Comparison of Multicore Processors using Sourcery VSIPL++

Brooks Moses, Don McCoy, Justin Voo, Stefan Seefeld
CodeSourcery, Inc.
{brooks, don, justin, stefan}@codesourcery.com

Introduction

The current HPEC processor landscape contains a wide variety of processors, ranging from homogeneous multicore CPUs to heterogeneous manycore GPU-based systems. Due to the difficulty of porting low-level code from one type of processor to another, it is often important to choose a processor type early in the lifetime of a programming cycle. Likewise, because of the architectural differences, the real-world performance of these processors cannot be summed up in a single number such as floating-point operations per second (FLOPS); one processor may perform substantially better than another on one algorithm, while performing substantially worse on another. As a result, selecting an appropriate processor for a given application is not only important, but difficult.

CodeSourcery's experiences porting Sourcery VSIPL++ to a range of current architectures can provide a broad insight into the multidimensional characterization of processor performance. The VSIPL++ API contains over a hundred functions for signal, vector, and image processing, ranging from elementwise functions to matrix algebra, Fourier transforms, and filters, and the various Sourcery VSIPL++ products contain implementations of these for multicore x86 and Power, Cell/B.E., and NVIDIA CUDA processors, covering much of the available range of architecture types. Thus, we can show performance results covering a range of identical operations and different platforms, drawing quantitative conclusions about the comparative performance characteristics and how these relate to the characteristics of the operation being performed.

Processor Types

To a first approximation, multicore processor cores can be placed along a one-dimensional spectrum of size and complexity -- or, more precisely, a spectrum of the proportion of the core devoted to instruction processing in relation to arithmetic logic.

At one end of this spectrum are the larger x86 and Power CPU cores, which contain a vast quantity of transistors devoted to out-of-order scheduling, branch prediction, multiple-pipeline dispatch, and memory cache management. The arithmetic logic units make up a relatively small fraction of the processor, but the sophisticated infrastructure around it enables the processor to use them very efficiently on nearly arbitrary code and algorithms.

In the middle are cores such as the the SPUs on a Cell/B.E., where each core has a complete instruction scheduler, but it is very simple with in-order execution and largely software-controlled caching. These require a significant amount of additional sophistication in the code to make full use of the processor capabilities, which limits dynamic responsiveness and adds to the amount of code to be executed, but the tradeoff is that a higher quantity of arithmetic logic can be provided for a given size and power. Intel's Larrabee architecture fits into this category as well, as do Tilera's manycore processors.

At the other end of the spectrum are GPUs, where a single instruction execution unit is shared across an array of cores, such that all of the cores execute the same instructions in synchrony. This allows a very high proportion of the processor to be devoted to arithmetic logic, but this can only be effectively used for code and algorithms with appropriate parallelism. Very-wide SIMD processors such as ClearSpeed’s CSX700 likewise fall into this category, with a high proportion of arithmetic logic units to instruction execution, and a resulting requirement for large amounts of fine-grained parallelism in the algorithms.

The processors supported by Sourcery VSIPL++ that will be discussed in this presentation provide representative samples from each of these categories. We will discuss results from a current Intel x86 multicore processor illustrating the high-instruction-scheduling end of the spectrum, an IBM Cell/B.E. processor illustrating the middle range, and an NVIDIA Tesla GPU illustrating the high-arithmetic SIMD end of the spectrum.

Algorithms

The operations covered in the VSIPL++ API fall into three major categories: elementwise and reduction operations, linear algebraic operations such as matrix multiplications and solvers, and signal-processing operations such as FFTs, convolutions, and FIR and IIR filters.

Within these algorithms, there are a number of characteristics that affect the performance of the algorithm. The ratio of input elements to output elements range from reduction operators such as summation and averages where the output is a single element, to operations such as vector outer products where there are a large number of output elements for each input element.

For operations where the input and output sizes are comparable, the number and distribution of the input elements required to compute each output element vary, from elementwise operations where an output element is computed from a single input element, to convolutions and FIR filters where a small contiguous range of input elements are involved, to matrix products where a matrix row and column are involved, and to FFTs and solvers where every input element is required in the computation of each output element.

When operations involve inputs from multiple elements, they also differ in the structure of the cross-linking between computations. Although FFTs require all of the input elements for computing each output element, the computations occur in a pattern of local computation and global communication, and this pattern is the same for each element; thus, although the communication can be costly, spreading the algorithm across parallel computing elements is relatively trivial. Conversely, each element in an IIR filter requires the full results from each previous element in the sequence, and so the naive algorithm provides no opportunity for parallelism at all. Many of the algebraic solver algorithms are in the middle of these extremes; early parts of the computation can be performed in parallel, but the data dependencies require the final portions of the algorithm to be performed more sequentially.

Finally, the VSIPL++ API includes a number of operations computed on matrices viewed as sets of vectors, where the operation on each vector may involve complicated computation patterns, but each row (or column) of the matrix is independent. An example of this is the FFTM operator, which performs FFTs on each row or column of a matrix. This provides a second, more coarse-grained level of parallelism above any fine-grained parallelism in the underlying operation.

Results

In the presentation, we will show quantitative results for a number of different operations within the VSIPL++ API, over a range of significant data sizes. Some preliminary qualitative examples follow.

The elementwise arithmetic operations are trivially parallelizable, and one might expect the performance to be directly related to the amount of arithmetic processing available on the hardware – with the GPU and Cell processors showing a dramatic advantage. This turns out not to be the case in practice, as the operations are so cheap that the cost is largely driven by memory access rather than arithmetic. In particular, both the Cell and CUDA versions have a large-vector performance of only about a 2x-3x performance increase over a single-core x86 version. Differences between the two processors show up with shorter vectors – for a vector of 32k points, the Cell performance is somewhat lower than that of the x86, whereas the CUDA version achieves its full performance at around 1k-2k points.

The importance of memory access shows up in matrix multiplication as well. Matrix multiplication requires O(N) computations for each output element, and can be performed in a block manner so that many of those computations can read data from cache rather than main memory. With the large register array on the Cell/B.E. SPUs, this makes a much more significant difference than it does with CUDA; we found that matrix-multiply computations are 3x-4x faster on the Cell/B.E over a range of sizes. However, an 8-core x86 implementation makes a strong showing as well; the CUDA version is only 2x faster on complex data.

The FFTM computations, meanwhile, appear to use the high floating-point execution speed of the Cell/B.E. and CUDA to much better advantage. For cases with 1k- to 4k-point FFTs (such that each FFT fits easily on the local store of a single Cell/B.E. SPU), the Cell/B.E. version obtains a speedup of 9x-15x over the x86 version, and the CUDA version obtains a similar 6x-18x speedup.

Conclusions

The most apparent conclusion from the preliminary results is that there is no clear universal winner among the available HPEC processor types. Not only do the performance comparisons differ strongly across different operations, but in many cases the absolute performance is reasonably close – certainly less than the variation one might expect within a given processor family.

As expected, we show significant differences in relative performance not only across different operations, but across different sizes of the same operation. For smaller data sizes, the flexibility of the CPU cores is a clear benefit. Likewise, we can see that the non-CPU processors have a necessary granularity of parallelism to obtain good performance, and the requirement is much tighter on the GPU than on the Cell/B.E.

The results given here show comparable numbers for Cell/B.E. and CUDA operations, and show that these almost always perform better than our x86 system. We do not expect this trend to continue; at the time of this writing, the CUDA version of the Sourcery VSIPL++ library is not yet complete, and the functions that have been implemented are largely those that are well-suited to GPU computation. In the presentation, we anticipate showing results for operations where the x86 processor is a clear winner on a similarly large scale.

These results also show that the performance of a given processor can be strongly influenced by considerations such as the memory bandwidth and (for coprocessors) the cost of initiating a computation on the coprocessor. This adds further considerations to the question of selecting the optimal processor type.