallel with little global temporal locality, while demanding good performance, low cost, and low power. Superscalar architectures, which expend a great deal of area and power in inferring parallelism and using reactive caches, are suboptimal for these applications; rather, both papers propose architectures with explicit support for data parallelism and a more proactively managed memory hierarchy.

2. VIRAM [1]

This paper addresses a mismatch between traditional superscalar and VLIW processors and the characteristics and demands of embedded multimedia applications. These highly data-parallel applications demand low complexity, low cost, and low power consumption from hardware. However, superscalar and VLIW architectures achieve their performance from high clock frequencies and complicated control structures to infer parallelism, both of which contribute to high power usage and high cost. The authors propose a multimedia vector architecture, VIRAM, which explicitly exploits data parallelism rather than having to infer it, resulting in a high number of operations per instruction and low control complexity.

The paper's contributions are the VIRAM architecture and compiler themselves, plus an excellent evaluation of the performance, cost, and power of VIRAM and other architectures for embedded multimedia applications. VIRAM's innovations for adapting vector architectures to this domain include a large number of vector registers, support for several narrow data widths, permutation instructions to communicate across lanes without going to memory, and paged virtual addressing. VIRAM also introduces a more effective memory hierarchy than traditional reactive caches, which work poorly on multimedia applications with little global data reuse. VIRAM's combination of a vector

register file and on-chip memories are more suited to the needs of multimedia applications.

The strengths of the paper are its initial idea and the rigorous comparison of VIRAM to other architectures for all the relevant parameters. Analyses of vector length and the degree of vectorization in compiled programs help validate the effectiveness of their compiler. The description of the VIRAM architecture and compiler is not detailed enough for any weaknesses in their design to be apparent from this paper. One minor quibble with their evaluation is that, because the EEMBC benchmark is inherently biased toward architectures with caches, they never provide a truly fair comparison between the cache-less VIRAM and other architectures. An additional graph showing EEMBC run 10 times with 10 different data sets would provide a less biased point of comparison that could only be more favorable to VIRAM.

Natural extensions and improvements to this work might involve improving scalability by partitioning the functional units into clusters and explicitly scheduling communication between these clusters; this could be coupled with FU-local registers to create a bandwidth hierarchy. It would also be interesting to evaluate VIRAM's performance on other application domains with high data parallelism, such as scientific computing. A final open question is how to configure a polymorphic architecture to look like VIRAM.

3. Imagine [2]

This paper introduces the *stream programming model* and a specific implementation of a *stream processor* (Imagine). Applications developed for this system are shown to sustain levels of performance comparable to special-purpose embedded processors.

Media applications such as signal processing, graphics, and audio and video compression demand high levels of performance and exhibit large amounts of parallelism that lend well to efficient custom implementations. However, these applications run inefficiently on conventional microprocessors since their memory access patterns, which typically exhibit little reuse, are a poor match to cache memory hierarchies.

The stream programming model expresses a computation as a set of *kernels* that operate on *streams*. A stream is a sequence homogeneous records. A kernel specifies a set of operations to be applied to every element of its input stream(s). A stream program specifies a set of kernels and orchestrates the flow of data streams among them. This model exposes the data communication at stream granularity as well as the data parallelism among operations applied to elements of a stream.

Imagine is a coprocessor that executes applications expressed using the stream programming model. It consists of 48 floating-point ALUs organized into 8 identical *clusters*. A microcontroller issues instructions that control all 8 clusters in SIMD fashion, with VLIW control of multiple units within clusters.

Each ALU is fed by a pair of dedicated *local register files* (LRF) with a high-bandwidth interconnect that allows results from any ALU to be written to any LRF within the cluster. The LRF's provide a total peak bandwidth of 544 Gbytes/s in the 8 clusters at 500 MHz. These LRF's form the highest-bandwidth level of a hierarchy optimized to capture data locality. The next level of the hierarchy is a 128 KB *stream register file* (SRF) that provides 32 GB/s peak, and captures inter-kernel stream locality. Finally, the off-chip memory provides a peak of 2.67 GB/s.

On 4 media application benchmarks, Imagine sustains 5.1 to 18.3 GOPs and 1 to 2 orders of magnitude bandwidth filtration at each level of the hierarchy. Power consumption is estimated to be 2.2 W to 3.6 W for the same benchmarks.

A key advantage of the stream programming model is that it explicitly expresses stream communication and data-level parallelism. However, in order to take advantage of this, applications need to be recoded in special dialects of C (*StreamC* and *KernelC*).

Imagine leverages data parallelism to tolerate DRAM access latencies and achieve high performance with low power consumption. Unlike traditional data-parallel architectures, Imagine has two levels of storage (LRF and SRF) under software control to try to capture a higher degree of data reuse and reduce memory traffic.

4. Discussion

The discussion started off with the difference between stream and vector architectures:

- Vector Architecture: each instruction is executed on all data elements of a vector before the next instruction is executed
- Stream Architecture: a set of instructions (kernel) is executed at a time on each data record of a stream

There was some discussion about caches. The two implementations do not use caches, but caches may be useful between DRAM and register file, since there is occasionally some temporal locality in the data, as in cases such as dictionaries in speech recognition or textures in graphics applications. One comment with respect to this was that using the space for caches to increase the size of the register file may be better, and this fact was possibly taken into account when designing the stream register file for Imagine.

An interesting way of seeing vector instructions was brought up, that a vector instruction is actually a collection of operations on many independent data, and can be thought of as many independent instructions. Since it is known that these instructions are independent, no analyses need to be done to find whether they can be executed in parallel, as would be needed for a collection of scalar instructions.

There was a question asking whether more complex register allocation algorithms were required for stream/vector processors as opposed to scalar processors. The answer to this

was that register traffic is more deterministic, so this reduces the difficulty of the register allocation problem.

Also, in these architectures, memory accesses are regular, so static scheduling is possible. Moreover, the instruction set enables the SW to communicate to the HW information such as memory access patterns (unit step, strided access, etc.) enabling more efficient memory accesses.

One issue with data parallel architectures is with small stream/vectors. Short streams are possible, but the overhead per data element will become significant for shorter streams. The same effect applies to vector architectures as well, although it may be smaller.

Conditional execution:
'For' loops are good for vectorization. But what about 'while' loops? Speculative vectorization is one way to vectorize while loops. The loop is vectorized for a sufficient number of iterations and you have some check whether you will exit or not.
In Imagine, conditional streams can handle these conditionals.

An example to demonstrate some issues in the vectorization of conditionals:

```
for(int i=0;i<k;i++)
{
        // This loop can be easily vectorized
        if(a[i]) b[i] = 2;
        else     b[i] = 3;
}

for(int i=0;i<k;i++)
{
        // These loops are not good for vector processors
        if(a[i]) b[i] = foo(a[i]) ;
        else     b[i] = bar(a[i]);
}
```

As the above example codes illustrate, some control dependencies do not hinder the exploitation of data parallelism, while others pose more of a problem. The first example code can be vectorized easily using a number of techniques such as predication, but the second case is more difficult since the two functions may be very different. Inlining can enable some methods if the inlined functions are short, but it is much more difficult if this is not the case. Conditional streams in Imagine can handle this case more easily.

One observation of Imagine was that it has a very large number of processing elements to feed, and there was a question on how high utilization can be reached given this.
The data parallel nature of stream processors enable high degrees of utilization for highly data parallel applications. Since Imagine clusters have multiple execution units, it can also exploit some degree of ILP.

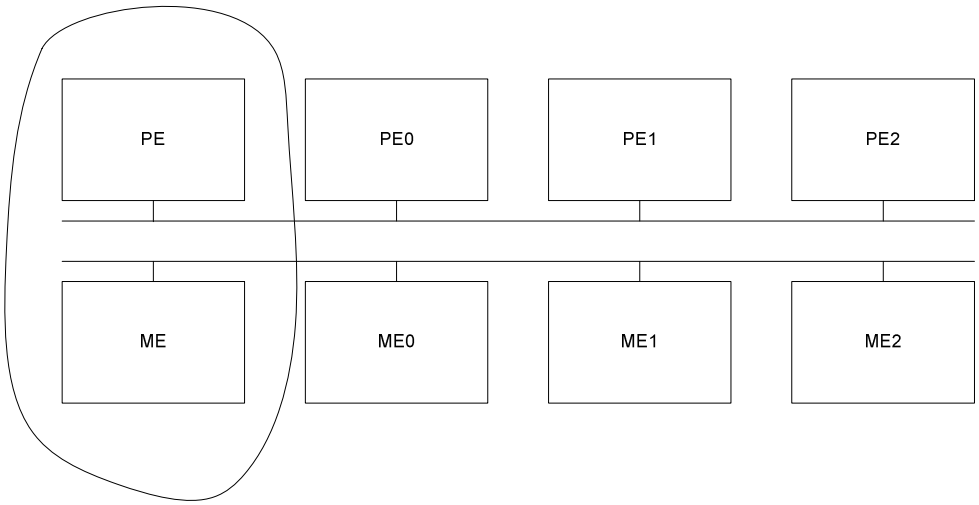
Scaling. How do you scale, and when do you stop scaling?
For vector processors, scaling is accomplished by increasing the number of lanes. In Imagine, there are two dimensions of scaling, number of clusters and number of ALU's in each cluster. The utility of scaling will be limited by the degree of data parallelism available in an application. For the vector processor you scale up to a certain point and when you start hitting the vector length you stop.

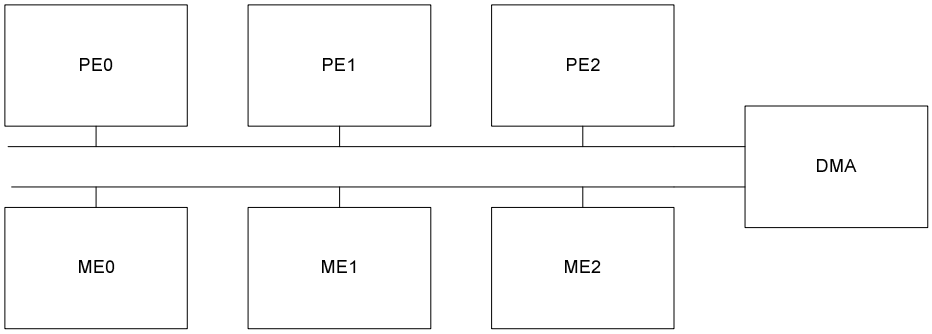
To make effective use of data parallelism, a good programming model and a high bandwidth memory system is important.

Are DLP and ILP mutually exclusive?
They can be exploited simultaneously. DLP can also be converted to ILP and TLP. After DLP is exploited using a data parallel architecture, ILP can also be exploited, e.g. using structures such as in Imagine.

How to exploit DLP with a CMP with many simple cores:
We can run processors with the same program on partitioned data. This is called "single program multiple data". Here, memory accesses pose a problem, and some synchronization is also necessary.

Merging memory accesses in ways similar to vectors may be beneficial in this case, since this will reduce the memory bandwidth requirement. For example, if N processors all issue a Load in a data-parallel execution, the loads could possibly merged to fewer loads to the memory system. One way to do this is have some cores set aside to manage the memory traffic, but another way that is more efficient would be using a DMA engine for these functions.





Data parallel execution requires some form of synchronization between PE's and this will require global wires. This presents a limit to the degree of exploitable DLP with this approach.

Having multiple cores present verification problems. In the hardware there are only a small number of types of modules and verification is likely to be easy. However, programming this is a different matter and verifying the compilation is a major problem.

There was some discussion on the specific implementations of the two systems, and a point was made that the properties of a specific implementation do not necessarily stem from the architecture.

Precise exceptions in vector processors:
Precise exceptions are important for arithmetic and virtual memory.
Exceptions should be restartable. Imprecise exceptions with recovery constructs similar to future files etc. do make this possible, but precise exceptions are easier to deal with for the OS programmer, since the interface is simpler and well defined.

There were some clarifications about the performance numbers in VIRAM.
A comment was made that it is good practice to provide two numbers, absolute performance in terms of time, and performance with respect to cycles.

A comment on general paper writing:
Comparisons are often unfair, and there are always many underlying assumptions, but it is always better to state assumptions that may be unfair as there is no way that the results can be reproduced.

A comment on architecture studies:
Simulations are inaccurate so small numbers may actually be random results.

References

[1] C. Kozyrakis, D. Patterson, "Vector vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks", *Proceedings of the 35th International Symposium on Microarchitecture*, Istanbul, Turkey, November 2002

[2] B. Khailany, W. Dally, S. Rixner, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, “Imagine: Media Processing with Streams”, *IEEE Micro*, Volume: 21, Issue: 2, March 2001