Universidade de Évora
Honesto Estudo com Longa Experiência Misturado

Departamento de Informática

Visual Programming in a Heterogeneous Multi-core Environment

Pedro Miguel Rito Guerreiro

<l18766@alunos.uevora.pt>

Supervisor: Salvador Abreu (Universidade de Évora, DI)

Évora

2009

Nota: Esta dissertação não inclui as críticas e sugestões feitas pelo júri.

Universidade de Évora
Honesto Estudo com Longa Experiência Misturado
Departamento de Informática

Visual Programming in a Heterogeneous Multi-core Environment

Pedro Miguel Rito Guerreiro

<l18766@alunos.uevora.pt>

Supervisor: Salvador Abreu (Universidade de Évora, DI)

Évora

2009

Nota: Esta dissertação não inclui as críticas e sugestões feitas pelo júri.

Acknowledgments

First of all, I would like to give a big and deep “thank you” to everyone who helped me during this journey of my life. Going to Germany (first time outside Portugal) and integrating myself in the IBM lab in Böblingen would not have been so easy without the help of everyone in the Cell/B.E. Test Team. Very special thanks to: Mijo Safradin and Rui Machado, who helped me to integrate; Dr. Werner Kriechbaum, whose contribution and help during the development of this thesis I cannot quantify; Dr. Salvador Abreu, who made this journey come true; Valério Valério, for downloading some articles I needed; Thomas Bodner, Johannes Bodner and Stephanie Bollinger, for their contribution to the proof reading and Hilário Fernandes, for helping me with the paper stuff at the university while I was in Germany.

And at last, but not least, thanks to my family who always have supported me, no matter what.

Contents

List of Figures

List of Tables

Abstract

It is known that nowadays technology develops really fast. New architectures are created in order to provide new solutions for different technology limitations and problems. Sometimes, this evolution is pacific and there is no need to adapt to new technologies, but things also may require a change every once in a while.

Programming languages have always been the communication bridge between the programmer and the computer. New ones keep coming and other ones keep improving in order to adapt to new concepts and paradigms. This requires an extra-effort for the programmer, who always needs to be aware of these changes.

Visual Programming may be a solution to this problem. Expressing functions as module boxes which receive determined input and return determined output may help programmers across the world by giving them the possibility to abstract from specific low-level hardware issues.

This thesis not only shows how the Cell/B.E. (which has a heterogeneous multi-core architecture) capabilities can be combined with OpenDX (which has a visual programming environment), but also demonstrates that it can be done without losing much performance.

Keywords: Cell, Visual Programming, OpenDX

Resumo

Programação visual numa arquitectura multi-processador heterogénea

É do conhecimento geral de que, hoje em dia, a tecnologia evolui rapidamente. São criadas novas arquitecturas para resolver determinadas limitações ou problemas. Por vezes, essa evolução é pacífica e não requer necessidade de adaptação e, por outras, essa evolução pode implicar mudanças.

As linguagens de programação são, desde sempre, o principal elo de comunicação entre o programador e o computador. Novas linguagens continuam a aparecer e outras estão sempre em desenvolvimento para se adaptarem a novos conceitos e paradigmas. Isto requer um esforço extra para o programador, que tem de estar sempre atento a estas mudanças.

A Programação Visual pode ser uma solução para este problema. Exprimir funções como módulos que recebem determinado input e retornam determinado output poderá ajudar os programadores espalhados pelo mundo, através da possibilidade de lhes dar uma margem para se abstraírem de pormenores de baixo nível relacionados com uma arquitectura específica.

Esta tese não só mostra como combinar as capacidades do Cell/B.E. (que tem uma arquitectura multi-processador heterogénea) com o OpenDX (que tem um ambiente de programação visual), como também demonstra que tal pode ser feito sem grande perda de performance.

1 Introduction

Computer science only indicates the retrospective omnipotence of our technologies. In other words, an infinite capacity to process data (but only data – i.e. the already given) and in no sense a new vision. With that science, we are entering an era of exhaustivity, which is also an era of exhaustion. – Jean Baudrillard (1929 – 2007)

It is known that nowadays technology develops really fast. This evolution brings more and more computation power, but this power is useless if we do not know how to use it. Cell Broadband Engine (Cell/B.E.) is a heterogeneous multi-core architecture proposed by Sony, Toshiba and IBM for computing intensive tasks. The idea behind its creation is to allow improvement of applications performance, by computing different tasks in different cores.

New technologies also bring new ways of thinking. Breaking a limitation usually implies breaking a paradigm, and new paradigms offer new solutions, which may solve many problems.

However, adapting to new concepts is not so easy, especially if we are strongly connected to the "old" ones. Cell/B.E. is not any exception and developing new applications for it can be quite challenging.

On the other hand, there are applications already developed for "older" architectures that help people in their daily routines. OpenDX (DX) is an open-source application for data visualization and it goes in the direction of what Jean Baudrillard was defending. Its goal is to grab/process data and give a new insight into it, a meaning.

Plus, DX uses an innovative way to manipulate the data by the use of a Visual Programming Environment. This approach is very high-level and DX users do not need to care about any low-level issues related with the architecture where the application is running.

So, if we can run a Cell/B.E. application on DX we can win three things:

- More computing power
- Ability to keep seeing the meaning of data
- No need to change any paradigm for DX users

The purpose of this thesis is to join the capabilities of Cell/B.E. with the power of data visualization and the simplicity of visual programming. This means not only running a Cell/B.E. application on DX, but also showing how performance can be improved in an already implemented module for DX.

There are two main challenges when trying to fulfil this task: the first one is how to embed a Cell/B.E. application in DX and the second is how to improve the performance of an already implemented module.

This thesis is organized as follows:

- Chapter 2: Background provides background information about the three main concepts in this thesis: Cell/B.E., Visual Programming and OpenDX.
- Chapter 3: Visualization in Cell/B.E. introduces all the practical work involved and puts together the three main concepts exposed in chapter 2.

- Chapter 4: Conclusion summarises the entire thesis and discusses about the encountered problems and limitations during the development of the practical work. It has one sub-chapter which shows some ideas for future work.
- Bibliography contains all the reference material.
- Glossary contains a list of terms and their definitions.

2 Background

2.1 Cell / Heterogeneous Multi-core Environment

In this chapter the basic concepts about the Cell/B.E. (Arevalo, et al. 2008) are described: hardware, capabilities and programming environment.

2.1.1 Cell/B.E. Overview

The Cell/B.E. processor is the first implementation of a new multiprocessor family conforming to the Cell Broadband Engine Architecture (CBEA). The CBEA and the Cell/B.E. processor are the result of a collaboration between Sony, Toshiba, and IBM known as STI, formally started in early 2001.

The basic configuration is a multi-core heterogeneous chip that was designed to improve performance in computing. Figure 2-1 shows an overview of the Cell/B.E. Its main components are:

- One main processor: Power Processor Element (PPE)
- Eight identical computing-intensive processors: Synergistic Processor Elements (SPEs)
- One Memory Interface Controller (MIC)
- One main connector between all parts: Element Interconnect Bus (EIB)
- Two I/O Interfaces

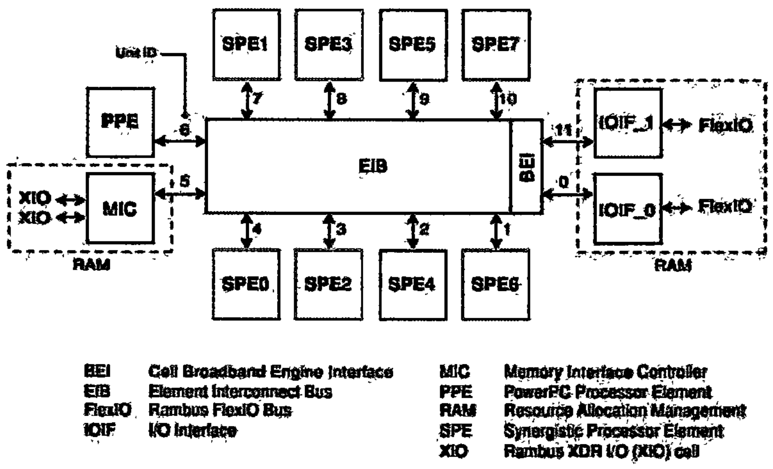

SPE1
SPE3
SPE5
SPE7
Unit ID
PPE
EIB
BEI
MIC
XIO
XIO
RAM
IOIF_1
IOIF_0
FlexIO
FlexIO
RAM
SPE0
SPE2
SPE4
SPE6
0
1
2
3
4
5
6
7
8
9
10
11
BEI Cell Broadband Engine Interface
EIB Element Interconnect Bus
FlexIO Rambus FlexIO Bus
IOIF I/O Interface
MIC Memory Interface Controller
PPE PowerPC Processor Element
RAM Resource Allocation Management
SPE Synergistic Processor Element
XIO Rambus XDR I/O (XIO) cell

Figure 2-1 Cell Broadband Engine Overview

The particular concept about its multi-cores is related to the fact that the PPE is a processor mainly designed to control threads and the SPEs are mainly designed for computing-intensive tasks.

This new concept not only opens new doors in application performance improvement but also breaks some of the traditional paradigms in programming. The next three main concepts must be kept in mind while Cell/B.E. is introduced: computing parallelization, DMA (Direct Memory Access) transfers and Vector SIMD (Single Instruction Multiple Data).

2.1.2 Hardware Specification

Figure 2-1 shows a high-level block diagram of the Cell/B.E. processor hardware. There is one PPE and eight identical SPEs. All processor elements are connected to each other and to the on-chip memory and I/O controllers by the memory-coherent element interconnect bus (EIB).

PPE, SPE and EIB, the main components of the processor, are described below.

2.1.2.1 Power Process Element (PPE)

The PPE contains a 64-bit, dual-thread PowerPC Architecture RISC core and supports a PowerPC virtual-memory subsystem. It has 32 KB level-1 (L1) instruction and data caches and a 512 KB level-2 (L2) unified (instruction and data) cache. It is intended primarily for control processing, running operating systems, managing system resources, and managing SPE threads. It can run existing PowerPC Architecture software and is well-suited to executing system control code. The instruction set for the PPE is a version of the PowerPC instruction set. It includes the Vector/SIMD multimedia extensions and associated C/C++ intrinsic extensions.

2.1.2.2 Synergistic Processor Element (SPE)

The eight identical SPEs are single-instruction, multiple-data (SIMD) processor elements that are optimized for data-rich operations allocated to them by the PPE. Each SPE contains a RISC core, 256 KB software-controlled LS for instructions and data, and a 128-bit, 128-entry unified register file. The SPEs support a special SIMD instruction set and a unique set of commands for managing DMA transfers and inter-processor messaging and control. SPE DMA transfers access main storage using PowerPC effective addresses. As in the PPE, SPE address translation is governed by PowerPC Architecture segment and page tables, which are loaded into the SPEs by privileged software running on the PPE. The SPEs are not intended to run an operating system.

The more significant difference between the SPE and PPE lies in how they access memory. The PPE accesses main storage (the effective-address space) with load and store instructions that move data between main storage and a private register file, the contents of which may be cached. PPE memory access is like that of a conventional processor technology, which is found on conventional machines.

The SPEs, in contrast, access main storage with direct memory access (DMA) commands that move data and instructions between main storage and a private local memory, called a local store or local storage (LS). The SPE instruction set accesses its private LS rather than shared main storage and the LS has no associated cache. This 3-level organization of storage (register file, LS, main storage), with asynchronous DMA transfers between LS and main storage, is a radical break from conventional architecture and programming models because it explicitly parallelizes computation with the transfers of data and instructions that feed computation, and stores the results of computation in main storage.

An SPE controls DMA transfers and communicates with the system by means of channels that are implemented in and managed by the SPE's memory flow controller (MFC). The channels are unidirectional message-passing interfaces.

The PPE and other devices in the system, including other SPEs, can also access this MFC state through the MFC's memory-mapped I/O (MMIO) registers and queues, which are visible to software in the main-storage address space.

2.1.2.3 Element Interconnect Bus (EIB)

The EIB is the communication path for commands and data between all processor elements in the Cell/B.E. processor and the on-chip controllers for memory and I/O. The EIB supports full memory-coherent and SMP operations. Thus, a Cell/B.E. processor is designed to be grouped coherently with other Cell/B.E. processors to produce a cluster.

The EIB consists of four 16-byte-wide data rings. Each ring transfers 128 bytes (one PPE cache line) at a time. Processor elements can drive and receive data simultaneously. Figure 2-1 shows the unit ID numbers of each element and the order in which the elements are connected to the EIB. The connection order is important to programmers who are seeking to minimize the latency of transfers on the EIB. The latency is a function of the number of connection hops, so that transfers between adjacent elements have the shortest latencies, and transfers between elements separated by six hops have the longest latencies.

The internal maximum bandwidth of the EIB is 96 bytes per processor-clock cycle. Multiple transfers can be in process concurrently on each ring, including more than 100 outstanding DMA memory transfer requests between main storage and the SPEs in either direction. These requests might also include SPE memory to and from the I/O space. The EIB does not support any particular quality-of-service (QoS) behaviour other than to guarantee forward progress.

However, a resource allocation management (RAM) facility, shown in Figure 2-1, resides in the EIB. Privileged software can use it to regulate the rate at which resource requesters (the PPE, SPEs, and I/O devices) can use memory and I/O resources.

2.1.3 Improving Performance

The Cell/B.E. is mainly designed to improve performance in computation, but that improvement does not come up by magic. If a performance improvement is desired for some application, some changes to its structure must be made.

2.1.3.1 *Divide and Conquer (Parallelizing)*

The first thing to keep in mind is that SPEs must be mainly used for computing-intensive tasks. The programmer must look for parts in the code responsible for this and see if it can be separated into "independent" tasks. For example, assuming that there is an integer array in the code with 8000000 positions where each position must be incremented by one, which solution is faster: do 8000000 iterations summing one in each position of the array or do the same operation in 1000000 iterations by summing 8 positions per iteration? Figure 2-2 and Figure 2-3 illustrate this example:

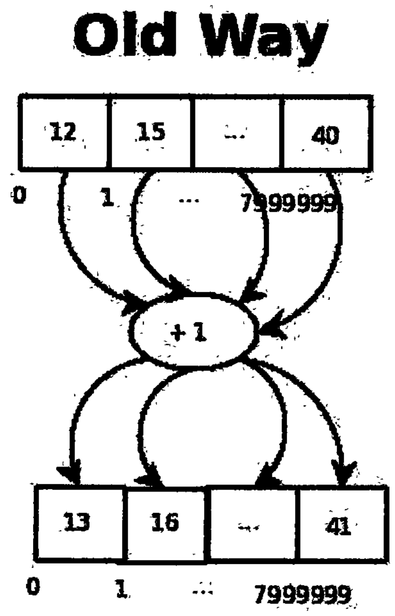


Figure 2-2 Processing a sum in a integer array with 8000000 positions

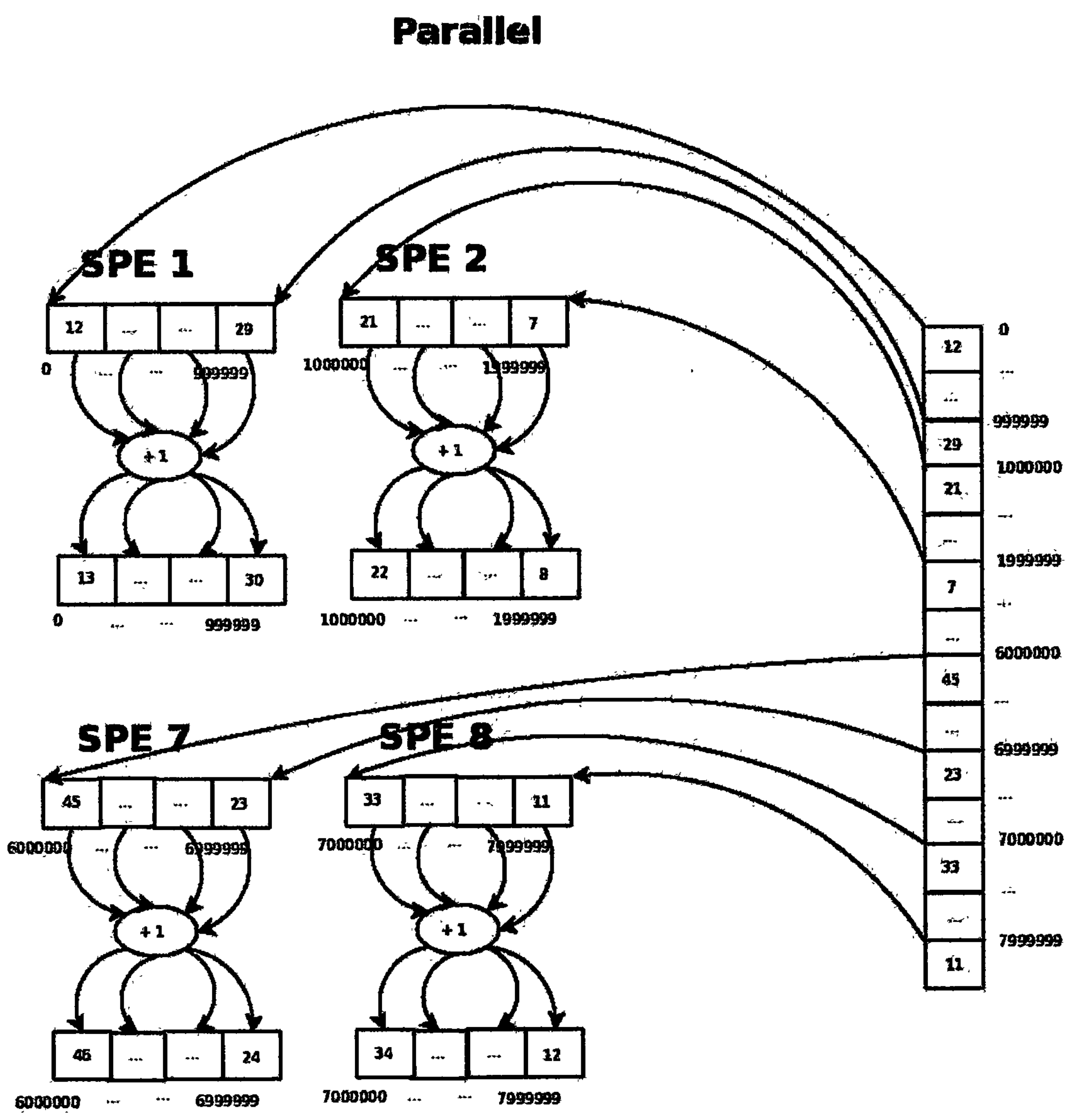


Figure 2-3 Processing the same sum as last figure, but in parallel

Figure 2-2 shows the "traditional" approach, which means incrementing the values in the array one-by-one. Figure 2-3 shows the computation for the same array spread inside 8 processors (only the first two and the last two SPEs are shown), and each one processes 1000000 positions.

Theoretically, the parallelized version will be 8 times faster since the 8000000 positions are computed in the same amount of time needed to compute 1000000 positions. This increase of performance is purely theoretical because Cell/B.E. requires some time to prepare the structures and start the parallelized threads (among other issues).

2.1.3.2 Compute as much as possible at a time (SIMDimizing)

Another improvement that Cell/B.E. offers is the ability to use SIMD (Single Instruction Multiple Data) instructions. SPEs have special vector registers with 128 bits length that can handle:

- Sixteen 8-bit values, signed or unsigned
- Eight 16-bit values, signed or unsigned
- Four 32-bit values, signed or unsigned
- Four single-precision IEEE-754 floating-point values
- Two double-precision IEEE-754 floating-point values

Basically, this means that each SPE can handle at least 2 pieces of data at a time.

2.1.3.2.1 Scalar VS Vector SIMD: A code example

SPEs can handle both vector and scalar codes, however vector code is a better solution because it can process 2 (or more) pieces of data at a time, and if we consider the parallelization of all the computation across 8 SPEs (for example), that means 16 pieces of data can be computed at a time (8 SPEs * 2 pieces of data).

The time needed to start a SPE thread and to prepare the structures needs to be taken into account, but using the full Cell/B.E. capabilities still measurably improves the performance of an application.

Example 2-1 shows a sample of a scalar C code which sums two arrays with four integers and puts the result on a third array:

Example 2-1 Summing 2 arrays with 4 elements (scalar version)

```
#include <stdio.h>
int main(void)
{
  int a[4] = {2, 2, 2, 2}, b[4] = {10, 20, 30, 40}, c[4], count;
  for (count = 0; count < 4; count++)
    c[count] = a[count] + b[count];
  printf("c[4] = {%d, %d, %d, %d}\n",c[0],c[1],c[2],c[3]);
  return 0;
}
```

The computation for this program needs four cycles to complete. Example 2-2 shows how to perform the same computation in a SPE with only one cycle:

Example 2-2 Summing 2 vectors with 4 elements (SIMD version)

```
#include <stdio.h>
#include <spu_intrinsics.h>
int main(void)
{
  vec_int4 a = {2, 2, 2, 2}, b = {10, 20, 30, 40}, c;
  c = spu_add(a,b);
  printf("c[4] = {%d, %d, %d, %d}\n",c[0],c[1],c[2],c[3]);
  return 0;
}
```

This program example produces exactly the same output as the one in the Example 2-1. The type **vec_int4** is used to declare vectors with four 32-bit signed integers, and the **spu_add** function is the intrinsic used which performs the assembler instruction responsible to add the four integers.

There is no direct support in C for the SPE instruction set, but a series of additional commands called "intrinsics" can be used (IBM 2007). Intrinsics allow the programmer to avoid writing assembler code and SPE C intrinsics are the ones most used to program for the SPEs. More details about C language intrinsics can be found in chapter 2.1.4.1.1 C-Language Intrinsics.

2.1.3.3 Use Time Wisely (Avoiding Stalls)

It is true that SPEs offer a great ability for computing-intensive tasks, but the programmer must know that the computed data is not available on-the-fly in the SPE. The data must be copied from the main store to the local SPE store via DMA transfers (see chapter 2.1.4.2 Different Processors, Different Address Spaces). What must be kept in mind in this sub-chapter is that this process of copying must be planned. If the computation is stopped because the SPE is waiting for the data, then we are not taking advantage of this architecture.

One possible approach is to apply for two sets of data, compute the first one arriving and once it is done, apply for the third set and start computing the second set (which must have arrived meanwhile). This process is called "Double Buffering" and is introduced in chapter 3.2.2.1 Double Buffering along with the practical work.

2.1.4 Programming Overview

Programming for Cell/B.E. can be quite challenging because its architecture requires a different way of thinking (Blachford 2006). One important thing to know is that while Cell/B.E. offers huge potential in computing performance, it does not come free. Do not expect to get a magic speedup from existing legacy code. If code optimized for a regular PowerPC processor (or any other compatible architecture) is just recompiled, it will only run on the PPE and may actually run slower.

Cell/B.E. is optimized for certain types of code and not everything will be able to take advantage of it immediately. The vast majority of code running on a Cell/B.E. is "control" code running inside the PPE, which does not involve any different or new paradigm in programming.

SPE development may be more complex, but performance sensitive code is usually only a tiny percentage of the code that runs. A lot of the code in an application is just glue code tying things together. An example for this is GUI (Graphical User Interface) code which mostly consists of calls to perform a function when a gadget is clicked. This is shown in chapter 3 Visualization in Cell/B.E. where the developed work is introduced. Most of the code is mainly responsible to guarantee that the input is correct (and launch exceptions in case it is not), in contrast with the computational part, which is only responsible for the computing intensive tasks.

2.1.4.1 Different Processors, Different Compilers

Because the existing differences between the PPE and the SPE, there was a need to create different compilers for these processors.

Like shown before in chapter 2.1.4 Programming Overview, PPE (and its compiler) can handle all the code optimized for regular architectures but, in order to improve performance, some of this code must be ported to the SPE side where it is treated differently.

It is true that the programmer can use the same programming language for the PPE and the SPEs (in this case, language C), but it does not mean that programming for both sides is the same thing. This happens because the SPEs are mainly optimized for SIMD instructions, in contrast to the PPE.

All the practical work introduced in chapter 3 Visualization in Cell/B.E. shows how to compile and run the code for the different implemented modules.

2.1.4.1.1 C-Language Intrinsics

The intrinsics are essentially in-line language instructions in the form of C-language function calls. They provide the programmer with explicit control over the Vector/SIMD and SPU instructions without directly managing registers.

In a specific instruction set, most intrinsic names use a standard prefix in their mnemonic, and some intrinsic names incorporate the mnemonic of an associated assembly-language instruction. For example, the Vector/SIMD that implements the add Vector/SIMD assembly-language instruction is named vec_add, and the SPU intrinsic that implements the stop SPU assembly-language instruction is named spu_stop.

The PPE's Vector/SIMD instruction set and the SPE's SPU instruction set both have extensions that define somewhat different sets of intrinsics, but they all fall into four types of intrinsics. These are listed in Table 2-1 (IBM 2007). Although the intrinsics provided by the two instruction sets are similar in function, their naming conventions and function-call forms are different.

Table 2-1 PPE and SPE intrinsic classes

Types of Intrinsic	Definition	PPE	SPE
Specific	One-to-one mapping to a single assembly-language instruction.	X	X
Generic	Map to one or more assembly-language instructions, depending on types of input parameters.	X	X
Composite	Constructed from a sequence of Specific or Generic intrinsics.		X
Predicates	Evaluate SIMD conditionals.	X	

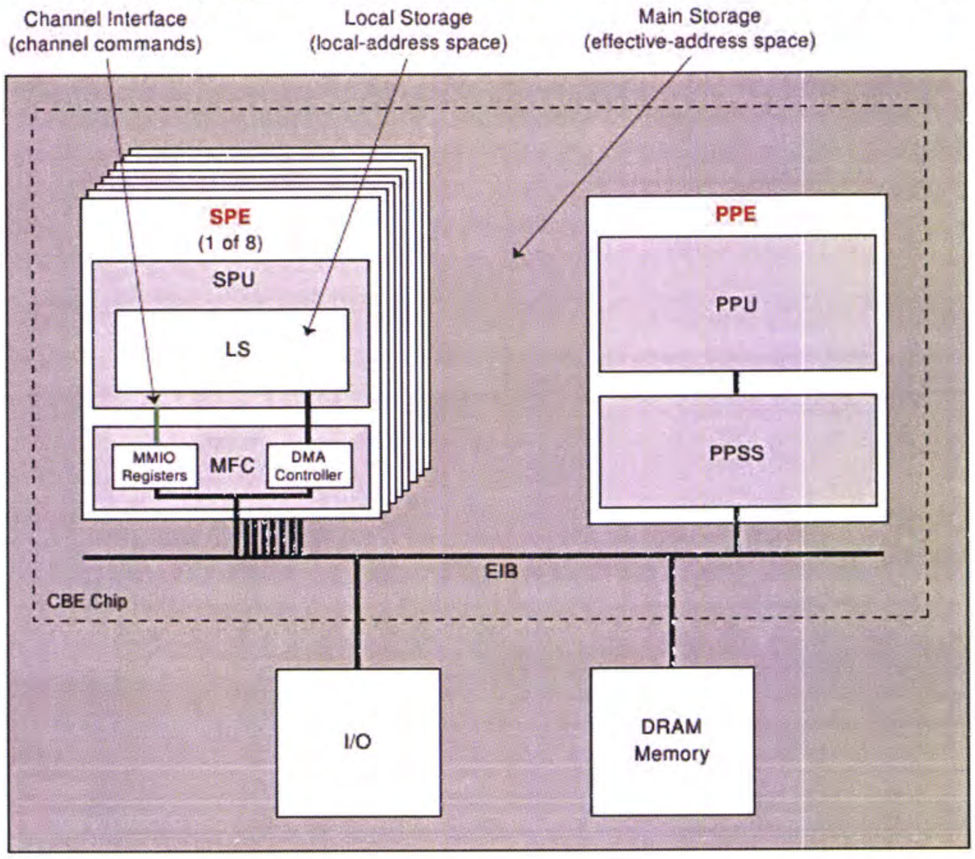
2.1.4.1.2 Porting SIMD code from the PPE to the SPEs

For some programmers, it is easier to write SIMD programs by writing them first for the PPE, and then porting them to the SPEs. This approach postpones some SPE-related considerations like dealing with the local store size, data movements, and debugging until after the port. The approach can also allow partitioning of the work into simpler (perhaps more digestible) steps on the SPEs.

Alternatively, experienced Cell/B.E. programmers may prefer to skip the Vector/SIMD Multimedia Extension coding phase and go directly to SPU programming. In some cases, SIMD programming can be easier on an SPE than the PPE because of the SPE's unified register file.

2.1.4.2 Different Processors, Different Address Spaces

Two important issues about the Cell/B.E. are its layout of the memory and its address space. Figure 2-4 shows the two existing types of storage: one main-storage domain and eight SPE LS domains:



Channel Interface (channel commands)
Local Storage (local-address space)
Main Storage (effective-address space)
SPE
(1 of 8)
SPU
LS
MMIO Registers
MFC
DMA Controller
PPE
PPU
PPSS
EIB
CBE Chip
I/O
DRAM Memory

DMA	Direct Memory Access	PPE	PowerPC Processor Element
EIB	Element Interconnect Bus	PPSS	PowerPC Processor Storage Subsystem
LS	Local Storage	PPU	PowerPC Processor Unit
MFC	Memory Flow Controller	SPE	Synergistic Processor Element
MMIO	Memory-Mapped I/O	SPU	Synergistic Processor Unit

Figure 2-4 Storage and domain interfaces

The main-storage domain, which is the entire effective-address space (EA), can be configured by PPE privileged software to be shared by all processor elements and memory-mapped devices in the system.

The layout shows that the SPE can access and modify the data contained in this area (main-storage) but, from the programmer's point of view, there are some aspects which the programmer must keep in mind:

- Data sharing and dependencies have to be carefully designed since all processors share the same main memory;
- References or pointers: a pointer passed from PPE to the SPE cannot just be dereferenced but has to involve a DMA transfer;
- Memory alignment: LS and main memory addresses must be aligned for DMA transfers;

- The Cell/B.E. Memory Flow Controller (MFC) supports naturally aligned transfer sizes of 1, 2, 4, or 8 bytes, and multiples of 16 bytes, with a maximum transfer size of 16KB;
- Peak performance can be achieved when both the EA and LSA (Local Storage Address) are 128-byte aligned and the size of the transfer is an even multiple of 128 bytes;
- DMA transfers should be SPE-initiated and be overlapped with computation (when possible) to avoid stalls;
- And, SPE's Local Store has only 256 KB for data and code.

Assuming that the programmer wants to improve performance (using Cell/B.E.) on some algorithm that is already optimized for a regular architecture, the second and third points can be quite challenging. The programmer, most probably, will have to change the structures and the algorithm in the computational part in order to take advantage of the Cell/B.E. Not only should the parallelization of it be considered, but also the vector SIMD capabilities.

Concerning the data sharing and dependencies, a taxonomy design is introduced in the chapter below to give a better idea about parallel programming.

2.1.4.2.1 Parallel Programming: A Taxonomy

In Patterns for Parallel Programming (Mattson, Massingill and Sanders 2004) there is a definition of a taxonomy of parallel programming models. First they define four "spaces" (described in Table 2-2) which the application programmer must visit.

Table 2-2 Four design spaces

Space	Description
Finding concurrency	Find parallel tasks and group and order them
Algorithm structure	Organize the tasks in processes
Supporting structure	Code structures for tasks and data
Implementation mechanisms	Low level mechanisms for managing and synchronizing execution threads as well as data communication

In the algorithm space it is proposed a look at three different ways of decomposing the work, each with two modalities. This leads to six major algorithm structures, which are described in Table 2-3.

Table 2-3 Algorithm structures

Organization principle	Organization subtype	Algorithm structure
By tasks	Linear	Task parallelism
	Recursive	Divide and conquer
By data decomposition	Linear	Geometric decomposition
	Recursive	Tree
By data flow	Linear	Pipeline
	Recursive	Event-based coordination

Task parallelism occurs when multiple independent tasks can be scheduled in parallel. The divide and conquer structure is applied when a problem can be recursively treated by solving smaller sub-problems. Geometric decomposition is common when a partial differential equation that has been made discrete on a 2-D or 3-D grid is tried to be solved, and grid regions are assigned to processors.

As for the supporting structures, Mattson et al. identified four structures for organizing tasks and three for organizing data. They are given side by side in Table 2-4.

Table 2-4 Supporting structures for code and data

Code structures	Data structures
Single Program Multiple Data (SPMD)	Shared data
Master/worker	Shared queue
Loop parallelism	Distributed array
Fork/join	

SPMD is a code structure that is well-known to MPI (Message Passing Interface) programmers. Although MPI does not impose the use of SPMD, this is a frequent construct. Master/worker is sometimes called "bag of tasks" when a master task distributes work elements independently of each other to a pool of workers. Loop parallelism is a low-level structure where the iterations of a loop are shared between execution threads.

Fork/join is a model where a master execution thread calls (fork) multiple parallel execution threads and waits for their completion (join) before continuing with the sequential execution.

Shared data refers to the constructs that are necessary to share data between execution threads. Shared queue is the coordination among tasks to process a queue of work items. Distributed array addresses the decomposition of multidimensional arrays into smaller sub-arrays that are spread across multiple execution units.

2.1.5 Summary

An overview of Cell/B.E. was shown in this chapter.

A new architecture concept with different processors specialized for different tasks and the existence of main-storage and local-storage are the most important concepts to keep in mind when working with Cell/B.E.

This different architecture made it possible to break three main performance walls: power, memory and frequency. The power wall is broken by using different cores specialized for different tasks (PPE handles control tasks and SPE handles compute intensive-tasks); the memory wall is broken by the new layout conception for the memory (main storage, local storage and larger register files in each SPE) and asynchronous DMA transfers between the main storage and local storage. This allows the PPE and SPEs to be designed for high frequency without excessive overhead, breaking the frequency limitation wall.

Nevertheless, with new concepts also new ways of programming come up. Because the deep differences between the PPE and the SPE, it was needed to create different compilers for the different processors. Also, the vector SIMD capabilities bring up a new way of thinking when programming, since more than one data can be handled at a time. This means that the improvement in performance does not come up by magic but instead an extra-effort is required to the programmer to deal with this new technology.

But, is there any way to make the programming for Cell/B.E. easier rather than harder and, at the same time, take profit of all its capabilities?

2.2 Visual Programming

It is well-known that conventional programming languages are difficult to learn and use. Programming for Cell/B.E. does not make that task easier at all as it requires skills that many people do not have (Lewis and Olson 1987). However, the number of applications that supports the act of programming by user interfaces is growing. For example, the success of spreadsheets can be partially attributed to the ability of users to write programs (as collection of "formulas").

It is known that new technology is developed really fast and if we are almost sure that Cell/B.E. is one of the best architectures made for computing intensive tasks, today, that may not be true tomorrow. If a new architecture arrives, most probably it will bring a new way of programming for it. It is probable that programmers will need to invest time to learn how to program for this new solution.

So, it would be good if we could find a way to make not only the act of programming easier, but also to avoid the need of learning how to program for new technology every time it arrives. One approach to this problem is to investigate the use of graphics as the programming language. This has been called "Visual Programming" (VP) or "Graphical Programming".

This chapter will give a background about what does VP consists of: overview, history, definitions, existing applications and differences.

2.2.1 Overview

There has been a great interest in systems that use graphics to aid in the programming, debugging, and understanding of computer programs. The terms "Visual Programming" and "Program Visualization" have been applied to these systems. Also, there has been a renewed interest in using examples to help alleviate the complexity of programming. This technique is called "Programming by Example" (Koelma, Balen and Smeulders 1992).

All these concepts have the main purpose to focus the programmer more into problem solving rather than just writing programs.

Visual programming has many advantages over textual programming. Their two-dimensional layout and use of icons seem to be closer to the human way of thinking (Smith, Pygmalion: a creative programming environment 1975). This simplifies the translation of the representation used in the mind, while thinking about a problem, to the representation used in programming a problem. The shorter translation distance makes visual languages easier to comprehend, learn, use, and remember. The use of pictorial elements is important because usually pictures convey more meaning than text: a single picture is often worth more than a thousand words. Furthermore, the two-dimensional layout facilitates the detection of potential concurrency in a program for parallel computation.

2.2.2 The origins of Visual Programming

Flowcharts are the first and best known diagrams of software. Goldstine (Goldstine 1972) claims he created the first flowchart for computers in 1947, while he was working with Von Neumann. Yet these early charts were entirely decoupled from the computer itself. It was not until the creation of graphic display technology in the 1960s that such a coupling became possible.

W. Sutherland, in 1966, created the first interactive visual programming language. Figure 2-5 (Curry 1978) shows his diagram for calculating a square root:

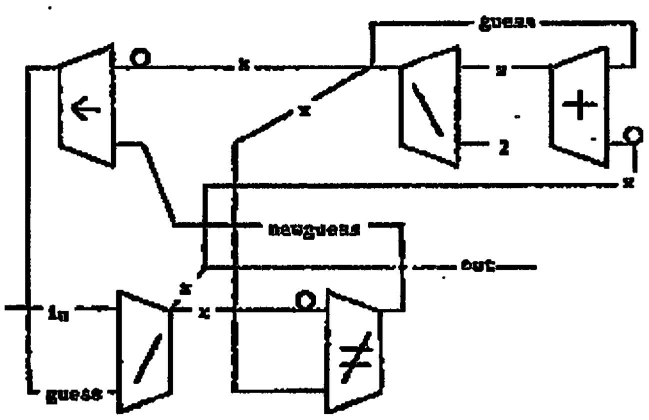


Figure 2-5 Sutherland's diagram for calculating a square root

Starting in the early 1970s, researchers at Xerox PARC created the first visual programming environments. Bitmapped graphics, mice, and window systems can be mainly credited to this research laboratory. The culmination of the work came in the form of Smalltalk, an operating system/programming environment/programming language (Goldberg and Robson 1983). The present graphic user interfaces differ little in concept from the Xerox PARC vision. In the 1980s, Apple Computer, Sun Microsystems, and M.I.T. (X Windows) spread graphic user interfaces to researchers and consumers; recently, the creation of working windowing systems for PCs (Microsoft, IBM, NeXT) has created a flurry of systems based on graphic user interfaces.

2.2.2.1 *The Visual Basic Phenomenon*

Visual Basic (today known as Visual Basic .NET (Wikipedia 2008)) is one famous programming language and is generally associated to a visual programming language. By definition, it is not a pure visual language. Instead of being based on diagrammatic representation, its underlying language is an enhanced textual version of the Basic language. In front of this textual language is a well-thought-out graphic user interface, which allows the programmer to construct windows and all their corresponding components such as buttons, slider bars, and menus by selecting graphic icons and dragging them onto a graphic representation of a window. The programmer then writes textual source code fragments that are essentially event handlers for the different possible mouse and keyboard events. This code is linked to the graphic representation of the window, so that instead of scrolling through long files of source, a programmer can access relevant code by clicking on a physical location. In other words, the interface provides, first of all, a way of constructing the framework of a user interface by manipulating graphic objects, and, second of all, a way of separating access to the textual code that needs to be written. Part of the success of Visual Basic is the flatness of the language - many verbs are provided, and many software vendors have been encouraged to create modules that add more verbs.

But, in spite of its success in the past, if Visual Basic was a pure visual programming language, it would not have the problem of code-refactoring with the existent last version of Visual Basic .NET (VB6). This happens because Visual Basic, in the end, is code based and its instructions can change with newer versions, forcing the programmer to change his/her code to keep applications running. The concept behind a pure Visual Programming does not involve code-refactoring at all, because there is no code, only pictures.

2.2.3 Definitions

To better understand what Visual Programming is it is necessary to know some related definitions in order to get a clearer picture.

2.2.3.1 Programming

A computer "program" is defined as "a set of statements that can be submitted as a unit to a computer system and used to direct the behaviour of that system" (Daintith 1983). While the ability to compute "everything" is not required, the system must include the ability to handle variables, conditionals and iteration, at least implicitly.

2.2.3.2 Interpretive VS Compiled

Any programming language system may either be "interpretive" or "compiled" (Free On-line Dictionary of Computing 2007). A compiled system has a large processing delay before statements can be run while they are converted into a lower-level representation in a batch fashion. An interpretive system allows statements to be executed when they are entered.

It takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total required to compile and run it. This is especially important when prototyping and testing code when an edit-interpret-debug cycle can often be much shorter than an edit-compile-run-debug cycle.

2.2.3.3 Visual Programming

The definition of Visual Programming (VP) is separated in two concepts: Visual Programming Language and Visual Programming Environment.

2.2.3.3.1 Visual Programming Language

"Visual Programming Language" (VPL) is any programming language that allows the user to specify a program in a two-(or more)-dimensional way. Conventional textual languages are not considered two-dimensional since the compiler or interpreter processes them as one-dimensional streams of characters. A VPL allows programming with visual expressions – spatial arrangements of textual and graphical symbols.

VPLs may be further classified, according to the type and extent of visual expression used, into icon-based languages, form-based languages and diagram languages.

2.2.3.3.2 Visual Programming Environment

"Visual Programming Environment" (VPE) is software which allows the use of visual expressions (such as graphics, drawings, animation or icons) in the process of programming. These visual expressions may be used as graphical interfaces for textual programming languages. They may be used to form the syntax of new visual programming languages leading to new paradigms such as programming by demonstration or they may be used in graphical presentations of the behaviour or structure of a program.

2.2.3.4 Program Visualization

"Program Visualization" (PV) is an entirely different concept from Visual Programming. In Visual Programming, the graphics are used to create the program itself, but in Program Visualization, the program is specified in a conventional, textual manner, and the graphics are used to illustrate some aspect of the program or its run-time execution (Myers, Taxonomies of visual programming and program visualization 1990). Unfortunately, in the past, many Program Visualization systems have been incorrectly labelled Visual Programming (Grafton and Ichikawa 1985). Program Visualization systems can be classified using two axes: whether they illustrate the code, data or algorithm of the program, and whether they are dynamic or static.

"Data Visualization" systems show pictures of the actual data of the program. Similarly, "Code Visualization" illustrates the actual program text by adding graphical marks to it or by converting it to a graphical form (such as a flowchart).

Systems that illustrate the "algorithm" use graphics to abstractly show how the program operates. This is different from data and code visualization, since with algorithm visualization the pictures may not correspond directly to data in the program and changes in the pictures might not correspond to specific pieces of the code. For example, an algorithm animation of a sort routine might show the data as lines of different heights, and swaps of two items might be shown as a smooth animation of the lines moving. The "swap" operation may not be explicitly in the code, however.

"Dynamic" visualizations refer to systems that can show an animation of the program running, whereas "static" systems are limited to snapshots of the program at certain points.

If a program created using Visual Programming is to be displayed or debugged, clearly this should be done in a graphical manner, which might be considered a form of Program Visualization. However, it is more accurate to use the term Visual Programming for systems that allow the program to be created using graphics, and Program Visualization for systems that use graphics only for illustrating programs after they have been created.

2.2.3.5 Example-Based Programming

A number of Visual Programming systems also use "Example-Based Programming". Example-Based Programming refers to systems that allow the programmer to use examples of input and output data during the programming process (Myers, Taxonomies of visual programming and program visualization 1990). There are two types of Example-Based Programming: "Programming by Example" and "Programming with Example".

Programming by Example refers to systems that try to guess or infer the program from examples of input and output or sample traces of execution. This is often called "automatic programming" and has generally been an area of Artificial Intelligence research.

Programming with Example systems, however, requires the programmer to specify everything about the program (there is no inferencing involved), but the programmer can work out the program on a specific example. The system executes the programmer's commands normally, but remembers them for later reuse. Halbert (Halbert 1984) characterizes Programming with Examples as "Do What I Did" whereas inferential Programming by Example might be "Do What I Mean".

2.2.4 Advantages of Using Graphics

Visual Programming and Program Visualization are very appealing ideas for a number of reasons. The human visual system and human visual information processing are clearly optimized for multi-dimensional data. Computer programs, however, are conventionally presented in a one-dimensional textual form, not using the full power of the brain. Two-dimensional displays for programs, such as flowcharts and even the indenting of block structured programs have long been known to be helpful aids in program understanding (Smith, Pygmalion: A Computer Program to Model and Stimulate Creative Thought 1977). A number of Program Visualization systems [(Myers, Chandhok and Sareen, Automatic data visualization for novice Pascal programmers 1988), (Myers, INCENSE: A system for displaying data structures 1983), (Baecker 1981) and (Brown and Sedgewick 1984)] have demonstrated that two-dimensional pictorial displays for data structures, such as those drawn by hand on a blackboard, are very helpful. Clarisse (Clarisse and Chang 1986) claims that graphical programming uses information in a format that is closer to the user's mental representations of problems, and will allow data to be processed in a format closer to the way objects are manipulated in the real world. It seems clear that a more visual style of programming could be easier to understand for humans, especially for nonprogrammers or novice programmers.

Another motivation for using graphics is that it tends to be a higher-level description of the desired actions (often deemphasizing issues of syntax and providing a higher level of abstraction) and may therefore make the programming task easier even for professional programmers. This may be especially true during debugging, where graphics can be used to present much more information about the program state (such as current variables and data structures) than is possible with purely textual displays. Also, some types of complex programs, such as those that use concurrent processes or deal with real-time systems, are difficult to describe with textual languages so graphical specifications may be more appropriate.

The popularity of "direct manipulation" interfaces (Shneiderman 1987), where there are items on the computer screen that can be pointed to and operated on using a mouse, also contributes to the desire for Visual Languages. Since many Visual Languages use icons and other graphical objects, editors for these languages usually have a direct manipulation user interface. The user has the impression of more directly constructing a program rather than having to abstractly design it.

2.2.5 Some principles for visual language design

Several principles can guide our search for the ideal visual programming language and enable us to compare and criticize existing languages. Some of these principles are:

- Give the visual programmer flexibility over issues and layout rather than forcing her or him into one fixed way of doing things. For instance, if all data lines must go from the top of the screen down, a programmer who thinks of data flowing left to right, or right to left, or even bottom to top, will be forced into an unnatural mode of thought.
- Visual programming languages should not rely too much on text but that does not mean abandoning all textual names. A visual program should not be a sequence of C statements with arrows between them.
- Coupled with the right use of text is the right use of graphics and colour. A visual language should enable the programmer to use colour meaningfully and to import his or her own icons to stand for the activity points. For instance, the programmer might want to link the data pathways as water pipes instead of wires.
- Windows pose problems beyond their mere proliferation. When a window refers to items in other windows, either by name or by visual means such as a line, it forces a person to somehow arrange the windows on the screen so as to see the larger picture. But windows usually overlap each other, obscuring part of the underlying diagram. A general principle would be to allow the visual programmer to control the level of detail in ways that permit him or her to ignore fine detail while at other times to see that detail. Moreover, seeing the detail should be permitted in a variety of ways that allow easier placement within the larger context.

In spite of all the principles when creating a visual language, there are still some limitations which must be considered. Deutsch once said something like:

"Well, this is all fine and well, but the problem with visual programming languages is that you can't have more than 50 visual primitives on the screen at the same time. How are you going to write an operating system?"

It points out the obvious density advantage of text. This barrier has become known as the "Deutsch Limit" (Wikipedia 2009), stated as:

"The problem with visual programming is that you can't have more than 50 visual primitives on the screen at the same time."

This is clearly a problem with visual representations. However, it is not immediately clear that a similar limit does not also exist in textual languages. Visual programming languages and textual languages have limitations, but that does not mean it cannot be worked out.

2.2.6 Application Examples

This chapter introduces some examples of visual programming applications/prototypes and its capabilities/limitations.

2.2.6.1 Prograph

Prograph is a visual, object-oriented, dataflow, multi paradigm programming language that uses iconic symbols to represent actions to be taken on data. Commercial Prograph software development environments such as Prograph Classic and Prograph CPX were available for the Apple Macintosh and Windows platforms for many years but were eventually withdrawn from the market in the late 1990s. Support for the Prograph language on Mac OS X has recently reappeared with the release of the Marten software development environment (Andescotia 2008).

Prograph introduced a combination of object-oriented methodologies and a completely visual environment for programming. Objects are represented by hexagons with two sides, one containing the data fields, the other the methods that operate on them. Double-clicking on either side would open a window showing the details for that object, for instance, opening the variables side would show class variables at the top and instance variables below. Double-clicking the method side shows the methods implemented in this class, as well as those inherited from the superclass. When a method itself is double-clicked, it opens into another window displaying the logic.

In Prograph a method is represented by a series of icons, each icon containing an instructions (or group of them). Within each method the flow of data is represented by lines in a directed graph. Data flows in the top of the diagram, passes through various instructions, and eventually flows back out the bottom (if there is any output).

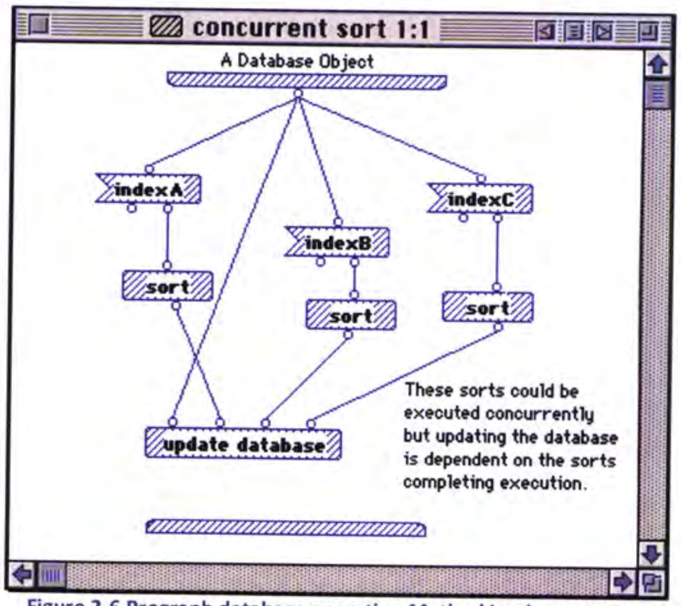

concurrent sort 1:1
A Database Object
indexA
indexB
indexC
sort
sort
sort
update database
These sorts could be executed concurrently but updating the database is dependent on the sorts completing execution.

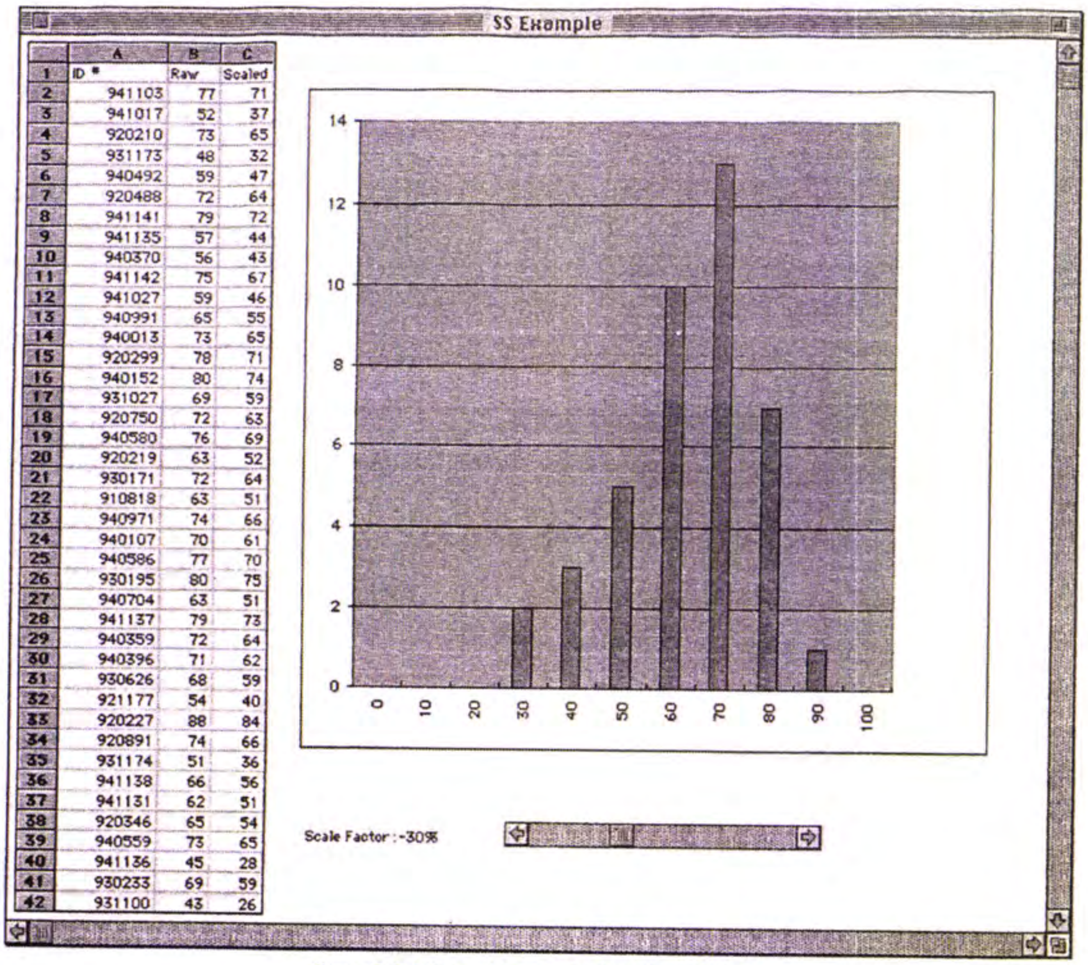
Figure 2-6 Prograph database operation Method implementation

2.2.6.1.1 Related Work

2.2.6.1.1.1 Spreadsheets

One of the primary uses of spreadsheets is in forecasting future events. This involves investigating "what-if" scenarios, experimenting with different values for inputs, and observing how they affect the computed values. Unfortunately, current spreadsheets provide little support for this type of interaction. Data values must be typed in, and computed values can be observed only as numbers, or on simple charts.

Smedley, Cox and Byrne [(Smedley, Cox and Byrne 1996) and (Cox and Smedley 1994)] proposed a system which allows the user to create interface objects which interact directly with values in a spreadsheet. For example, the user could create a spreadsheet where the results were displayed in a chart, and then create graphical controls, such as dials, or sliders, that are connected to the inputs of the computation, and then watch the effect on the chart as the inputs are varied by adjusting the scroll bar. The purpose is to enhance the value of spreadsheets for investigating "what-if" scenarios. This is achievable by implementing some extensions for Prograph. Figure 2-7 shows an example.



	A	B	C
1	ID #	Raw	Scaled
2	941103	77	71
3	941017	52	37
4	920210	73	65
5	931173	48	32
6	940492	59	47
7	920488	72	64
8	941141	79	72
9	941135	57	44
10	940370	56	43
11	941142	75	67
12	941027	59	46
13	940991	65	55
14	940013	73	65
15	920299	78	71
16	940152	80	74
17	931027	69	59
18	920750	72	63
19	940580	76	69
20	920219	63	52
21	930171	72	64
22	910818	63	51
23	940971	74	66
24	940107	70	61
25	940586	77	70
26	930195	80	75
27	940704	63	51
28	941137	79	73
29	940359	72	64
30	940396	71	62
31	930626	68	59
32	921177	54	40
33	920227	88	84
34	920891	74	66
35	931174	51	36
36	941138	66	56
37	941131	62	51
38	920346	65	54
39	940559	73	65
40	941136	45	28
41	930233	69	59
42	931100	43	26

Figure 2-7 A Spreadsheet with Interface Objects

2.2.6.1.1.2 Limitations

R. Mark Meyer and Tim Masterson (Meyer and Masterson 2000) discuss some limitations of Prograph and they implement an improved version from it called SIVL (SImple Visual Language).

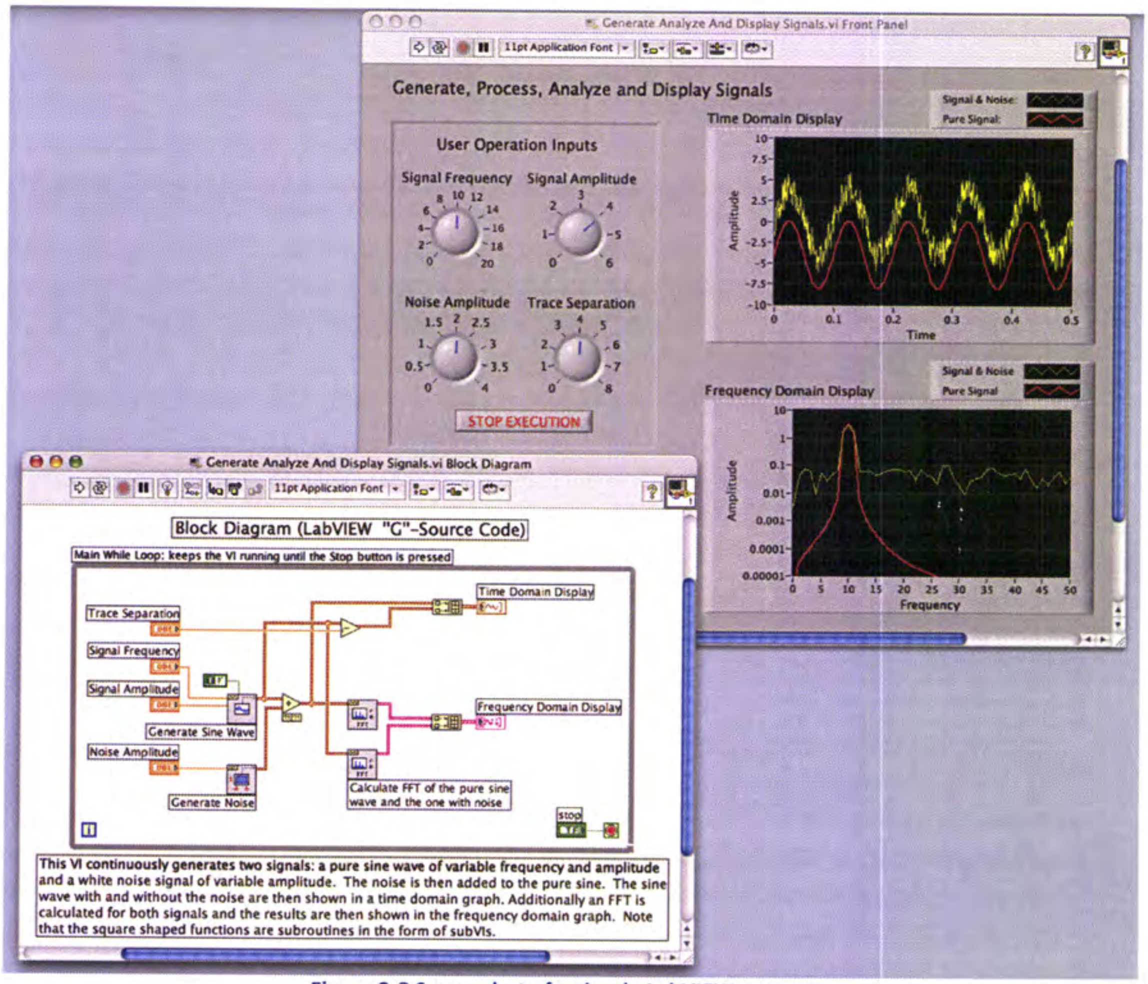
The main limitation is related with the size of the programs and its organization. One way to avoid having spaghetti code is by having windows referring items to other windows and hiding some structures, but this would bring the problem of not making the programmer see the "whole picture". For example, a chunk of visual code could be delineated by a box, sort of like an in-line function. This chunk could be the body of a loop or a case limb in a decision construct. If the chunk can be visualized as a black box whose inputs and outputs are the only concern, then the programmer can temporarily ignore its innards. But the interior must be dealt with sometime. It can be opened up as separate window, or it could be expanded in place. Prograph gives the programmer only the first option of opening a separate window, making it difficult to see the chunk of code in context. A better approach would be to allow the programmer to make the interior visible as the chunk sits in the larger program, perhaps in muted colours or as smaller icons and lines.

2.2.6.2 LabVIEW

LabVIEW (short for Laboratory Virtual Instrumentation Engineering Workbench) is a platform and development environment for a visual programming language from National Instruments. The graphical language is named "G". Originally released for the Apple Macintosh in 1986, LabVIEW is commonly used for data acquisition, instrument control, and industrial automation on a variety of platforms including Microsoft Windows, various flavours of UNIX, Linux, and Mac Os. The latest version of LabVIEW is version 8.6, released in August of 2008 (Wikipedia 2008).

LabVIEW ties the creation of user interfaces (called front panels) into the development cycle. LabVIEW programs/subroutines are called virtual instruments (VIs). Each VI has three components: a block diagram, a front panel, and a connector pane. The last is used to represent the VI in the block diagrams of other, calling VIs. Controls and indicators on the front panel allow an operator to input data into or extract data from a running virtual instrument. However, the front panel can also serve as a programmatic interface. Thus a virtual instrument can either be run as a program, with the front panel serving as a user interface, or, when dropped as a node onto the block diagram, the front panel defines the inputs and outputs for the given node through the connector pane. This implies each VI can be easily tested before being embedded as a subroutine into a larger program.

Figure 2-8 shows a screenshot of a simple LabVIEW program that generates, synthesizes, analyzes and displays waveforms, showing the block diagram and front panel. Each symbol on the block diagram represents a LabVIEW subroutine (subVI) which can be another LabVIEW program or a LV library function.



This VI continuously generates two signals: a pure sine wave of variable frequency and amplitude and a white noise signal of variable amplitude. The noise is then added to the pure sine. The sine wave with and without the noise are then shown in a time domain graph. Additionally an FFT is calculated for both signals and the results are then shown in the frequency domain graph. Note that the square shaped functions are subroutines in the form of subVIs.

Figure 2-8 Screenshot of a simple LabVIEW program

2.2.6.2.1 Related Work

2.2.6.2.1.1 Survey

K. N. Whitley and Alan F. Blackwell presented a paper with three surveys of beliefs about the cognitive effects of visual programming (Whitley and Blackwell 2001). The first survey was aimed for the opinion of visual programming academic researchers and the second and third surveys were aimed to users of programming languages: professional programmers and LabVIEW programmers respectively.

The main conclusions are: students from the first survey revealed speculative, but optimistic views of visual programming; professional programmers were the most sceptical and LabVIEW programmers were confident that the visual programming language provided by LabVIEW is beneficial.

Academic researchers often have optimistic theories regarding the influence that new programming languages can exert on the mental processes of the programmer.

Professional programmers, whether they are familiar with VPLs or not, tend to see the advantages of new languages in different terms - they are more concerned with the potential improvements in productivity that arise from straightforward usability issues, rather than from theories of cognition. The professional programmers exhibit a preference for the tools that they have had most experience of using.

This might produce significant biases when programmers are questioned about the value of their tools. These biases can even extend to significant scepticism about the advantages of new techniques, whether or not the programmer fully understands the technique being described.

However, experienced LabVIEW programmers admitted (when specifically prompted) the weaknesses of their tool. Equally as interesting is the result that LabVIEW programmers rated the visual representation of LabVIEW as more of an advantage that its reusability facilities.

2.2.6.2.1.2 Limitations

Some of the reported LabVIEW limitations (Wikipedia 2008) are:

- Small changes can pull complex restructuring. Always when a new VI is inserted, it is needed to reconnect all the "wires" and symbols in order to re-establish the system;
- Unlike common programming languages, LabVIEW is not managed or specified by a third party standards committee. Compiled executables produced by the Application Builder are not truly standalone since they also require that the LabVIEW run-time engine be installed on any target computer on which users run the application (National Instruments 2006). The use of standard controls requires a runtime library for any language and all major operating system suppliers supply the required libraries for common languages such as "C". However, the runtime required for LabVIEW is not supplied with any operating system and is required to be specifically installed by the administrator or user. This requirement can cause problems if an application is distributed to a user who may be prepared to run the application but does not have the inclination or permission to install additional files on the host system prior to running the executable;
- The simple entrance into LabVIEW programming tempts to "simply straight on programming", but also graphic programming does not replace the need to plan the project before starting to program.

2.2.6.3 Java Town

JavaTown project (Feinberg 2007) began as an attempt to address students misunderstanding, in introductory courses of Computer Science, about the nature of object references (pointers) in Java.

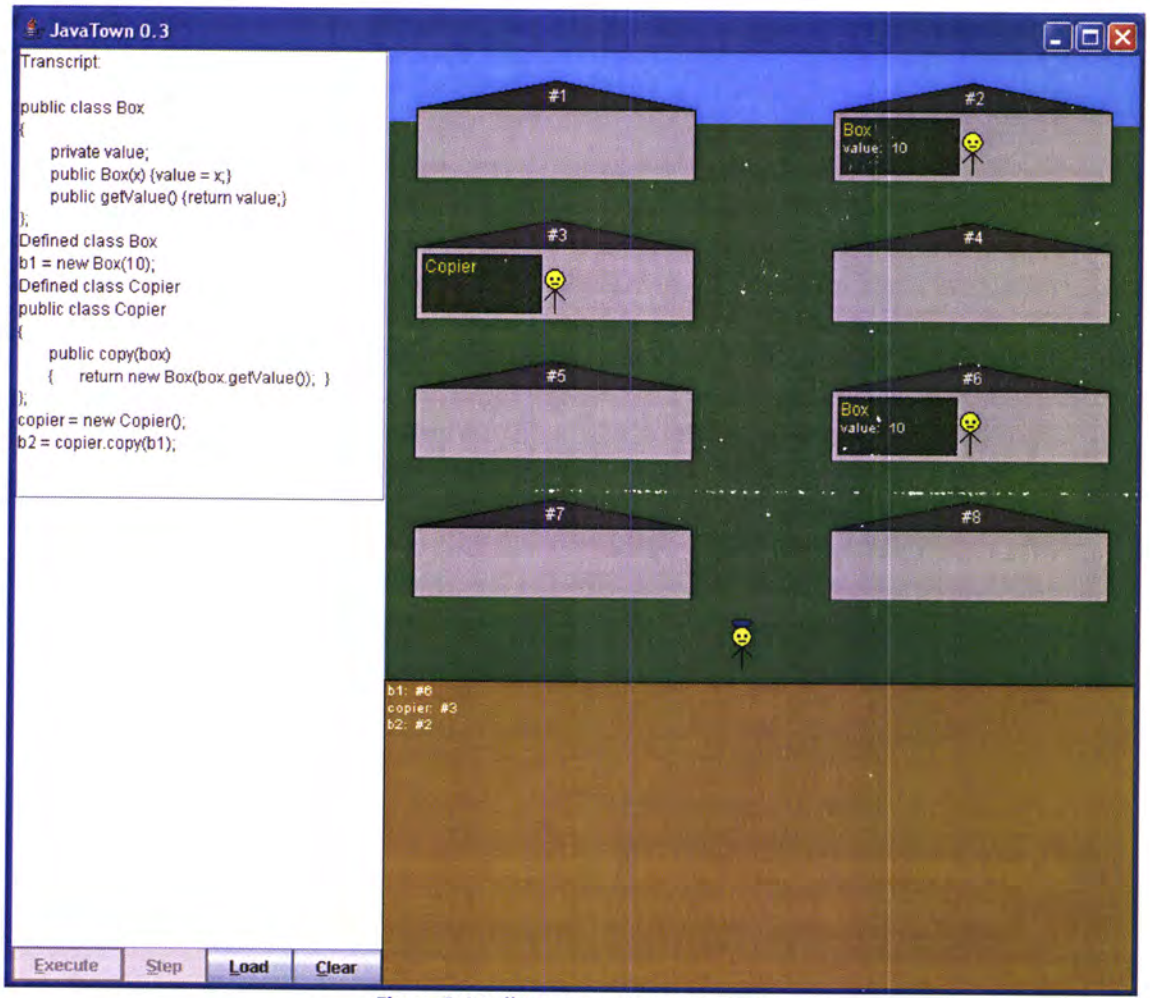
Teachers started to notice this problem when students showed difficulties in assigning objects to variables and passing them as method arguments. Usually, the approach to this kind of situations (since an object reference is simply a memory address) is to show the students how objects and references are actually represented in RAM, but this proved to be too technical and involved for an introductory course.

The solution found was to develop a visual programming environment in which characters would be shown performing computations – a sort of object oriented Little Man Computer (Madnick 1993).

The model is based on metaphors, which is most popular among the programmers, such as sending messages and remembering values. Objects are represented as people who live in numbered houses where each home address mirrors an object's address in RAM. These people are exclusively referred by their home address which is remembered by people and passed in messages to other people in the same manner that pointer addresses are used by conventional programming languages. Although students see addresses on screen, they quickly discover that such addresses never actually appear directly in code. The execution of the code grows out of the collective interaction of the people living in a group of houses, which is JavaTown.

The software was developed to bring JavaTown model to life. The programs executed by it are not actually written in Java. Instead, the software uses a language featuring only some of Java's syntax and behaviours, in order to prepare students to use Java later in the course.

JavaTown is seen as a prototype to develop more comprehensive and production-quality visual programming environments.



```
Transcript:

public class Box
{
     private value;
     public Box(x) {value = x;}
     public getValue() {return value;}
};
Defined class Box
b1 = new Box(10);
Defined class Copier
public class Copier
{
     public copy(box)
     {     return new Box(box.getValue());  }
};
copier = new Copier();
b2 = copier.copy(b1);
```

```
b1: #6
copier: #3
b2: #2
```

Figure 2-9 Full JavaTown Screen Layout

2.2.6.3.1 Limitations

The negative responses came from more experienced programmers, who felt that three weeks of JavaTown was too much and were frustrated by JavaTown's restrictive and type-free syntax. A number of students also suggested that JavaTown's slow execution speed to be adjustable. Another problem is related with sometimes vague error messages, especially the ones regarding to parse errors.

2.2.6.4 Device Independent Generation of User Interfaces

The promise of information anytime and anywhere has become a reality. Today, it is possible to access information through multiple kinds of devices. However, the design of user interfaces for such devices is restricted to the use of specific ad- hoc programming languages that may vary from one device to the other. In this sense, the existence of generic programming languages for creating device-independent user interfaces became a strong necessity.

One emerging approach to device independent developments require the construction of generic vocabularies for transcoding into specific target codes for web browsers, PDAs, voice systems, mobile phones, etc. The authors of this work (Mayora-Ibarra, et al. 2003) presented an authoring tool for designing generic user interfaces with automatic transcoding to multiple target languages. The tool is a visual programming environment with drag and drop generic widgets created in UIML and transcoded into VoiceXML, J2ME, HTML and WML languages. This tool takes advantage of the UIML language and visual programming paradigms for providing flexibility, consistency and decrease in development time.

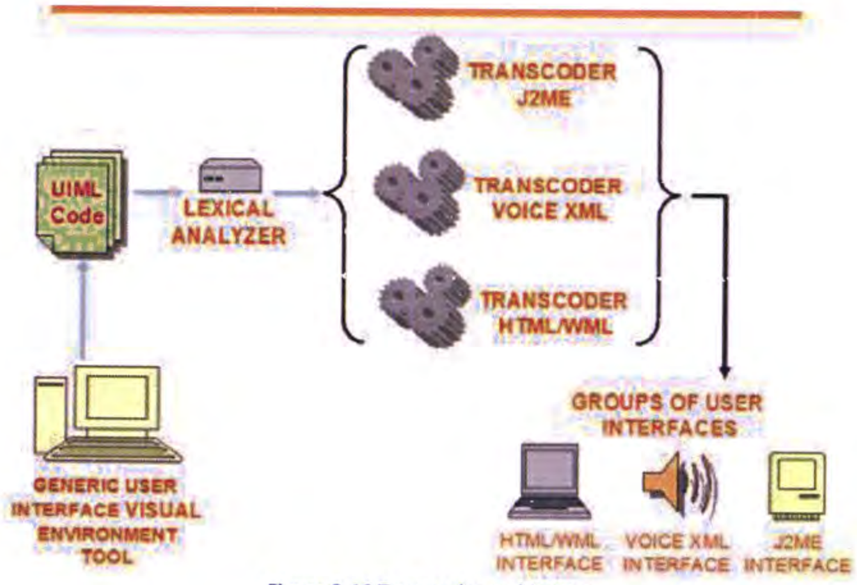


Figure 2-10 Transcoder architeture

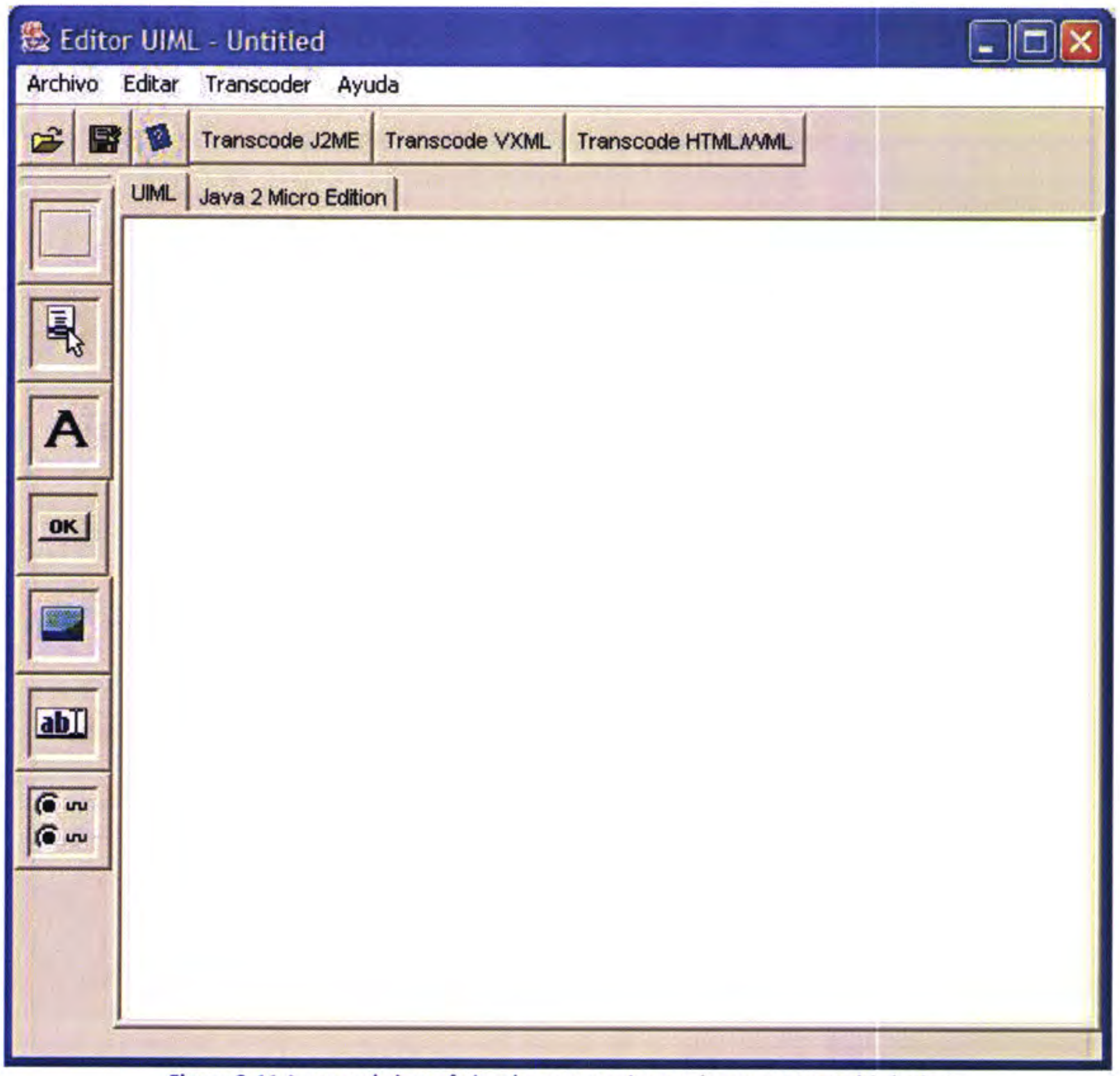


Figure 2-11 A general view of visual programming environment to multi-device

2.2.6.4.1 Limitations

The principal benefit of this approach consists on doing transcoders maintenance only and, if a new target language is needed, the programmers just have to create the respective transcoder.

There are a few visual programming environments to develop multi-devices interfaces, such as Harmonia Liquid (Harmonia, Inc. 2008). Harmonia have marked a good step with Liquid, but it has the inconvenience of multiple kinds of "generic language" to obtain a target language (Mayora-Ibarra, et al. 2003).

2.2.7 Summary

With the evolution of technology there is an increasing demand for software applications to work with this technology. There are lot of causes behind a demand for a new/changed software application: package update; new architecture; solving a problem; customer demands; etc.

Creating/changing a computer program often requires the act of programming and this task, beside the possibility of being cumbersome and time-consuming, may require experienced technicians.

Visual Programming came up as a possible solution for this. Its purpose is to ease up the way of programming which can save time and may allow end-users to create/change their programs.

Several principles can guide our search for the ideal visual programming language and enable us to compare and criticize existing languages. Some of these principles are analogous to those of current text-based programming languages like, for example, the need for restricting the set of possible structures to avoid spaghetti code.

It is possible to find around some Visual Programming applications in the market. Some of them are free and open-source and some others are close-source and commercial.

Anyhow, there is a long way to go for the proliferation of Visual Programming. Text-code based programming is still the preference among most of the programmers and some end-users find Visual Programming difficult and not intuitive. One solution can use a mix solution, like the one offered by Microsoft with Visual .NET, which combines visual and text code. Words exist for some reason and we cannot pretend that we can live only with pictures. Text programming offers more flexibility and Visual Programming offers more intuition so, why not combine both?

2.3 OpenDX

Now that the capabilities of Cell/B.E. and Visual Programming were introduced (see chapters 2.1 Cell / Heterogeneous Multi-core Environment and 2.2 Visual Programming), it is time to introduce the platform chosen to connect these two concepts: Open Visualization Data Explorer (DX).

DX sounded a good candidate for Cell/B.E. for three main reasons:

- has a Visual Programming Environment
- handles huge data-sets
- new modules for it can be implemented in C language

If a user has a data-set and wants to apply a complex operation on it in order to try to visualize something different, then DX shows up as a possible solution. It is used in many different areas and is possible to find, among the DX community (IBM 1991), projects related with: weather forecasting; physics and mathematics; oceanography; geographic information systems; data mining; chemistry; biology; etc. These researchers can take much profit from the power of High Performance Computing (HPC) since they can get their computing results much faster if Cell applications run with success in DX.

This chapter introduces the main topics about this application: overview; visual program editor; data visualization; module builder and installation steps in a Cell/B.E.

2.3.1 OpenDX Overview

Open Visualization Data Explorer (DX) was introduced by IBM Research in 1991 and it is open-source software. DX is a portable, general-purpose software package for data analysis and visualization and uses a Graphical User Interface based on X windows and Motif.

The main idea behind the creation of DX was to try to give meaning to data. If a user has a data-set and wants to see beyond its values, then DX comes up as a possible solution. DX allows users to see data in different colours, shapes, layouts, etc. Figure 2-12 shows a diagram about how does DX works:

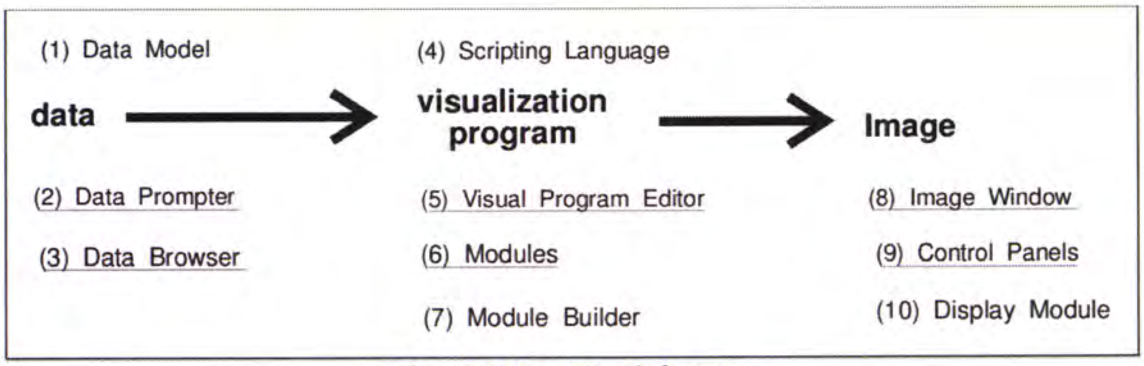


Figure 2-12 OpenDX main features

The Data Model (1), Visual Program Editor (5), Module Builder (7), Image Window (8), Control Panels (9) and Display Module (10) are introduced in this chapter. The last three topics are introduced in the sub-chapter 2.3.4 Data Explorer Visualization.

Details about the Modules (6) are presented in chapter 3.1 Writing Modules for OpenDX.

The remaining topics (2 to 4) will not be exposed in detail since they are not directly related with the proposed work.

2.3.2 Data Model

The most important thing to know about DX's Data Model is that it is based in a Field structure. A Field has three basic components: positions, connections and data.

The "positions" component is an Array Object specifying a set of *n*-dimensional positions.

The "connections" component provides a means for interpolating data values between the positions.

The "data" component stores the user's data values. The data values can be position or connection dependent. If the values are position-dependent, then the "connections" component supplies a means of interpolating data values between the samples. If the values are connection-dependent, the data value is constant for each interpolation element. Figure 2-13 shows an example of a Field object:

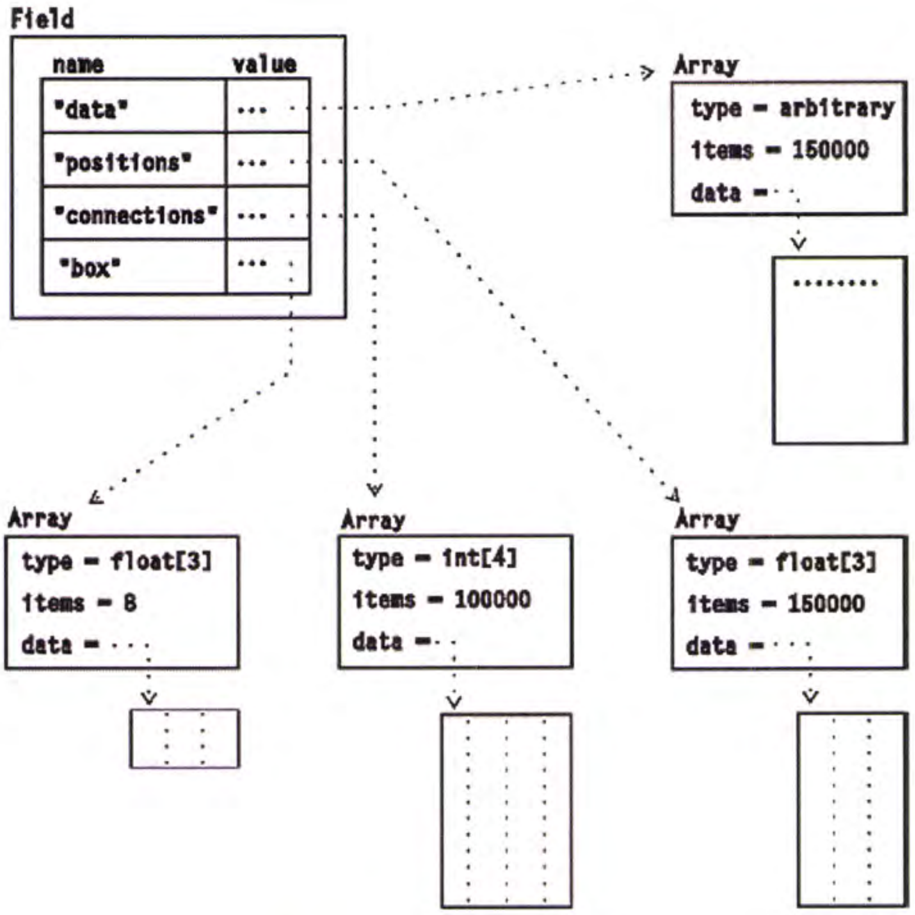
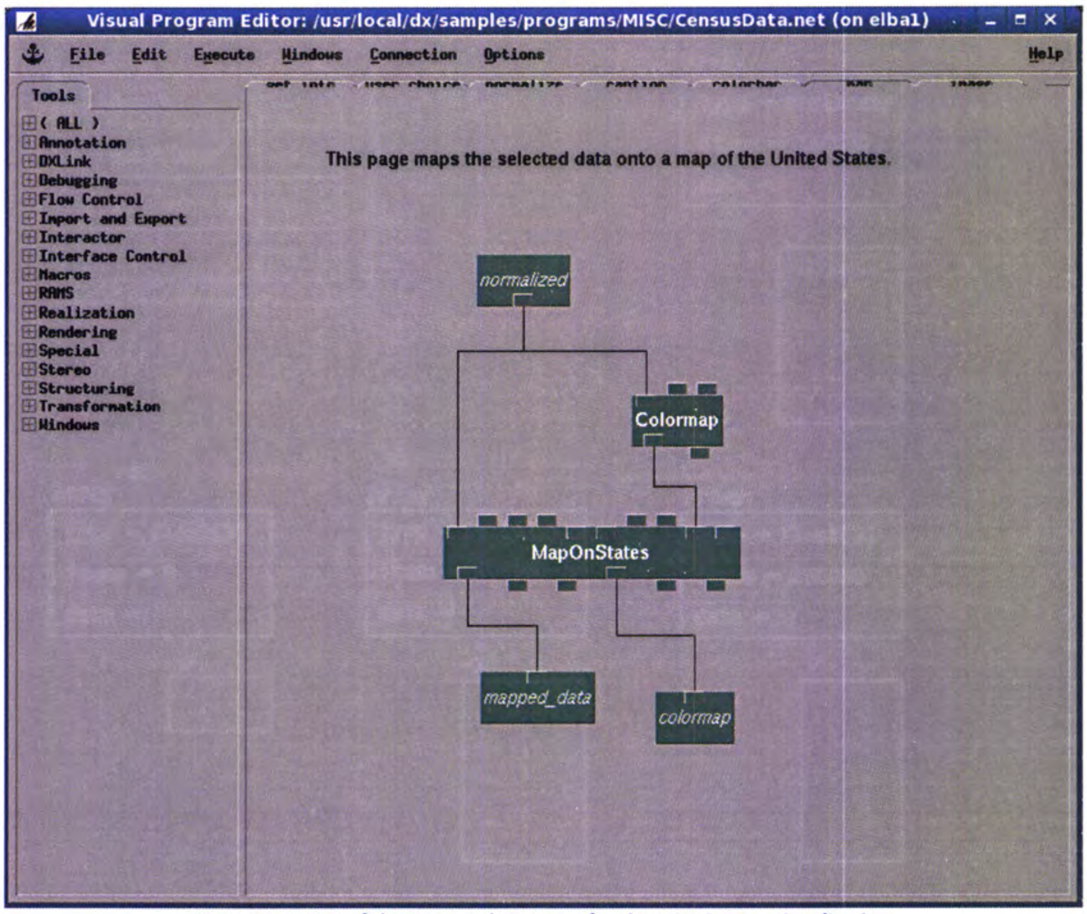


Figure 2-13 Example of a Field Object

The *Construct* module in DX allows the user to create a Field (very easily and straightforward) and it is introduced with the first practical example in chapter 3.2.2.4 Compiling and Running Add.

2.3.3 Visual Program Editor

The Visual Program Editor allows the users to create their Visual Programs. Figure 2-14 shows a program sample in the Visual Program Editor:


Visual Program Editor: /usr/local/dx/samples/programs/MISC/CensusData.net (on elba1)

This page maps the selected data onto a map of the United States.

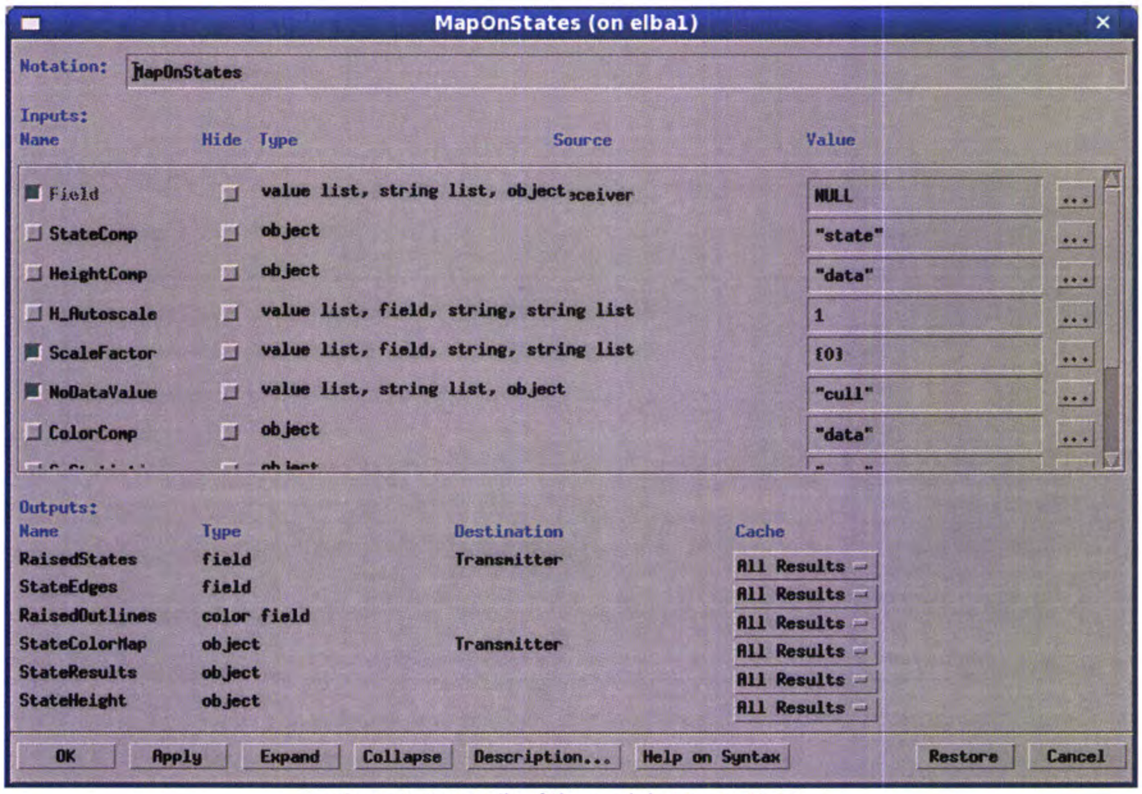
Figure 2-14 Part of the DX visual program for the USA Census visualization

In the left is a "Tools" menu which allows the user to select the modules to connect. In the middle is a page used to connect the modules and the menu is in the top.

It is possible to have more than one page in the visual program and these pages are separated by different tabs and when that happens, the module referencing a different page has italic fonts and thinner characters. The modules used for these operations are called *Transmitter* and *Receiver*. In the example of Figure 2-14, the *normalized* module is a *Receiver*, because this module "receives" information from another page, and the module *colormap* is a *Transmitter*, because it "transmits" information to another page.

Each module can be seen as a "function call". The end-user just give the input to the module and it will produce an output which can be used as input to another module and so on, but, instead of writing code, the user has only to deal with graphics.

Left-clicking two times in a module opens a dialog box where the user can manage its details. Figure 2-15 shows the details of the module *MapOnStates* presented in the Figure 2-14.



Notation: MapOnStates

Inputs:

Name	Hide	Type	Source	Value
■ Field	☐	value list, string list, object	ceiver	NULL
☐ StateComp	☐	object		"state"
☐ HeightComp	☐	object		"data"
☐ H_Autoscale	☐	value list, field, string, string list		1
■ ScaleFactor	☐	value list, field, string, string list		{0}
■ NoDataValue	☐	value list, string list, object		"cull"
☐ ColorComp	☐	object		"data"

Outputs:

Name	Type	Destination	Cache
RaisedStates	field	Transmitter	All Results
StateEdges	field		All Results
RaisedOutlines	color field		All Results
StateColorMap	object	Transmitter	All Results
StateResults	object		All Results
StateHeight	object		All Results

Figure 2-15 Details of the module MapOnStates

Here the user can: type/pick up specific inputs for the module; choose the cache procedure for the output and see the module description.

2.3.4 Data Explorer Visualization

DX offers different and flexible ways to visualize the data through the use of: Image Window; Control Panels and Display Module. They are introduced in this sub-chapter.

2.3.4.1 Image Window

The Image Window is an interactive window for viewing and modifying the presentation of the image produced by a visual program. For example, Figure 2-16 shows an image with the distribution of the USA population in 1989 (census of 1990), produced by the visual program in Figure 2-14.

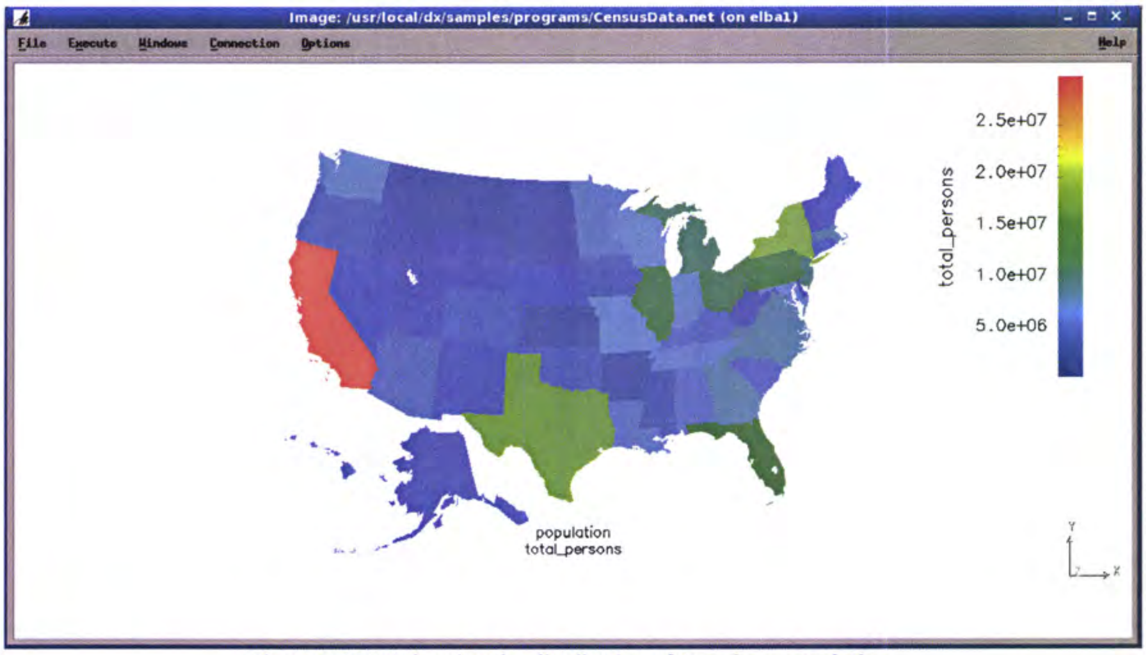


Image: /usr/local/dx/samples/programs/CensusData.net (on elba1)
File Execute Windows Connection Options Help
2.5e+07
2.0e+07
1.5e+07
1.0e+07
5.0e+06
total_persons
population
total_persons
Y
X

Figure 2-16 DX showing the distribution of American population

Image Window allows the user to control:

- the object's appearance
- the colour(s) of an object
- the placement of axes around an object

In order to be able to use such controls, the visual program created must use the *Image* module for rendering.

2.3.4.2 Control Panels

The Control Panels allow the user to change the parameter values used by a visual program. Figure 2-17 shows the Control Panel for the Census program which have been shown along this chapter.

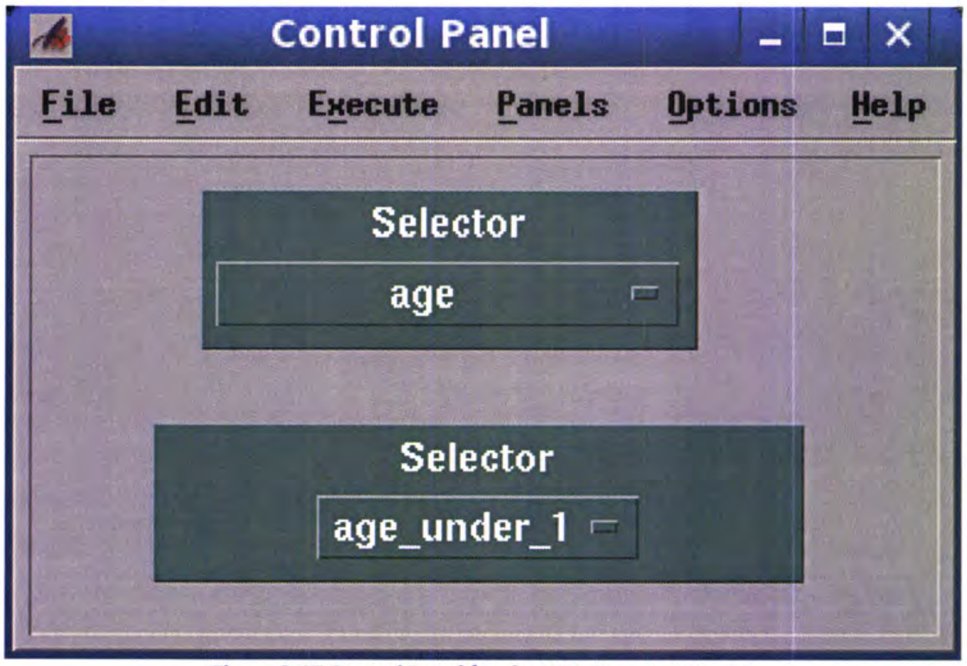


Figure 2-17 Control Panel for the USA Census program

In this example, the user can select, from the top selector, which type of information he/she wants to see: age (selected), population, sex, income, etc., and, in the selector below, can be selected information according to the first selector. In this case, because age is selected, the options available are: age_under_1; age_30_to_34 and age_over_85. Image displayed in Figure 2-16 will change according to the selected inputs in the Control Panel.

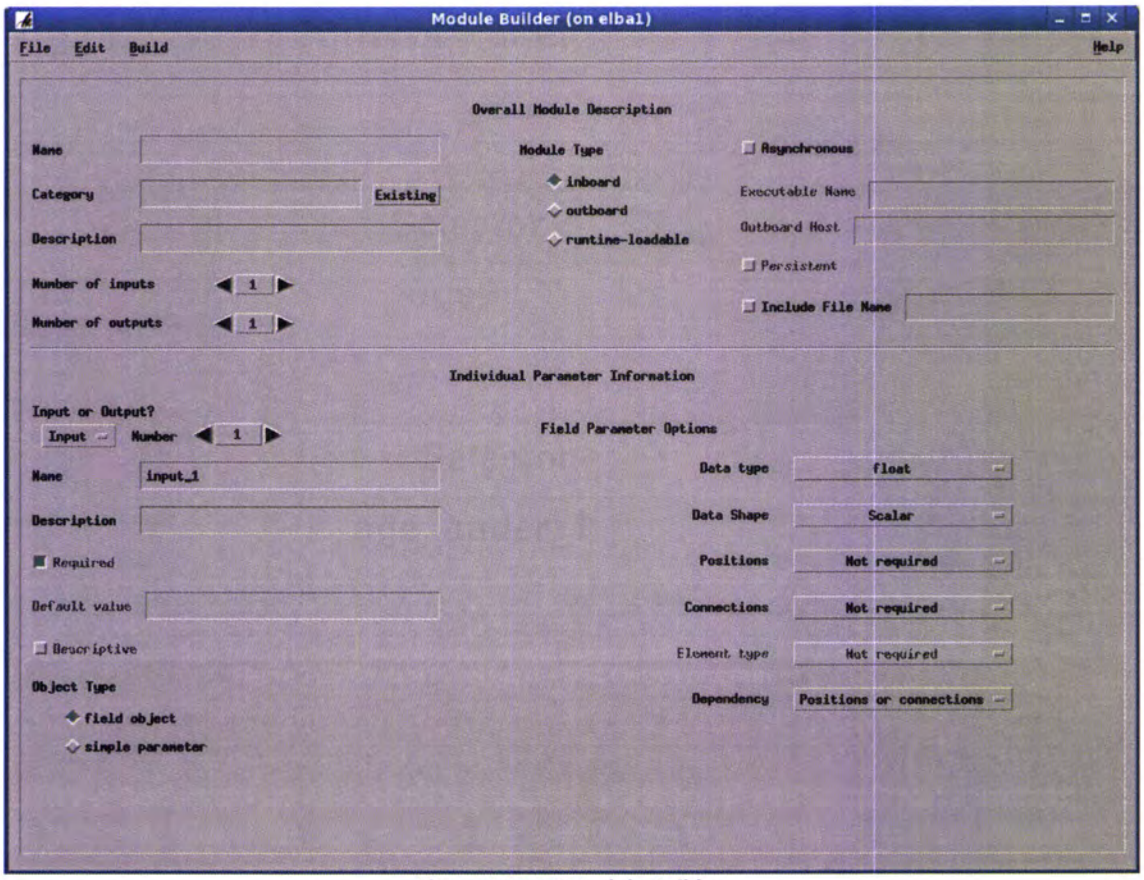
2.3.4.3 Display Module

Display Module is an alternative to the Image Window (see chapter 2.3.4.1 Image Window) and, like its name says, is used only to display images. This module does not offer the ability to "play" with the image like: rotating, zooming, moving; etc.

2.3.5 Module Builder

The Module Builder is a user interface which allows the creation of customized modules to be used in visual programs.

DX users have two main options when creating their own modules for their applications: they can use this module builder or they can program them on their own using C language. In fact, the builder does not do anything more than just generating C code according to its inputs.


Module Builder (on elbal)

File Edit Build Help

Overall Module Description

Name

Category Existing

Description

Number of inputs 1

Number of outputs 1

Module Type [✓ inboard] outboard runtime-loadable

Asynchronous

Executable Name

Outboard Host

Persistent

Include File Name

Individual Parameter Information

Input or Output? Input Number 1

Name input_1

Description

[✓ Required]

Default value

Descriptive

Object Type [✓ field object] simple parameter

Field Parameter Options

Data type float

Data Shape Scalar

Positions Not required

Connections Not required

Element type Not required

Dependency Positions or connections

Figure 2-18 DX Module Builder

Figure 2-18 show its interface. The *Build* menu on the top allows the user to generate the module description file (mdf), C code and a Makefile for the module (after typing the necessary information).

Module Builder proved to be a good tool in the beginning since it made it easier to understand the underlying DX structures, but it has some limitations about the specificity of the modules. So the best option, for advanced programmers, is to write new modules on their own (see chapter 3.1 Writing Modules for OpenDX). Examples using DX Module Builder are shown in chapter 3.2 Cell/B.E. Applications in OpenDX with more detailed explanation about how to use it.

2.3.6 OpenDX Installation in Cell/B.E.

This practical work of this thesis was developed using OpenDX v4.4.4 in a Fedora-Core based Linux. Besides a Fedora-Core minimal installation (with an X-server), the following packages must be installed:

- mesa
- glibc-devel
- freeglut-devel
- lesstif
- sqlite-devel
- expat-devel
- numatcl-devel
- libXp-devel

- libXext-devel
- libX11-devel
- libSM-devel
- libICE-devel

It may also be required for the user to create an empty *sys.h* file in "/usr/include/linux". Compiler may complain about it in spite of not using it.

For the rest: "./configure CC=ppu32-gcc CXX=ppu32-g++"; "make" and "make install" should be enough to complete the DX installation.

3 Visualization in Cell/B.E.

Cell/B.E., Visual Programming and OpenDX: how do these three components get together? In this chapter all the practical work developed during this thesis is introduced. The main goal is to run Cell/B.E. applications in OpenDX and apply some data visualization on it (DX purpose). DX modules are written in C language with the help of its API and development library (IBM 1991).

This chapter first starts to introduce how to write modules for DX and then introduces the implemented modules. Each module includes a performance study where the runtime for each application is compared between running in the console and running inside DX. The purpose is to show that there is no big loss of performance by switching from textual to visual programming.

Then, at last, a case study about using the Cell/B.E. capabilities to improve DX performance is introduced. The module picked up for this study is the *Gradient* module and its performance study compares its run-time without any optimizations with an optimized version (using all the available SPEs). Tests are performed in a QS22 Blade, which has 2 PPE cores (3.2 GHz each one) and 8 SPE cores per PPE, and in a Playstation3, which has one PPE core (3.2 GHz too) and 8 SPE cores. Both architectures offer 512Kb of L2 cache for the PPE and 256Kb of local store memory per each SPE.

3.1 Writing Modules for OpenDX

After DX is installed, the next step for the proposed work is to write Cell/B.E. applications in form of DX module. *DX Programmer's Reference* (IBM 1991) helps to achieve this with the help of tutorials and an API.

There are two available ways to write the modules: using DX Module Builder or writing the modules manually in C language.

DX Module Builder (see chapter 2.3.5 Module Builder) is recommended for beginners in DX development. It has a user interface and the user/programmer can specify the inputs/outputs for the new module, its name and description. The builder can generate the corresponding C code for the module, its corresponding *module description file* (mdf – see chapter 3.1.2 Module Description File) and a makefile.

However, advanced programmers may feel more comfortable writing these modules manually. Manual programming gives more flexibility to the programmer and it is possible to specify things about the modules which are not possible using the DX Module Builder.

Creating a module for DX involves three steps (does not matter if using DX Module Builder or not):

- Define what is the input/output of the module
- Create (or generate) a *module description file* (mdf)
- Write (or generate) the module in C language

These steps are described during this sub-chapter.

3.1.1 Defining Input/Output

First step to do when it is pretended to write a module for DX is to clarify which is the input and output of the module. This part is responsible for planning the behaviour of the module before starting to implement it.

The main questions in this phase are: what are the inputs? What are the types for each one? What will be the output(s) and its type(s)? It is desirable, at this phase, that the programmer has an idea about what the module is supposed to do and how its behaviour is.

3.1.2 Module Description File

The module description file (mdf) contains information about the modules which will be processed by DX. It contains information like: name of the module; description; flags; indication of outboard or loadable (if none, it is assumed that is inboard); inputs; options and outputs.

Its syntax is:

> **MODULE** *name*
> **CATEGORY** *category name*
> **DESCRIPTION** *module description*
> **FLAGS** *optional flags*
> **OUTBOARD** *"executable"; host*
> **LOADABLE** *"executable"*
> **INPUT** *name [visible]; type; default; description*
> **OPTIONS** *option1; option2; ...;*
> **OUTPUT** *name [cache]; type; description*
> **REPEAT** *n*

Table 3-1 shows a brief description of the components of an mdf:

Table 3-1 Module Description File (mdf)

NAME	REQUIRED	DESCRIPTION
MODULE	X	Assigns a name to the module being described.
CATEGORY	X	Assigns the module to a DX or user-defined category.
DESCRIPTION		Serves as a help function.
FLAGS		
OUTBOARD		Identifies the module as a separate executable program.
LOADABLE		Identifies the module as being runtime loadable (i.e., compiled separately and loaded into DX at run time).
INPUT	X	
OPTIONS		Identifies a list of possible values for the parameter. This list can be accessed by clicking on the "..." button to the right of the Value field in the module's configuration dialog box.
OUTPUT	X	
REPEAT		Specifies some number of INPUT or OUTPUT statements to be repeated.

For the **MODULE** and **CATEGORY**, the user has only to specify a string name which must start with a letter. In the **MODULE** case, it may be only a single alphanumeric word.

For the **INPUT** the syntax is as follows:

1. *name* (of a parameter) must be one word and must conform to the executive's lexical conventions.
   [visible] is optional. **visible:n** specifies the accessibility and initial visibility of input tabs:
   0: Not initially visible.
   1: Initially visible (default).
   2: Not available to the user interface.
2. *type* specifies the type(s) of the input and is used for type matching in the Visual Program Editor (see chapter 2.3.3 Visual Program Editor). The valid types are:

camera	**integer list**	**scalar**	**value**
field	**matrix**	**scalar list**	**value list**
flag	**matrix list**	**series**	**vector**
group	**object**	**string**	**vector list**
integer			

   To specify more than one type, the word **or** is used as a separator.
   If the type of the input value is not explicit (e.g., a string without quotation marks or a vector without brackets), the user interface attempts to match the input against the type(s) specified in the **INPUT** statement. It reads from left to right and stops at the first successful match. For this reason, string should be specified last, because any series of characters can always be converted to a string by adding double-quotation marks.
3. *default* identifies the value to be used if none has been specified.
   By convention, parentheses identify a description of default behaviour rather than an actual value. If no default is applicable, (no default) is specified. If the parameter is required, (none) is specified. **NOTE:** this part is purely informal.
4. *description* should contain a short phrase describing the parameter.

For the **OUTPUT** there are only two small differences compared to the **INPUT**:

- there is no default option
- *cache* is optional. **cache:n** specifies the caching to be performed by the executive:
  0: Do not cache the output
  1: Cache all outputs (default)
  2: Cache the output from the last execution only.

In spite of the possibility of appearing in the code, the remaining components are not discussed in this thesis since they are not relevant for the proposed work. All implemented modules are inboard modules, which means that they are compiled and charged before starting DX. **OPTIONS** and **REPEAT** are superfluous components.

3.1.3 Implementing the Module

The procedure after creating the mdf is to implement the corresponding module in C. DX offers a library and an API to help in this task.

Below is a list of the basic steps when implementing a module for DX:

- DX header file must be included in order to use the library (usually: #include <dx/dx.h>)
- the "main" function has type *Error* and its name is preceded by "m_" (for example, if the name of the module is "Hello", then the file which implements this module must have a function of type *Error* called "m_Hello")
- the main function have two arguments:
  - First argument is a pointer of the type *Object* to the input

    - Second argument is a pointer of the type *Object* to the output
- If the module is computed with success then it must return "OK", otherwise "ERROR"

Implementation specific details are introduced within the case-study module.

3.1.4 Compiling and Running

In order to compile the modules there are two steps which must be taken:

- First, the C file with all the information about the modules in the mdf must be generated. In order to do that, the next command must be performed:

  $(BASE)/bin/**mdf2c** *name_of_the_file.mdf* > *name_of_the_file2.c*

    - the default BASE directory is /usr/local/dx
    - *name_of_the_file* refers to the input mdf and a *name_of_the_file2* is generated in C
- Second, the compile procedure is the same as compiling a regular C code. The generated C file (from the mdf) must be included and the flag -IDX must be added in order to access the DX library. Other libraries may be needed since the DX library is used (like libGLU) and since these programs use SPU binaries, the libraries libspe2 and libpthread also must be linked

To start the application, the user must type:

dx -mdf *name_of_the_file.mdf* -exec *./generated_executable*

This command starts DX with all the implemented modules by the programmer.

3.1.5 Hello World Example

This chapter shows how to implement a module which connects a string to the word "Hello" and then an example of usage in the Visual Program Editor of DX.

The steps are introduced as described in previous chapter 3.1 Writing Modules for OpenDX.

3.1.5.1 Hello Input/Output Definition

This module receives an input string, concatenates it to the string "Hello" and then returns the string "Hello" concatenated with the input string.

3.1.5.2 Hello mdf

The proposed mdf, according to the syntax in chapter 3.1.2 Module Description File, is the following one:

Example 3-1 Hello mdf

```
MODULE        Hello
CATEGORY      Greetings
DESCRIPTION   Prefixes "hello" to the input string
INPUT         value; string; "world"; input string
OUPUT         greetings; string; prefixed string
```

The proposed name for this file is *hello.mdf* and it is used to generate the C file with the modules information.

3.1.5.3 Hello Implementation

Now that the mdf is defined, the next step is to implement the module using the C language:

Example 3-2 Hello World Implementation

```
1  #include <dx/dx.h>
2  Error m_Hello(Object *in, Object *out)
3  {
4         char message[30], *greeting;
5         if(!in[0])
6               sprintf(message, "hello world");
7         else
8         {
9               DXExtractString(in[0], &greeting);
10              sprintf(message, "%s %s", "hello", greeting);
11        }
12        out[0] = DXNewString(message);
13        return OK;
14 }
```

Details regarding the implementation of a module were introduced in chapter 3.1.3 Implementing the Module.

If no argument is specified for **value** (see the mdf in chapter 3.1.5.2 Hello mdf), then **in[0]** is **NULL** and the default output ("hello world") is placed in **message**. If an argument is specified, a library routine (**DXExtractString**) extracts it from **in[0]** and **greeting** becomes a pointer to that string. In line 10, string pointed by **greeting** is appended to "hello", creating **message**.

This file is named as **hello.c** and next chapter introduces how to compile and run this example.

3.1.5.4 Compiling and Running the Hello Example

In order to compile and run the application the following commands are performed:

(assuming BASE = /usr/local/dx)

```
$(BASE)/bin/mdf2c hello.mdf > hello_mdf.c

ppu32-gcc -I /usr/local/dx/include hello_mdf.c hello.c –c
```

```
ppu32-g++ hello_mdf.o hello.o -L /usr/local/dx/lib_linux -lDX -lGLU –exportdynamic -o dxexec

dx -mdf hello.mdf -exec ./dxexec
```

Compiling and running topics are introduced in chapter 3.1.4 Compiling and Running.

As stated in chapter 2.1.4.1 Different Processors, Different Compilers, compiling a regular C program in the PPE can be done without changing code and compile flags.

3.1.5.5 *Visual Program Example Using Hello*

Figure 3-1 shows the main window when DX starts:

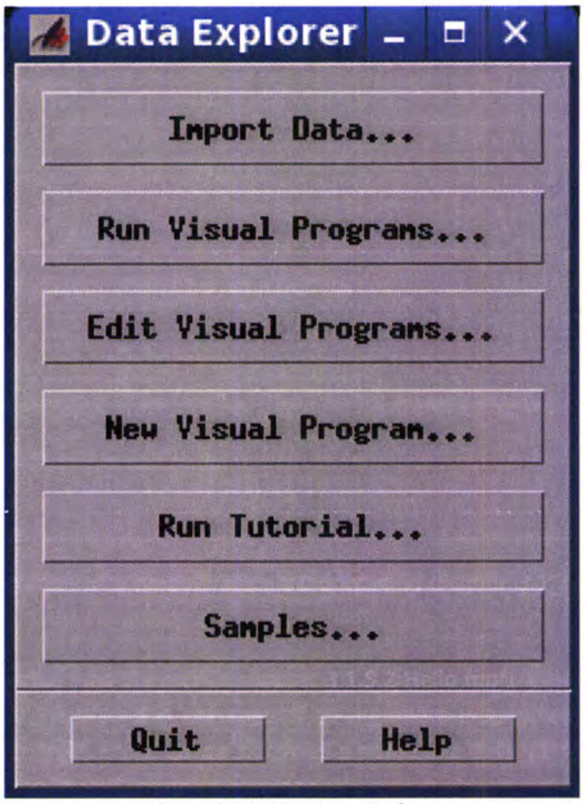


Figure 3-1 DX Startup window

Clicking on **Edit Visual Programs...** opens the Visual Program Editor where the user can connect the modules. Figure 3-2 shows a visual program using the *Hello* module:

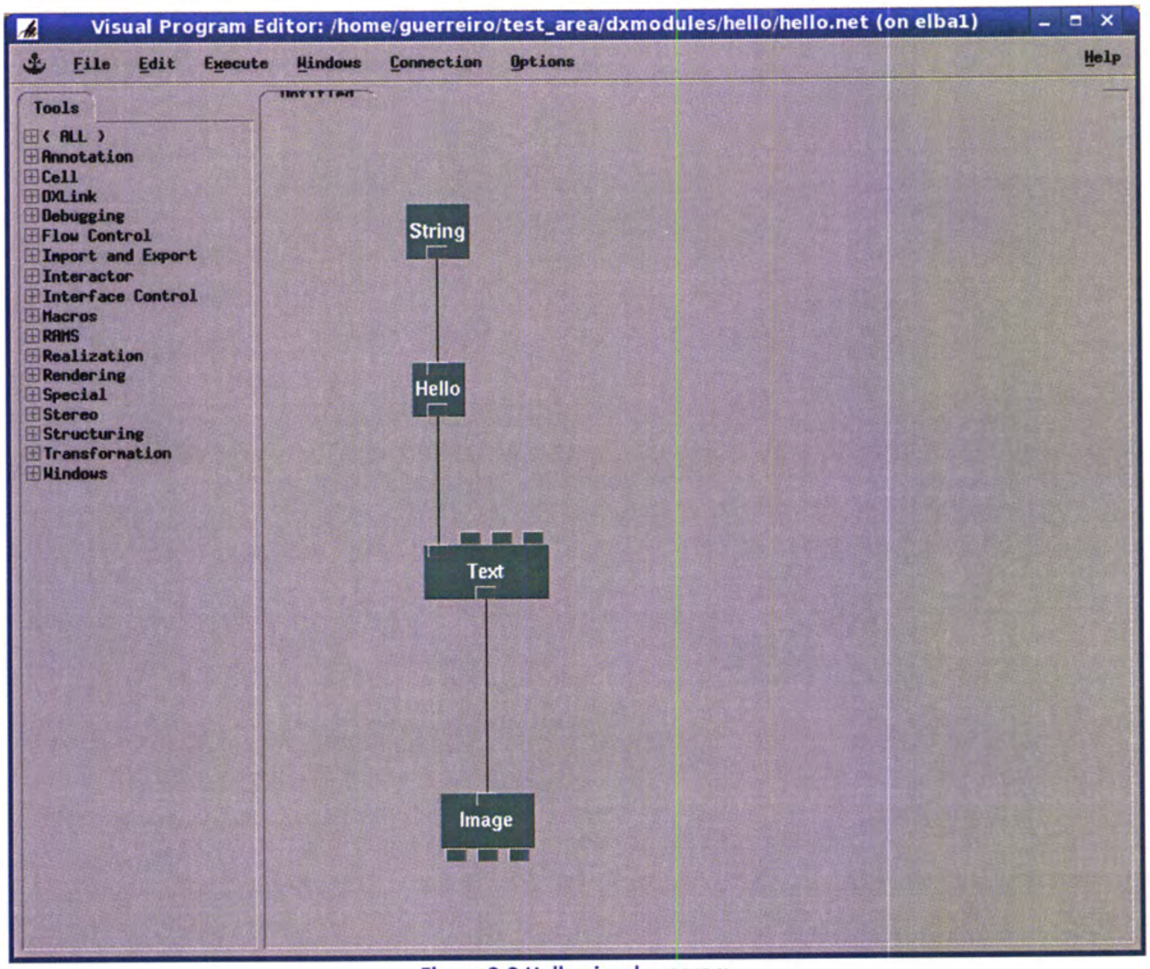


Figure 3-2 Hello visual program

By double-clicking in the *String* module it is possible to specify the input which is used in the Hello module.

Figure 3-3 shows the string dialog box:

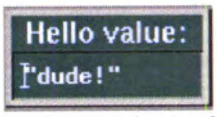


Figure 3-3 DX String module

As may be noticed, the program recognizes that the input value is actually the **Hello value** since the output of the *String* module is connected to the input of the *Hello* module.

Selecting **Execute Once** from the **Windows** menu in the Visual Program Editor in Figure 3-2 produces the output showed in Figure 3-4 which is created by the *Image* module present in the Visual Program Editor.

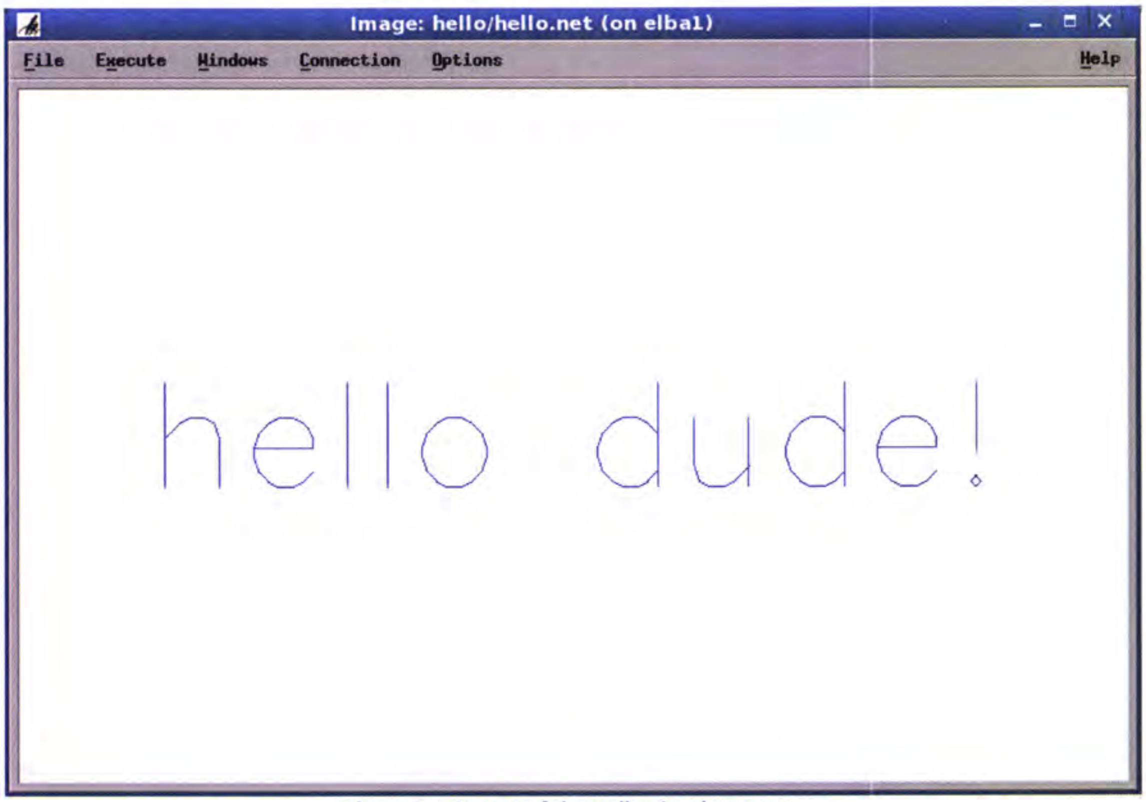


Figure 3-4 Output of the Hello visual program

3.2 Cell/B.E. Applications in OpenDX

After introducing how to write modules for DX, it is time to show how Cell/B.E. can interact with them. For each implemented module this chapter introduces: its behaviour (what is the module supposed to do); its implementation (and some notes after); how to compile and run and a performance study.

Regarding the implementation, there is a lot of code that is not shown during this chapter and big part of it is related to the auto-generated code by the DX module builder. That code is responsible to check the input and handle the errors if something is wrong, which is executed in the PPU side. It is extensive and responsible for some loss of performance during the execution, but the key question is: is this overhead so much that it will not compensate the switch from a textual to a visual programming environment?

The performance study will compare the performance between a standalone (console) C application and the same application running inside of a DX module. The process taken to test this is very simple: for each module, the SPU programs are the same both in the textual and visual programming environments (which means: are the same both for the console and DX) and the PPU programs (which are responsible to start the execution, check inputs and handle errors) are the ones where the key differences lies. A PPU program for the console does not include extensive checks in the input (it is simply initialized and then processed in the SPU side) contrasting to the PPU programs in DX.

3.2.1 Hello World

The first example is very similar to the one introduced in chapter 3.1.5 Hello World Example. The user gives a string as input to the module and the module concatenates the word "Hello" with the input string in one SPE, which gives back the result to the PPE. The main purpose of this module is to show how to start a SPE thread in a DX module.

3.2.1.1 Hello mdf

The mdf for this example is the same as the one exposed in chapter 3.1.5.2 Hello mdf since the module is the same.

3.2.1.2 Hello Implementation

The C code for this example is slightly different from the one in chapter 3.1.5.3 Hello Implementation. Since this example uses a SPE and since that will require a proper code implementation for the SPE side (see chapter 2.1.4.1 Different Processors, Different Compilers), this chapter is divided into two sub-chapters: PPU Hello Implementation and SPU Hello Implementation.

3.2.1.2.1 PPU Hello Implementation

The implementation in the PPU side computes the following steps: process the DX input; start a SPE thread and wait for its conclusion.

Example 3-3 C code for the PPU Hello Implementation

```
#include <dx/dx.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
#include <string.h>
#include <stdint.h>

extern spe_program_handle_t hello_spu;
spe_context_ptr_t spe_ctx;
void *spe_argp, *spe_envp;
char str[256] __attribute__ ((aligned(16)));

// macro for rounding input value to the next higher multiple of
// either 16 or 128 (to fulfill MFC's DMA requirements)
#define spu_mfc_ceil128(value) ((value + 127) & ~127)
#define spu_mfc_ceil16(value) ((value + 15) &  ~15)

void *ppu_hello_pthread_function(void *arg) {
  spe_context_ptr_t ctx;
  unsigned int entry = SPE_DEFAULT_ENTRY;
  ctx = *((spe_context_ptr_t *)arg);
  if (spe_context_run(ctx,&entry, 0, spe_argp, spe_envp, NULL) < 0) {
    perror ("Failed running context");
    exit (1);
  }
  pthread_exit(NULL);
}

// DX Function with New Code
Error m_Hello(Object *in, Object *out)
{
  if (!in[0])
    sprintf(str, "hello world");

  else { // Process the stuff on the SPE
    pthread_t thread;
    char *message;

    if (!DXExtractString(in[0], &message)) {
      DXSetError(ERROR_BAD_PARAMETER, "value must be a string");
      return ERROR;
    }

    // Prepare SPE parameters
    strcpy(str,message);
    spe_argp=(void*)str;
    spe_envp=(void*)strlen(str);
    spe_envp=(void*)spu_mfc_ceil16((uint32_t)spe_envp);

    /* Create context */
    if ((spe_ctx = spe_context_create (0, NULL)) == NULL) {
      perror ("Failed creating context");
      return ERROR;
    }

    /* Load program into context */
    if (spe_program_load (spe_ctx,&hello_spu)) {
      perror ("Failed loading program");
      return ERROR;
    }

    /* Create thread for each SPE context */
    if (pthread_create (&thread,
            NULL,&ppu_hello_pthread_function,&spe_ctx)) {
      perror ("Failed creating thread");
      return ERROR;
    }

    /* Wait for SPU-thread to complete execution. */
    if (pthread_join (thread, NULL)) {
      perror("Failed pthread_join");
      return ERROR;
    }

    if (spe_context_destroy(spe_ctx)) {
      perror("Failed spe_context_destroy");
      return ERROR;
    }
  }

```

```
83      out[0] = (Object)DXNewString(str);
84      return OK;
85    }
```

The first library is the DX library and the three libraries following are just standard libraries.

The two libraries in lines 5 and 6 are used to manage SPE threads and the last two libraries included in lines 7 and 8 are the String and Int libraries. String library is used for a string copy command and the Int library is used to specify short versions for the integer type (for example, a unsigned 64-bit integer can be specified as uint64_t).

Five variables are defined between lines 10 and 13: **hello_spu** is the name of the SPU binary file and executable which runs the SPE thread; **spe_ctx** is a structure which contains information about a SPE thread; ***spe_argp** contains the EA of a structure with all the information needed to be copied to a SPE LS (in this case will just be the address of the input string) and ***spe_envp** contains its size; finally, **str** is the string which is copied to a SPE LS and is 16-byte aligned.

Every time the programmer wants his/her program to do transfers between the Main Storage and a SPE Local Storage, there is one thing that must be kept in mind: the memory supposed to be transferred must be aligned both in the PPE and in the SPE sides and the size of the DMA transfer must have at least 16 bytes and be a multiple of 16 (see chapter 2.1.4.2 Different Processors, Different Address Spaces). For example, is possible to transfer 4 integers in a DMA transfer if is assumed that each integer occupies four bytes (4 integers * 4 bytes/integer = 16 bytes), but it is not possible to transfer one, two or three integers (and not any number of integers not multiple of four). If the programmer wants to transfer an amount of data not multiple of 16 bytes (4 integers in the last example) from the SPE to the PPE, then the macro in line 18 (spu_mfc_ceil16) can be called to find the next multiple size of 16 bytes. The macro in line 17 (spu_mfc_ceil128) does the same for 128 bytes. **NOTE:** The programmer must be sure that there is extra allocated memory on the PPU size to handle with it, otherwise an exception may come up.

The function defined in line 20 is responsible to run a SPE thread (spe_context_run does the magic).

The remaining code, starting in line 32, is responsible to handle the input and corresponds to the main function. If no input is given, then the returning string is "hello world". Otherwise, the string is extracted from the input using **DXExtractString** (see chapter 3.1.5.3 Hello Implementation) and then it is copied to **str**. The reason for this copy is because there is a need to assure a proper alignment of the string to perform a clean copy from the Main Storage to the Local Storage. Since the memory layout for the string pointed by **message** may not fulfil this requirement, the copy is then performed (**strcpy**) to another space and this one (**str**) is then used for the output.

After processing the input, the program prepares the structure for the SPE thread and then runs it and waits for its conclusion. The process of starting, running and finishing SPE threads is as follows (usually):

1. create a SPE context
2. load the SPE binary file
3. create the SPE thread
4. call **pthread_join** to wait for the execution of the SPE thread
5. clean up the context information using **spe_context_destroy**

Finally, the program puts the string in the output and the execution, for this module, finishes. The following sub-chapter shows the SPU hello implementation.

3.2.1.2.2 SPU Hello Implementation

The SPU implementation is divided in straightforward steps: the program starts in the main function; uses a DMA transfer to get the data from the main memory; computes it and then puts the result back. The implementation is below:

Example 3-4 C code for the SPU Hello Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <spu_mfcio.h>

// Macro for waiting to completion of DMA group related to input tag:
// 1. Write tag mask
// 2. Read status which is blocked until all tag's DMA are completed
#define waitag(t) mfc_write_tag_mask(1<<t); mfc_read_tag_status_all();

// macro for rounding input value to the next higher multiple of
// either 16 or 128 (to fulfill MFC's DMA requirements)
#define spu_mfc_ceil128(value) ((value + 127) & ~127)
#define spu_mfc_ceil16(value) ((value + 15) &  ~15)

// Local store buffer: DMA address and size alignment:
//    - MUST be 16B aligned otherwise a bus error is generated
//    - may be 128B aligned to get better performance
// In this case we use 16B because we don't care about performance
char str[256] __attribute__ ((aligned(16)));
char msg[256] __attribute__ ((aligned(16)));

// argp - effective address pointer to the string in main storage
// envp - size of string in main memory in bytes
int main( uint64_t spuid __attribute__ ((__unused__)) ,
          uint64_t argp , uint64_t envp )
{
  uint32_t tag_id = mfc_tag_reserve();
  void *size;

  // reserve a tag from the tag manager
  if (tag_id==MFC_TAG_INVALID){
    printf("SPE: ERROR can't allocate tag ID\n");
    return -1;
  }

  // get data from main storage to local store
  mfc_get((void *)(str), argp, (uint32_t)envp, tag_id, 0, 0);

  // wait for all the DMA commands to complete. wait only this tag_id.
  waitag(tag_id);

  // append "hello" to the string
  sprintf(msg, "%s %s","hello", str);

  // put data to main storage from local store
  size = (void *)strlen(msg);
  size = (void *)spu_mfc_ceil16((uint32_t)size);
  mfc_put((void *)(msg), argp, (uint32_t)size, tag_id, 0, 0);

  // wait for all the DMA commands to complete. Wait on this tag_id.
  waitag(tag_id);

  // release the tag from the tag manager
  mfc_tag_release(tag_id);
  return (0);
}
```

There are two main arrays which are used for the DMA transfers: **str** and **msg**.

The program starts to get the information from the main storage using the arguments given (**mfc_get** command), copying the input string from the main storage to **str**. Then the same string is concatenated to the word "hello" and the result is saved in **msg**, and its contents are copied back to the main storage, more specifically to the same address given by the argument in the beginning of the application.

3.2.1.3 Compiling and Running Hello

The procedure to compile the PPU program was already introduced in chapter 3.1.5.4 Compiling and Running the Hello Example. However, a few changes are needed in order to include the SPU program in the PPU program.

Assuming the SPU Hello implementation is stored in a file **hello_spu.c**, then the compilation process is as follows:

```
spu-gcc -W -Wall -Winline -Wno-main -I. -I /usr/spu/include -I /usr/local/dx/include -O3 -c hello_spu.c
spu-gcc -I /usr/include -o hello_spu hello_spu.o -Wl,-N
ppu-embedspu -m32 hello_spu hello_spu hello_spu-embed.o
ppu-ar -qcs lib_hello_spu.a hello_spu-embed.o
```

The binary file **hello_spu** is the one which is called in the PPU program and the library **lib_hello_spu.a** is included in the PPU compilation process for **hello.c** (assuming that is the name for the PPU Hello implementation):

```
/usr/local/dx/bin/mdf2c hello.mdf > hello_mdf.c
ppu32-gcc -g -O3 -D_GNU_SOURCE -I /usr/local/dx/include -c hello_mdf.c hello.c
ppu32-g++ hello_mdf.o hello.o -L /usr/local/dx/lib_linux -lDX -R
spu/lib_hello_spu.a -lGLU -lpthread -lspe2 -export-dynamic -o dxexec
```

Chapters 3.1.5.4 Compiling and Running the Hello Example and 3.1.5.5 Visual Program Example Using Hello already described how to run and use this example.

3.2.1.4 Hello Implementation Notes

This example is a reference about how to start a SPE-thread and how to work the memory alignment. These two issues are the basic to know when programming for the Cell/B.E. The programmer must plan which data is shared, prepare the structure, create the SPE thread, copy the data to the SPE LS, process it and then write the results back to the main storage.

The next module (Add) introduces (among others) the parallelization and SIMD (Single Instruction Multiple Data) instructions. Fully understanding all those concepts gives the programmer the basic knowledge and skill for Cell/B.E. and parallel programming.

3.2.2 Add

The Add module presented in this sub-chapter introduces the use of parallelization, SIMD instructions and also the double buffering technique.

Its input data is: a field with a 1-D vector of integers; an integer to sum to each position in the vector and another integer where user can specify how many SPEs shall be used. This example was theoretically introduced in chapter 2.1.3 Improving Performance.

All the data in the vector is summed with the integer inside the SPEs and the results are transferred back to the main storage and then the application moves on.

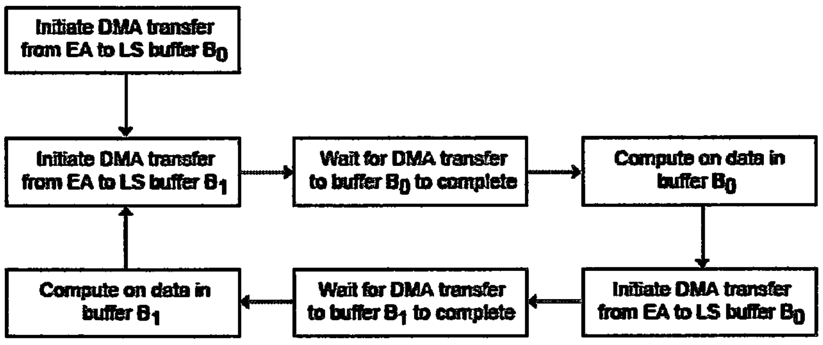
This module was implemented with the help of the DX Module Builder (see chapter 2.3.5 Module Builder). Inputs and outputs were specified there and then the mdf and C code were generated from its specifications.

3.2.2.1 Double Buffering

The goal of the double buffering technique is to avoid stalls (a stall happens when the computation stops because the application is waiting for data). Basically, this technique consists of getting the next buffer of data while processing the current one. When the current buffer is processed then the program starts processing the next buffer and applies for another set of data at the same time, and so on, until the end.

Double buffering is a private class of multi buffering, which extends this idea by using multiple buffers in a circular queue instead of only two buffers. In most cases, the usage of two buffers in the double buffering case is enough to guarantee overlapping between the computation and data transfer. In this example, since it is for demonstrating purposes instead of performance improvement, only double buffering is used.

Figure 3-5 shows a double buffering scheme:



Initiate DMA transfer from EA to LS buffer B_0
Initiate DMA transfer from EA to LS buffer B_1
Wait for DMA transfer to buffer B_0 to complete
Compute on data in buffer B_0
Initiate DMA transfer from EA to LS buffer B_0
Wait for DMA transfer to buffer B_1 to complete
Compute on data in buffer B_1

Figure 3-5 Double buffering scheme

3.2.2.2 Add mdf

The mdf for the Add module was generated using DX Builder (see chapter 2.3.5 Module Builder). Its description is bellow:

Example 3-5 Add mdf

MODULE	Add
CATEGORY	Cell
DESCRIPTION	Adds a single number to each data value of a set
INPUT	data; field; (none); input data
INPUT	value; integer; 0; value to add
INPUT	MAX_SPU_THREADS [visible:0]; integer; 16; maximum number of SPEs to use
OUTPUT	result; field; new data

(none) value in the first input means it is required.

[visible:0] in the last input was added after the mdf was generated. It means that when the module is opened, this input is invisible and can only be seen if the module is expanded. The default value for it is 16, which actually is the maximum number of SPEs in the Cell/B.E. If there are less SPEs available than desired, then the program only uses the available ones.

3.2.2.3 Add Implementation

In spite of the C code for this module being generated, its implementation has some particularities. Some are related to the module itself and some others are related to Cell/B.E. programming, so this chapter is divided into four sections: shared structure; auxiliary functions; ppu program and spu program.

3.2.2.3.1 Add Shared Structure

Since the PPE and the different SPEs share information contained in the main storage (which means in the Effective Address Space), this information must be well structured and defined in order to assure proper DMA transfers from the main storage to the SPE local storage. The proposed structure is the following:

Example 3-6 Add shared structure

```
#ifndef ADD_H_
#define ADD_H_
typedef struct {
  int spu_num;
  int data_knt;
  int *data_data;
  int value;
  int result_knt;
  int *result_data;
  int result;
  unsigned char pad[100]; /* pad to a full cache line */
} add_worker;
```

```
#endif /* ADD_H_ */
```

This structure contains all the needed information for the SPE when it starts running: **spu_num** identifies which SPE is processing the data; **data_knt** refers to how much data is going to be processed; **data_data** is a pointer containing the EA of the first element of data to process; **value** is the value to be summed to all the data; **result_knt** (not needed, but since the DX Builder generates this information the decision was to share it) has the same value as **data_knt** and contains the number of output data and **result_data** points to the EA of the first output data. For alignment purposes, this structure has a size of 128 bytes and to guarantee that an unsigned char **pad** is added to the structure with the remaining size until 128 bytes.

3.2.2.3.2 Auxiliary Functions

Some functions are needed across all the practical work, so a header file was created with these functions. There is one for the PPU side and other for the SPU side.

3.2.2.3.2.1 Auxiliary PPU Functions

Example 3-7 Auxiliary functions for the PPU

```
#ifndef _AUX_H_
#define _AUX_H_

#include <libspe2.h>
#include <pthread.h>
typedef struct {
  spe_context_ptr_t spe_ctx;
  pthread_t thread;
  void *spe_argp;
} param_context;
void *ppu_pthread_function(void *arg);
int find_next_multiple(int n, int s);
int round2(float f);

#endif /* _AUX_H_ */
```

First, the structure **param_context** is the structure containing all the context information needed for the PPU to start a SPU thread.

ppu_pthread_function is responsible to run the SPU program. Its argument is the address of a **param_context** variable (it is similar with the **ppu_hello_pthread_function** in chapter 3.2.1.2.1 PPU Hello Implementation).

The last two functions are used to define how much data is going to be processed in each SPU. **find_next_multiple** returns the next **n** multiple of **s** (it is desired that the shared amounts of data are multiple of 4) and **round2** is a very simple macro which converts a float into a rounded integer.

3.2.2.3.2.2 Auxiliary SPU Functions

Example 3-8 Auxiliary functions for the SPU

```
#ifndef _AUX_SPU_H_
#define _AUX_SPU_H_

#include <spu_mfcio.h>

// Macro for waiting to completion of DMA group related to input tag:
// 1. Write tag mask
// 2. Read status which is blocked until all tag's DMA are completed
#define waitag(t) mfc_write_tag_mask(1<<t); mfc_read_tag_status_all();

// macro for rounding input value to the next higher multiple of
// either 16 or 128 (to fulfill MFC's DMA requirements)
#define spu_mfc_ceil128(value) ((value + 127) & ~127)
#define spu_mfc_ceil16(value) ((value + 15) &  ~15)

// Size of the buffers for DMA transfers
#define ELEM_PER_BLOCK 4096

#endif /* _AUX_SPU_H_ */
```

All macros were used in the SPU Hello Implementation (see chapter 3.2.1.2.2 SPU Hello Implementation).

waitag macro is used to know when determined DMA transfer is completed. If a transfer is completed then the program goes on, otherwise the program stays blocked until the transfer is done.

spu_mfc_ceil rounds the input value to the next higher multiple of 16 or 128.

ELEM_PER_BLOCK defines the size of the buffers used in DMA transfers.

3.2.2.3.3 PPU Add Implementation

Since the DX Builder was used to implement this module, only the manipulation of the data is introduced.

The first thing to know is: when the DX Builder generates the C code for the module, it creates a user function where the programmer decides how to handle the data. The following user function was created from the specification given in the module (which generated the mdf too, see chapter 3.2.2.2 Add mdf):

Example 3-9 Generated C code for the PPU Add Implementation

```
int
Add_worker(
    int data_knt, int *data_data,
    int value_knt, int *value_data,
    int MAX_SPU_THREADS_knt, int *MAX_SPU_THREADS_data,
    int result_knt, int *result_data)
{
  /*
```

```
    * Comments describing the variables...
    */

   /*
    * User's code goes here
    */

   /*
    * successful completion
    */
   return 1;

   /*
    * unsuccessful completion
    */
error:
   return 0;
}
```

The function provides a pointer for each input and output to its data and an integer with the number of items for each pointer. The name before **_knt** and **_data** is the name given by the user to the variable in the DX Builder. The data provided by DX through the pointers is word aligned, which conforms to the DMA transfers requirements.

Once the code for the module is generated, the programmer can start writing its code after the comment which says "**User's code goes here**".

So, to start processing and summing all the values given in the input data and then store the results in the output data, the following steps are taken:

- determine how many SPEs are available
- determine how much data will be processed in each SPE
- prepare the arguments for the SPEs
- start the SPE threads and wait for their completion

To determine how many SPEs are available, the function **spe_cpu_info_get** is called. If there are more SPEs available than desired, only the requested number is used.

After knowing how many SPEs are going to be used, it is time to define how much data is going to be processed in each SPE. If there was no requisites for the DMA transfers (see chapter 2.1.4.2 Different Processors, Different Address Spaces), then the solution would be as simple as dividing the amount of data to process by the number of SPEs. But, since that is not enough, after dividing and getting the result, the next multiple of four of that result is found and then: the first N-1 SPEs handle the multiple of four amount of data and the last SPE takes the remaining data. Why multiple of four? Because a DMA transfer must handle at least 16 bytes of data it is only possible to transfer the minimal amount of 4 integers (32-bit size) per each DMA transfer.

Preparing the arguments for the SPEs is as simple as allocating one **add_worker** structure (see chapter 3.2.2.3.1 Add Shared Structure) for each SPE and then giving the proper values: **data_data** points to the memory position where the SPE starts handling the data along **data_knt** positions and the results are stored starting in the address provided by **result_data**.

Finally, the SPE threads are started. Their argument is a pointer to the main storage where the **add_worker** structure is stored with all the needed information. The PPU then waits for the threads to finish and returns successful or unsuccessful completion.

The code for this task is below. Since the SPU threads handling was already introduced in the Hello World Example (see chapter 3.2.1.2 Hello Implementation), only the first three steps are shown:

Example 3-10 C code for the PPU Add Implementation

```
 /*
  * User's code goes here
  */

 int MAX_SPU_THREADS, value;

 // How many SPEs
 MAX_SPU_THREADS = (MAX_SPU_THREADS_knt == 0) ?
                        16 : MAX_SPU_THREADS_data[0];

 // value to sum with the "data"
 value = (value_knt == 0) ? 0 : value_data[0];

 volatile add_worker stuff[MAX_SPU_THREADS] __attribute__
((aligned(128)));
 extern spe_program_handle_t add_spu;
 param_context param[MAX_SPU_THREADS];

 int i, spu_threads, nItems;

 /* Determine the number of SPE threads to create */
 spu_threads = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
 if (spu_threads > MAX_SPU_THREADS)
       spu_threads = MAX_SPU_THREADS;

 // nItems to process in each SPE. Must be a multiple of 4
 nItems = round2((float)data_knt / (float)spu_threads);
 nItems = find_next_multiple(nItems,4);

 for(i = 0; i < spu_threads; i++)
 {
       // Prepare SPE parameters
       stuff[i].spu_num = i;
       stuff[i].data_data=(void *)&data_data[i*nItems];
       stuff[i].value=value;
       stuff[i].result_data=(void *)&result_data[i*nItems];
       if (i == spu_threads - 1)
       {
             stuff[i].data_knt = data_knt-nItems*(spu_threads-1);
             stuff[i].result_knt = result_knt-nItems*(spu_threads-1);
       }
       else
       {
             stuff[i].data_knt=nItems;
             stuff[i].result_knt=nItems;
       }

       stuff[i].result=0; //Assuming "Failure..."

       param[i].spe_argp=(void *)&stuff[i];

       /* Create context, load program, create thread */
 }

 /* wait for the threads completion and destroy context */
```

3.2.2.3.4 SPU Add Implementation

The program in the SPE side implements the following steps:

1. get the argument information
2. get the first buffer of data
3. if there is more buffers to get, apply for the next one (if not, step 7)
4. compute the data in the current buffer
5. put the result data on the main storage
6. go to the next buffer and repeat the third step
7. compute the current buffer and put the result on the main storage
8. terminate the application

The implementation for this program is not much different from a regular C program. The key differences are the DMA transfers (with the double buffering technique, see chapter 3.2.2.1 Double Buffering) and the SIMD instructions.

The commands **mfc_get** and **mfc_put** are used to get and put data respectively from/in the main storage and the data is stored in SIMD vectors in order to sum more than one piece of data at a time by using the intrinsic **spu_add** (see chapters 2.1.3.2.1 Scalar VS Vector SIMD: A code example and 2.1.4.1.1 C-Language Intrinsics).

Bellow is the SPU Add Implementation:

Example 3-11 C code for the SPU Add Implementation

```
#include "../add.h"
#include "aux_spu.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <spu_mfcio.h>
#include <spu_intrinsics.h>
#include <math.h>

volatile add_worker stuff __attribute__ ((aligned(128)));
volatile vec_int4 ls_data_data[2][ELEM_PER_BLOCK / 4] __attribute__
((aligned(128)));
volatile vec_int4 ls_result_data[2][ELEM_PER_BLOCK / 4] __attribute__
((aligned(128)));

// argp - effective address pointer to the "SPE stuff"
int main( uint64_t spuid __attribute__ ((__unused__)) ,
          uint64_t argp , uint64_t envp __attribute__ ((__unused__)) )
{
  uint32_t tag_id[2];

  volatile int *data_data, *result_data, *nxt_data, *nxt_result;
  vec_int4 value;
  int data_knt, left, buf, nxt_left, nxt_buf, i, spu_num;
  void *size;

  tag_id[0] = mfc_tag_reserve();
  tag_id[1] = mfc_tag_reserve();

  // reserve a tag from the tag manager
  if (tag_id[0]==MFC_TAG_INVALID){
    printf("SPE: ERROR can't allocate tag ID\n");
    return -1;
  }

  // reserve a tag from the tag manager
  if (tag_id[1]==MFC_TAG_INVALID){
    printf("SPE: ERROR can't allocate tag ID\n");
    return -1;
  }

  // get data from main storage to local store
  mfc_get((void *)&(stuff),argp,sizeof(add_worker),tag_id[0],0,0);

  // wait for the command to complete. wait on this tag_id.
  waitag(tag_id[0]);

  // initialize the parameters
  spu_num = stuff.spu_num;
  data_data = stuff.data_data;
  result_data = stuff.result_data;
  data_knt = stuff.data_knt;
  value = (vec_int4){stuff.value,stuff.value,stuff.value,stuff.value};

  left = (data_knt < ELEM_PER_BLOCK) ? data_knt : ELEM_PER_BLOCK;

  // adapt the size in order to fulfill the alignment requirements
  size = (void *)(left*sizeof(int));
  size = (void *)(spu_mfc_ceil16((uint32_t)size));

  // Prefetch first buffer of input data
  buf = 0;
  mfc_getb((void *)ls_data_data, (uint32_t)(data_data),
           (uint32_t)size, tag_id[0], 0, 0);

  while (left < data_knt) {
    data_knt -= left;

    nxt_data = data_data + left;
    nxt_result = result_data + left;
    nxt_left = (data_knt < ELEM_PER_BLOCK) ?
                data_knt : ELEM_PER_BLOCK;

    // adapt the size in order to fulfill the alignment requirements
    size = (void *)(nxt_left*sizeof(int));
    size = (void *)(spu_mfc_ceil16((uint32_t)size));

    // Prefetch next buffer so the data is available for computation
    // on next loop iteration.
    // The first DMA is barriered so that we don't GET data before the
    // previous iteration's data is PUT.
    nxt_buf = buf^1;
    mfc_getb((void *)(&ls_data_data[nxt_buf][0]),
            (uint32_t)(nxt_data),(uint32_t)size,tag_id[nxt_buf],0,0);

    // wait for previously prefetched data
    waitag(tag_id[buf]);

    for (i = 0; i < left / 4; i++)
      ls_result_data[buf][i] = spu_add(ls_data_data[buf][i], value);

    // Put the buffer's position data back into system address space
```

```
 93      mfc_putb((void *)(&ls_result_data[buf][0]),
 94              (uint32_t)(result_data),left*sizeof(int),tag_id[buf],0,0);
 95
 96      data_data = nxt_data;
 97      result_data = nxt_result;
 98
 99      buf = nxt_buf;
100      left = nxt_left;
101    }
102
103    // Wait for previously prefetched data
104    waitag(tag_id[buf]);
105
106    // process buffer
107    for (i = 0; i < (int)ceil((float)left/(float)4); i++)
108      ls_result_data[buf][i] = spu_add(ls_data_data[buf][i], value);
109
110    // adapt the size in order to fullfil the alignment requirements
111    size = (void *)(left*sizeof(int));
112    size = (void *)(spu_mfc_ceil16((uint32_t)size));
113
114    // Put the buffer's position data back into system address space
115    // Put barrier to ensure all data i written to memory before writing
116    // status
117    mfc_putb((void *)(&ls_result_data[buf][0]), (uint32_t)(result_data),
118             (uint32_t)size,tag_id[buf],0,0);
119    waitag(tag_id[buf]);
120
121    stuff.result = 1;
122
123    mfc_put((void *)&stuff,argp,sizeof(add_worker),tag_id[buf],0,0);
124
125    waitag(tag_id[buf]);
126
127    mfc_tag_release(tag_id[0]);
128    mfc_tag_release(tag_id[1]);
129
130    return (0);
131  }
```

The first step goes from line 43 until 54. The program gets the arguments and initializes the variables. Second step is performed until line 65 and here the application gets the first buffer of data. Third step starts then and it is repeated as long as there are more buffers to apply. Steps 3, 4, 5 and 6 run inside the cycle and once the program gets out of it, the last two steps are then performed.

Before all the mfc commands it is noticeable that the size of the transfer is always rounded. This process is only needed in the last processed buffer in the last SPE since this one may not have the required size for a DMA transfer, but in order to keep the program simple that detail was left aside.

3.2.2.4 Compiling and Running Add

To compile the add module, the same procedure is followed as in the Hello Example (see chapter 3.2.1.3 Compiling and Running Hello). An example of a visual program using this module is shown in Figure 3-6:

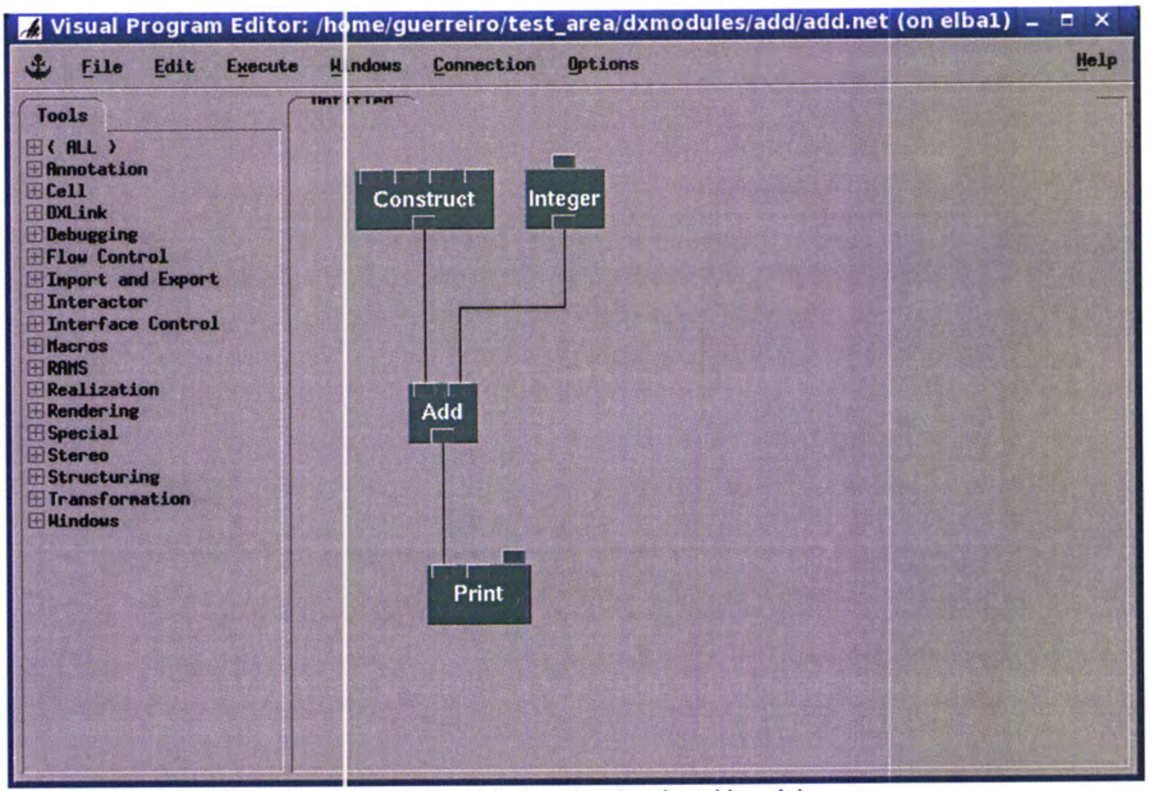
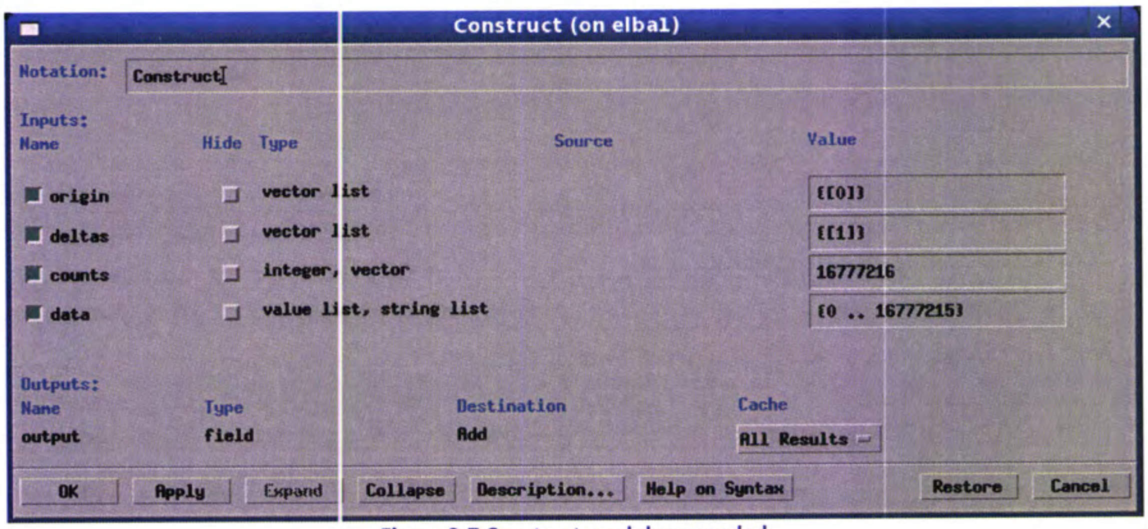


Figure 3-6 Visual program using the Add module

The *Add* module has two (visible) inputs and one output.

The first input is given by the *Construct* module and its expansion is showed in Figure 3-7:


Construct (on elba1)

Notation: Construct

Inputs:

Name	Hide	Type	Source	Value
origin	☐	vector list		[[0]]
deltas	☐	vector list		[[1]]
counts	☐	integer, vector		16777216
data	☐	value list, string list		[0 .. 16777215]

Outputs:

Name	Type	Destination	Cache
output	field	Add	All Results

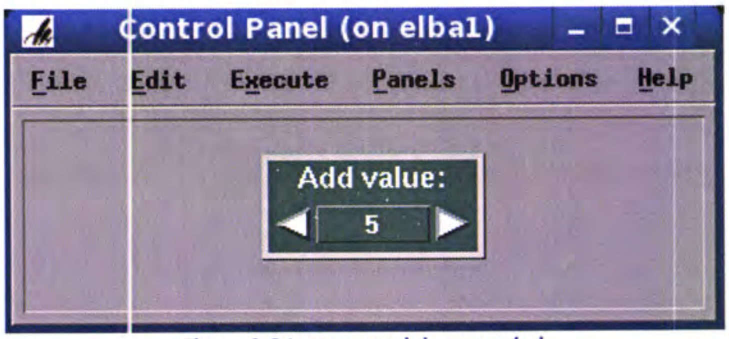
Figure 3-7 Construct module expanded

Since the data shape specified for the first input is a 1-D vector, the **origin** must be a 1-D point ("0" in this case).

The **deltas** can be omitted since they do not have influence in this module. In this case it means that the next position is one point ahead of the current one.

The length of the data is 16MB (stated in **counts**) and its values go from 0 until 16777215 (stated in **data**).

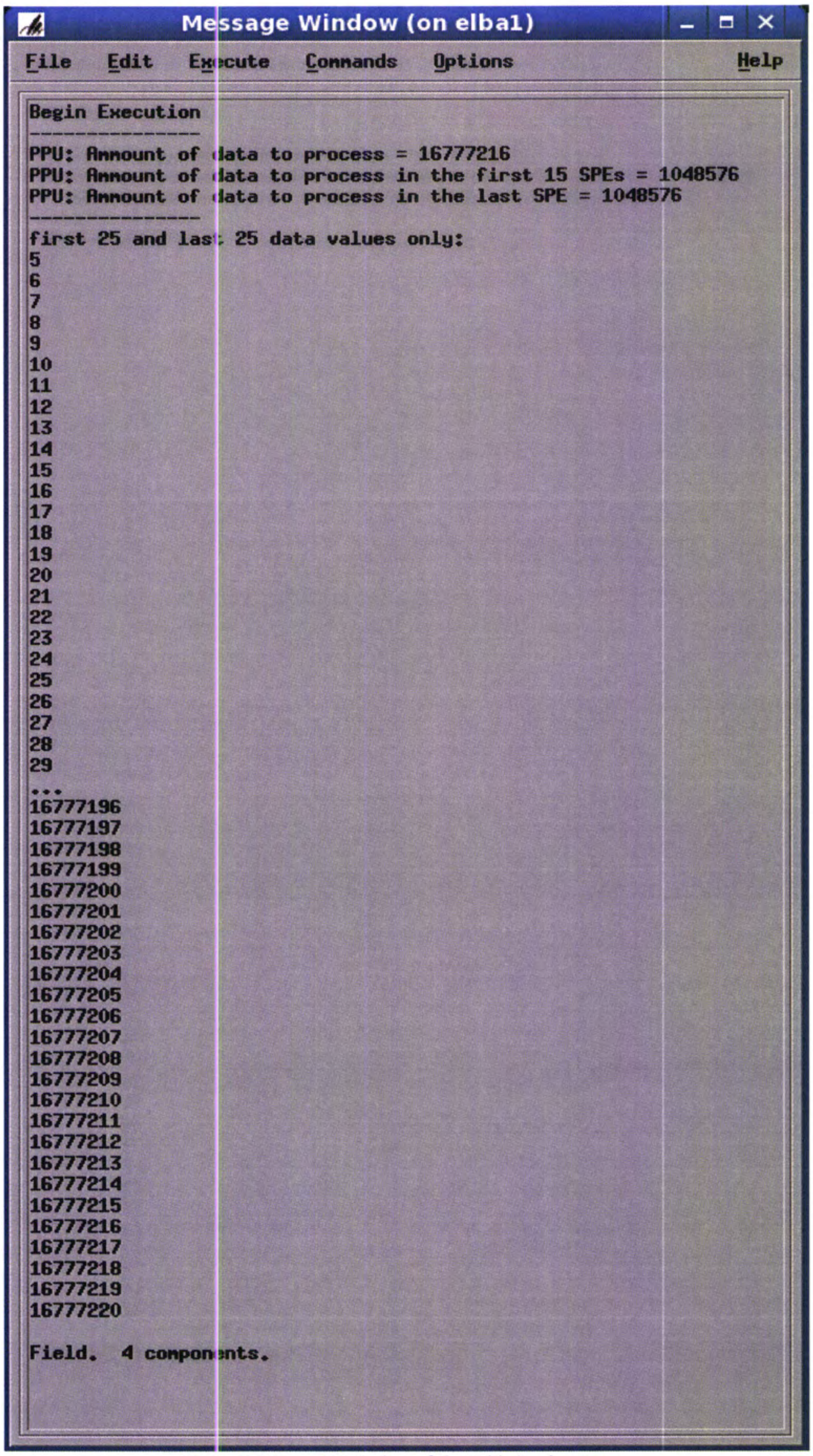
The second input of the *Add* module is just an integer and is showed in Figure 3-8:



Add value: 5

Figure 3-8 Integer module expanded

Finally, the output of the *Add* module is connected to a *print* module which prints the contents of the field in the message window:


Message Window (on elba1)
File Edit Execute Commands Options Help

```
Begin Execution
---------------
PPU: Amount of data to process = 16777216
PPU: Amount of data to process in the first 15 SPEs = 1048576
PPU: Amount of data to process in the last SPE = 1048576
---------------
first 25 and last 25 data values only:
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
...
16777196
16777197
16777198
16777199
16777200
16777201
16777202
16777203
16777204
16777205
16777206
16777207
16777208
16777209
16777210
16777211
16777212
16777213
16777214
16777215
16777216
16777217
16777218
16777219
16777220

Field.  4 components.
```

Figure 3-9 Message window of the Add visual program

Figure 3-9 also shows that the adding operation is performing well (in terms of results consistency) since the *Construct* module stated data between 0 and 16777215 and the data obtained after computing the *Add* module is between 5 and 16777220.

3.2.2.5 Add Implementation Notes

The *Add* module was the first module implemented using parallelization and SIMD instructions. This brought some issues during the implementation.

The first one is related to the control structure, which is shared between the PPU and the SPU. If the size of this structure is not a power of two or properly aligned, the first attempt to start the parallelization fails.

The second one is related to the use of SIMD instructions. The data is stored in 128 bit vectors where each one contains four 32-bit integers. Adjusting the program to support any size and distributing the computation in an equivalent form for each SPE is cumbersome. The implemented solution was to make sure that the computed size is a multiple of four. If not, then the program finds the next higher size which makes this true. This did not prove to be a problem since DX allocates more memory than requested, so no bugs were found because of using data addresses bigger than the size requested in the allocation process.

The last issue is related to the compilation process. If the user decides to use a 64 bit compiler instead of the 32 bit, some problems may occur since the size of the pointers is different in each version which may provoke conflicts in the DMA transfers (because of the use of pointers by the shared structure).

3.2.2.6 Add Performance Study

The approach taken to measure the performance is composed by one SPU program and two "different" PPU programs: the regular PPU program for the DX module and its console version.

The console program, responsible to perform the same computation than the DX module, has a very simple structure: allocate memory and compute. Basically, 95% of its implementation is the same that is inside the user code for the Add module (see chapter 3.2.2.3 Add Implementation) and the main difference resides in the fact that there is no input checking. Compiler flags are also the same for both sides.

Memory is allocated using **posix_memalign** (from stdlib.h). This function provides a way to allocate aligned memory, which is very useful for the DMA transfers between the PPE and the SPEs.

gettimeofday (from sys/time.h) is the function used to measure the performance and is called in the beginning and end of the computation of both console and DX module programs. The goal is to demonstrate that the performance, from the user's point of view, is not heavily compromised when there is a switch from a console environment to a visual programming environment (DX). A deeper study may count the clock cycles for the computation but that does not give the desired perspective since time is more intuitive than clock cycles in order to understand the real difference.

Figure 3-10 and Figure 3-11 show the performance results for the Add module in both QS22 Blade and Playstation3 (PS3):

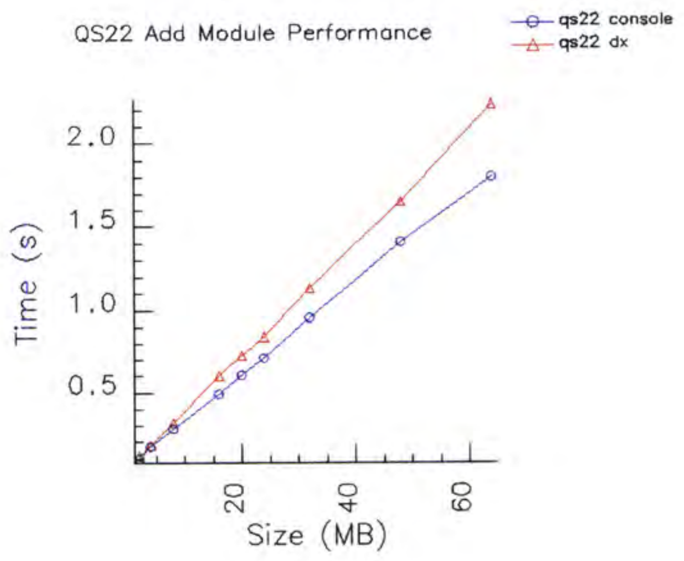


Figure 3-10 QS22 Add Module Performance

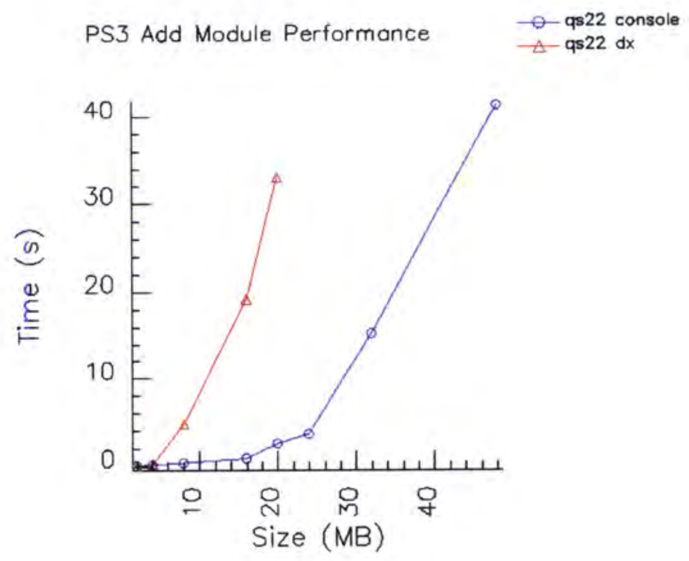


Figure 3-11 PS3 Add Module Performance

The X-axis corresponds to the input size (in Megabytes) for the operation, which follows the pattern in Figure 3-7. It is composed by a single array of integers which goes from 0 until the size of the input.

The Y-axis corresponds to the time required (in seconds) to do the computation depending on the size of its input.

The plot in Figure 3-10 (QS22 case) shows a small difference for the computation time between the console and DX. In spite of the growing difference, it can be reduced. The code used to implement this module was generated with the DX Module Builder and this code is far away from being optimized. Besides, this example is very simple and does not involve too much complex computation. The more complex the computation, the less impact will have the input checking, performed by DX, in the performance measurement.

Figure 3-11 does not show what is expected due to some particularities related with the PlayStation3: small RAM memory (256MB) and the use of a native hypervisor.

The lack of memory explains why the computation time grows exponentially faster in DX than in the console. Since DX uses the X-server and lots of memory, the program easily runs out of memory when some computation is performed. Since a console environment uses less memory, then the computation can finish faster for bigger inputs.

The native hypervisor is also responsible for the slow performance since it controls all the access to the hardware. This means that it occupies a layer between the hardware and the running operating system kernel. Since the Cell/B.E. SDK is optimized for Fedora and once that Fedora does not have direct access to the hardware, this compromises all performance tests which can be made with the PS3.

This means that the PS3 can be used for demonstration purposes but the QS22 Blade is a better choice if more computing power is desired. It is possible to improve performance in the PS3 if the DX server (using the X-windows) runs in a different system and then, by defining clusters, use one (or more) PS3 with a clean system to perform the computation. It is possible do define in DX which modules run in which system but the main server must be running the user interface and the remaining clients can be running only in script mode.

In order to keep performance tests simple, the remaining ones for the other modules are evaluated only in a QS22 Blade.

3.2.3 Add2

The Add2 module is not much different from the Add module (see chapter 3.2.2 Add). Instead of summing all the positions of a 1-D array with an integer, it sums two 1-D arrays with the same size and position by position. For example, if **A** and **B** are the input arrays, **R** is the output array and **i** is an integer which goes from the beginning until the end of the array, then R[i] = A[i] + B[i].

3.2.3.1 Add2 mdf

The only difference between the Add2 mdf and the Add mdf (see chapter 3.2.2.2 Add mdf) is the second input, which is a field instead of an integer:

Example 3-12 Add2 mdf

```
MODULE        Add2
```

CATEGORY	Cell
DESCRIPTION	Adds two data sets
INPUT	data1; field; (none); first data set
INPUT	data2; field; (none); second data set
INPUT	MAX_SPU_THREADS [visible:0]; integer; 16; maximum number of SPEs to use
OUTPUT	result; field; summed data sets

3.2.3.2 Add2 Implementation

The main differences in the Add2 module implementation compared with the Add module implementation (see chapter 3.2.2.3 Add Implementation) are:

- The shared structure has the size of the second input and a pointer for it, instead of a single integer.
- Because the shared structure is a little different, the PPU program will have to handle it when initializing the structure.
- The SPU program will process 2 buffers at a time instead of one (double buffering on two sets of data).

Apart from these three differences, the implementation is exactly the same as the Add module implementation. It uses the same auxiliary functions and the PPU code was also generated using the DX Module Builder (see chapter 2.3.5 Module Builder).

3.2.3.3 Compiling and Running Add2

Compiling this module is not any different from the other presented modules. Below is a visual program in DX using the Add2 module:

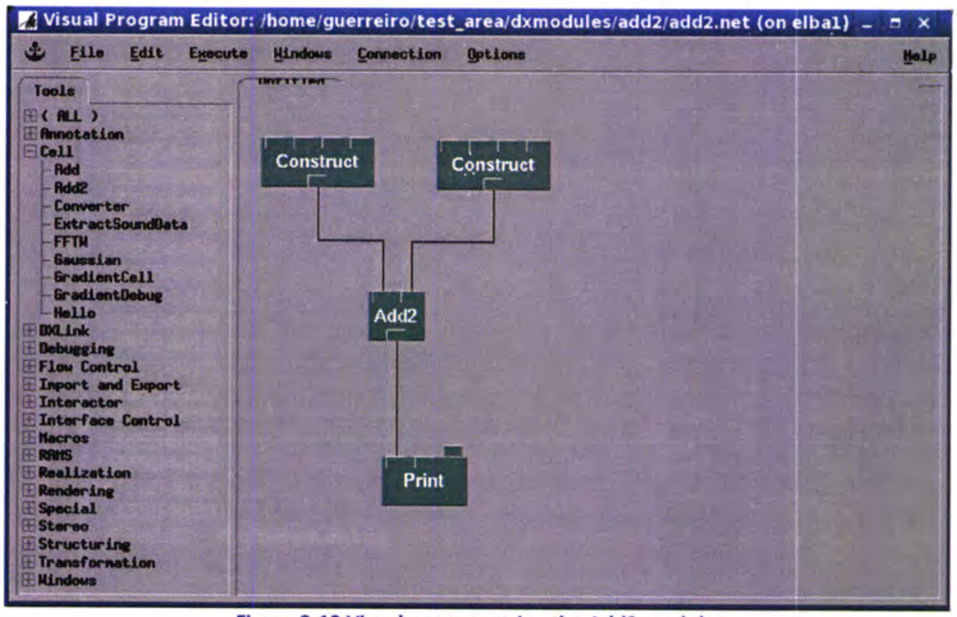
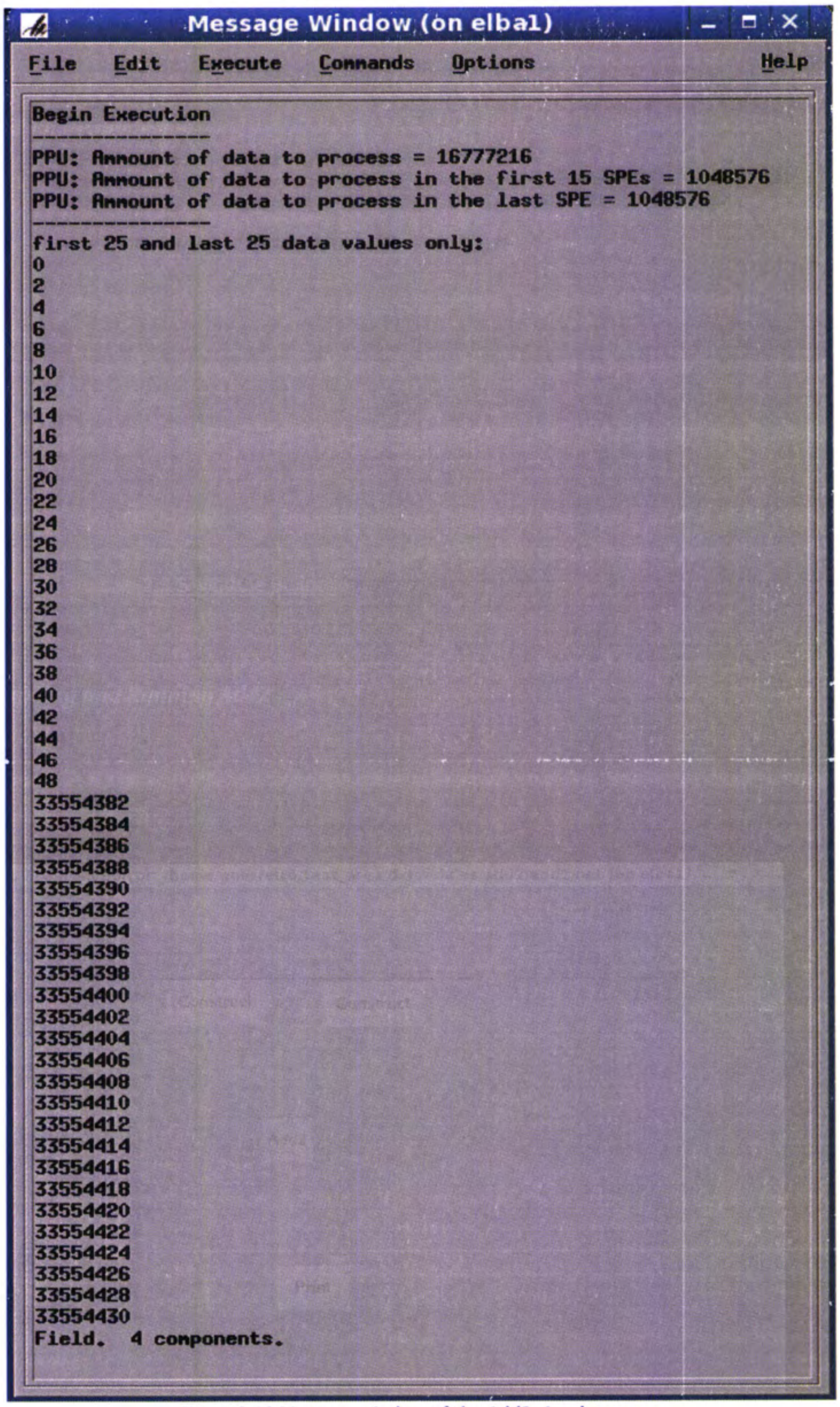


Figure 3-12 Visual program using the Add2 module

The *Construct* and *Print* modules have the same input as the visual program example using the *Add* module inFigure 3-6. Figure 3-13 shows the program output:


Message Window (on elba1)
File Edit Execute Commands Options Help

```
Begin Execution
---------------
PPU: Ammount of data to process = 16777216
PPU: Ammount of data to process in the first 15 SPEs = 1048576
PPU: Ammount of data to process in the last SPE = 1048576
---------------
first 25 and last 25 data values only:
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
33554382
33554384
33554386
33554388
33554390
33554392
33554394
33554396
33554398
33554400
33554402
33554404
33554406
33554408
33554410
33554412
33554414
33554416
33554418
33554420
33554422
33554424
33554426
33554428
33554430
Field.  4 components.
```

Figure 3-13 Message window of the Add2 visual program

The results look consistent since the output values go two by two from 0 to 33554430 which is the expected result since the summed arrays go from 0 to 16777215.

3.2.3.4 Add2 Implementation Notes

This example does not bring much new stuff. The shared structure was modified and the SPU processes two buffers at a time (and because of that is used one more **tag** for DMA transfers).

The main reason behind the implementation of this module is not the challenge or its utility, but to conduct a performance study which is introduced in the next sub-chapter.

3.2.3.5 Add2 Performance Study

Figure 3-14 shows the results of running the Add2 module in a QS22 blade:

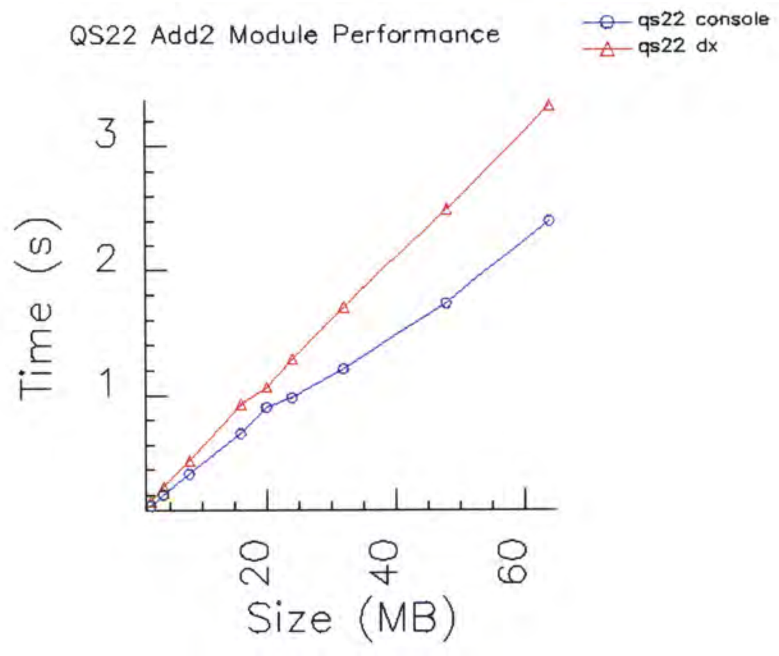


Figure 3-14 QS22 Add2 Module Performance

The input sizes in the plot refer to each argument for the Add2 module. For example, for a 16MB input it means 16MB size for each input.

Basically the plot shows the same as in the Add module example. There is some loss of performance but not enough to argue that visual programming is not worth it.

The example introduced in the next sub-chapter not only is more computing intensive, but also uses a optimized memory structure (created by the DX developers), so it is expected that the difference between console and DX is much smaller.

3.2.4 Fast Fourier Transform in the West (FFTW)

FFTs (Fast Fourier Transforms) are used in many applications that process raw data looking for a signal. There are many FFT algorithms ranging from relatively simple powers-of-two algorithms to powerful, but CPU intensive, algorithms capable of working on arbitrary inputs. FFT algorithms are well suited to the Cell/B.E. processor because they are floating point intensive and exhibit regular data access patterns.

The next module consists of porting a real Cell/B.E. application to a DX module. The FFTW library (Frigo and Johnson 2008) is used in the implementation process. It supports FFTs of any size and takes advantage of the Cell/B.E. capabilities.

Since DX has already a module to compute FFTs then its code is reused to implement the FFTW module, instead of implementing it from the scratch.

3.2.4.1 Installing the libFFTW

Installing the libFFTW with Cell/B.E. support is very straightforward. Version used is 3.2 and it can be installed by typing the following commands in the shell console:

- ./configure --enable-cell --enable-single --enable-altivec
- make
- make install

The reason for installing only single-precision is to keep the same configuration both in the Cell QS22 Blade and in the PS3. QS22 supports double-precision but the PS3 does not. If different configurations were used, this could complicate the demonstration issues performed with the PS3.

Also, if a comparison between Cell/B.E. and other architecture is desired, then making the single-precision a standard for the tests is a good choice since most of the architectures support it.

3.2.4.2 FFTW mdf

The mdf for the FFTW module is the same as the DX mdf for the FFT module:

Example 3-13 FFTW mdf

```
MODULE      FFTW
CATEGORY    Cell
```

DESCRIPTION	Computes the Fast Fourier Transform using the LIB FFTW
INPUT	field; field; (none); input data
INPUT	direction; string; "forward"; direction of the transform: "forward", "inverse" or "backward"
INPUT	center; flag; 0; center the result of the transform
OUTPUT	result; field; computed field

Besides computing a forward FFT, the FFTW module can also compute an inverse FFT.

The **center** flag is used mainly for visualization purposes.

3.2.4.3 FFTW Implementation

Browsing and understanding the DX code for the FFT module can be a cumbersome task. Anyhow, after all the code which handles the input, there is one function called **TransformAggregate** which is responsible for the FFT's computational part. FFTW Implementation replaces the code in this function with the libFFTW functions and puts the results in the output buffer.

The code bellow shows the implementation for a regular FFT for a 1-D input field:

Example 3-14 C code for the FFTW Implementation

```
static Error
TransformAggregate (FieldInfo *f, XFArgs *xfa)
{
    XFData          data;
    int             dims;
    int             i, j, k, n, counts[3];
    Error           ret   = ERROR;

    data.args  = *xfa;
    data.info  = f;
    data.procs = DXProcessors (0);

    dims = f->ndims;

    /* New code starts here */
    /* DXGetArrayData returns a pointer to the field's data */
    float *out_data = (float *) DXGetArrayData(data.info->data);

    /* FFTW uses its own structures to compute the data */
    fftw_complex *in;
    /* fftw_plan specifies which kind of FFT is performed */
    fftw_plan p;

    counts[0] = data.info->counts[0];
    counts[1] = data.info->counts[1];
    counts[2] = data.info->counts[2];

    if (dims == 3)
    {
      if (counts[2] > 1) {
            /* Compute the 3D field */
      }

      else if (counts[1] > 1) {
            /* Compute the 2D field */
    }

    else { //Compute the 1D field
      n = counts[0];

      /* allocate input (and output) data */
      in = (fftw_complex*) fftw_malloc(n * sizeof(fftw_complex));

      if (!data.args.inverse) {
            /*
             * Here is specified which kind of FFT is
             * going to be performed: input size is n; in contains
             * the input data and the output data is saved in the
             * same area; FFT_FORWARD means a regular FFT is going
             * to be computed and FFTW_ESTIMATE means the library
             * processes the FFT on-the-fly without much
             * optimizations
             */
            p = fftw_plan_dft_1d(n,in,in,FFTW_FORWARD,FFTW_ESTIMATE);
            for(i = 0; i < n; i++){
                  //in[i][0] contains the real component
                  //in[i][1] contains the imaginary component
                  in[i][0] = out_data[2*i];
                  in[i][1] = 0.0f;
            }
      }
      else {
            /* instead, compute a inverse FFT */
            p = fftw_plan_dft_1d(n,in,in,FFTW_BACKWARD,
                                 FFTW_ESTIMATE);
            for(i = 0; i < n; i++){
                  /*
                   * Contrasting the libFFTW, DX saves complex
                   * information in a row, which means the first
                   * real and imaginary components are the first 2
                   * sets of data in the array, and so on...
                   */
                  in[i][0] = out_data[2*i];
                  in[i][1] = out_data[2*i+1];
            }
      }

      fftw_execute(p); //compute the FFT

            if (!data.args.center && !data.args.inverse)
              {
                  /* Copy the results back to the DX output */
                  for(i = 0; i < n; i++){
                        out_data[2*i] = in[i][0];
                        out_data[2*i+1] = in[i][1];
                  }
              }
            else
            {
                  /* Take care of the other cases ... */
            }

            fftw_destroy_plan(p);
            fftw_free(in);

```

```
                ret = OK;
            }
        }

        else
            DXSetError (ERROR_BAD_PARAMETER, "unsupported dimensionality");

        return (ret);
}
```

The code is very simple and straightforward. The other cases are not much different from this one and can be seen on the source code.

3.2.4.4 Compiling and Running FFTW

The important thing to keep in mind when compiling the FFTW module is to add the flag **-lfftw3** in the compilation process.

Figure 3-15 shows a visual program using the FFTW module:

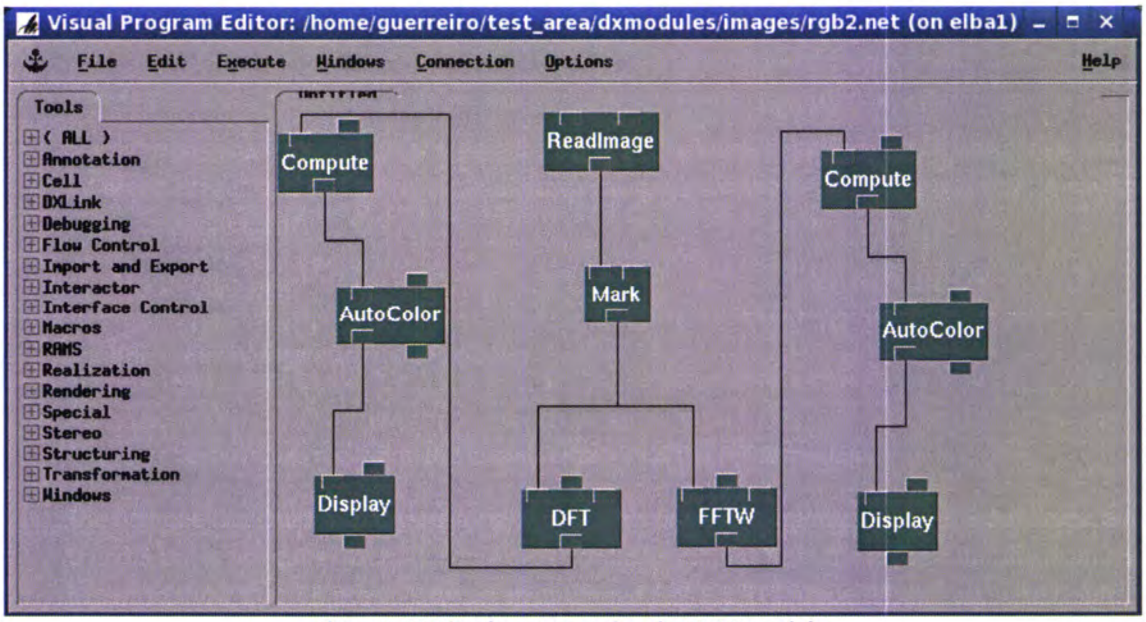


Figure 3-15 Visual program using the FFTW module

In this example an FFT of an image is compute. On one side the DX *DFT* module (Discrete Fourier Transform) is used and in the other the *FFTW* module is used.

Both of them receive a field with the colour data of an image as first input. This data is composed by a 3-D vector (red, blue and green) with the type unsigned byte, which means that each value goes from 0 to 255.

Second input is a predefined one, which means a forward FFT is computed and the third input is an activated flag in order to centralize the results.

The *Compute* module applies the mathematical absolute function to all the given values (converting the negative values into positive ones) and the *AutoColor* module, like the name suggests, colours the results.

Figure 3-16 shows the output images from the two *Display* modules:

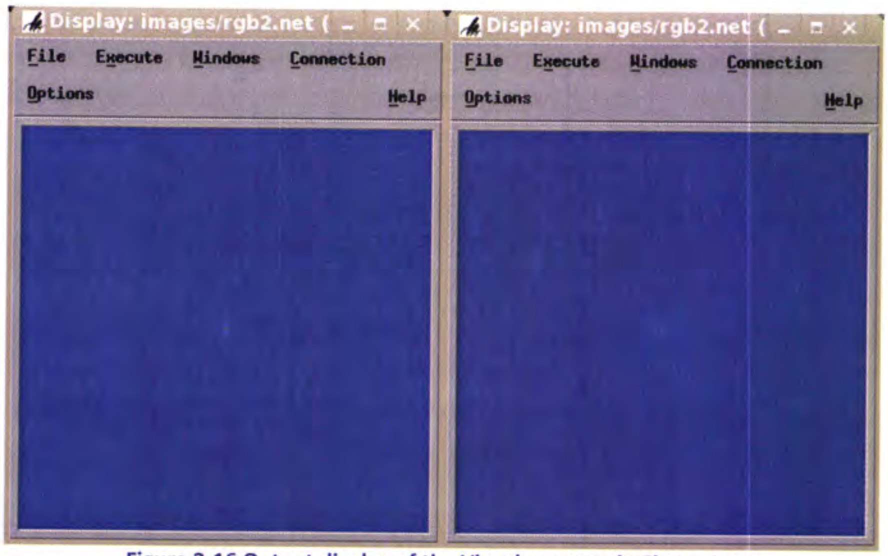

Display: images/rgb2.net (
File Execute Windows Connection
Options Help
Display: images/rgb2.net (
File Execute Windows Connection
Options Help

Figure 3-16 Output display of the Visual program in Figure 3-15

Since the outputs from both displays are the same for the same input image (this was widely tested with different images and by comparing the numerical values of the results), it is possible to say that the results are consistent.

3.2.4.5 FFTW Implementation Notes

There are some issues regarding the implementation of the FFTW module.

The first one is related with the centring of data. This process is optimized for 1-D input fields, but for 2-D and 3-D it is not quite like that. Some cycles could be cut out from the process but the main objective here is to show that it is possible to run an optimized Cell/B.E. FFT inside DX with accurate results, so the performance tuning for this small operation was left aside. Besides, this fact does not influence the performance study since both console and DX module programs use the same algorithm for the computation part.

The second issue is related to the Inverse FFT. Since a computed FFT always has the data type COMPLEX (which means real and imaginary floating point components), even after applying an inverse FFT, a small module called *Converter* was created. This module converts COMPLEX data into REAL data with the type unsigned byte. Using this module makes it possible to compute a forward FFT in an image and then get the image back by computing an inverse FFT on the forward FFT results.

3.2.4.6 FFTW Performance Study

The performance study for this module is not much different from the previous cases. Since the FFTW library takes care of all the computation, one standalone C console application was written to run the FFT, processing the same input as the one inside DX. Then, both applications were executed and run-time results were compared.

The input follows the same line as the *Add* module case (see chapter 3.2.2.6 Add Performance Study), which means that it is composed of integers from 0 until the size which is going to be tested.

Figure 3-17 shows the performance results in a QS22 Blade for computing a 1D forward FFT:

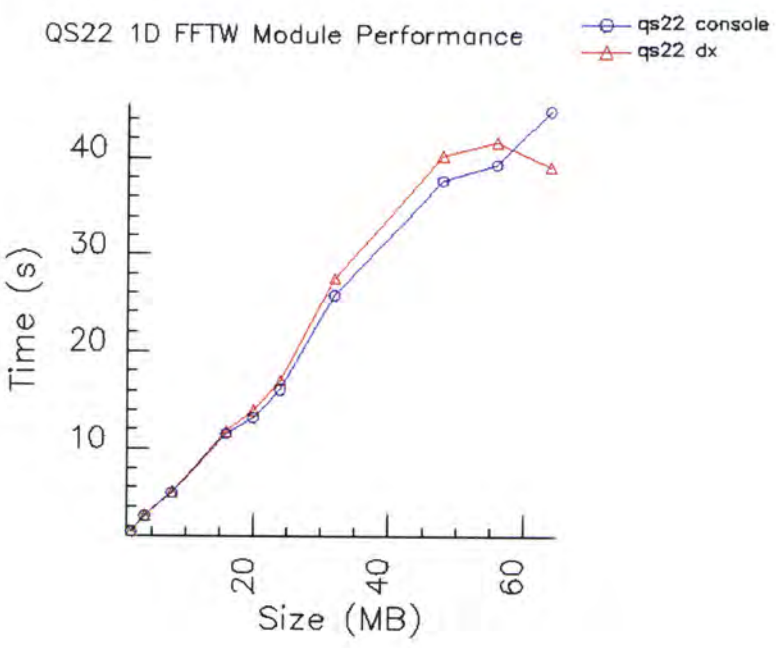


Figure 3-17 QS22 FFTW Module Performance

This example goes more inside the purpose of this thesis. Switching from the console to DX environment to perform a 1D forward FFT does not affect the performance at all, no matter the size of the input array.

One particular observation for a 64 MB input: it runs faster inside DX than in the console. Furthermore, it computes faster than the previous case with a smaller input (48 MB). Unfortunately, the FFTW library does not support bigger inputs (maybe a bug), so it is not possible to study what happens further. Anyway, in spite of being a weird result, this still shows that switching from a console to a visual programming environment does not affect the performance very badly.

3.2.5 Gaussian Blur

Gaussian blur describes blurring an image. It is a widely used effect in graphics software, typically to reduce image noise and detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen.

This process is achieved by the convolution of a Gaussian kernel with the data of an input image. Cell/B.E. Example Library, which comes with the SDK (IBM 2008), offers a set of functions to perform convolutions and DX offers a way to access image data, so the only thing left to implement is a connecting bridge between these two "sides". The Gaussian kernel used for the convolution is a 3x3 matrix obtained from the following formula (Wikipedia 2008):

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}},$$

with a standard deviation (σ) of 0.84089642 (x and y are integers which go from -1 to 1). This module serves merely for demonstration purposes, so no options were introduced to change the size of the kernel and the standard deviation.

3.2.5.1 Image Processing in DX

Before going forward in this chapter, it is important to understand how DX process images. Figure 3-18 shows the pixel order of an image:

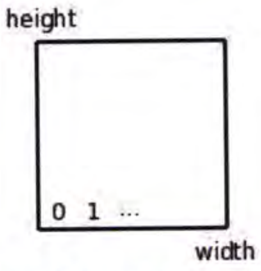


Figure 3-18 Pixel order in DX images

Pixels are processed from the lower left corner to the top right corner. Each line has a **width** size and a **height** number of lines.

Each pixel is composed of three elements: one for red, another for green and another for blue. Each element has the type unsigned byte, which means that each value goes from 0 to 255.

So, for example, if it is desired to access the contents of the first pixel in the second line, then the pixel which should be looked for is the number **width** (pixels in the first line goes from 0 to width – 1). But, since each pixel has three elements, the desired pixel is in the position **width*3**. **Width*3** contains the red element, **width*3 + 1** contains the green element and **width*3 + 2** contains the blue element. The three elements together form the first pixel in the second line.

In resume, to access the red value of a pixel on a **i** position inside of an **image**: red_val = image[3*i]. For the green value: green_val = image[3*i+1] and for the blue value: blue_val = image[3*i+2].

3.2.5.2 *Gaussian mdf*

The mdf for the Gaussian module is pretty straightforward. Only one input field and one output field. The input must be a field image or a computed FFT field of an image and the output is a field with the same attributes as the input one:

Example 3-15 Gaussian mdf

MODULE	Gaussian
CATEGORY	Cell
DESCRIPTION	Applies the Gaussian smoothing to a image
INPUT	data; field; NULL; input image
OUTPUT	result; field; new image

3.2.5.3 *Gaussian Implementation*

The implementation for this module does not reuse any code, so this time everything was implemented from the scratch.

Again, a small part of the code is responsible to handle the input. If everything is fine, then the program creates a pointer to the input and another one to the output and calls one of two functions: **do_gaussian** or **do_gaussian_complex**. Both have the same behaviour but work with different types of data. First one handles unsigned byte data (type of data used to process images in DX) and the second one handles floating point data (one is a real component and the other is the imaginary component, like the output from the FFT).

A code sample for the function **do_gaussian** is bellow:

Example 3-16 C code for the Gaussian Implementation

```
int do_gaussian(ubyte *in_data, int in_data_size,
                ubyte *out_data, int out_data_size,
                int width)
{
  int i, j, width2 = find_next_multiple(width,16);

  /*
   * Three lines of data for each color (red, green and blue)
   * are processed at a time
   */
  float line1r[width2] __attribute__ ((aligned(16)));
  float line2r[width2] __attribute__ ((aligned(16)));
  float line3r[width2] __attribute__ ((aligned(16)));
  float line1g[width2] __attribute__ ((aligned(16)));
  float line2g[width2] __attribute__ ((aligned(16)));
  float line3g[width2] __attribute__ ((aligned(16)));
  float line1b[width2] __attribute__ ((aligned(16)));
  float line2b[width2] __attribute__ ((aligned(16)));
  float line3b[width2] __attribute__ ((aligned(16)));

  /*
   * Each convolution returns a output line for each color.
```

```
    * The function which processes the convolution requires
    * the data to be 16 bytes aligned for both input and output
    * and the size of the line must be a multiple of 16
    */
  float outr [width2] __attribute__ ((aligned(16)));
  float outg [width2] __attribute__ ((aligned(16)));
  float outb [width2] __attribute__ ((aligned(16)));

  /*
   * The 3*3 Kernel. All the values must be replicated along
   * all the vector. Matrix calculated using the Gaussian
   * function with a standard deviation of 0.84089642
   */
  vec_float4 m[9];
  m[0][0] = 0.0671783991; m[0][1] = 0.0671783991; m[0][2] =
0.0671783991; m[0][3] = 0.0671783991;
  m[1][0] = 0.1234373547; //Replicate...
  m[2][0] = 0.0671783991; //Replicate...
  m[3][0] = 0.1234373547; //Replicate...
  m[4][0] = 0.2375369846; //Replicate...
  m[5][0] = 0.1234373547; //Replicate...
  m[6][0] = 0.0671783991; //Replicate...
  m[7][0] = 0.1234373547; //Replicate...
  m[8][0] = 0.0671783991; //Replicate...

  //Unused positions are filled with neutral values
  for (i = width; i < width2; i++)
    {
      line1r[i] = 127.0;
      line2r[i] = 127.0;
      line3r[i] = 127.0;
      line1g[i] = 127.0;
      line2g[i] = 127.0;
      line3g[i] = 127.0;
      line1b[i] = 127.0;
      line2b[i] = 127.0;
      line3b[i] = 127.0;
    }

  //Load the first three lines
  i = width;
  for (j = 0; j < width; j++)
    {
      line1r[j] = (float)in_data[3*(i+width+j)];
      line2r[j] = (float)in_data[3*(i+j)];
      line3r[j] = (float)in_data[3*(i-width+j)];
      line1g[j] = (float)in_data[3*(i+width+j)+1];
      line2g[j] = (float)in_data[3*(i+j)+1];
      line3g[j] = (float)in_data[3*(i-width+j)+1];
      line1b[j] = (float)in_data[3*(i+width+j)+2];
      line2b[j] = (float)in_data[3*(i+j)+2];
      line3b[j] = (float)in_data[3*(i-width+j)+2];
    }

  /*
   * Each variable contains three pointers for three lines.
   * One variable for each color
   */
  const float *inr[3], *ing[3], *inb[3];
  inr[0] = &line1r[0];
  inr[1] = &line2r[0];
```

```
inr[2] = &line3r[0];
ing[0] = &line1g[0];
ing[1] = &line2g[0];
ing[2] = &line3g[0];
inb[0] = &line1b[0];
inb[1] = &line2b[0];
inb[2] = &line3b[0];

//Start the computation
float *tempr, *tempg, *tempb;
for (i = width; i < in_data_size - width; i += width)
  {
    conv3x3_1f(inr,outr,m,width2);
    conv3x3_1f(ing,outg,m,width2);
    conv3x3_1f(inb,outb,m,width2);
    for (j = 0; j < width; j++)
    { //Convert the floats into ubytes
      out_data[3*(i+j)] = (ubyte)outr[j];
      out_data[3*(i+j)+1] = (ubyte)outg[j];
      out_data[3*(i+j)+2] = (ubyte)outb[j];
    }

    //Prepare the next computation
    tempr = (float *)inr[2];
    inr[2] = inr[1]; //The new 3rd line is the previous 2nd line
    inr[1] = inr[0]; //The new 2nd line is the previous 1st line
    inr[0] = tempr;  //The new 1st line needs to be loaded
    tempg = (float *)ing[2];
    ing[2] = ing[1];
    ing[1] = ing[0];
    ing[0] = tempg;
    tempb = (float *)inb[2];
    inb[2] = inb[1];
    inb[1] = inb[0];
    inb[0] = tempb;

    for (j = 0; j < width; j++) //Load the new line
    {
      tempr[j] = in_data[3*(i+width+j)];
      tempg[j] = in_data[3*(i+width+j)+1];
      tempb[j] = in_data[3*(i+width+j)+2];
    }
  }

/*
 * No transformation performed in the first and last
 * lines of the image
 */
for (i = 0; i < width; i++)
  {
    out_data[3*i] = in_data[3*i];
    out_data[3*i+1] = in_data[3*i+1];
    out_data[3*i+2] = in_data[3*i+2];
  }
for (i = in_data_size - width; i < in_data_size; i++)
  {
    out_data[3*i] = in_data[3*i];
    out_data[3*i+1] = in_data[3*i+1];
    out_data[3*i+2] = in_data[3*i+2];
  }

```

```
    return OK;
}
```

3.2.5.4 *Compiling and Running Gaussian*

The only thing needed to do when compiling this module is to add the libimage.a to the linked libraries (usually found in: /opt/cell/sdk/usr/lib/libimage.a).

Figure 3-19 shows an example of a visual program using the Gaussian module (and also the Converter module, see chapter 3.2.4.5 FFTW Implementation Notes):

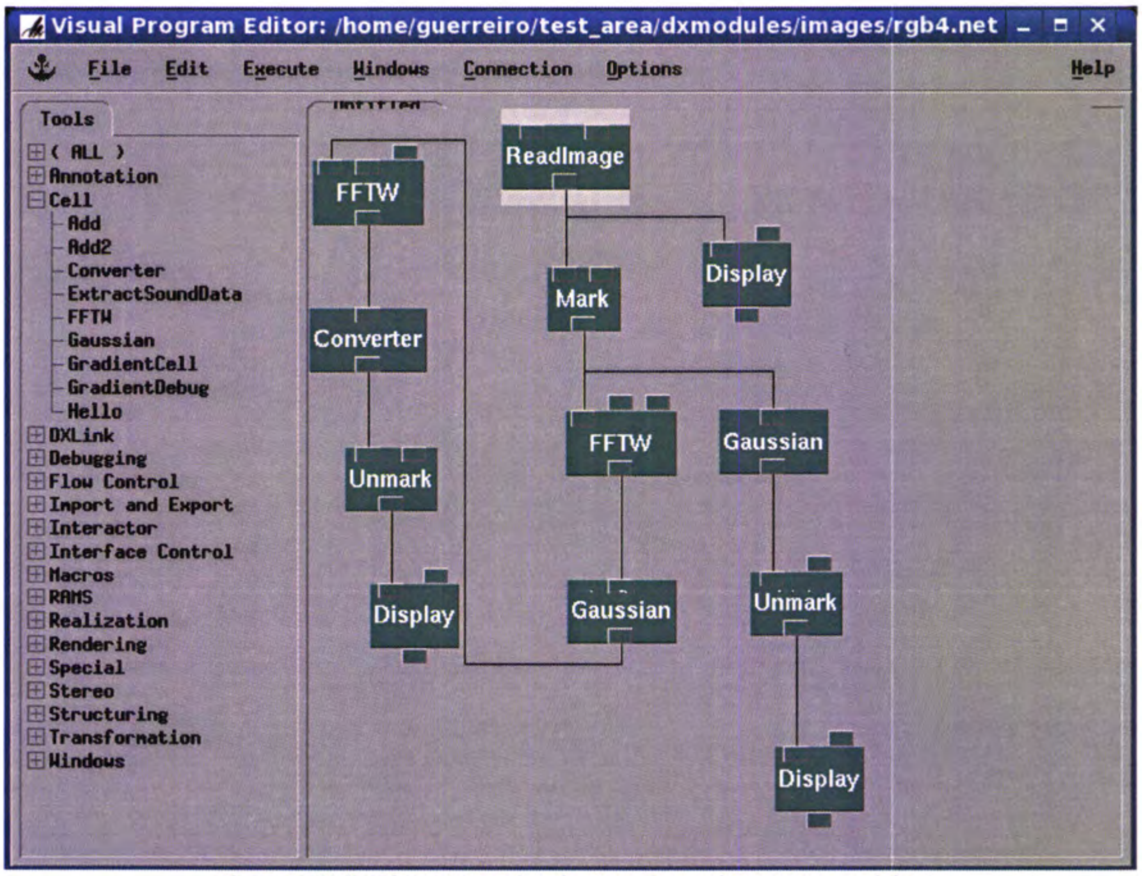


Figure 3-19 Visual program using the Gaussian and Converter modules

This program produces three display outputs: the first output (Figure 3-20) is the original image; the second output (Figure 3-21) is the previous image transformed after processing a Gaussian Blur effect and the third output (Figure 3-22) is the result image after applying a forward FFT, then a Gaussian Blur and finally an Inverse FFT. A Converter module is used in this last example in order to transform the floats from the inverse FFT back to unsigned byte data.

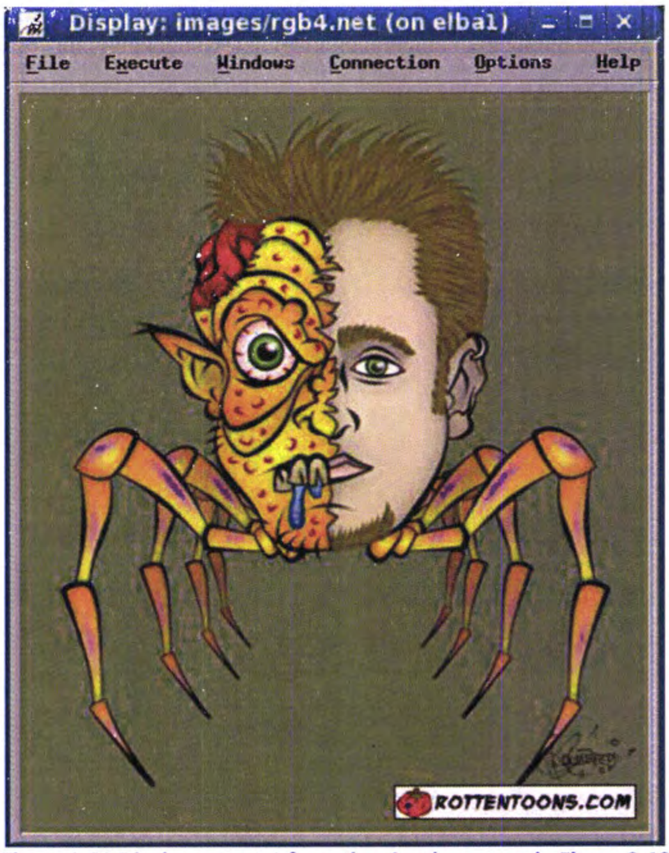


Figure 3-20 Display output 1 from the visual program in Figure 3-19

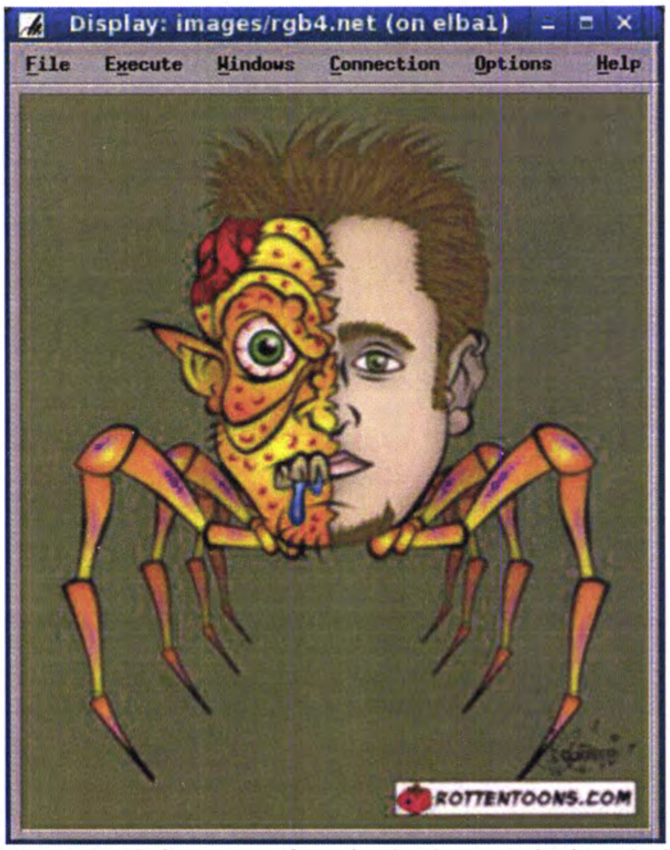


Figure 3-21 Display output 2 from the visual program in Figure 3-19

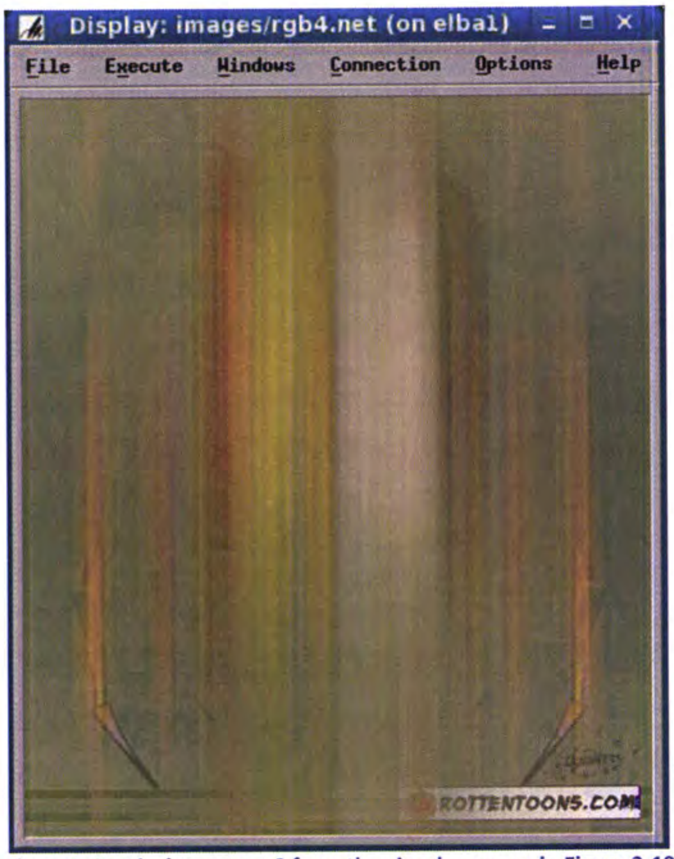


Figure 3-22 Display output 3 from the visual program in Figure 3-19

3.2.5.5 *Gaussian Implementation Notes*

There is not much to discuss about the implementation of this module. The main issue is to manage the pixels and apply the convolution.

In order to keep the program simple, there is no policy regarding the sharps of the image (like the first and last lines and the first and last columns). First and last lines are not computed and for the first and last columns the default policy of the Cell/B.E. SDK Example library, which calculates an average value to these positions, is used.

Concerning the returning floats from the convolution function, the results are directly converted (and possibly rounded) back to unsigned bytes. For example, if a returning value from the convolution operation is lower than 0 then it is rounded to 0 and if it is bigger than 255, then it is rounded to 255. Returned values between 0 and 255 are rounded to units.

3.2.5.6 *Gaussian Performance Study*

The Gaussian Blur effect in different images with different sizes was computed for this case study.

Figure 3-15 and Figure 3-19 show a module called *ReadImage* in the top of the visual program. This module makes life easier when reading images inside DX, but a solution had to be found in order to perform the same computation in the console side. So, with the help of the *Export* module it was possible to create an output file with all the colours for all the pixels inside of an image. The number of lines of this output file is the same number as input pixels and each line has 3 integers: one for the red value; another for the green value and the last one for the blue value. This file is loaded in the console program before computing the Gaussian blur.

Since this process of loading the image in the console side takes too much time, this time is not counted. Time starts counting right after the image is loaded and stops until the output buffer is loaded with the returning values.

Figure 3-23 shows the performance results for running the Gaussian in the QS22 Blade:

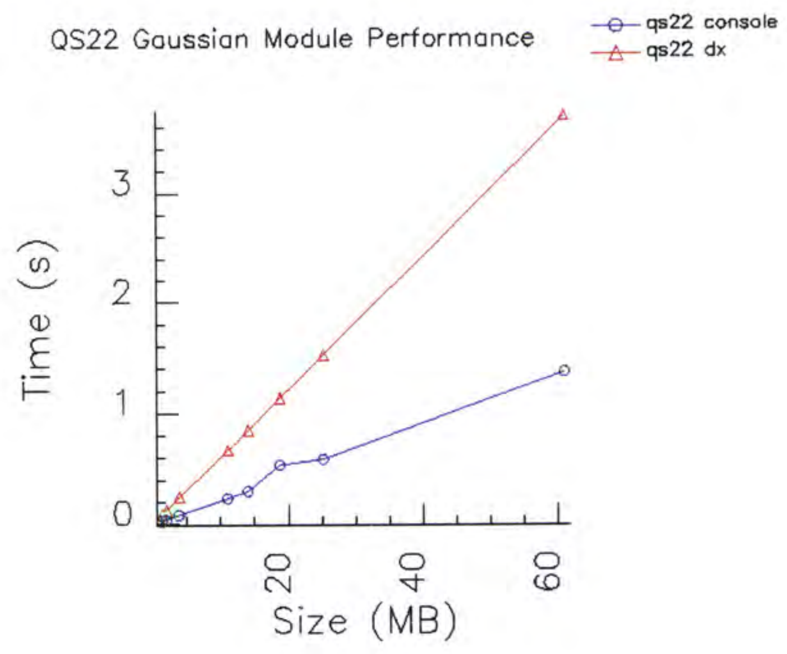


Figure 3-23 QS22 Gaussian Module Performance

The size of the images referenced in the X-axis refers to images of type TIFF. Images of this type occupy more space in disk but work on-the-fly inside DX.

Looking to the plot, this case shows up as the worst example among the all modules. The difference of the computing time between the console and DX keeps increasing as the input sizes increases.

This difference is absolutely not related to the input checking that DX performs, but to its processing. Copying the input data from the data structure given by DX to the Gaussian input structure is the operation which consumes most of the time. Copying the Gaussian output data back to the DX output structure consumes the remaining difference.

Figure 3-24 shows a different approach on this test. Instead of using DX structures for input and output, the program uses allocated local structures. The image is loaded using the same method as in the console and, again, the time needed to load the image is not counted.

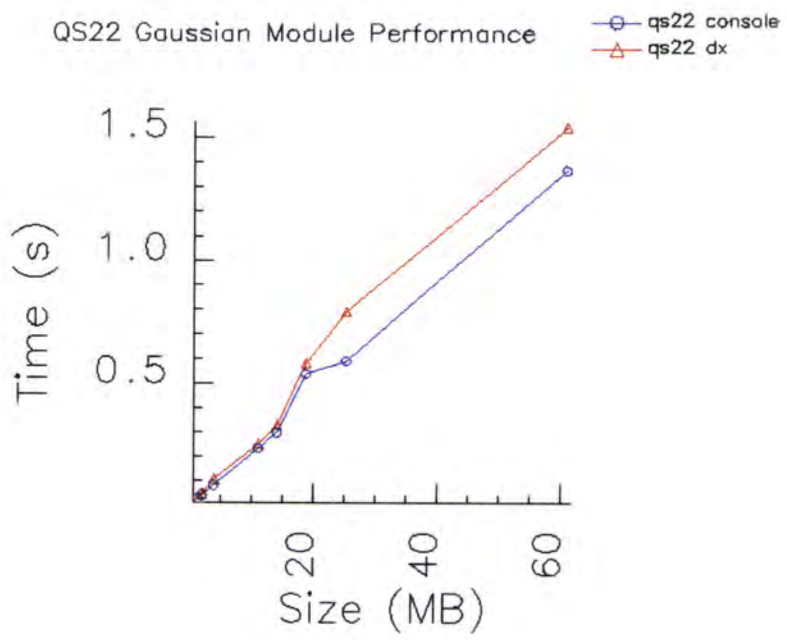


Figure 3-24 QS22 Gaussian Module Performance (DX Improved)

The test in Figure 3-24 shows no significant difference between the console and DX in the stated conditions.

Nevertheless, it is always possible to optimize DX structures and the way they organize themselves in the memory. The FFTW performance case study in Figure 3-17 proves it. Such structures are not easy to implement and future work can be developed to ease this process, but most importantly it is possible.

3.2.6 Extract Sound Data – A Different Example

Besides all the background work developed in this thesis exploring the advantages of the visual programming, this small sub-chapter introduces one practical case study about how visual programming can make some tasks easier.

The proposed situation is to visualize the FTT of three different musics and compare their signals. If the approach taken to do so is based in an implementation of a standalone console program, then the programmer has to code: the loading method of the musics into memory; the FFT of the musics and a visualization method. Also, the programmer must implement a set of flags to define which type of FFT shall be performed and how the visualization of it looks.

In DX, the user only needs to connect the modules without worrying about the computation and inside these modules it is possible to set different flags defining how the computation is processed. For computing the FFT and to visualize the data the necessary tools are already implemented, so the only thing left to do is to create a module which loads music.

The *ExtractSoundData* module generates a floating point 1-D field from a sound file (specified in the input) which can then be used by the FFT module (or any other compatible one). The libsoundfile (Mega Nerd 2008) is used in the implementation for this module.

The visual program in Figure 3-25 extracts the information from three music files, performs a forward FFT and then displays a plot with the signal of the three different music files:

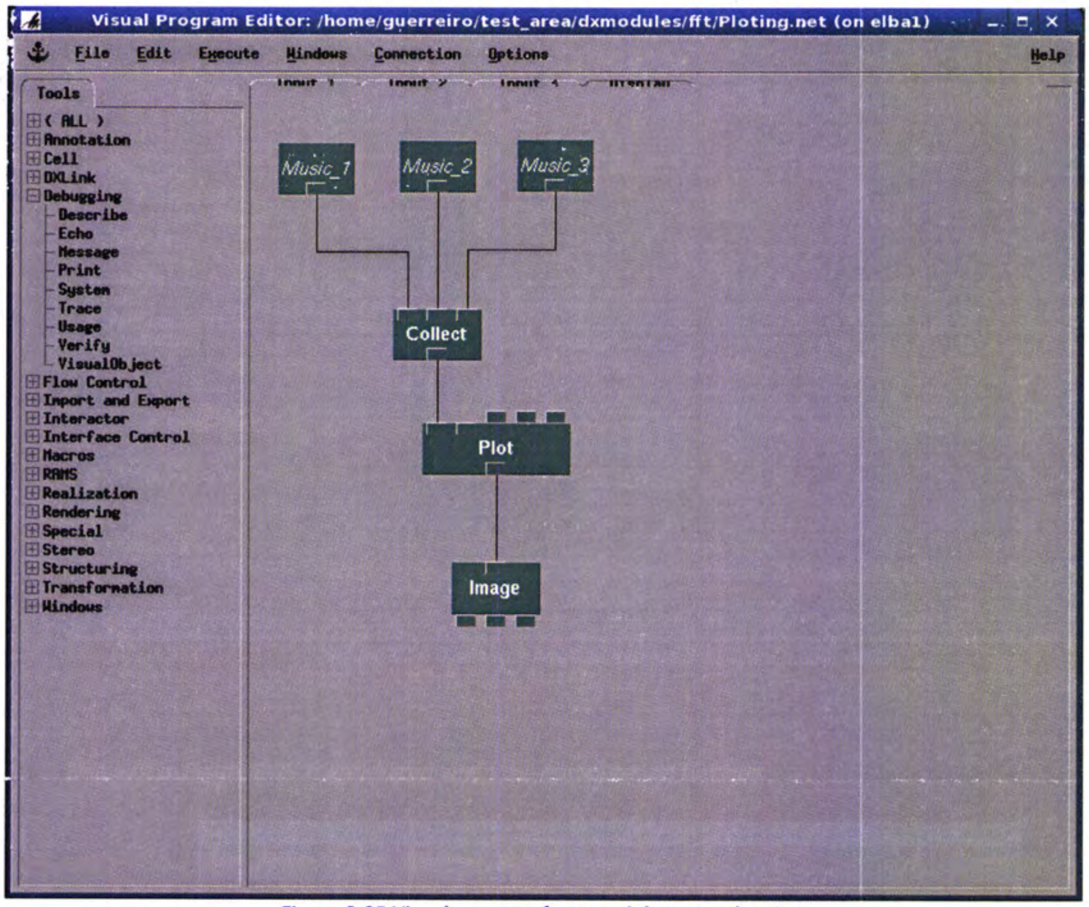


Figure 3-25 Visual program for sound data visualization

The modules *Music_1*, *Music_2* and *Music_3* have exactly the same structure. Only the path to the input music in the *ExtractSoundData* module differs. Figure 3-26 shows the visual program for the ***Music_1*** module:

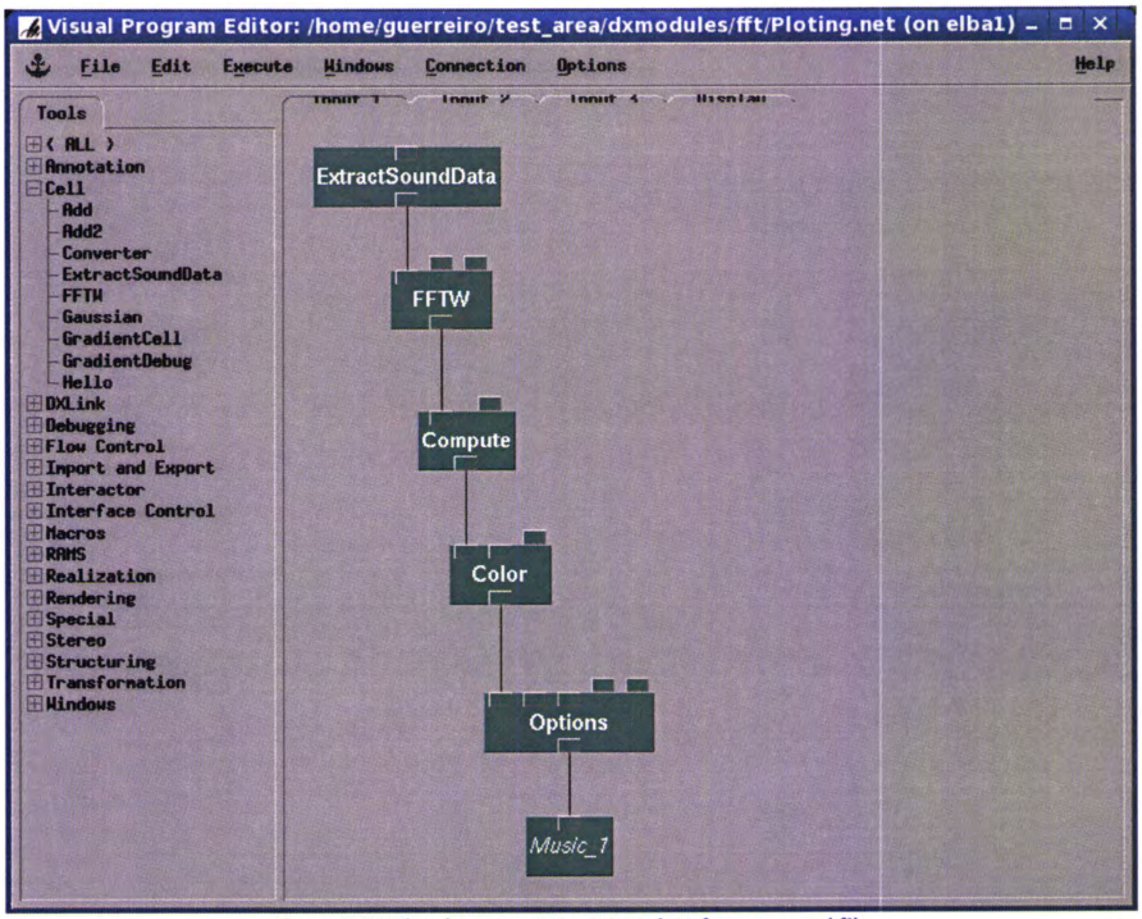


Figure 3-26 Visual program to extract data from a sound file

The plot in Figure 3-27 is generated when the computation finishes:

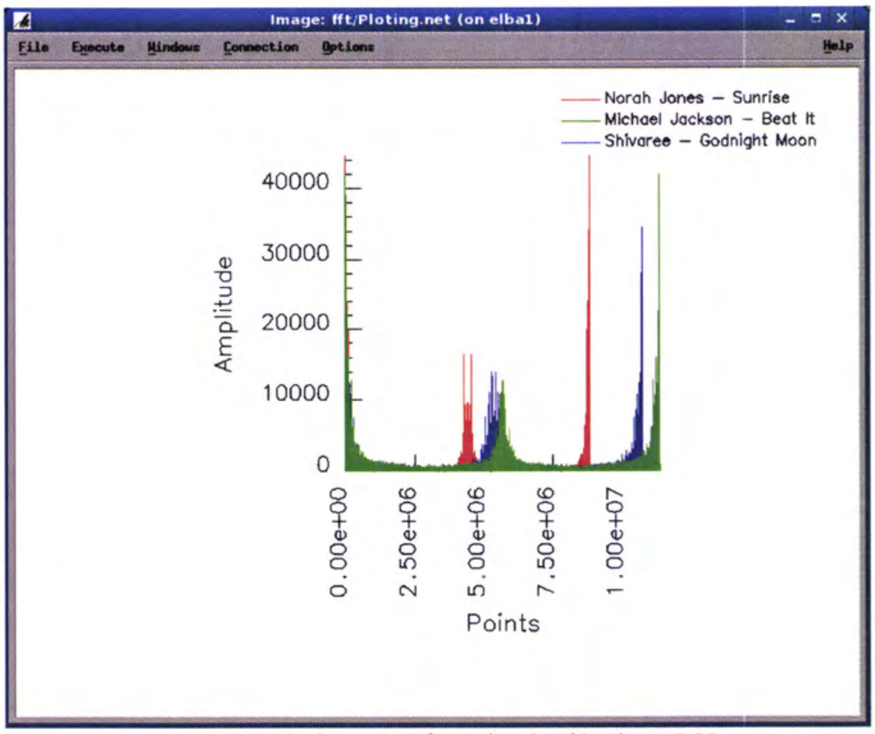


Figure 3-27 Display output from the visual in Figure 3-25

The plot shows the signal for the three different music files. The important thing here is not what these results mean, but how simple can it be to implement a program using a visual programming environment.

3.3 Improving OpenDX Performance

The second part of the practical work in this thesis is to show how OpenDX performance can be improved using the Cell/B.E. capabilities.

The process consists of grabbing an already implemented DX module and then optimize its code. Unfortunately, it is not possible to simply compile and run a DX module inside a SPE because of its differences with the PPE (see chapter 2.1 Cell / Heterogeneous Multi-core Environment).

The Gradient module inside DX is a good candidate since its computation resides inside a loop applying an algorithm to each element of an array. Dividing and parallelizing this loop into different cores, as different and independent tasks, may improve the performance.

This chapter introduces the GradientCell module which is an optimized version of the Gradient module. Like the structure for each implemented module in the chapter 3.2 Cell/B.E. Applications in OpenDX, this case study shows the GradientCell: module description file; implementation; how to compile and run; implementation notes and performance study.

3.3.1 GradientCell

The approach taken to optimize this module does not involve a deep knowledge of the Gradient operation. There is a part in the code which has a computing-intensive cycle and its number of iterations depends on the size of the input, so the goal is to: divide and parallelize this cycle across the SPEs (in simultaneous tasks); put the result back in the main memory and then check whether the results are consistent.

Only 1-D and 2-D regular fields were optimized. The remaining code is unchanged.

3.3.1.1 GradientCell mdf

The mdf for the GradientCell module does not introduce anything new when compared with the regular Gradient module, so its contents (except the name) are exactly the same:

Example 3-17 GradientCell mdf

```
MODULE        GradientCell
CATEGORY      Cell
DESCRIPTION   Computes the gradient of a scalar field
INPUT         data; scalar field; NULL; field to compute gradient of
```

INPUT	**method [visible:0]; string; "manhattan"; method to use**
OUTPUT	**gradient; vector field; gradient field**

3.3.1.2 GradientCell Implementation

Like the Add module (see chapter 3.2.2 Add), the implementation for the GradientCell module is divided in three sections: GradientCell Shared Structure; PPU GradientCell Implementation and SPU GradientCell Implementation.

3.3.1.2.1 GradientCell Shared Structure

Bellow is the shared structure between the PPE and the SPE:

Example 3-18 GradientCell shared structure

```
#ifndef GRADIENT_H_
#define GRADIENT_H_

#include <dx/dx_spu.h> /* SPU may need some information from here */

typedef struct {
  void *data_address;
  void *vectors_address;
  float *deltas_address;
  int *permute_address;
  int data_nItems;
  Type data_type;
  int nDim;
  int pFlag;
  int first_cycle;
  int last_cycle;
  int spe_num;
  int xKnt;
  int yKnt;
  int zKnt;
  int start;
  int offset;
  unsigned char pad[64];
} spu;

typedef union
{
  unsigned long long ull;
  unsigned int ui[2];
}
addr64; /* linkage stuff used when calling the SPU program */

#define ELEM_PER_BLOCK 1024
#define MAX_SPU_THREADS 16

#define spu_mfc_ceil128(value) ((value + 127) & ~127)
#define spu_mfc_ceil16(value) ((value + 15) &  ~15)

// calculates the n=(x,y)
// input: n and y-axis size
void find_2D_coordinates(int n, int y_size, int *x, int *y)
{
  int pos_x = abs(n/y_size);
  int pos_y = n % y_size;

  if (pos_y == 0){
    pos_x--;
    pos_y = y_size - 1;
  }
  else
    pos_y--;

  x[0] = pos_x;
  y[0] = pos_y;
}

#endif /* GRADIENT_H_ */
```

This structure may look confusing, but everything makes sense when the PPU and SPU GradientCell implementations are introduced in the next sub-chapter.

The structure contains four addresses to the main memory: **data_address** contains the address for the input values (which have a **data_type**); **vectors_address** contains the address for the output values; **deltas_address** contains the address for the delta values and **permute_address** contains the address for permuting values which may be used or not (depending on the **pFlag** value). The remaining values are: **data_nitems**, which contains the input's number of items; **offset**, which describes how many values are going to be computed in the corresponding **spe_num**; **nDim**, which contains the dimension of the input field (must be 1-D, 2-D or 3-D); **first_cycle** and **last_cycle**, which are activated flags in the case of the correspondingly first or last set of data is computed in the SPE; **xKnt**, **yKnt** and **zKnt**, which correspondingly contain the size of the X-axis, Y-axis and Z-axis and **start**, which contains the starting point for the computation in the SPE.

The function **find_2D_coordinates** is used in a 2-D regular field input case. It transforms a specified array position into the corresponding (x,y) coordinates to the DX data structure.

3.3.1.2.2 PPU GradientCell Implementation

First of all, the original computation for the Gradient inside a field is divided in two cases: regular or irregular field. If the field is irregular then the code is unchanged.

Otherwise, the structure in the previous chapter is initialized and the computation starts the SPE threads. The function **doGradientRegular** inside the file **gradient.c** in the source code has a macro called **RUN_SPU** responsible for that:

Example 3-19 C code for the PPU GradientCell Implementation

```
define RUN_SPU(type)                                              \
  {                                                               \
    type  *data;                                                  \
                                                                  \
    vectors = (float *)DXGetArrayData(outArray);                  \
    data    = (type  *)DXGetArrayData(inArray);                   \
                                                                  \
    int acc = 0;                                               \
    for (i = 0; i < spu_threads; i++)                          \
      {                                                        \
            stuff[i].spe_num = i;                              \

            stuff[i].permute_address = permute;                \
            stuff[i].data_address = (void *)data;              \
            stuff[i].vectors_address = (void *)vectors;        \
            stuff[i].data_nItems = nItems;                     \
            stuff[i].nDim = nDim;                              \
            stuff[i].deltas_address = deltas;                  \
            stuff[i].offset = offset;                          \
                                                               \
            if ( i == 0 )                                      \
                  stuff[i].first_cycle = 1;                    \
            else                                               \
                  stuff[i].first_cycle = 0;                    \
                                                               \
            acc += offset;                                     \
            switch(nDim)                                       \
            {                                                  \
                  case 1:                                      \
                  if ( i == spu_threads - 1 )                  \
                  {                                            \
                        stuff[i].last_cycle = 1;               \
                        stuff[i].xKnt = nItems-offset*(spu_threads-1);\
                  }                                            \
                  else                                         \
                  {                                            \
                        stuff[i].last_cycle = 0;               \
                        stuff[i].xKnt = offset ;               \
                  }                                            \
                  break;                                       \
                                                         \
                  case 2:                                      \
                  stuff[i].start = acc-offset+1;               \
                  stuff[i].xKnt = counts[0];                   \
                  stuff[i].yKnt = counts[1];                   \
            }                                                  \
            param[i].spe_argp=&stuff[i];                       \
            /* Omitted code to start and run the SPE threads */  \
      }                                                        \
      /* Omitted code to wait for the SPE threads */           \
      /* to complete and destroy the context information */    \
  }
```

All the values except spu_threads, stuff and offset already exist in the original code version and that is the reason why the shared structure in the previous chapter (see 3.3.1.2.1 GradientCell Shared Structure) looks so confusing. These values are necessary in the computational part and must be copied from the main memory to the SPE LS.

The new values: spu_threads contains the number of SPUs to use; stuff refers to a variable of the type spu which is the previously introduced shared structure and offset contains the number of items to process in determined SPE.

All the computation which is supposed to be taken in this section (in the original version) is "postponed" to the SPEs execution.

3.3.1.2.3 SPU GradientCell Implementation

On the SPU side one of three things can happen: the input corresponds to a 1-D, 2-D or 3-D field. If it is 3-D field, then no computation is made since this process is not optimized.

3.3.1.2.3.1 1-D Input Field

For the 1-D field input case, the computation works in the following way: assuming **input** as the input array and **output** as the output array, then for each position **i** (from the beginning to the end of array): output[i] = (input[i+1] – input[i-1]) * value, where **value** is a previous calculated number. If **i** equals the first position or the last position of the array, then the first or the last positions are used in the formula.

Figure 3-28 illustrates the computation procedure:

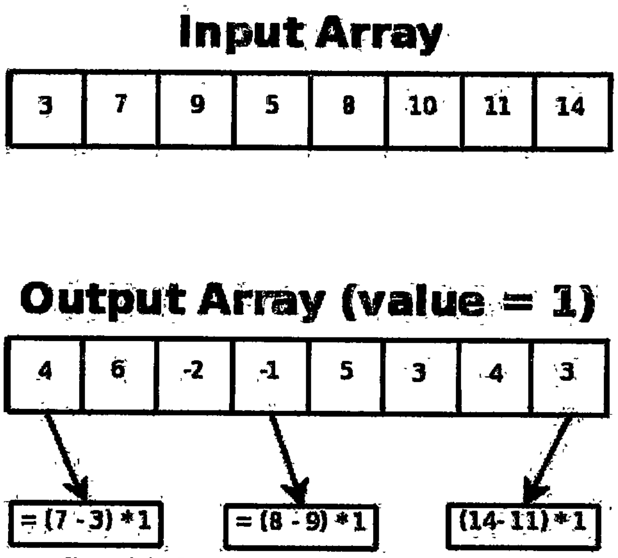


Figure 3-28 Computation of a gradient in a 1-D field

This brings up a problem when implementing the DMA transfers to prepare the data for the computational part. Since the DMA transfers get data from point A to point B, then computing the point A can be cumbersome because the data right before A is needed. The same happens for computing the point B, since the value right after B is also needed.

So, for each DMA transfer (excepting the first one and the last one), two more DMA transfers are performed to get the data before A and after B in order to perform the computation. The variable with the information needed by point B is called **special_one** and the variable with the information needed by point A is called **special_two**.

Besides the Double Buffering technique (see chapter 3.2.2.1 Double Buffering), the remaining code to compute the gradient, even in the SPE side, is the same. This means that no vector SIMD is used.

3.3.1.2.3.2 2-D Input Field

For a 2-D input field, the implementation gets more complex.

First of all, the output size is two times bigger than the input size and, for each computed value, two output slots are used.

The computation takes two embedded cycles where the first cycle goes from 0 until the size of **xKnt** and the embedded cycle inside goes from 0 until the size of **yKnt**. These variables are inside of the shared structure (see chapter 3.3.1.2.1 GradientCell Shared Structure) and correspond to the size of the X and Y axis.

For example, computing the gradient for a 2*5 input Field (size of X axis is 2 and size of Y axis is 5) with the data {3, 7, 9, 5, 8, 10, 11, 14, 15, 17} and assuming a constant **value** = 1 results in the following output:

- For the Y values (second component) the computation stays the same as in the 1-D case. Then the output for the Y values is: {4, 6, -2, -1, 3, 1, 4, 4, 3, 2} (**NOTE**: there are two Y arrays with 5 elements each one)
- For the X values the computation is a little bit different. Instead of getting the previous and following values, the algorithm gets the Y size (in this case 5) previous value and the Y size following value for the current computation. Like the 1-D case, if the cycle is in the beginning, then the previous value is the current value and if the cycle is in the end, then the current value is used instead of a following one. The output for the X values are: {7, 4, 5, 10, 9, 7, 4, 5, 10, 9}

Figure 3-29 illustrates the example:

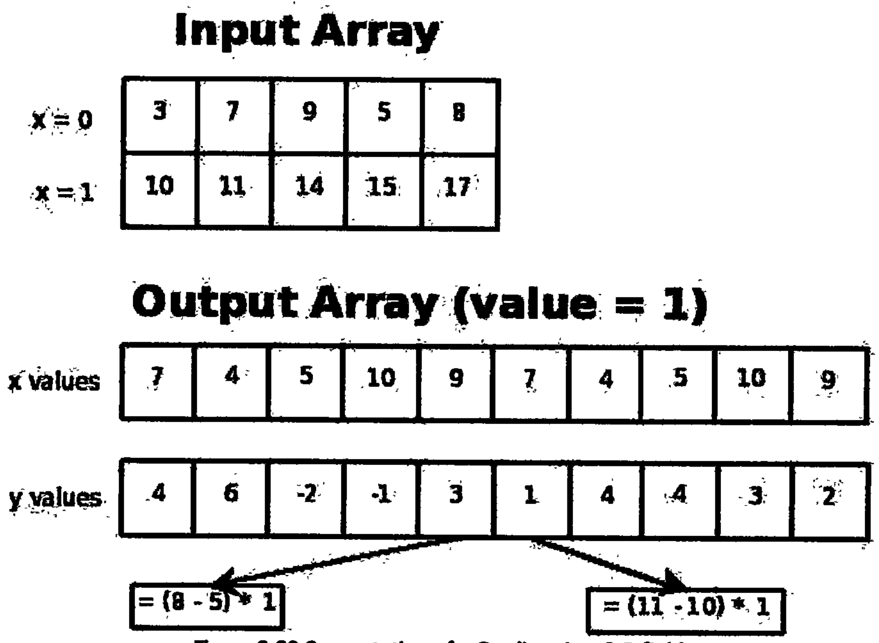


Figure 3-29 Computation of a Gradient in a 2-D field

This issue makes the DMA transfers even more complicated. Besides the **special_one** and **special_two** (see chapter 3.3.1.2.3.1 1-D Input Field), two more DMA transfers are used in order to compute the X values. If the current buffer corresponds to data in the middle of the X axis, then Y size previous and following values are also transferred. In resume, for each computed buffer four DMA transfers (on the most) are performed in order to get: the value right before the beginning of the buffer; the value right after the end of the buffer; Y size values before the beginning of the buffer and Y size values after the end of the buffer (**NOTE**: the first two DMAs can be omitted but that would require a big restructure on the code without implying a big performance improvement).

Also, a function was implemented which transforms a 1-D coordinate into a 2-D coordinate in order to keep the original algorithm and code in the computational part. For the previous example, the 6th element (which is 10) corresponds to the point (1,0). This means first point (0) in the second line (1).

This implementation also uses Double Buffering (see chapter 3.2.2.1 Double Buffering) and the rest of the code is unchanged.

3.3.1.3 *Compiling and Running GradientCell*

Compiling the GradientCell module is not any different from the Hello Example (see chapter 3.1.5.4 Compiling and Running the Hello Example). Figure 3-30 shows an example of a visual program using the GradientCell module:

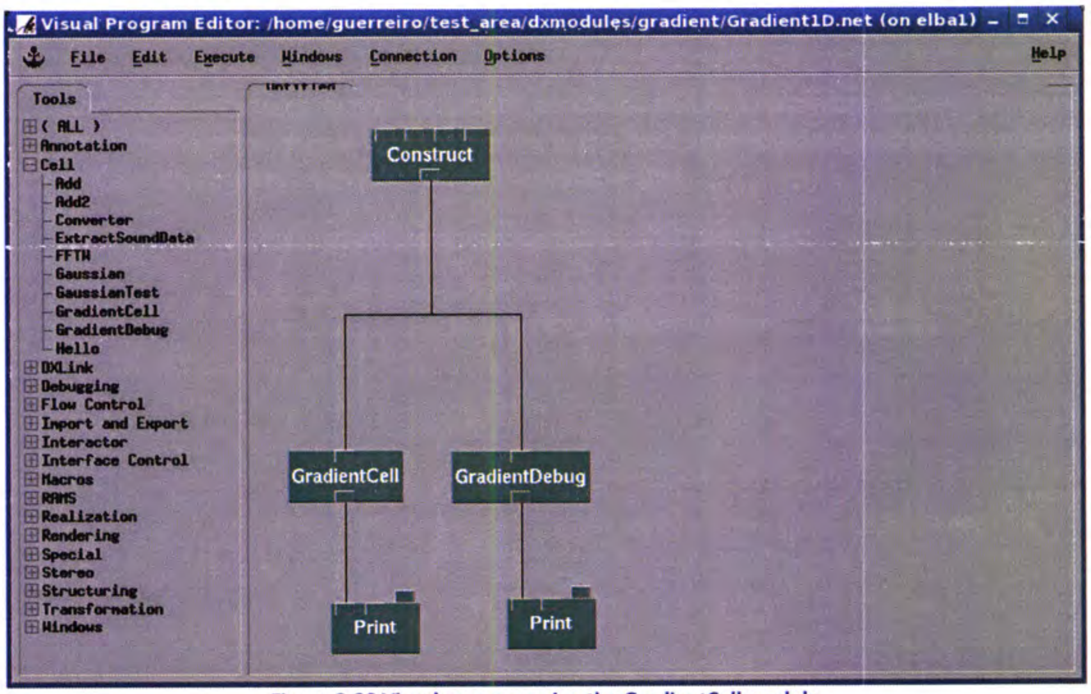


Figure 3-30 Visual program using the GradientCell module

The *Construct* module creates a 1-D field with 67108864 integers (64 MB) which have values reaching from 3 to 67108866. The input is given to the implemented module and to the *GradientDebug* module. This last module, which has exactly the same code of the original *Gradient* module, was implemented in order to compare the run-time for the performance study (see chapter 3.3.1.5 GradientCell Performance Study).

The *Print* module prints the contents for both the computations and is used in order to verify that the *GradientCell* module returns the same output as the *GradientDebug* module.

3.3.1.4 GradientCell Implementation Notes

The input size for the GradientCell module must be a multiple of the number of SPEs used. Contrasting to the implementation of the other modules, this module is not prepared to parallelize any input size.

This sub-chapter demonstrated how Cell/B.E. can be used in order to optimize some implemented applications. The implementation of this module consisted of optimizing some already written code, but another approach could be to write the application from the scratch.

The implemented solution took an easy approach, but far away from taking advantage of all the Cell/B.E. capabilities. Better results may appear if all the code is written from the scratch (but that also may consume more time and resources). When the programmer wants to improve the performance of an already existent application using the Cell/B.E. capabilities he/she must decide which option is the best depending on the main objectives.

3.3.1.5 GradientCell Performance Study

The performance study for the GradientCell module is based on a comparison between its run-time and the run-time of the original Gradient module, in the QS22 Blade. Gradient module runs only inside the PPE and the GradientCell module uses all the available SPEs.

Figure 3-31 shows the gradient performance graph for a 1-D field input:

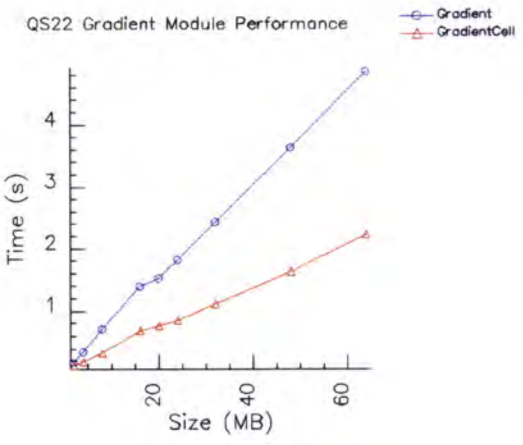


Figure 3-31 QS22 GradientCell Module Performance

Like all the other performance study tests presented in chapter 3.2 Cell/B.E. Applications in OpenDX, the **Size** axis refers to the input size in Megabytes, which is composed by integers going from 0 until **Size-1**, and the **Time** axis refers to its execution time in seconds.

Figure 3-31 shows that there is some performance improvement by the GradientCell module. For a 60MB input this difference is more than 2 seconds.

Figure 3-32 shows the same performance study for a 2-D input field:

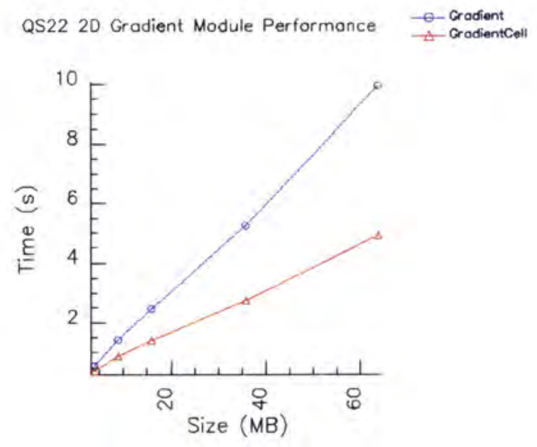


Figure 3-32 QS22 GradientCell Module Performance (2D Input)

In this example, the difference gets bigger in spite of the input size being the same. For the 60MB input case, the run-time difference between the GradientCell module and the Gradient module is more than 4 seconds.

It is possible to argue that the effort taken to optimize the performance for this module is not worth it, but there are a couple of things that must be kept in mind:

- No vector SIMD instructions were used;
- Memory organization could be re-arranged;
- Only a small part of the code was optimized.

It is possible to get better performance results, but an extra effort is required for the programmer in order to do so. Also, it may be a better idea to re-implement the module from the scratch instead of optimizing already written code.

4 Conclusion

Two different concepts were introduced in this thesis: Cell/B.E. and visual programming. Cell/B.E. came up as a solution to improve application performance, but programming for it can be quite challenging. On the other side, visual programming demonstrated to be an easier and intuitive programming paradigm.

So, combining these two concepts without losing performance offers the programmers around the world an easier way to take advantage of the Cell/B.E. capabilities. OpenDX was the platform chosen in order to do so since it is possible to implement modules for it in C language, which is supported by Cell/B.E.

Several modules were then implemented: Hello, Add, Add2, FFTW and Gaussian. Hello module introduced how to start SPE threads. Add and Add2 modules introduced parallelization and SIMD instructions. FFTW and Gaussian modules introduced the use of existing libraries, which are optimized for Cell/B.E. All this modules show that it is possible to use the full Cell/B.E. capabilities inside a visual programming environment. The final user of the application just connects the modules without caring about particularities regarding the architecture where the application is running.

For each implemented module a performance study comparing their run-times (in a QS22 blade) between running them inside console or DX was introduced. The study showed that it is possible to switch the environment without losing performance, but attention must be paid to the memory structures. DX memory structures carry the inputs and outputs from module to module without problems, but they must be optimized in order to not lose performance. That is proven by FFTW and Gaussian modules performance study. The first one uses an optimized memory organization and there is no big loss of performance and the second one only achieves that by using local memory structures, which have the problem to not transmit the output along the visual program. It is possible to write a module with an optimized memory organization inside DX, but that requires an extra effort.

At last, a case study using the Gradient module was presented in order to show how performance of an already implemented module can be improved. The process consisted in parallelizing a big loop responsible for the computational part of the module and results showed some gaining in performance in spite of not being used all the Cell/B.E. capabilities. This process was far from easy and a possible better solution may be to re-implement the module from the scratch.

4.1 Future Work

In spite of DX optimistic results, there is still work that can be done in order to ease some tasks for users and developers.

The code generator of the DX Module Builder (see chapter 2.3.5 Module Builder) is a great tool and it should introduce a few more features. It would help DX developers if this builder could generate Cell/B.E. code in the user's function. For example, generate code to start, run and finish SPE-threads, and also generate a header file with a structure containing pointers to all the inputs and outputs, and integers with their size. This would give an easier way for Cell/B.E. developers to write modules.

Another issue regarding the DX Module Builder is the DX memory structure. Generating code with optimized memory structures would be a big help for the developers to keep a good performance on the Cell/B.E. applications.

The DX Compute module is a great tool and it should be a case study. Giving this module the ability to compute whatever the user wants in his/her data using the full Cell/B.E. capabilities would allow the computing of any Cell/B.E. optimized mathematical expression simply by typing it in the module.

At last, remaining future work may focus on the user interface of DX in order to make the act of visual programming more intuitive (see chapter 2.2.5 Some principles for visual language design). Some suggestions on this field are:

- Expand DX module transmitters and receivers in the main window by double clicking on them (instead of forcing the user to search for the corresponding tab)
- Option to attribute personalized icons to the modules and colours to the visual programs
- Option to edit the source code of the module in the visual program and recompile it on-the-fly

Bibliography

Agilent Technologies Inc. *Vee Pro User's Guide.* 2008. http://www.home.agilent.com/agilent/home.jspx (accessed February 16, 2009).

Andescotia. *Marten IDE 1.4.* 2008. http://www.andescotia.com/products/marten/ (accessed February 16, 2009).

Arevalo, Abraham, et al. *Programming the Cell Broadband Engine – Examples and Best Practices.* New York: IBM Redbooks, 2008.

Baecker, R. "Sorting out Sorting." *ACM SIGGRAPH '81.* Dallas, TX: ACM, 1981. Sound film, 25 minutes, 16mm color.

Blachford, Nicholas. *Programming The Cell Processor - Part 1: What You Need to Know.* 2006. http://www.blachford.info/computer/articles/CellProgramming1.html (accessed October 6, 2008).

Brown, Marc H., and Robert Sedgewick. "A system for algorithm animation." *ACM SIGGRAPH Computer Graphics, Volume 18, Issue 3,* 1984: 177-186.

Clarisse, O., and S. K. Chang. "VICON: A Visual Icon Manager." *Visual Languages.* New York: Plenum Press, 1986. 151-190.

Cox, Philip T., and Trevor J. Smedley. "Using visual programming to extend the power of spreadsheet." *Proceedings of the workshop on Advanced visual interfaces.* Bari, Italy: ACM, 1994. 153-161.

Curry, Gael A. *Programming by Abstract Demonstration.* Dissertation, Washington: University of Washington, 1978.

Daintith, John. *Oxford dictionary of computing.* New York: Oxford University Press, 1983.

dalke scientific. *Visual dataflow programming.* 22 September 2003. http://www.dalkescientific.com/writings/diary/archive/2003/09/22/VisualProgramming.html (accessed August 20, 2008).

Feinberg, Dave. "A visual object-oriented programming environment." *ACM SIGCSE Bulletin, Volume 39, Issue 1,* 2007: 140-144.

Free On-line Dictionary of Computing. 26 May 2007. http://foldoc.org/ (accessed October 8, 2008).

Frigo, Matteo, and Steven G. Johnson. *Fastest Fourier Transform in the West.* November 2008. http://www.fftw.org/ (accessed February 16, 2009).

Goldberg, Adele, and David Robson. *Smalltalk-80: The language and its implementation.* New York: Addison-Wesley, 1983.

Goldstine, Herman H. *The Computer from Pascal to von Neumann.* Princeton, NJ: Princeton University Press, 1972.

Graduate Education and Research Services - Penn State. "Open Visualization Data Explorer." *Special Projects Group.* 2008. http://gears.aset.psu.edu/sp/software/opendx/details.shtml (accessed February 16, 2009).

Grafton, Robert B., and Tadao Ichikawa. "Visual Programming - Guest Editor's Introduction." *Computer, Volume 18, Issue 8, ISSN: 0018-9162,* 1985: 6-9.

Halbert, Daniel C. *Programming by example.* Dissertation, Berkeley, CA, USA: University of California, 1984.

Harmonia, Inc. *Harmonia.* 2008. http://www.harmonia.com/ (accessed February 16, 2009).

IBM. *Cell BE Programming Tutorial.* 19 October 2007. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788 (accessed February 16, 2009).

—. *Cell SDK.* 2008. http://www.ibm.com/developerworks/power/cell/ (accessed February 16, 2009).

—. *OpenDX.* 1991. http://www.opendx.org/ (accessed February 16, 2009).

Koelma, Dennis, Richard V. Balen, and Arnold Smeulders. "SCIL-VP: a multi-purpose visual programming environment." *In Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: technological challenges of the 1990's.* ACM, 1992. 1188-1198.

Lewis, Clayton, and Gary Olson. "Can principles of cognition lower the barriers to programming?" In *Empirical studies of programmers: second workshop, ISBN:0-89391-461-4,* by Ablex Series Of Monographs, 248-263. Norwood, NJ, USA: Ablex Publishing Corp., 1987.

Madnick, Stuart. "Understanding the Computer (Little Man Computer)." *Unpublished manuscript.* 1993.

Mattson, Timothy G., Berna L. Massingill, and Beverly A. Sanders. *Patterns for Parallel Programming.* Addison Wesley, ISBN 0321228111, 2004.

Mayora-Ibarra, Oscar, Oscar de la Paz-Arroyo, Edgar Cambranes-Martínez, and Alejandro Fuentes-Penna. "A visual programming environment for device independent generation of user interfaces." *Proceedings of the Latin American conference on Human-computer interaction.* Rio de Janeiro, Brazil: ACM, 2003. 61-68.

Mega Nerd. *libsndfile.* 2008. http://www.mega-nerd.com/libsndfile/ (accessed February 16, 2009).

Meyer, Robert M., and Tim Masterson. "Towards a better visual programming language: critiquing Prograph's control structures." *Journal of Computing Sciences in Colleges, Volume 15, Issue 5,* 2000: 181-193.

Myers, Brad A. "INCENSE: A system for displaying data structures." *ACM SIGGRAPH Computer Graphics, Volume 17, Issue 3,* 1983: 115-125.

Myers, Brad A. "Taxonomies of visual programming and program visualization." *Journal of Visual Languages and Computing, Volume 1, Issue 1,* 1990: 97-123.

Myers, Brad A., Ravinder Chandhok, and Atul Sareen. "Automatic data visualization for novice Pascal programmers." *Visual Languages, 1988., IEEE Workshop on.* Pittsburgh, PA: IEEE, 1988. 192-198.

National Instruments. *Using the LabVIEW Run-Time Engine.* 2006. http://zone.ni.com/reference/en-XX/help/371361B-01/lvhowto/using_the_lv_run_time_eng/ (accessed October 20, 2008).

Nickerson, Jeffrey. *Visual Programming Ph.D. Dissertation.* 1994. http://www.nickerson.to/visprog/visprog.htm (accessed October 8, 2008).

Roy, Geoffrey G., Joel Kelso, and Craig Standing. "Towards a Visual Programming Environment for Software Development." *Software Engineering: Education & Practice, 1998. Proceedings. 1998 International Conference.* Dunedin: IEEE, 1998. 381-388.

Shneiderman, Ben A. "Direct manipulation: A step beyond programming languages." In *Human-computer interaction: a multidisciplinary approach, ISBN:0-934613-24-9,* by William A. S. Buxton, 461-467. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987.

Smedley, Trevor J., Philip T. Cox, and Shannon L. Byrne. "Expanding the utility of spreadsheets through the integration of visual programming and user interface objects." *Proceedings of the workshop on Advanced visual interfaces.* Gubbio, Italy: ACM, 1996. 148-155.

Smith, David C. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought.* Basel, Stuttgart: Birkhauser, 1977.

Smith, David C. *Pygmalion: a creative programming environment.* Dissertation, Stanford, CA, USA: Stanford University, 1975.

Tritera. *The State of Prograph/CPX.* 19 April 2005. http://www.tritera.com/prograph.html (accessed October 23, 2008).

Ward, Matthew. "Data Visualization." *Worcester Polytechnic Institute, Computer Science Department.* http://www.dalkescientific.com/writings/diary/archive/2003/09/22/VisualProgramming.html (accessed August 18, 2008).

Whitley, Kirsten N., and Alan F. Blackwell. "Visual Programming in the wild: A survey of labview programmers." *Journal of Visual Languages and Computing, Volume 12, Issue 4,* 2001: 435-472.

Whitley, Kisrten M., and Alan F. Blackwell. "Visual programming: the outlook from academia and industry." *Papers presented at the seventh workshop on Empirical studies of programmers, ISBN:0-89791-992-0.* Alexandria, Virginia, United States: ACM, 1997. 180-208.

Wikipedia. *Deutsch Limit.* 10 January 2009. http://en.wikipedia.org/wiki/Deutsch_Limit (accessed February 22, 2009).

—. *Gaussian blur.* 2008. http://en.wikipedia.org/wiki/Gaussian_blur (accessed January 26, 2009).

—. *Hypervisor.* 2008. http://en.wikipedia.org/wiki/Hypervisor (accessed February 16, 2009).

—. *LabVIEW.* 2008. http://en.wikipedia.org/wiki/LabVIEW (accessed October 20, 2008).

—. *Visual Basic .NET.* 2008. http://en.wikipedia.org/wiki/Visual_Basic_.NET (accessed October 8, 2008).

—. *Visualization.* 2008. http://en.wikipedia.org/wiki/Visualization (accessed August 20, 2008).

Young, Mark, Danielle Argiro, and Steven Kubica. "Cantata: Visual programming environment for the Khoros system." *Computer Graphics,* 1995: 22-24.

Glossary

CBEA

Cell Broadband Engine Architecture.

Cell/B.E.

Cell Broadband Engine. The Cell Broadband Engine is one implementation of the Cell Broadband Engine Architecture (CBEA).

Data Explorer

Data Explorer allows the users to manipulate their data and create visualizations by using a visual programming environment.

DMA

Direct Memory Access. A technique for using a special-purpose controller to generate the source and destination addresses for a memory or I/O transfer.

DX (or OpenDX)

See *Data Explorer*.

Effective-address space (EA)

An address generated or used by a program to reference memory. A memory-management unit translates an effective address (EA) to a virtual address (VA), which it then translates to a real address (RA) that accesses real (physical) memory. The maximum size of the effective address space is 2^{64} bytes.

Element Interconnect Bus EIB

Element Interconnect Bus. The on-chip coherent bus that handles communication between the PPE, SPEs, memory, and I/O devices (or a second Cell Broadband Engine). The EIB is organized as four unidirectional data rings (two clockwise and two counter clockwise).

Example-Based Programming

Example-Based Programming refers to systems that allow the programmer to use examples of input and output data during the programming process.

Field

A self-contained collection of information necessary to represent scientific data. A Data Explorer Field typically is made up of a series of components and other information as required. It includes the data itself in the form of a "data" component, a set of sample points in the form of a "positions" component, optionally, a set of interpolation elements in the form of a "connections" component, and other information as needed.

Graphical Programming

See *Visual Programming*.

I/O

Input/Output.

Intrinsic

A C-language command, in the form of a function call, that is a convenient substitute for one or more inline assembly-language instructions. Intrinsics make the underlying ISA (Instruction Set Architecture) accessible from the C and C++ programming languages.

Local Store

The 256-KB local store (LS) associated with each SPE. It holds both instructions and data.

LS

See *Local Store*.

Main Memory

See *Main Storage*.

Main Storage

The effective-address (EA) space. It consists physically of real memory (whatever is external to the memory-interface controller, including both volatile and nonvolatile memory), SPU LSs, memory-mapped registers and arrays, memory-mapped I/O devices (all I/O is memory-mapped), and pages of virtual memory that reside on disk. It does not include caches or execution-unit register files. See also local store.

mdf

See *module description file*.

MFC

Memory Flow Controller. It is part of an SPE and provides two main functions: moves data via DMA between the SPE's local store (LS) and main storage, and synchronizes the SPU with the rest of the processing units in the system.

Module Builder

A graphical user interface to assist in the creation of user-defined modules.

module description file

A module description file is used by a programmer who is adding a module to Data Explorer to describe information about the module that is needed by the system.

A module description file contains the name of the module, a short description of it, a category for the user interface to put the module in, and the names and descriptions of the input and output parameters. The module description file is used by the executive and the user interface to name parameters. The module description file is also used by the graphical user interface to form a tool icon in the proper category with the right number of input and output tabs.

PPE

Power Process Element. The general-purpose processor in the Cell Broadband Engine.

PPU

PowerPC Processor Unit. The part of the PPE that includes the execution units, memory-management unit, and L1 cache.

Program Visualization

Programming environment which uses graphics to illustrate some aspect of the program or its run-time execution.

RISC

Reduced Instruction Set Computing. Represents a CPU design strategy emphasizing the insight that simplified instructions that "do less" may still provide for higher performance if this simplicity is used to execute instructions faster.

SIMD

Single Instruction Multiple Data. Processing in which a single instruction operates on multiple data elements that make up a vector data-type. Also known as vector processing. This style of programming implements data-level parallelism.

SIMDize

To transform scalar code to vector code.

hypervisor

A control (or virtualization) layer between hardware and the operating system. It allocates resources, reserves resources, and protects resources among (for example) sets of SPEs that may be running under different operating systems. The Cell Broadband Engine has three operating modes: user, supervisor and hypervisor. The hypervisor performs a meta-supervisor role that allows multiple independent supervisors' software to run on the same hardware platform. For example, the hypervisor allows both a real-time operating system and a traditional operating system to run on a single PPE. The PPE can then operate a subset of the SPEs in the Cell Broadband Engine with the real-time operating system, while the other SPEs run under the traditional operating system.

SPE

Synergistic Processor Element. It includes an SPU, an MFC, and an LS.

SPE thread

(a) A thread running on an SPE. Each such thread has its own 128 x 128-bit register file, program counter, and MFC Command Queues, and it can communicate with other execution units (or with effective-address memory through the MFC channel interface). (b) A thread scheduled and run on an SPE. A program has one or more SPE threads. Each thread has its own SPU local store (LS), register file, program counter, and MFC command queues.

SPU

Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS).

thread

A sequence of instructions executed within the global context (shared memory space and other global resources) of a process that has created (spawned) the thread. Multiple threads (including multiple instances of the same sequence of instructions) can run simultaneously if each thread has its own architectural state (registers, program counter, flags, and other program-visible state). Each SPE can support only a single thread at any one time. Multiple SPEs can simultaneously support multiple threads. The PPE supports two threads at any one time, without the need for software to create the threads. It does this by duplicating the architectural state. A thread is typically created by the pthreads library.

UIML

User Interface Markup Language. Is an XML language for defining user interfaces on computers.

vector

An instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU SIMD instructions operate on vector operands. Vectors are also called SIMD operands or packed operands.

Vector/SIMD

The SIMD instruction set of the PowerPC Architecture, supported on the PPE.

Visual Program

A user-specified interconnected set of Data Explorer (DX) modules that performs a sequence of operations on data and typically produces an image as output.

Visual Program Editor

Data Explorer (DX) window used to create and edit visual programs and macros.

Visual Programming (VP)

The Visual Programming concept is split in two: Visual Programming Language (VPL) and Visual Programming Environment (VPE).

Visual Programming Environment

Visual Programming Environment (VPE) is software which allows the use of visual expressions (such as graphics, drawings, animation or icons) in the process of programming.

Visual Programming Language

Visual Programming Language (VPL) is any programming language that allows the user to specify a program in a two-(or more)-dimensional way.

VPE

See *Visual Programming Environment*.

VPL

See *Visual Programming Language*.