A framework for stream programming on DSP

Changqing Xun, Wei Wu, Mei Wen, Chunyuan Zhang
Computer School, National University of Defense Technology
Chang Sha, Hu Nan, P. R. of China, 410073
xunchangqing@nudt.edu.cn

Abstract

There has recently been much interest in stream processing, both in industry (Cell, Storm series, NVIDIA G80, AMD FIRESTREAM) and academia (IMAGINE). Some researchers have accelerated a lot of applications in media processing, scientific computing and signal processing with a special programming style called stream programming. This paper presents a framework to program DSP with this special programming style. Stream program can run on DSP without any architectural support. H264 encoding is selected to evaluate our technique. The result shows that significant speedup is achieved, ranging from 3.2x for cavlc up to 7.1x for analysis.

1. Introduction

We can achieve very high performance through programming on stream processor. Some researchers have accelerated a lot of applications in media processing, scientific computing and signal processing, etc[1,2,3,4,5]. Typical stream architecture such as CELL [6], STORM series [7], NVIDIA G80 [8], AMD FIRESTREAM [9] and IMAGINE [10], usually contains a special unit for intensive computing. The key of its efficiency, the programmer is with responsibility for explicitly managing hardware properly. Stream programming model provides a high efficient way to make it. Stream programming model does not only work on stream processor, but general purpose processor [11]. Another research also shows that streaming at the programming model level is particularly beneficial, even with the cache-based model, as it enhances locality and creates opportunities for bandwidth optimizations [21].

With the development of technology, peak performance of Modern DSP (digital signal processor) such as TI TMS320C6000 [12] improves rapidly. Key application kernels, or even entire applications, are routinely programmed in assembly languages on DSPs to maximize the performance and minimize memory utilization. Existing DSP instruction set architectures (ISA) are developed on a vendor-specific basis, leading to incompatibility even between different product lines from the same vendor. Therefore, assembly code-level optimization is required to achieve maximum performance and conform to the proprietary hardware architectures. So programming with high level language is very important [19] which improve software design productivity significantly. High efficiency and easy-to-use language can support larger scale and portable applications.

This paper presents a stream programming framework on DSP, which actually includes a set of macro definitions. With this framework, programmers can write stream program and run it on DSP. Writing stream program with this framework is equal to optimizing the C program with traditional methods such as blocking and locality-aware scheduling, etc. But it is easier for programmers and more portable because of the hidden device specific. Our experiments on H264 coding show that the total number of instructions executed is reduced dramatically and functional unit utilization increase significantly. Our technique results in 5-6x mean speedup over common C program.

The rest of the paper is organized as follows. The stream programming model is described in section 2, especially the differences between it and C. Section 3 implements the framework for stream programming on DSP. Section 4 evaluates the performance provided by this technique. Section 5 discusses the related work and section 6 concludes.

2. Stream programming model

Stream programming models encourage programmers to think about the locality, data movement, and storage capacity issues in their applications [23] [24]. Typically memory accesses are decoupled from computation in it. This enables

...
software or compiler management of data locality and asynchronous communication with architecturally visible on-chip memories. Despite its affinity for stream architectures, the writers find that stream programming is beneficial for traditional architectures as well. Computation kernels performing mainly on-chip memory accesses could run very efficiently. Moreover, memory operations on a bulk of data reduce the memory access latency, and utilize memory bandwidth efficiently. This paper call computation kernels bulk operation and memory operations bulk transfers [15]. These conceptions are ubiquitous in various stream programming model, such as cudaMemcp and device function in CUDA [13], stream Read & stream Write in brook+ [14], etc. Programmers load data referred to as stream from off-chip memory to on-chip memory by bulk transfer first, then one or more user defined kernel consisting of several hundred operations execute on it. The output data will be stored back to off-chip memory by bulk transfer lastly. In kernel, the process is similar: reading data from stream, computing, and storing data to stream back.

Moreover, memory operations on a bulk of data reduce the memory access latency, and utilize memory bandwidth efficiently. This paper call computation kernels bulk operation and memory operations bulk transfers [15]. These conceptions are ubiquitous in various stream programming model, such as cudaMemcp and device function in CUDA [13], stream Read & stream Write in brook+ [14], etc. Programmers load data referred to as stream from off-chip memory to on-chip memory by bulk transfer first, then one or more user defined kernel consisting of several hundred operations execute on it. The output data will be stored back to off-chip memory by bulk transfer lastly. In kernel, the process is similar: reading data from stream, computing, and storing data to stream back.

3. Stream programming framework

Stream programming framework consists of two parts: kernel computation and stream data management. Kernel organizes computing as a series of kernels, while the later one manages data inter memory hierarchy and provides data for kernels. Next our framework is present through the two aspects.

3.1 Stream level framework

Stream level framework supports the definition of a new type of data: stream, and the operations on it. Loading to and storing from stream are the key of stream level framework. Bulk and asynchronous data transfer between on-chip and off-chip memory plays a critical role in sustaining high performance, which can significantly increase the throughput of off-chip memory and enable execution concurrency between computation and data transfer. Using stream as a basic data type makes this data transfer natural. A method to manage on-chip memory explicitly is configuring the on-chip memory of DSP as SRAM which will be mapped to a certain address space. Using DMA operation between this address space and another address space forbidden to be cached can easily implement the bulk data transfer. Waiting for the IRQ (interrupt request) can be used to meet the data dependency. There is no difference for loading instructions, except the memory address.

Figure 1 gives an example C and stream program for the same function. Variable c is current and previous a and b. In C programming, we only need to define three arrays, and then repeat operating on every array element cited by subscript. While in stream program, a and b have to be loaded to stream astr and bstr, then kernel invoked on every element of them outputs stream cstr which should be stored back to c. Computing in kernel consists of reading stream, getting neighbor a by communication among clusters, and writing stream.

C programmers view the computer as a process unit and a flat memory which is essentially a one dimensional array. All operations are performed by the process unit, and all data is stored in the memory. But programmers using stream model must manage memory hierarchy explicitly, which typically includes three levels: off-chip DRAM, on-chip SRAM and registers. More levels memory hierarchy was discussed by Timothy J. Knight[15]. If functional units are organized as clusters, we still have to manage data inter clusters. Take the code in figure 1 for example, array a, b and c is only stored in DRAM, while stream data and temporary data in kernel belong to SRAM and registers respectively. Data transfers are implemented through stream load&store and read&write stream.

Figure 1 gives an example C and stream program with the same function.

```c
int a[LENGTH], b[LENGTH], c[LENGTH];
for (int i = 0; i < LENGTH; i++)
    c[i] = a[i] + a[i-1] + b[i];
```

```c
Streamstore(c,cstr);
KernelAdd(astr,bstr,cstr);
Writestream(cstr,creg);
```

Figure 1. C and stream program with the same function

3.1 Stream level framework

Stream level framework supports the definition of a new type of data: stream, and the operations on it. Loading to and storing from stream are the key of stream level framework. Bulk and asynchronous data transfer between on-chip and off-chip memory plays a critical role in sustaining high performance, which can significantly increase the throughput of off-chip memory and enable execution concurrency between computation and data transfer. Using stream as a basic data type makes this data transfer natural. A method to manage on-chip memory explicitly is configuring the on-chip memory of DSP as SRAM which will be mapped to a certain address space. Using DMA operation between this address space and another address space forbidden to be cached can easily implement the bulk data transfer. Waiting for the IRQ (interrupt request) can be used to meet the data dependency. There is no difference for loading instructions, except the memory address.

Code below shows a simple example of stream loading using DMA. First we map SRAM to L2RAM, link strbuffer to it, and declare a stream variable srxstream in it which is in and only in on-chip memory. Second we declare array variable dataarray.

```c
CACHE_enableCaching(CACHE_EMIFA_CE00);
CACHE_setL2Mode(CACHE_256KCACHE);
MEMORY
{L2RAM : origin = 0x00000000, len = 0x00000000}
SECTIONS
{
    .vectors > ISRAM
    .text > ISRAM
    .strbuffer > L2RAM
}
#define STREAM(var,len) \n    unsigned int var[len];
```
At last, DMA operation is used to transfer data between `dataarray` and `srcstream`. Typically DSP could support various kinds of data transfer modes, such as 1-D, 2-D, etc., which should be chosen based on situations. The sample code below demonstrates how to load data from off-chip memory to on-chip using DMA. Program initiates, configures and launches the transfer, then waits for the synchronal event, which indicates completion of the transfer. Actually something else with no dependency could be done until the transferred data have to be used.

```c
#define STREAMLOAD(str, bin)\EDMA_OPEN\EDMA_Config myConfig = {
 0x41200000, /* opt */\bin, /* src */\0x00000040, /* cnt */\str, /* dst */\0x00000004, /* idx */\0x00000000 /* rld will be ignored */\};\EDMA_qdmaConfig\.........\EDMA_intTest\EDMA_CLOSE\STREAMLOAD(srcstream, srcBin);
```

This program can be compiled with original C compiler. Programmers have to allocate SRAM for stream manually, without special stream program compiler. But the number of streams is usually less then 8. It is not a hard work for user to allocate space for them.

This framework is responsible for hiding hardware details from programmers. All device-specific could be hidden. And the implementation of its self can be optimized along with the development of DSP. This makes codes good portability and framework easy to use.

### 3.2 Kernel level framework

The kernel framework includes two parts: the first is operation read&writes stream, the second is all the ALU operations. Read&write stream can be implemented through two methods. Assembly instruction load and store could be used to achieve our purpose. Using this method, users have to pre-allocate every register, and manage them during program running. Although it is very high efficient, it is also very hard to deal with, especially when we get a lack of registers. Assigning variable directly is the basic approach, which actually allocates registers by the C compiler. It is relatively simple and easy-to-use. But a lot of extra memory copy may be brought in. Computation described by kernel can only access on-chip memory, load and store data explicitly. This could enhance the efficiency of VLIW scheduling.

```c
#define READSTREAM(str, var, index)\var = str[index];
```

### 4. Performance Evaluation and Discussion

#### 4.1 Experimental Setup

**Experimental platform**

TMS320C6416 [12], our DSP platform uses high performance, advanced VLIW architecture. Figure 2 shows its structure. The eight functional units in the C6416 data paths can be divided into two groups of four; each functional unit in one data path is almost identical to the corresponding unit in the other data path.

![Figure 2. Structure of TMS320C6416](image)

There are two general-purpose register files (A and B) in the C6416 data paths. Each of these files contains 16 32-bit registers (A0–A15 for file A and B0–B15 for file B). Each functional unit reads directly from and writes directly to the register file within its own data path. That is, the .L1, .S1, .D1, and .M1 units write to register file A and the .L2, .S2, .D2, and .M2 units write to register file B. The register files are connected to the opposite-side register file’s functional units via the cross paths. These cross paths allow functional units from one data path to access a 32-bit operand from the opposite side register file. EDMA controller is also integrated on-chip to support DMA operation between on-chip memory and off-chip.
### Table 1. Performance of C and stream analysis_intra

<table>
<thead>
<tr>
<th></th>
<th>Cycle total</th>
<th>Cycle CPU</th>
<th>Instruction packet</th>
<th>Instruction executed</th>
<th>CPU access</th>
<th>Cycle stall</th>
<th>L1P stall</th>
<th>L1D stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>C prog</td>
<td>36340</td>
<td>31894</td>
<td>28071</td>
<td>68115</td>
<td>53799</td>
<td>4446</td>
<td>3328</td>
<td>149</td>
</tr>
<tr>
<td>S prog</td>
<td>5201</td>
<td>4987</td>
<td>4986</td>
<td>9310</td>
<td>6539</td>
<td>214</td>
<td>189</td>
<td>22</td>
</tr>
</tbody>
</table>

### Streaming H264

This paper selects H264 to do performance evaluation. We implement a streaming H264 on DSP; The compared C program is from an open source project x264[16]. H.264 standard [17] developed by the Video Coding Experts Group (VCEG) and the Motion Picture Experts Group (MPEG) is widely adopted in applications from high definition living room entertainment (BluRay/ HD-DVD) to Handhold terminals (DVB-H). It can save 25%-45% and 50%-70% of bit rates when compared with MPEG-4 Advanced Simple Profile (ASP) and MPEG-2, respectively. H264 is a typical media process application which stream processor and DSP are both good at.

Figure 3 shows our streaming H264 implementation. The construct of H264 encoding is outlined in figure 3(a). A picture frame is divided into a number of macro blocks, which are processed by analyse, encode, cavlc and filter one by one. Because the whole H264 is too big, and the implementations of different functional modules are similar, we take the analysis_intra for example to introduce our streaming H264 program.

The analysis_intra is the first step of I-slices’ encoding. It uses the samples in the current MB (macro block) that have already been encoded, decoded and reconstructed to perform the prediction [25]. The prediction get the best mode by calculating and comparing the SAD (Sum of Absolute Differences) which are generated by the source image samples and the prediction samples of each mode. The input is a 16x16 MB and other 33 pixels; the output is the best predict mode. Various kinds of modes are tried to do prediction, including 4 of 16x16 modes and 9 of 4x4.

### 4.2 Performance

Figure 4 shows the speedup achieved over original C implementation by reprogramming the H264 with stream program and the breakdown of execution time. Y-axis represents the cycles consumed by processing one MB. X-axis represents the four components of H264. The left four having suffix _c means C program, and others having suffix _s means stream program. Significant speedup is achieved, ranging from 3.2x for cavlc up to 7.1x for analysis. Cache access stall is little due to good spatial and temporal locality in media process applications. Traditionally media process applications are known as poor locality because of stream data; actually it is not [20]. Although H264 consists of several processes, every process is similar. Next we still detail the analysis_intra to show the reason for the performance boost.

**Instructions executed**

The performance of analysis_intra implemented with C and stream are listed in Table 1. The biggest difference between them is the number of instructions executed, which is 68115 in C, and 9310 in stream.
This lead to less memory accesses and less stalls, cause the greatest degradation in C program, and finally the performance boost of stream. analysis_intra_s (The kernel corresponding to analysis_intra in stream) doesn’t have any control statement such as if, while (replaced by select) and function calling except only one loop on the sequence of MBs. Moreover the data copies from reference to temporary and global variable references are removed. Although we have got the performance boost, the big percentage of access instructions is unusual. This paper will discuss this issue in 4.3.

**Level 1 program cache**

The second reason for the performance boost is the reduced stall cycles due to L1P (level 1 program cache) accesses. The size of code processing one macro block is bigger than 200K, which is the capacity of L1P. This result in a lot of cache misses when every MB is processed just like figure 5 shows. It is totally different in stream program which processes a sequence of MBs with a short code size kernel. The size of analysis_intra_s is 10.5K and it is small enough to fit the L1P; after the first time compulsory misses there will be no cache misses during processing the following MBs.

In conclusion, reduced number of cache accesses and higher utilization of ALUs due to more compact kernels lead to the performance boost of stream program. And our experiment shows that the execution time would be shorter with more registers. On the other hand managing on-chip memory explicitly affects performance little because of its small percentage of total execution time.

5. Related work

Jayanth Gummaraju has presented a similar method for programming general purpose processors with stream programming model [11]. But it is not practical for two reasons. First, this method demands additional architectural support, such as SLS (stream load/store unit) unit to manage stream load and store. It is a big obstacle to apply this method to practice. The other one is the application domain of GPP (general purpose processor). Developing a large desktop application with stream program is not realistic because it is much harder to use than C. Comparative speaking, there are no need for architectural support and special compiler in our method. And the DSP domain is more similar to stream processor.

Stream processing is applied to wider area now. GPGPU (general purpose computation on graphics hardware) [18] is about how to do general purpose computing with GPU. However the architecture of stream processor is very different from GPP. Some applications especially irregular ones are hard to implement on stream processor. Uses can benefit not only form the hardware architecture of stream processor, but also the programming model. Porting
the model to other processor is a realistic way to extend the stream processing domain.

6. Conclusion

Stream programming model can be applied in DSP as an efficient and productive programming language. Our research shows that a high speedup over C program has been achieved by this new programming model. Moreover users need not to know various optimizing methods. However the peak performance of DSP is hard to achieve due to the differences between the architectures of DSP and stream processor.

Acknowledgement
This research was supported by NSFC No. 60673148, 60703073, 863 Project No. 2007AA01Z286, SRFDP No. 20069998025.

References

[12] TI,TMS320C6000 CPU and Instruction Set Reference Guide
[15] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez Kayvon Fatahalian, Alex Aiken, William J. Dally, Pat Hanrahan. Compilation for Explicitly Managed Memory Hierarchies. PPoPP’07 March 14–17, 2007, San Jose, California, USA.
[18] GPGPU. http://www.gpgpu.org