A Multithreaded Vector Co-processor

Bernard Goossens
IBP-LITP, Université Paris 7 Denis Diderot, 2 place Jussieu
75251 Paris cedex 05, France
email: bg@litp.ibp.fr

Abstract. A multithreaded vector co-processor design is described. It is intended to be placed with its private vector memory, on an expansion board, linked to the scalar processor and its cache-based memory hierarchy. The vector co-processor can run up to 8 vector tasks (threads) in parallel. Vector registers can be accessed either as independent sets of scalar values or as array sets. The Tomasulo’s algorithm, simplified to keep the issue and termination logics simple in a multithreaded context, dynamically schedules the dependent instructions. A locking feature is provided to handle both the reductions and the complex recurrences in a vector form.

1 Introduction

In the 70’s and 80’s, vector supercomputers like the Cray were designed to handle data parallel computations more efficiently than on standard hardware. However, these architectures were very costly ones and in the 90’s, commercial superscalar microprocessors have become very serious competitors for vector machines. Not that these new processors are as fast as their contemporaries supercomputers, but they do are as fast as 5 years old ones and more than ten times less costly. So the question is: has the superscalar technology rung the knell of vector computation? A key to the answer is to understand what essentially differentiates the two approaches. Vector hardware is designed to pull data from memory as smoothly as possible, fill the computing units pipelines and send results back to memory, whereas superscalar hardware is designed to provide enough scalar instructions to keep the issue unit busy.

To pull data smoothly from memory, a uniform memory access (UMA) structure must be employed rather than a hierarchical one based on caches. A large amount or all the data must be resident, so the memory size must be large. Then, the memory access time will also be large and to provide enough memory bandwidth to the computing units, first many parallel and interleaved accesses must be provided and second, vector registers are necessary to reduce the memory traffic. From the vector registers access time, the rate of the machine is derived. They provide operands to the computing units every cycle, so each computing unit that takes more than one cycle to deliver its result must be pipelined. Because read-after-write dependencies between vector instructions can be simply handled by chaining, no single scalar result by-pass from the pipelines outputs to their inputs has to be implemented. The last step of the design is to fix the
vector machine language, with register oriented vector instructions. These multi-cycles instructions leave the processor fetch and decode units rather unbusy so that pre-fetching, branch prediction and superscalar issue hardware is irrelevant for vector code.

To adapt a superscalar design to high vector performance, we start from the instructions, which are scalar ones, and their issue logic. The cache-based memory structure is already imposed by the scalar computations that the processor must run efficiently. Data parallel computation is exhibited by the compiler through loop unrolling and software pipelining. From such prepared codes analysis, the computing units set is enhanced, extending more particularly the data cache ports set. The multiple instructions issue logic fixes the machine rate. As in the vector machine, any computing unit that cannot deliver its result in a single cycle is pipelined. Pipelines by-passes are essential to scalar computations but not used when independent vector data are treated. Scalar instruction fetch, decoding and issue includes a lot of useless work when applied to a set of instructions equivalent to a vector one.

So, firstly, in a certain sense, a vector machine performs well on vector computations and does its best on scalar ones and a superscalar machine has the reverse features. But none of the two is the best in both computation styles. Secondly, in a vector machine, the more the vector registers, the larger the machine cycle whereas in a superscalar machine, the higher the superscalar degree, the larger the machine cycle.

In this paper, we propose to separate vector computations from scalar ones (i.e. processes that do not manipulate vectors; the so-called non vectorizable part of vector code is taken in charge by the vector co-processor in a way explained at the end of section 3). We justify this separation by the fact that, as noted above, vector and scalar computations require rather different hardwares for memory structure, computing units and processor control.

Whatever the choice, superscalar or vector processor, data parallelism exploitation suffers from the non uniform distribution of the resources utilization. This is best illustrated by memory addresses conflicts leading to banks conflicts in a vector processor and to cache set conflicts in a superscalar one. In both cases, the full available bandwidth is not used. A similar situation arises when a functional unit has to handle multiple vector requests. Other units may stay idle while the requested one has to serialize all the computations. Duplicating functional units and memory banks or cache sets leads to unfrequently busy hardware which is bad both for cost and efficiency. Recent works on multithreading [7] [3] have shown that simultaneous multithreading (SMT) may somewhat enhance the distribution uniformity of the requests by sharing the resources and allocating them to multiple concurrent threads running in parallel. In the vector co-processor context, on each cycle one memory request per bank may be selected from different threads, for a full bandwidth filling. The same policy may also be applied to functional units allocation.

Some vector co-processors have been designed in the past, mainly as extensions to existing scalar machines. The IBM 3090VF [5] is a special unit taking