Project Presentation (2)

Lecture #18: Tuesday, 4th June 2002
Lecturer: Prof. Bill Dally
Scribe: Chaiyasit Manovit, John Kim
Reviewer: Mattan Erez

This was the last day of the class and we wrapped up the project presentations with the five remaining groups. A summary of the presentation is provided below. For more details of each project, please refer to the full reports and presentation slides posted on the class website at: http://cva.stanford.edu/ee482c/projects.html

1 On-Chip Support for ILP, DLP, and TLP in an Imagine-Like Stream Processor

This group looked at the three level of parallelism that can exist - ILP, DLP, and TLP, and the effects of exploiting the parallelism. They evaluated the cost involved with different configuration of the hardware in terms of area which includes the clusters, internal cluster switch, microcontroller, and cluster communication. The details of their cost model can be found in their report.

The group developed a StreamC/KernelC of a JPEG-like encoder to evaluate the performance of the different configurations and measure the ILP. From the ILP, the DLP and the TLP were extrapolated.

Kernel classification of their testing code based on how well they ran on each configuration:

1. Wimpy kernels - perform similarly over all configurations because of its low computation, short stream effect, etc.
2. Functional unit-limited kernels
3. Zero-communication kernels
4. Communication-limited kernels
5. Bigger kernels

The speedup seen from doubling the number of FUs on Imagine was 1.75x, which was reduced to 1.2x when the StreamC overhead was included. StreamC was not pipelined in the experiment and they believe pipelining would help.

A general conclusion was that, by increasing number of hardware units, we see diminishing return in the performance while the cost grows much faster, resulting in performance/cost drop after a certain point.

2 Compiling Brook to StreamC

With the difficulty involved in programming in StreamC, this group looked into compiling Brook, which is a high level stream language into StreamC. The benefit of Brook was very clear - it was architecture independent where as StreamC was specific to the Imagine processor and Brook was also much more easy to code (as was demonstrated by the homework assignments!)

The group, using the existing Brook meta-compiler, developed a basic compiler framework and generated some StreamC code from it. The application that they looked into to test their compiler was StreamMD, which is a molecular dynamics simulation of water. In addition, optimization of the code was investigated which included optimized self-product oeprator and general strip-mining. Self-product was used to compute interactions between molecule pairs and its straightforward implementation expanded the stream into $O(n^2)$ records before reduced back again to n. An optimization was done here by strip-mining the computation and combining the expansion and reduction into one kernel. The strip-mining was done in such a way that a base strip of molecules was fixed in the local storage (scratchpad memory) and interactions to other pair strips were then computed, also taking symmetry into account to avoid redundant computation. Another level of strip-mining was also applied, to efficiently utilize the bandwidth hierarchy.The optimal strip size was determined by profiling.

They were able to get some code converted from Brook to StreamC but was unable to verify their results since the original code, StreamMD, did not run properly in Brook because of a bug. But their project demonstrated that Brook to StreamC is possible and that stateless kernels are easy to convert.

3 Stream Cache Architectures for Irregular Stream Applications

This project investigated the usage of a Stream Cache and its benefit in a stream processor. Two different types of applications, application 1 committed results after all the computations and application 2 committed as soon as possible, were tested on four different architecture arrangements:

1. no cache (baseline model)
2. cache accessible from SRF
3. dedicated cache accessible from clusters
4. cache accessible from clusters and SRF

All cache models had 8 banks, which matched the memory and had 1 word per cache line. The SRF was 256K words and all caches were 32K words. All the different architectures were tested on a cycle-by-cycle performance simulator which modeled latencies,

throughputs and resource bottlenecks, but did not do the actual computation, i.e. it was not a functional simulator. The results in general, was that with a stream cache, they were observing better performance when the number of computations per record was relatively small (less than 4000 cycles). The speedup decreased as this number increased and eventually saturated at 1 where the program became computation-bounded. Application 2 had lower speedup compared to application 1 because it required synchronization across all nodes once every few strips, which broke software pipelining. For actual charts of their results, please see their report and presentation.

Based on their results, the group concluded that, in general, a stream cache can improve performance on an irregular stream application, upto a factor of 3.5 and is more useful when kernel performs little computation on each record, i.e. the application is memory-bounded. Each cache architecture had its own pros and cons and the optimal architecture depends on the application and data set.

4 Improving Unstructured Mesh Application Performance on Stream Architectures

This group looked into improving the performance of unstructures mesh application on stream architecture. An example of an unstructures mesh application was when nodes have some values that are updated based on neighbors' values but each node has different number of neighbors, which results in accesses that are not regular but still a lot of data reuse. The application studied here was a static mesh.

Currently, Imagine does not operate on these application in an efficient manner as it wastes a lot of room in the SRF. The group investigated the impact of adding a stream cache connected to the memory system. With the stream cache, in certain applications where it was memory limited, there was clearly some benefit from using a stream cache but as the application became computation limited, the cache did not provide much gain.

Another aspect of the Imagine processor that the group investigated was the SRF and the use of SRF indexing. There were several benefits of having SRF indexing - it would reduce the data replication in the SRF as well as reduce the bandwidth demand on the memory system. However, it comes at a cost as it would require extra hardware and also reduce the SRF bandwidth by indexing into it. Also, SRF bandwidth is further reduced due to bank conflicts. The group investigated different ways of reducing the bank conflicts and measured the different performance values. (please see their report for graphs and actual results)

They finally concluded that unstructured mesh have a lot of data reuse (3 reuses out of 4 accesses in the studied application) which can be effectively exploited by the usage of a cache. SRF indexing mechanism also helps improve the bandwidth with bank conflicts being handled by hashing and hardware arbitration.

5 Mapping Vector Codes to Streams

The motivation of this work came from the fact that vector processing has long been studied and there are already tools such as vectorizing compilers. The goal of this work was then to map vector codes, which are largely available, to stream processing with maximal resource utilization, minimal inter-cluster communications, and minimal memory bandwidth requirement, both between SRF-LRF and SRF-μCode.

While a better scheduling was obtained by using a larger record size, which is equivalent to loop unrolling, the total execution time was actually worse. This may seem counter intuitive, but it was due to the longer μCode consuming more SRF-μCode bandwidth as the record size increased. To alleviate the problem, the cost of long μCode can be amortized by reusing the same kernel or using larger data sets.

Computations can be done by either having all operations in one big kernel or making each operation a separate kernel. The experiment (without software pipelining in kernel scheduling) showed that for serial computation, i.e. no parallelism at operation level, small kernels with small record sizes yielded the best performance, whereas for non-serial (parallel) computation, bigger kernels always had better resource utilization and performance. However, with software pipelining, a single pipelined kernel seemed to perform best in all cases.

For larger dataflows, kernel fusion was applied to merge single-operation kernels into one kernel resulting in better performance because there were more operations to be scheduled. A single large kernel was also a better choice when the data size was comparable to SRF. However, the kernel size could not be arbitrarily large; it was limited by the LRF size and the number of streams per kernel.

The work was finally concluded as being, mostly, a confirmation of the intuition. Kernel fusion would help improve the mapping result, but it requires constraints satisfaction where a set of heuristics may be sufficient to solve the problem.