Packed Arithmetic – Architectural Influence on Compilation

John Lumley
Appliance Computing Department
HP Laboratories Bristol
HPL-97-36
February, 1997

packed arithmetic, compilation

Packed arithmetic instructions attempt to harness the unused power within 32 and 64 bit ALUs to speed principally multimedia application codes. Additional hardware costs are generally modest, partly because only very specific instructions focused on the needs of a very few algorithms are added. Using such features by systematic compilation rather than assembly coding reveals some problems resulting from architectural design choices, mainly with modality and supporting control flow.

IEE Colloquium on *"Multimedia Instruction Sets: Design and Application,"* Birmingham, United Kingdom, February 25, 1997

1 Packed Arithmetic

Packed arithmetic instructions have been developed for recent general-purpose microprocessors because: i) some critical media processing applications (noteably MPEG) have voracious computing requirements, ii) these algorithms use small or low-precision data types normally arranged in arrays, iii) there is excess ALU capacity for such types within 32 and 64bit machines and iv) useful SIMD instructions can be implemented cheaply on existing hardware. Initial exploitation has been confined to hand-crafted assembly routine libraries[1]. Using this power more generally implies some form of compilation: initial investigations suggest some annoying mismatches. This paper examines the problem, noting where different architectural features or specific instruction repertoires make (automated) software development easier or more effective.

The main packed-arithmetic instruction repertoires show their different design centres. HP's MAX[2]originallyfocussed on the DCT within MPEG, Sun's VIS™ [3] has extensive features aimed at graphics rendering (particularly with RGBalpha pixels packed into a single word), whilst Intel's MMX™ [4] is rather more general-purpose but smaller. MicroUnity's Mediaprocessor family [5] has been designed more specifically for media applications. All are implemented as SIMD operations on a designated register file.

2 Compiling for Packed Arithmetic

Given these architectural features and application code (probably in C) whose basic structure should be able to exploit the SIMD parallelism provided, our aim should be to use all the available (ALU) power. If the application is dominated by serious loop-based computational hotspots with simple arithmetic and data can be packed four-at-a-time into a word, then of course we'd like to run upto four times faster. My goal is to try to get close to such an n-fold performance improvement by a reduction in executed instruction count, treating as secondary other effects, such as memory/cache interaction.

How do we approach compilation ? We could introduce the new constructs at two levels: the code-generator stage where dependencies and operations are typically held in DAGs at basic- and super-block granularity, or at a high level where loop-structures can be attacked. Here I'm going to consider the latter, mainly because promising important applications are often written as array-traversing loops and useful information, particularly for parallelism, is available explicitly at this level.

A most promising, but not exclusive, approach is to 'stripmine' these loops, i.e. carry out n loop iterations at a time, with all appropriate data packed in individual 'stripes'. We get the data into the appropriate packing and alignment and hold it there as long as possible. This makes sense if the loops have effective trip counts large enough to make start-up and shutdown costs a minor consideration and the amount of computation (as opposed to memory traffic) is relatively high. With such a loop we must consider the following about the correctness and efficiency of the stripmine:

- Is the loop strictly parallel (no iteration-borne dependencies) or can it be transformed to be so ?
- How densely can the data be packed ?
- Is the data properly arranged and aligned to apply SIMD processing and if not, can it be made to be so (cheaply) ?
- In the loop body can we cover all necessary operations with our SIMD instruction repertoire ?
- Can we accomodate any control-flow variation within the loop body ?
- What additional data manipulation and mode-switching operations need to be added into the code ?

The first of these is a property of the algorithm and its application context - the actual packed instruction repertoire (normally) has no bearing on this problem. The others are all influenced to some degree by the implemented instructions. Of these it appears that the most crucial for correctness/performance are the coverage of the SIMD repertoire, modality of operation and mechanisms for handling control flow. We need however to look for cases where a systematic application strategy, exploited by a compiler, can yield high performance.

3 Instruction Coverage

To be efficient, all operations within the body of a stripmined loop need to be supported for packed data. For compilation we need 'basic' constructs supported simply in the instruction set. Modular addition and subtraction is available as are most of the logical bitwise operations (and/or, shifts etc.). For normal arithmetic operations this seems to suffice. The absence of multiplication in some processors can sometimes be a problem, though synthesis of (constant) multiplication is an option. There can be drawbacks when, for hardware reasons and a very specific design centre (such as MPEG), an instruction is only supported at a particular width, and not others. One operation unsupported, for obvious reasons, is table lookup. Also important is the repertoire of data mixing and rearrangement instructions provided. These can be needed both to set up initial parallel data streams and support limited interaction across streams, such as in: a[i] = (b[i]+b[i+1])/2; In practice compilation will involve some standard mappings between abstract operations and packed equivalents or short-range instruction constructs.

4 Modality

Whether or not the packed arithmetic features have to be used in a modal fashion may be the most influential factor that the architecture has on compiled efficiency. MMX use the floating-point register file to hold the packed data - whilst giving adequate space for 64bit words it suffers from a potentially expensive 'context switch' involving storing FPU 'status' when packed arithmetic is used. This may cause problems for those cases in graphics where mixed-mode work is needed. VIS also holds packed data in the floating-point register file leaving the integer register file free. MAX operates solely within the integer register file, allowing much simpler mixing with

other operations and clean register allocation. Whether the resulting additional register pressures causes performance problems remains to be seen.

With amodal operation of packed arithmetic very little special scheduling needs to be carried out for these instructions and they can be processed at the compiler back-end along with all other normal code. With modal facilities, ameliorating the costs of such context switching typically means code must be moved extensively to bunch similar sections together and surround them with appropriate context switches - this would have to operate at a a fairly high level and is unlikely to be easy and effective with current compilers.

5 Control flow

Most loops that have interesting computing behaviour exhibit some form of control-flow variation. Clearly, in the search for performance, loop invariant control flow should be removed outside the loop by simple inversion, or in the case of procedural invocation, by some form of inlining. The real problem is to handle loop-variant control flow, ie. the sort implied by

```
if(a[i] < 45)

      b[i] = c[i] + 19;
```

With current architectures this turns out to be tricky to implement when we're using SIMD instructions to process n-at-a-time. Conventional single condition codes (zero, lt etc.) are meaningless for the results of these operations and different control mechanisms need to be synthesised. Since separating the control flow for each element in a packed word would be exceptionally expensive in hardware we must either split the control flow through software (i.e. breaking the parallelism for the code concerned) which defeats the original goal, or attempt to map into some equivalent data flow with (cheap) hardware support. Some options are to include some form of element-by-element predication on the SIMD operation, or providing generation of conditional 'mask words' which can be logically combined (in software) to produce the required data flow. The latter seems to be by far the cheapest to implement in hardware and is what is used in MMX. So the above code is basically mapped into dataflow as:

```
temp  = (a[i] < 45) ? all_1s: 0;
temp1 = (c[i] + 19) &   temp;
temp2 =  b[i]       & (~temp);
b[i]  =  temp1 | temp2;
```

which can then be stripmined n-at-a-time, albeit at the cost of three extra logical operations per assignment within the execution loop (assuming an and-complement operator). This substitution can be carried out easily and systematically by a compiler and can be subject to late-stage optimisation of the resulting dataflow. More drastic control flow variation, such as abnormal loop exit, is not susceptible to such a simple technique, but may be less important in practical situations.

In VIS the comparison operations are oriented to determining graphics clipping conditions, resulting in single bit conditions and would need a different approach (there are arguments for both modes of comparison result.). In some specific small-scale arithmetic cases, such as maximisation, these control flow 'problems' can be overcome by careful use of saturation modes (as in MAX-2 - see [2] for details), which are relatively cheap to build into the hardware, and can be surprisingly efficient in software, though they are less likely to be exploitable systematically. MicroUnity's Mediaprocessor appears to combine condition and selection into a single operation.

6 Conclusion

For systematic efficient exploitation of these packed arithmetic features via automated compilation, lack of modality and support for limited 'control-as-data'-flow within the architecture seem to be important and potentially cheap in hardware.

7 References

[1] http://www.sun.com/sparc/vis/mediaLib.html

[2] R. Lee, *"Subword Parallelism with MAX-2"*, IEEE Micro Vol 16 No 4, Aug. 1996, pp51-60

[3] M. Tremblay et al, *"VIS Speeds New Media Processing"*, IEEE Micro Vol 16 No 4, Aug. 1996, pp10-20

[4] A. Peleg and U. Weiser, *"MMX Technology Extension to the Intel Architecture"*, IEEE Micro Vol 16 No 4, Aug. 1996, pp42-50

[5] C. Hansen, *"MicroUnity's MediaProcessor Architecture"*, IEEE Micro Vol 16 No 4, Aug. 1996, pp34-41