Voice Command Recognition with Dynamic Time Warping (DTW) using Graphics Processing Units (GPU) with Compute Unified Device Architecture (CUDA)

Gustavo Poli[1], Alexandre L. M. Levada[2], João F. Mari[1], José Hiroki Saito[1].
[1]*Departamento de Computação - Universidade Federal de São Carlos, São Carlos, SP, Brasil.*
[2]*Instituto de Física de São Carlos - Universidade de São Paulo, São Carlos, SP, Brasil.*
{gustavo_silva,joao_mari,saito}@dc.ufscar.br, alexandreluis@ursa.ifsc.usp.br

Abstract

Recently, we are attending to a huge evolution on the development of high performance computing platforms. Among these platforms, the GPU (Graphics Processing Units) stimulated by game industries, constantly demanding more graphical processing power, evolved from a simple graphical card to a general purpose computation parallel data processing device. This article shows the GPU's viability to general purpose computation, developing a speech recognition application inside. Dynamic Time Warping (DTW) is applied on a voice password identification. Normally, DTW requires large amount of data and processing time, so that it is an efficient technique to simple vocabulary, when the voice commands set is small. Using NVIDIA GeForce 8800 GTX, with 128 processing unit cores, and a CUDA (Compute Unified Device Architecture) software platform development architecture, the DTW application was implemented, and tested its performance.

1. Introduction

Recently, we are attending to a huge evolution on the development of high performance computing platforms. Among these platforms, the GPU (Graphics Processing Units) has evolved into an absolute computing workhorse, with multiple cores driven by very high memory bandwidth. Today's GPUs offer incredible resources for both graphics and non-graphics processing. One of the main reasons is the fact that GPU is specialized to compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore is designed such that more transistors are devoted to data processing rather than data catching and flow control.

These GPUs are very powerful. For example, the nVidia GeForce3 chip contains more transistors than the Intel Pentium IV and its successor the GeForce4 is advertised as being able to perform more than 1.2 trillion internal operations per second. The internal pipelined processing fashion makes the GPU also suitable for stream processing. Furthermore, GPU speed grows much faster than the famous Moore's law for CPU, that is, 2.4 times/year versus 2 times per 18 months. Moreover, a GPU usually contains multiple (typically 4 or 8) parallel pipelines and is indeed a SIMD processor. Another important feature of most contemporary GPUs is their programmability of the partial or full graphics pipeline, thanks to the introduction of vertex shaders and pixel shaders in DirextX-8. The powerfulness, SIMD operation and programmability of GPUs have motivated an active research area of using GPU for non-graphics oriented operations such as numerical computations like basic linear algebra subprograms (BLAS) [3] and image/volume processing [4]–[9]. In [4], the authors presented a technique for multiplying large matrices quickly using the graphics hardware of a PC. Strzodka and Rumpf have implemented complicated numerical schemes solving parabolic differential equations fully in graphics hardware [5]. Chris Thompson et al. introduced a programming framework of using modern graphics architectures for general purpose computing using GPU [6]. In [7], the authors tested five color image processing algorithms using the latest programmability feature available in DirectX-9 compatible GPUs. Performing FFT on GPU was reported by Doggett et al. in [8]. Wavelet decomposition and reconstruction was implemented on modern OpenGL capable graphics hardware [9].

The objective of this article is to show how the GPU can do speech recognition through Dynamic Time Warping (DTW) technique. For this, it was developed an application for numerical voice password recognition using NVIDIA GeForce 8800 GTX, with 128 processing unit cores.

The following sections of this article is organized in Abstract, Section 1 - Introduction, Section 2 –

1550-6533/07 $25.00 © 2007 IEEE
DOI 10.1109/SBAC-PAD.2007.21

Dynamic Time Warping, Section 3 – Compute Unified Device Architecture, Section 4 – Methodology, Section 5 – Results, Section 6 - Conclusion and Section 7 - References.

2. Dynamic Time Warping

Dynamic Time Warping (DTW), [1-2] is a pattern matching technique used to compare signals, not necessarily of same size, based on its characteristic shapes. The basic idea is to compare the samples of an unknown input signal with the samples of a set of template signals. The unknown signal is classified as the most similar pattern found. Our motivation here is that, in speech recognition, each pattern (voice signal) is characterized by a specific waveform.

The main advantages of using DTW are:

- It does not require complex mathematical models, resulting in a simple algorithm.
- It requires only one sample for each class, that is, a single representative template for each class is enough (most approaches require a training set with many samples for each class, what is not always possible).

However, the method also has some complications, such as:

- High computational cost due to size of audio signals makes the method a "slow" approach.
- Unviable for real time processing
- Requires a restrict vocabulary

With this work, we expect to reduce some of these complications by using a parallel high performance computing platform (GPU). The DWT algorithm is detailed below.

Algorithm

1. Read the input symbol (voice signal of size *M*).
2. Repeat for each template signal of size T_i.

   a. Allocate $M \times T_i$ DIST matrix, to store the absolute difference between all pairs of values related to the input and template signals.

   b. Allocate $M \times T_i$ ACCDIST matrix, the matrix of accumulated distances, obtained from the DIST matrix, as indicated on the next steps.

   c. Compute the absolute difference between each input sample and the current template sample

$$DIST(i,j) = \left| input(i) - template_k(j) \right|$$

   d. Fill the first line of ACCDIST matrix

$$ACCDIST(0,0) = DIST(0,0)$$
$$ACCDIST(i,0) = DIST(i,0) + ACCDIST(i-1,0)$$

   e. Fill the first column of ACCDIST matrix

$$ACCDIST(0,j) = DIST(0,j) + ACCDIST(0,j-1)$$

   f. For each element (i,j) of ACCDIST do:

$$ACCDIST(i,j) = DIST(i,j) + \min_{(k,l)\in V} \{ACCDIST(k,l)\}$$

   where $V = \{(i-1,j);(i,j-1);(i-1,j-1)\}$ is the set of 3 previous neighbors. The preference is always for the diagonal direction.

   g. Calculate the size of the path between the last and initial points of ACCDIST matrix:

```
i = M;
j = Ti;

while (i+j != 0)
  direction = min(ACCDIST(k,l))

  switch(direction)
    case 'down': j--;

    case 'back': i--;

    case 'diagonal': i--;
                     j--;
  end

  path_size++;
end
```

3. Select the template with the smallest distance.

4. Repeat steps 1 to 3 for all input symbols.

3. Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without need of mapping them to a

graphics API. When programmed through CUDA, the GPU is viewed as a compute device of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU, or host.

Each multiprocessor, on computer device (GPU), is implemented as Single Instruction, Multiple Data architecture (SIMD), so that at any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data.

The CUDA software stack is composed of several layers as illustrated in Figure 1, a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries of common usage, Fast Fourier Transform 1D, 2D and 3D transforms of complex-valued signal data (CUFFT) and an implementation of basic linear algebra (CUBLAS). The hardware has been designed to support lightweight driver and runtime layers, resulting in high performance.

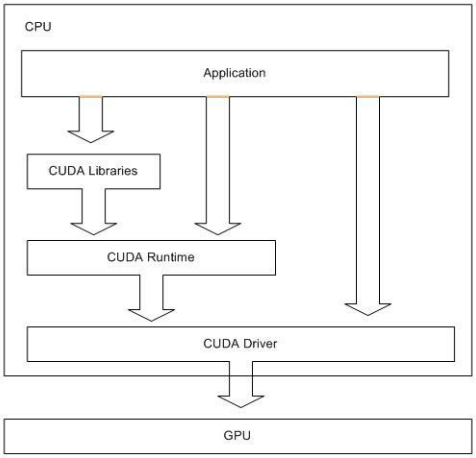


Figure 1. CUDA Stack

The batch of threads that executes a kernel is organized as a grid of thread blocks in a batch of threads that can cooperate together sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses efficiently. Each thread is identified by its thread ID, which is the thread number within the block and it specify a block as a 2 or 3 dimensional array of arbitrary size. For a 2 dimensional block size $(D_x, D$ the thread ID of a thread of index $(x,$ is (x, y) and for a 3 dimensional block size $(D_x, D_y, D$ the thread ID of a thread of index $(x, y,$ is $(x + yD_x + zD_xD$ (Fig 2).

A grid of thread blocks is executed on device by executing one or more block on each multiprocessor using time slicing. Each block is split into SIMD groups of threads called warps, each of these warps contains the same number of threads, called the warp size, and is executed by the multiprocessor in a SIMD fashion; a thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessor´s computational resources.

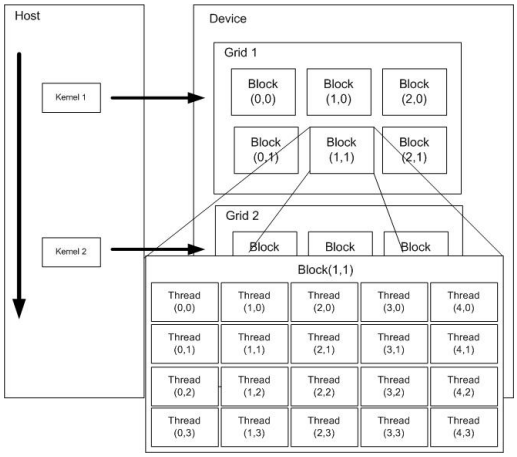


Figure 2. Thread Batching

Device memory (Fig 3) can be allocated either as linear memory or as CUDA array and a thread that executes on the device has only access to the device's DRAM and on-chip memory through the following memory spaces:

- Read-write per-thread registers,
- Read-write per-thread local memory,
- Read-write per-block shared memory,
- Read-write per-grid global memory,
- Read-only per-grid constant memory,
- Read-only per-grid texture memory.

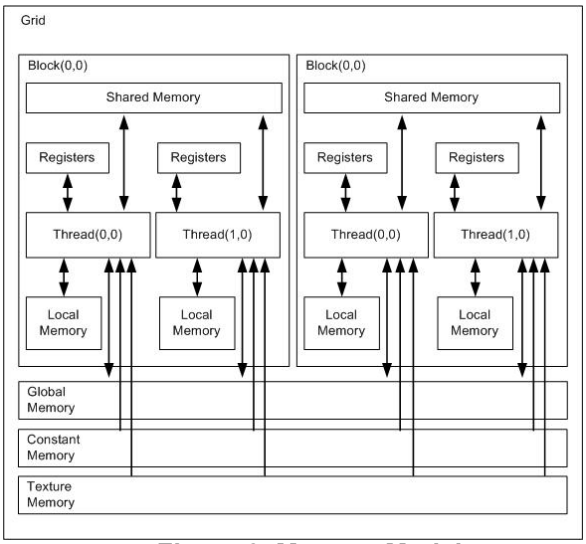


Figure 3. Memory Model

Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats.

The CUDA API comprises an extension to the C programming and for the compilation the CUDA source files it is used the NVCC, a compiler driver that simplifies the process of compiling. It provides simple and familiar command line options and executes them by invoking the collection of tools that implement the different compilation stages. NVCC's basic workflow consists in separating the device code from the host code and compiling the device code into a binary form or cubin. The generated host code is output either as C code that is left to be compiled using another tool or as object code directly by invoking the host compiler during the last compilation stage.

4. Methodology

In this section, we present the development of the proposed application for voice commands recognition using DTW processing in GPU. To the development of this application, the following tools were used:

a) NVIDIA GeForce 8800 GTX
b) Athlon X2 Dual Core 2.2Ghz
c) Microsoft Visual Studio 2005 C++ Express
d) CUDA Tookit 0.8
e) CUDA SDK 0.8.1
f) CUDA Driver 97.73 (for Windows XP)

The main problem consists in answer 2 questions: what part of application should be run in device (GPU)? And how many kernels will be necessary to the DTW process?

The answer to the first question is: process all vector or matrix data on the GPU, unless input/output (I/O) instructions. More precisely, a portion of the application that is executed many times, but independently on different data, can be isolated into a function that is executed on the device as many different threads. Having this vision, we divided the DTW process in three different kernel modules: population of the DIST matrix, population of the ACCDIST matrix and path finding (Fig. 4)

5. Results

A significant increase of performance was verified mainly in the first two kernel's, where the computation of the values of both DIST and ACCDIST matrices can be done on a general form, that is, using a big block size (the same operation is applied to all elements and no local decisions are required).

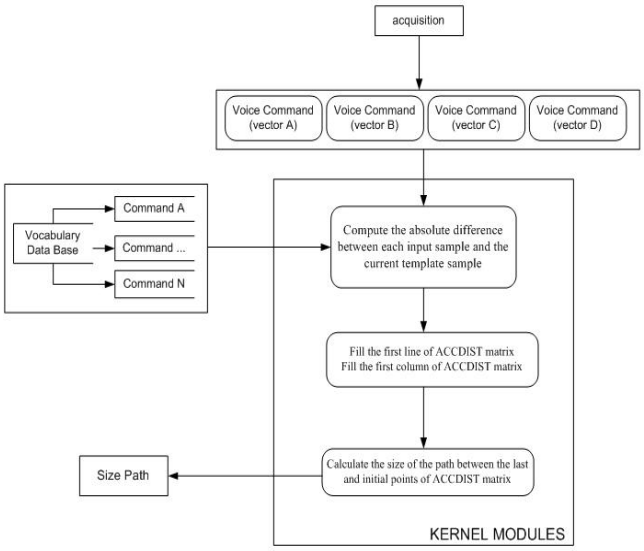


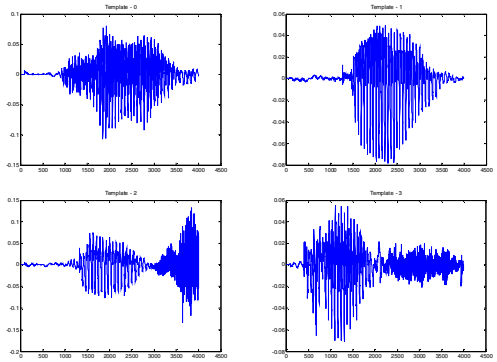
Figure 4. Workflow application

Assuming the two first modules (kernel´s), the increase of performance obtained using the GPU against the CPU was of approximately 75%. However in the last module, responsible for the calculation of the path, the time of processing between the GPU and the CPU it was practically the same, being GPU only 5% faster than the CPU.

The explanation for this "low" performance is that, in the last kernel module, we verified the existence of non intensive or highly parallel computation. In fact, the problem here is the block size, due to the extremely local and dependent nature of the path finding algorithm (there is a strong dependency structure between the components, which unable massive and parallel computation of the total path size).

The target password was the sequence defined by the digits: 9-8-0-5. The template signals, representing digits 0 to 9, and input signals (9805) are shown below (Fig. 5).

Template Signals



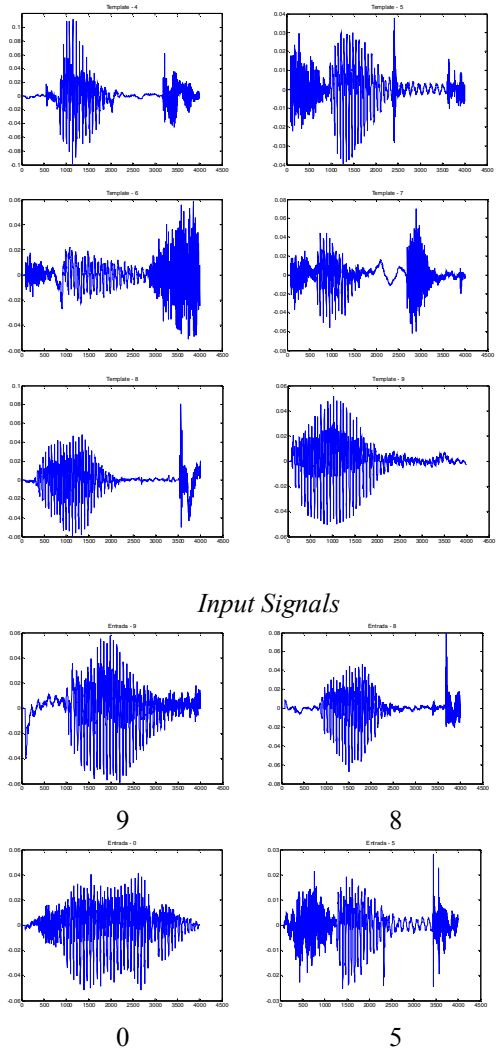


Figure 5. Template and input signals

The following table (Table 1) shows the path sizes obtained for each digit. Note that the correct patterns produced the smallest path size on all situations. Path size expressed in distance units (*d.u*)

Table 1. Path sizes for input digit sequence

Templates	*Input Password*			
	9	**8**	**0**	**5**
0	6205	6565	**5923**	6593
1	6265	6503	6124	6620
2	6217	6222	6127	6503
3	6094	5992	6036	6123
4	6575	6382	6435	6455
5	6467	6320	6418	**5798**
6	6672	6592	6358	5851
7	6373	6668	6332	6277
8	6556	**5933**	6525	6570
9	**6014**	6154	6276	6196

The results and comments can be verified in the following performance graphics, which show the GPU´s characteristics for the adopted configurations. Table 2 shows the processing characteristics of the kernel 1; Fig. 6, shows the multiprocessor warp occupancy varying the block size; Fig. 7, varying the register count; and Fig. 8 varying the number of register per thread.

Table 2. Configuration Kernel 1 call device

Kernel 1	
Threads per block	128
Registers per Thread	10
Shared Memory Per Block (bytes)	4096

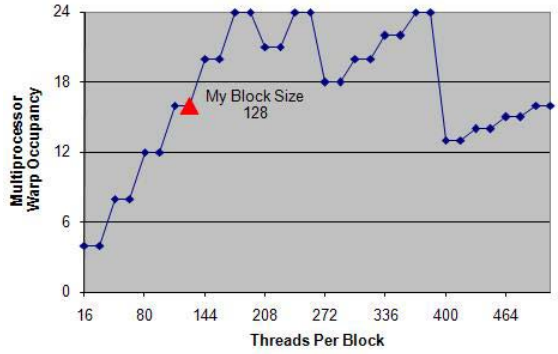


Figure 6. Varying block size

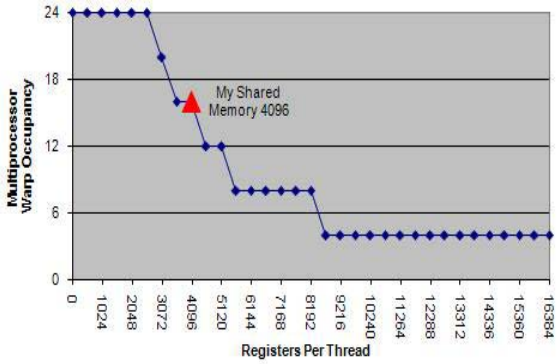


Figure 7. Varying register count

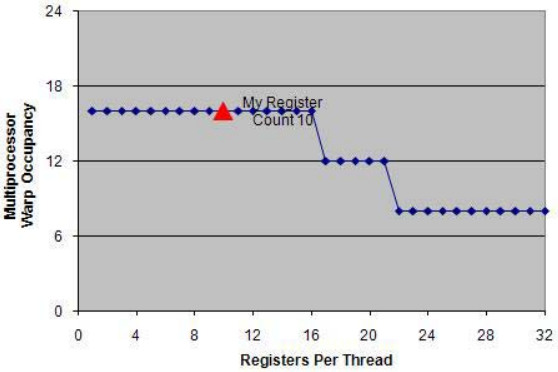


Figure 8. Varying shared memory usage

Table 3 shows the processing characteristics of the kernel 2; Fig. 9, shows the multiprocessor warp occupancy varying the block size; Fig. 10, varying the register count; and Fig. 11 varying the number of register per thread.

Table 3 Configuration Kernel 2 call device

Kernel 2	
Threads per block	256
Registers per Thread	10
Shared Memory Per Block (bytes)	4096

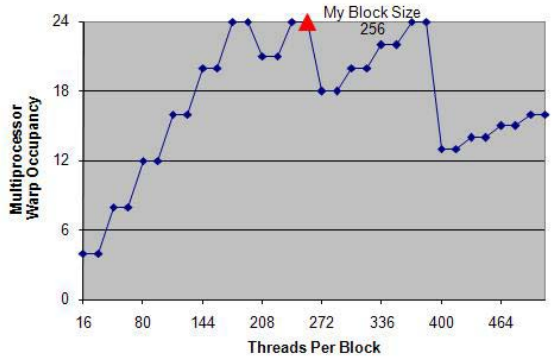


Figure 9. Varying block size

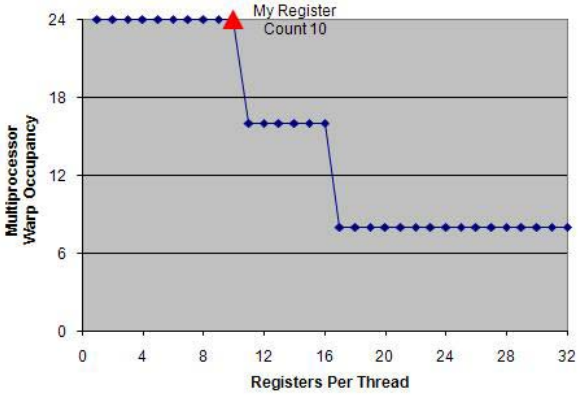


Figure 10. Varying register count

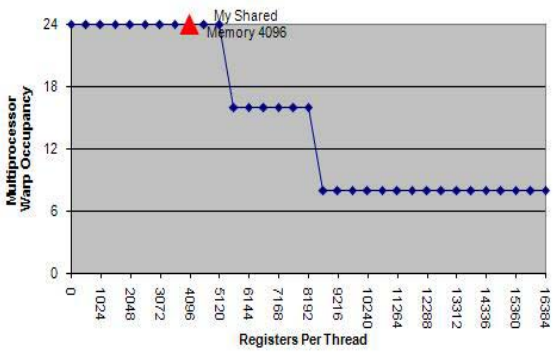


Figure 11. Varying shared memory usage

Table 4 shows the processing characteristics of the kernel 3; Fig. 12, shows the multiprocessor warp occupancy varying the block size; Fig 13, varying the register count; and Fig 14 varying the number of register per thread.

Table 4 Configuration Kernel 3 call device

Kernel 3	
Threads per block	8
Registers per Thread	10
Shared Memory Per Block (bytes)	4096

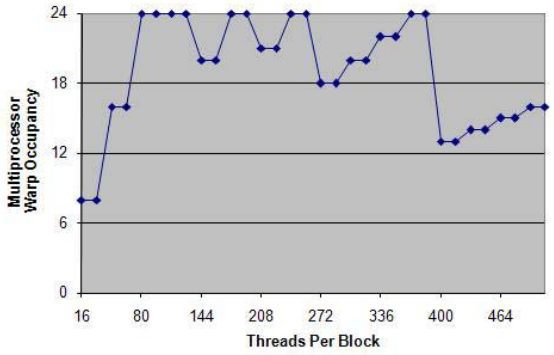


Figure 12. Varying block size

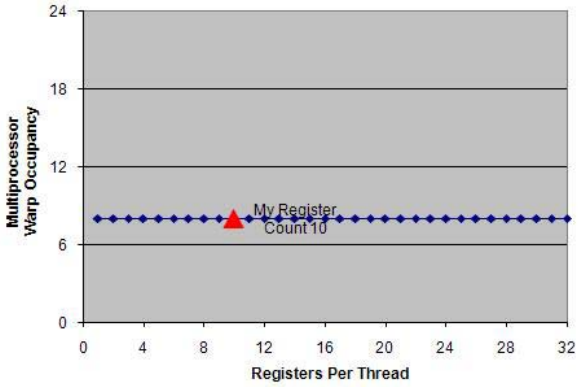


Figure 13. Varying register count

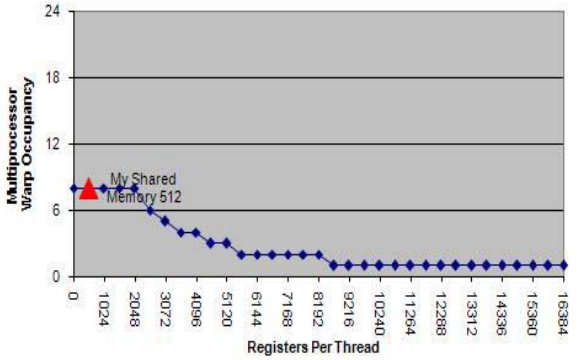


Figure 14. Varying shared memory usage

6. Conclusions

This paper proposed a study on the GPU's viability to general purpose computation, developing a speech recognition application. An increase of performance exists on the use of GPU in projects with great volume of data to be processed in parallel form (ie. big and independent blocks), without the need for taking local decisions, or shunting lines during the processing. Finally, regarding GPU´s viability, we conclude that it is possible to separate the applications in 3 classes: fully portable, semi portable (which is the case) and non portable.

7. References

[1] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition", *IEEE Trans. On Acoustics, Speech and Signal Processing*, ASSP-26, pp. 43-49, 1978.

[2] J. Coleman, *Introducing Speech and Language Processing*, Cambridge University Press, New York, 2005.

[3] M. Harris, "GPGPU: General-purpose computation using graphics hardware", http://www.cs.unc.edu/~harrism/gpgpu, 2003.

[4] E. S. Larsen and D. K. McAllister, "Fast matrix multiplies using graphics hardware," in Proc. IEEE Supercomputing, Nov. 2001, p. 55.

[5] M. Rumpf and R. Strzodka, "Level set segmentation in graphics hardware," in Proc. ICIP, vol. 3, 2001, pp. 1103–1106.

[6] C. J. Thompson, S. Hahn, and M. Oskin, "Using modern graphics architectures for general-purpose computing: A framework and analysis," in Proc. ACM/IEEE MICRO-35, Nov. 2002, pp. 306–317.

[7] P. Colantoni, N. Boukala, and J. D. Rugna, "Fast and accurate color image processing using 3-D graphics cards," presented at the 8th Int. Fall Workshop: Vision Modeling and Visualization, Munich, Germany, Nov. 2003.

[8] K. Moreland and E. Angel, "The FFT on a GPU," in Proc. SIGGRAPH/Eurographics Workshop Graphics Hardware, July 2003, pp. 112–119.

[9] M. Hopf and T. Ertl, "Hardware accelerated wavelet transformations", In Proc. EG/IEEE TCVG Symp. Visualization, 2000, pp. 93–103.