

Architecture Development In Stream Languages



Sepideh sheykholeslami+ sheykholeslami.se@gmail.com

Fateme ghaffarian+ f.ghaffarian@gmail.com

Name of the Presenter: sepide sheykholeslami

Abstract

Stream languages demands a platform integrated various functional modules and an increasing support of multiple standards. Stream architecture is able to solve the problem. However the application suited for typical stream architecture are limited..This paper provides an architecture development approach in stream languages.

Key words: Parallel processing, Stream processing, StreamIt, Brook

1. Introduction

Modern processors can be limited by communication rather than computation. However, communication or dependency information can often be difficult to determine at compile time and therefore limit parallelism. Communication and dependency information should be explicit within Brook to alleviate this problem.

There are several programming languages that describe how computation should be performed in streaming architectures. Some examples are StreamIT, Brook, and Opencl. We advocate a stream programming paradigm with separation of concerns between the tasks related to computation and communication. This separation of concerns decomposes the software into manageable and comprehensible parts where we can more easily identify and expose areas for performance improvement. Presently, we use a stream programming model that allows a programmer to explicitly define the data streams between computation kernels or from memory. Tasks such as data loads, stores, and alignment are assigned to dedicated mechanisms called stream units. Tasks related to the actual computation are grouped as a kernel and assigned to dedicated hardware mechanisms called datapaths. The datapath consists of functional units with a flexible interconnection network. The stream units make use of data prefetching and alignment techniques to move data elements ahead of the computation and arrange them in the order needed by the datapath. A computation kernel is a set of localized processor operations that are independent and self-contained. The processing in each kernel is regular or repetitive, which often comes in the form of a loop structure. These computation kernels can operate without global variables and without frequent external interactions with other kernels. Instead, the stream and other scalar values, which hold persistent state, are identified explicitly as variables in a data stream or as signals between kernels. The programmer defines a computation

kernel using a streaming data flow graph (sDFG) language. An sDFG consists of nodes, representing basic arithmetic and logical operations, and directed edges representing the dependency of one operation on the output of a previous operation. Each node is denoted by a descriptor, which specifies the following: input operands, the operation, the minimum precision of its output value, and the signedness of the output result. Kernel regularity, in turn, produces uniform memory access of stream data elements. Stream data appear to be sequential to the computation kernels even though data is usually scattered throughout memory. To express memory access or communication tasks, the programmer uses stream descriptors as an application programming interface (API) to express the shape and location of data in memory. Stream languages demands a platform integrated various functional modules and an increasing support of multiple standards. Stream architecture is able to solve the problem. However the application suited for typical stream architecture are limited.

This paper presents Architecture Development In Stream Languages. The remainder of this paper is organized as follows. Section 2 presents and describes Stream Processor architecture. Section 3 illustrates Evolution of Streaming Architecture. Section 4 discusses the Imagine architecture, Section 5 discusses the MaSa architecture. Section 6 discusses about an Adaptable Architecture for Mobile Streaming Applications. The last section summarizes the conclusions drawn in this paper.

2. Stream Processor architecture

Stream processors are programmable processors that are optimized for executing programs expressed using the stream programming model. A block diagram of a stream processor The stream processor operates as a coprocessor under the control of the host processor, which is often a standard general-purpose CPU. A stream program executing on the host processor orchestrates the sequence of kernels to be executed and then necessary transfer of input and output data streams between the stream processor and on-chip memory. Kernel execution takes place directly on the stream processor from instructions stored in the microcontroller. New kernels may be loaded into them microcontroller as needed, possibly under explicit control of the host processor. The host interface of the stream processor issues the commands received from the host to the appropriate units as resources become available, subject to dependencies among the commands. Arithmetic units of the stream processor are grouped in to n identical compute clusters. Each cluster consists of several functional units and associated registers. A block diagram of an example cluster organization. The local register files (LRFs) attached to each functional unit provide the input operands for that unit, and results are written to one or more of the LRFs via the intra cluster network. figure 1 shows the Block diagram of stream processor architecture.

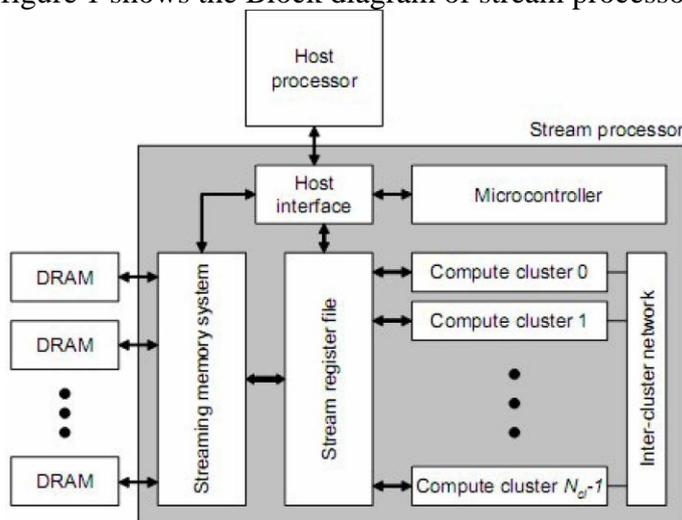


Figure 1: Block diagram of stream processor architecture

3. Evolution of Streaming Architecture

Multicore architectures offer a significant amount of coarse-grained parallelism on chip. They also increase the burden on programmers who now have to explicitly extract coarse-grained parallelism from their codes in order to leverage the compute potential available in emerging processors. Multimedia applications are especially challenging because they require stream abstractions that are not easily represented using current programming paradigms, and general purpose compilers eschew stream-aware optimizations. As a result, applications are hand-coded for performance, and this practice precludes portability and obfuscates readability.

4. Imagine Architecture

The Imagine architecture directly exploits the parallelism and locality exposed by the stream programming model to achieve high performance. As shown in Figure 2, Imagine runs as a coprocessor to a host. The host processor executes scalar code compiled from a StreamC program and issues stream instructions to Imagine via an on-chip stream controller. The stream register file (SRF), a large on-chip storage for streams, is the nexus of Imagine. All stream instructions operate on data in the SRF. Stream load and store instructions transfer streams of data between the SRF and memory (possibly using indirect (gather/scatter) addressing). Kernel-execute instructions perform a kernel on input streams of data from the SRF, and generating output streams in the SRF. The eight arithmetic clusters execute kernels in an eightwide SIMD manner. Each arithmetic cluster, contains six 32-bit floating-point units (FPUs) (three adders, two multipliers, and a divide square-root unit). Each FPU except for the divide square-root unit is fully pipelined to execute a single-precision floating-point, 32-bit integer, or multiple lower-precision integer operations per cycle.

These six FPUs execute kernel VLIW instructions issued from a single micro-controller each clock cycle. Two-port local register files (LRFs) feed the inputs of each FPU and an intra-cluster switch connects the outputs of the ALUs and external ports from the SRF to the inputs of the LRFs. In addition, a scratchpad (SP) unit is used for small indexed addressing operations within a cluster, and an intercluster communication (COMM) unit is used to exchange data between clusters.

When a kernel-execute instruction is received from the host processor, the micro-controller starts fetching and issuing VLIW instructions from the microcode instruction store. For each iteration of a typical kernel's inner loop, the eight clusters read eight subsequent elements in parallel from one or more input streams residing in the SRF, each cluster executes an identical series of VLIW instructions on stream elements, and the eight clusters then write eight output elements in parallel back to one or more output streams in the SRF. Kernels repeat this process for several loop iterations until all elements of the input stream have been read and operated on. In this manner, data-level parallelism is exploited across the eight clusters through SIMD execution and instruction-level parallelism is exploited with VLIW instructions per cluster. Locality within kernels is exploited during each loop iteration when intermediate results are passed through the intra-cluster switch and stored in the LRFs. Producer-consumer locality between kernels is captured by storing kernel output streams in the SRF and reading input streams from the SRF during subsequent kernels without going back to external memory. While kernel execution is ongoing, the host processor can concurrently issue stream load and store

instructions so that when the next kernel is ready to start, its input data is already available in the SRF.

4.1. Imagine Software System

The Imagine software system provides the compile-time and run-time support necessary for running stream programs. The software system includes compilers for converting StreamC and KernelC programs into host CPU assembly code and kernel microcode, respectively, and provides run-time support for issuing stream instructions to Imagine via the host CPU's external memory interface.

The KernelC compiler uses communication scheduling to produce VLIW microcode from KernelC. It performs high-level optimizations such as copy-propagation, loop unrolling and automatic software pipelining, schedules arithmetic operations on functional units, specifies the data movement between ALUs and LRFs, and performs register allocation. All of the kernel microcode for an application is loaded from the host CPU into the Imagine memory space during startup. At the start of application execution, the kernel microcode is transferred from Imagine memory to the microcode store (2K VLIW instructions total). If all the kernel microcode for an application does not fit in the microcode store, the host ensures that kernels are loaded dynamically from Imagine memory to the microcode store before kernel execution occurs. If new kernels are being loaded while another kernel is being executed, a performance degradation of less than 6% occurs.

The StreamC compilation process is split into two stages. In the first stage, a stream compiler performs a number of high-level tasks such as dependency analysis between kernels and stream load/stores, software pipelining between stream loads or stores and kernel operations, optimal sizing of stripmined streams, and allocating and managing the SRF. After these optimizations and analyses are completed, the stream compiler generates stream instructions (stream loads or stores, kernel invocations, and Imagine register reads and writes). These stream instructions are embedded in intermediate C++ code, which preserves the control flow of the StreamC. During the second stage, a standard C++ compiler compiles and links the intermediate code with a stream dispatcher, generating a host processor executable (assembly code). At run time, a command line interface is used to run a StreamC application of user's choice, which in turn invokes the stream dispatcher with relevant stream instructions. The stream dispatcher manages a 32-slot scoreboard on Imagine. When a slot is free, it issues a new stream instruction to be written into the scoreboard. The dependencies between the new stream instruction and other scoreboard entries are encoded with the instruction itself by the stream compiler. The stream controller on Imagine uses these dependencies to determine the next stream instruction to issue from the scoreboard when necessary resources become available. The dispatcher performs all reads/writes from/to Imagine registers, which are mapped to addresses in the host memory space. For data transfers between the host CPU and Imagine, the stream dispatcher performs memory mapped reads/writes to a on-chip fifo. For many media applications such as stereo depth extraction, the control-flow for the entire application is data-independent. In these cases, the StreamC compiler takes advantage of the static control-flow by using a playback mode, in which the intermediate C++ code is replaced by a record of the encoded stream instructions, in order. The playback dispatcher reads from this recorded sequence of stream instructions and dispatches them as scoreboard slots become free. Although less general than running application code on the host processor, this playback method allows for a more light-weight efficient dispatcher implementation when control-flow is data independent. Figure 3 Shows the Diagram of Imagine architecture with bandwidth hierarchy numbers.

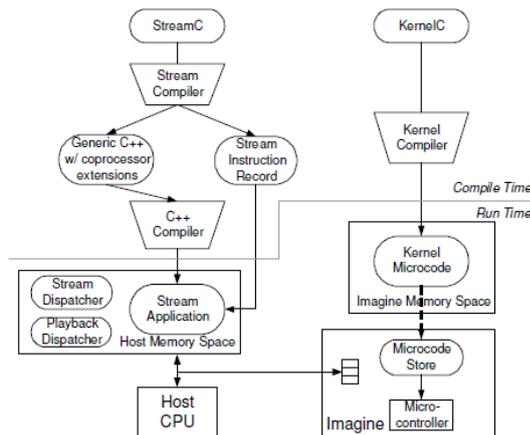


Figure2. Imagine software system

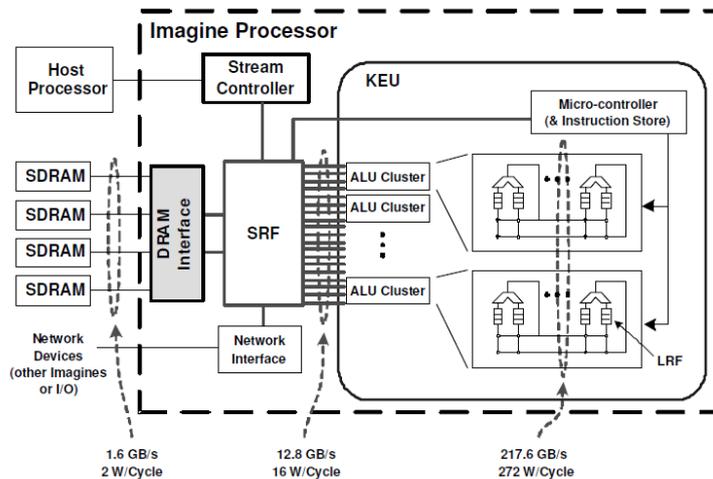


Figure 3. Diagram of Imagine architecture with bandwidth hierarchy numbers

5. Multiple Macro-Tile Stream Architecture

The scaling of conventional stream processor has reached the limit. Typical stream architecture, Imagine supporting ILP and DLP is consists of 8 clusters (4 ALU/cluster) while both the number of clusters and ALUs intra-cluster can be increased (two scaling dimensions: inter-cluster and intracluster). However prior research has shown that stream architecture like Imagine with 16 clusters (4 ALU/cluster) would achieve the best performance efficiency. The scaling of more than 64 ALUs will cause the decrease of the performance efficiency, and the downside will be more obvious as the number of ALUs increases.

The MasA architecture uses large, coarse-grained macro-tile to achieve high performance for stream applications with high computing intensity, and augments them with multiple execution pattern features that enable the tile to be subdivided for explicitly concurrent applications at different granularities. Contrary to conventional large-core designs with centralized components that are difficult to scale, the MaSA architecture is heavily partitioned to avoid large centralized structures and long wire runs. These partitioned computation and memory elements are connected by 2D onchip networks with multiple

virtual point-to-point communication channels that are exposed to software schedulers referred in section 5 for optimization. Figure 4 shows a diagram of the MaSA architecture, which consists of following major critical components.

Macro-Tile is a partition of computation elements. It consists of multiple stream cores and a network bridge. As shown in Figure 2, four stream cores compose a macro-tile. The organization is tightly coupled between stream cores in a tile while loosely coupled between tiles.

Stream Core adopting simplified classical stream processor architecture, is the basic computation module of MaSA. It is optimized for executing kernels of stream applications. Each stream core has its own instruction controllers—*Stream Controller* and *Micro controller*, and data storage—*Stream Register File* (SRF), and multiple arithmetic *clusters*. Clusters are controlled by the microcontroller in SIMD+VLIW pattern. A cluster is composed of a set of full-pipelined ALUs performing one multiply-add operation per cycle, and some non-ALU function units including 1 iterative unit to support operations like divide and square-root, 1 juke-box unit to support conditional streams, 1 COMM unit connected to an inter-cluster switch for data communication between clusters, and a group of local register files. Taking a 4 clusters x 4 ALUs configured stream core for example, the peak arithmetic performance of 32 floating-point operations per cycle can be achieved in a single core. In order to exploit DLP and ILP efficiently, the components of the stream core mentioned before are scaleable leading to varied configurations and at tradeoffs design time. Furthermore, in our future plan the stream core could be heterogeneous, even some special function unit or re-configurable circuit could be used. Besides this, there are some standard components in all types of cores including *Network Interface* (NI), *Host Interface* (HI) and *co-scalar core* (optional, run-time system software and few scalar code can be chosen to execute in it), which are necessary to guarantee uniform interface.

Memory/IO Core has multiple *Address Generators* (AG) or IO interfaces which are able to provide multiple DRAM or IO access channels in DMA mode without processor control. It is optimized for stream access by hardware memory schedule design.

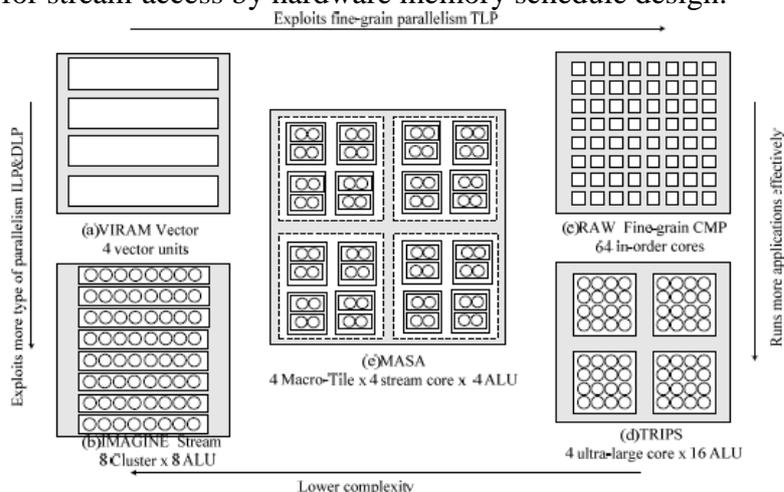


Figure 4 Granularity of parallel processing elements on a chip

6. An Adaptable Architecture for Mobile Streaming Applications

Many emerging mobile applications and services require playback of streaming media. In this section we describe an adaptable system architecture to implement mobile streaming

services. The main components of this architecture are the streaming server, the multicast proxy and the mobile client. The main novelty of our approach lies on the client which is designed to fit most mobile devices. The streaming servers and client are all compliant with 3GPP standards, and therefore use the Session Description Protocol (SDP), Real Time Streaming Protocol (RTSP), and Realtime Transport Protocol (RTP), as well as the Adaptive Multi-Rate (AMR) audio media standard.

The main purpose of approach is to facilitate the work of mobile applications developers by providing a set of reusable software elements. These elements can be easily adapted to different applications that require the streaming principle. To achieve this goal we conceived a generic architecture that can fit diverse mobile streaming applications. Fig.1 depicts the main components of this architecture. In frame A of Fig. 5 we show the components of a basic streaming system. In the simplest case a mobile client communicates only with a RTSP server to get a particular file. The media that can be streamed consists of AMR audio files. It is also possible to add a PHP server in order to store and administrate media files received from the clients. HTTP connection is then required to send files from the mobile phone to the server. Furthermore the PHP server can be used to provide security to the system by maintaining a list of authorized users, called members, into a MySQL database. In this case the user will require authenticating before been able to start a RTSP session.

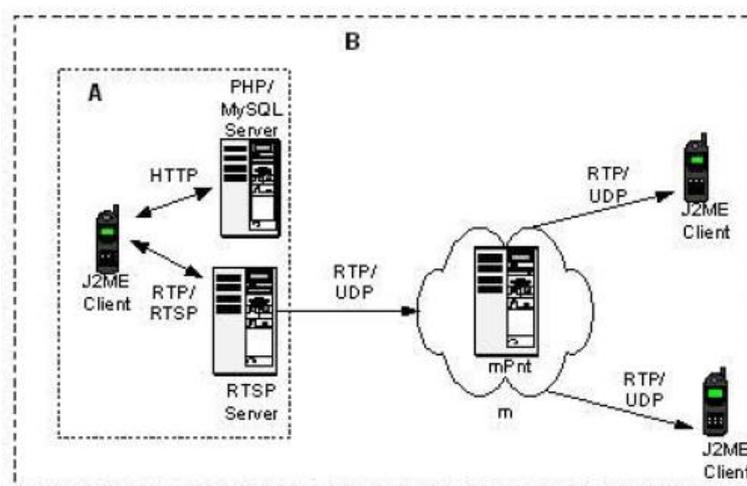


Figure. 5 Architecture components

7. conclusion and future works

In this paper we studied stream processor architecture. Stream processing has been shown to outperform mainstream programmable computing solutions while consuming less power for data parallel applications. Realizing the performance potential of stream processing, however, depends on the ability to manage bandwidth demands in the memory hierarchy to sustain the operands needed for highly parallel computation, we then discuss about Imagin architecture. the Imagine stream processor can efficiently support a wide variety of applications (ranging from molecular dynamics to video compression). Programming entirely in a high-level language, with no assembly performance tuning, Then we discuss about MasA architecture so we propose a programmable processor that provides multiple stream execution models and implement key parts for an h.264 encoder on the MASA simulator to investigate its performance. While MASA offers both high performance and flexibility that many media processing applications require, we expect it to replace ASICs in

the most demanding of media-processing applications. Our future work is to tune and evaluate the MASA architecture for additional applications and complete the custom design. Another challenge is to create an automatic partitioning and mapping tool assist the user, since kernel partition and stream organization are rather difficult in complex applications. At the end of the paper we discuss about an adaptable architecture for mobile streaming applications. We consider that the architecture presented can be adapted easily to fit specific needs of a streaming application that could be for example: a music sharing system, an electronic library system, a mobile learning system, among others. As future work we plan to add other streamed media like MPEG-4 video and MP3 audio. For this it will be only necessary to add the corresponding modules to the RTSP server and the J2ME client, without any modification on the other components.

References

AMIT KUMAR.(2008). Stream Computing: *In partial fulfillment for the award of the degree of bachelor of technology in computer science & engineering school of engineering cochin university of science and technology.*

Balazinska, M., Balakrishnan, H., Salz, J., & Stonebraker, M. (2004). The Medusa Distributed Stream-Processing System. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France.

H. Schulzrinne, A. Rao, R. Lanphier and M. Westerlund,.(2003) .“Real Time Streaming Protocol. RFC 2326.

Haiyan Li, Mei Wen, Nan Wu, Li Li, Chunyuan Zhang. (2005). Accelerated motion estimation of h.264 on Imagine stream processor, International conference on Image analysis and recognition, LNCS, Toronto

Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das(2003). Evaluating the Imagine Stream Architecture Computer Systems Laboratory Stanford University, Stanford, CA 94305, USA.

Mabel Vazquez-Briseno and Pierre Vincent.(2007). An Adaptable Architecture for Mobile Streaming Applications, GET-INT, RST Department, Evry, France / UABC, Ensenada, Mexico, MobKit, Lille, France.

Muchow, J. (200., Core J2ME Technology & MIDP, The SUN Microsystems Press, Prentice Hall, 1st. Edition, London.

Nan Wu, Mei Wen, Haiyan Li, Li Li, Chunyuan Zhang.(2005). A stream architecture supporting multiple stream execution models, Computer School, National University of Defense Technology Chang Sha, Hu Nan, P. R. of China.