GPU Based Feature Extraction Implementation

Haofeng Kou, Weijia Shang, Jike Chong, Ian Lane

Santa Clara University & Carnegie Mellon University

BACKGROUND

MFCC

The mel-frequency cepstrum (MFC) is a representation of the short-term power spectrum of a sound, based on a linear cosine transform of a log power spectrum on a nonlinear mel-scale of frequency.
MFCCs are coefficients that collectively make up an MFC

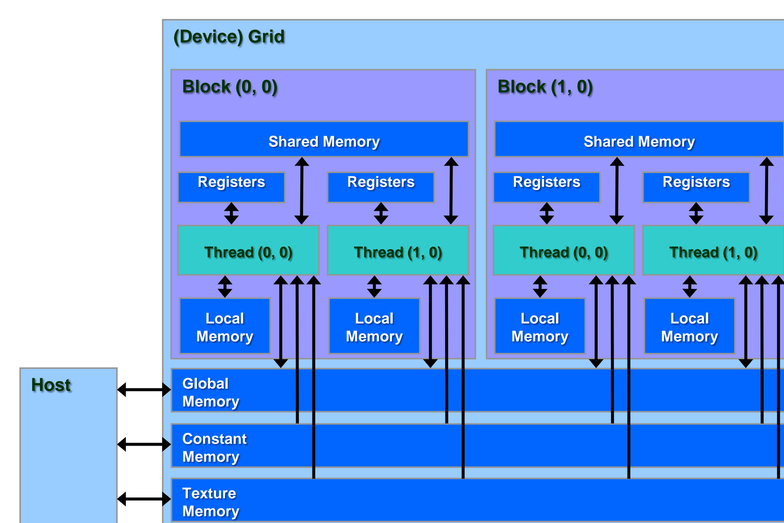
GPU Computing

- Multicore: yoke of oxen
  - Each core optimized for executing a single thread
- Manycore: flock of chickens
  - Cores optimized for aggregate throughput, deemphasizing individual performance

CUDA & OpenCL

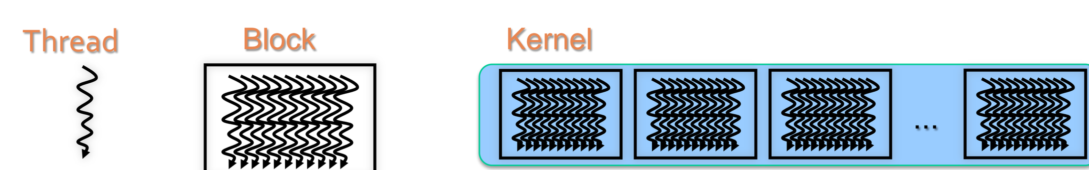
- CUDA is a recent programming model, designed for
  - Manycore architectures
  - Wide SIMD parallelism
  - Scalability
- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchronization & data sharing between small groups of threads
- CUDA programs are written in C + extensions
- OpenCL is the open standard for parallel programming of heterogeneous systems

HW Architecture of GPU



SW Computing Model of CUDA

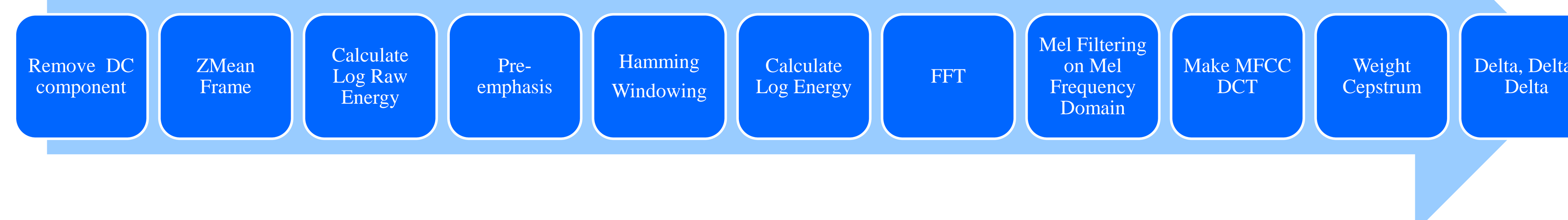
Single Instruction Multiple Data architectures make use of data parallelism



A program consist of a sequential CPU program running multiple parallel kernels running on GPU

Kernel is SIMD – with multiple threads

DETAIL IMPLEMENTATION



Feature Extraction Process Software Sequence

Requirement Analysis

- Size of data
  - Input: 400 samples per frame
  - Output: 13 Cepstral coefficients
- Hardware limitation
  - Geforce GTX 460 @ SCU
  - Shared Memory per Block 49 Kbytes
  - 7 (MP) x 48 (Cores/MP) = 336 (Cores)

Implementation

- Each block for one frame MFCC
- Number of frames loaded in GPU is designed as a variable
- MFCC Parameter Used:
  - Preemphasis: α=0.97
  - Windowing: use hamming window.
  - Number of Mel channels=24
- Computational bottleneck is FFT
  - Use SPIRAL Generated CUDA Code

Design Analysis

- Make Full Use of Shared Memory
  - 400 Float Data =1200 bytes, 50KBytes /4 (Warps)≈12 Kbytes /Block
- Minimize Data Transfer
  - One kernel for each stage of MFCC
  - Shared Memory life cycle—the execution of one block
  - One kernel for the whole MFCC extraction
  - Some threads are left idle, yet it saves the time for data transfer.
- Explore Maximum Concurrency
  - Do multiple frames' MFCC extraction together

VTLN Interface

Basing on different VTLN alpha input, the feature extraction code generates different MFCC.

The calculation inside Kernel up to FFT part is optimized to execute once for different VTLN alpha.

Challenge 1:

Optimization of GPU memory usage

There are several different data need to be passed into the GPU Kernel function, in order to achieve the best performance, some data need to be loaded from global memory to shared memory and the number/size of shared variables need to be carefully designed. For example, the following are shared for frames:

- MFCC configuration parameter values
- Work area for MFCC computation
- Workspace for filterbank analysis

The following are for each frame:

- Windowed waveform data
- Filterbank data

Solution 1:

Try to move as much data as possible into shared memory if there is enough space.

After shared memory reach its limit, sorting the data by how frequency are they being used, and move the less used data out of shared memory.

The windowed waveform data is the major input data, make a copy in shared memory, keep the temp data (for example RE, IM, FB…) also in shared memory.

Challenge 2:

GPU performance tuning strategy

This is a general topic for all GPU CUDA code and every CUDA programmer, sooner or later, does face this issue.

Solution 2:

Optimize the GPU memory usage – as mentioned in the solution 1.

Re-use the local, shared, register variables as more as it can, which will save space, but need to be carefully on the coding to rule out the life-time of each variable and make sure there is no overlap, and if there is overlap, how to prevent race condition.

Try to replace the heavy operation with light weight ones, for example the atomic operation normally cost more, and it can be replaced by using a different data structure or different implementation approach. Another example is to avoid the table lookup, replace it with on-fly computing.

Try to use the fast math API instead of the heavy ones.

Remove the unnecessary __syncthreads().

Reuse the piece of code which are called often, for example the fast VTLN implementation.

For the case of using double buffer, use them properly to prevent unnecessary buffer sync.

RESULTS

Num. Frames	CPU Program (msec)	GPU Program (msec)	Ratio (Tcpu / Tgpu)
1	0.57	<0.1	>5.7
5	0.81	<0.1	>8.1
10	1.1	<0.1	>11
20	1.69	0.1	16.9
50	3.46	0.2	17.3
100	6.41	0.3	21.4
250	15.26	0.6	25.4
500	28.11	1.2	23.4
1000	53.85	2.3	23.4

CPU:
- processor : 4
- model name : Intel(R) Core(TM)2 Quad CPU Q8300 @ 2.50GHz
- cpu MHz : 2003.000
- cache size : 2048 KB

GPU:
- GeForce GTX 460 CUDA @ SCU
- CUDA Driver Version / Runtime Version 4.0 / 4.0
- CUDA Capability Major/Minor version number: 2.1

OS:
- 2.6.31-14-generic #48-Ubuntu SMP Fri Oct 16 14:05:01 UTC 2009 x86_64 GNU/Linux
- ubuntu 9.10 karmic
- gcc version 4.4.1 (ubuntu 4.4.1-4ubuntu8)

Date: 12/09/2011

CONCLUSIONS

GPU based Feature Extraction shows up to about 25 times faster than CPU version for large amount of frame data.

The similar implementation could be used for other signal processing software to improve performance.

FUTURE WORKS

Porting the x86/GPU based Feature Extraction to the SOC platform which means most likely the code will be ported to the ARM SOC with GPU build-in, for example the Tegra with GPU from Nvidia or the TI OMAP5 with GPU and so on.

Besides CUDA, OpenCL and OpenGL are also on the list of evaluation.

printed by MegaPrint Inc.
www.postersession.com