Stream Processing in General-Purpose Processors

Jayanth Gummaraju and Mendel Rosenblum
Computer Systems Laboratory
Stanford University, Stanford, CA 94305

To date stream processing has been applied to a variety of special purpose hardware architectures including stream processors, DSP, and graphics engines. We believe that the stream processing programming paradigm will also be a win for general-purpose processors, for executing both applications that have been identified previously for streaming such as media processing, as well as for wider classes of general-purpose computation.

By encouraging programmers to think in a slightly different way, stream programming allows a simple compilation system to efficiently map the computation on to a general-purpose architecture without the advanced compiler techniques that have largely failed to deliver high performance. Furthermore, developing codes on existing architectures will form an evolutionary path to the deployment of new streaming architectures for general-purpose computing.

In the rest of this paper, we describe the essential characteristics of stream programming and how these characteristics can be efficiently mapped onto a modern general-purpose CPU.

Stream processing advocates a *gather*, *operate*, and *scatter* style of programming. First, the data is *gathered* into a stream from sequential, strided, or random memory locations. The data is then *operated* upon by one or more *kernels*, where each kernel comprises of several operations. Finally, the live data is *scattered* back to memory. In order to execute a stream program efficiently, the streams have to be blocked to fit in a local memory in such a way that producer-consumer locality is exploited – a stream produced by a kernel is immediately consumed by the next kernel. Stream processors accomplish this locality using Stream Register File (SRF), a compiler controlled addressable memory.

Stream processing may be a good fit for modern general-purpose processors for many reasons. First, several modern micro-architectural features directly map to stream programming paradigm. The large caches in modern general-purpose processors can be used to emulate the SRF for exploiting producer-consumer locality. The intermediate streams produced by kernels are consumed locally and therefore, not scattered back to memory. This reduces the memory bandwidth requirements considerably, directly targeting the memory bottleneck issue in general-purpose computing. Also, the streaming stages can be easily mapped onto hardware threads (for example, hyper-threads in Intel Pentium IV processors). A simple implementation for a hyper-threaded processor with two threads would be to have one thread perform all the memory accesses and the other thread perform all the computation. Hardware threads can thus be used to effectively overlap computation and memory operations. Second, stream processing enables the ALU units to execute at maximum efficiency in the *operate* stage. This is because the processor doesn't stall waiting on memory references. In essence, the memory references that cause load misses are pro-actively *prefetched* in the *gather* stage. Third, stream programming style greatly helps compilers with the traditionally very challenging task of scheduling the operation of hardware resources to maximize performance.

Compiling stream programs for general-purpose processors has several interesting implications. Stream programming simplifies a few compiler analyses and makes a few other redundant. Streams, by definition, don't alias each other and therefore, can be exempt from any *alias analysis.* By moving up all the memory references to the *gather* stage, software *prefetching* becomes redundant. The compiler can *software pipeline* at a larger granularity – at a stage level (gather, operate, and scatter stage), rather than at instruction level for better performance. The *cache efficiency* also improves if the compiler blocks the data records by taking into account the producer-consumer locality of streams. Furthermore, the *cache utilization* increases if the compiler creates a stream of only those fields of a record that are actually accessed by the kernels.

There are some important challenges in using a general-purpose processor for stream processing. First, SRF, or parts of it, can be evicted from cache due to conflict misses. This is because the SRF is emulated with a contiguous segment of main memory and the general-purpose hardware does not allow regions of memory to be pinned in cache. Second, in the presence of conditionals, and unknown stream lengths at compile time, blocking the streams efficiently to fit in cache is non-trivial. Third, an important concern in overlapping a computation thread and memory thread is that they would have to communicate with each other after the *gather* and *operate* stages. This overhead is exacerbated if streams are blocked with fewer elements in order to exploit producer-consumer locality,

We performed some simple experiments to test the effects of communication overhead on stream processing using a hyperthreaded Intel Pentium IV processor. Although the use of hyperthreading technology reduced the communication overhead significantly, we found that it could be substantial ($> 10\%$). It is encouraging, however, to note that our initial experiments with some simple micro-benchmarks show that in spite of these overheads, a program written in a streaming style and hand-compiled for Intel Pentium IV runs more than a factor of 1.5 faster than the same program written in a conventional style.

In conclusion, we believe that stream processing will become an important programming paradigm in general-purpose computing. Streaming applications ranging from media to scientific applications can potentially benefit from this approach. Looking forward, we could also envision improving the performance of traditional general-purpose applications by writing portions of them in streaming-style.