“Διάφανη υποστήριξη διασυνδεμένων δομών δεδομένων σε ετερογενή παράλληλα συστήματα.”

“Transparent support of linked data structures on heterogeneous parallel systems.”

A thesis
submitted in fulfillment
of the requirements for the degree
of
Computer and Communication Engineering
at
The University of Thessaly

by

Vasiliou Zoi

Supervisors:
Antonopoulos Christos

Bellas Nikolaos

Volos, 2013

Aknowledgements

It gives me great pleasure in expressing my gratitude to all those people who have supported me and had their contributions in making this thesis possible. First and foremost, I express my profound sense of reverence to my supervisor Christos D. Antonopoulos for his constant guidance, support, motivation and untiring help during the course of my thesis and especially for his confidence in me. Also, I would like to sincerely thank my co-advisor, Nicolaos Bellas, for his help and useful advices throughout this study. I aknowledge my parents, Fotios and Cleopatra, and my sister Mariza for their love and support throughout my life. Last but not least, I would like to thank my friends for their encouragement and all the cheerful moments that I recharged myself. Especially, I would like to thank Vasso for being such a good listener and her encouragement, and also Ifigeneia and Kathryne for the nice moments.

Abstract

Acceleration of applications with low power consumption can only be attained through programming heterogeneous parallel platforms. These architectures include a single host coordinating a collection of accelerators in the form of GPUs, DSPs, and FPGAs. However, due to the fact that the components of the heterogeneous platforms usually have separate memory spaces and have different architectures, the support of complex data structures is difficult on these systems. The purpose of this undergraduate thesis is the transparent support of linked data structures on heterogeneous parallel systems. Our requirements are low-overhead and no extensive changes of code. Linked data structures require many transfers between the components of the system. Since transfers cost due to latency, we have developed a method minimizing the transfers among the components of the system. Also, since components have different architectures and compilers, the data may not be represented in the same way, and hence we face problem at the mapping of data from one component to another. So, we have developed a method which ensures that the data will have the same form among the components of the system. As for the method minimizing the transfers, we have developed our own memory manager which stores the nodes of the linked data structures in such a way so as one transfer is needed. However, the linked data structures have as elements pointers which are invalid from one component to another. The memory manager enables the procedure of making pointers valid thanks to the way of storing the data. As for the same representation of data among the components, we have developed functions that enforce the same representation for data of structure type. Also we propose a method for supporting linked data structures at a heterogeneous system with components with different virtual address space bit-width. As case study we have used the application Gerris whose primary data structure is a tree which we attempted to transfer to a graphical processing unit. At the main part of thesis, we describe the goals, the problems we faced, and the final solutions.

Περίληψη

Η επιτάχυνση των εφαρμογών σε συνδυασμό με χαμηλή ισχύ μπορεί μόνο να επιτευχθεί μέσω ετερογενών παράλληλων πλατφόρμων. Αυτές οι αρχιτεκτονικές περιλαμβάνουν έναν μόνο κεντρικό επεξεργαστή (host) που συντονίζει μια συλλογή από επιταχυντές όπως GPUs, DSPs και FPGAs. Εξαιτίας του γεγονότος ότι τα συστατικά μέρη των ετερογενών πλατφόρμων συνήθως ανήκουν σε ξεχωριστούς χώρους μνήμης και βασίζονται σε διαφορετικές αρχιτεκτονικές, η υποστήριξη πολύπλοκων δομών δεδομένων σε ετερογενή παράλληλα συστήματα είναι δύσκολη. Ο στόχος της διπλωματικής είναι η διαφανή υποστήριξη διασυνδεδεμένων δομών δεδομένων σε ετερογενή παράλληλα συστήματα. Οι απαιτήσεις μας είναι χαμηλή επιβάρυνση και μη εκτεταμένες αλλαγές στον κώδικα. Οι διασυνδεδεμένες δομές δεδομένων απαιτούν πολλές μεταφορές μεταξύ των συστατικών μερών του συστήματος. Επειδή οι μεταφορές κοστίζουν λόγω latency, έχουμε αναπτύξει μια μέθοδο που μειώνει τις μεταφορές μεταξύ των συστατικών μερών του συστήματος. Επίσης, επειδή τα συστατικά μέρη έχουν διαφορετικές αρχιτεκτονικές και διαφορετικούς μεταγλωττιστές, τα δεδομένα ίσως δεν αναπαρίστανται με τον ίδιο τρόπο μεταξύ των συστατικών μερών, γι'αυτό αντιμετωπίζουμε το πρόβλημα της αντιστοιχίας των δεδομένων από το ένα συστατικό μέρος στο άλλο. Γι'αυτό το λόγο , αναπτύξαμε μια μέθοδο που επιβεβαιώνει ότι τα δεδομένα θα έχουν την ίδια αναπαράσταση στα συστατικά μέρη του συστήματος. Όσον αφορά τη μέθοδο που μειώνει τις μεταφορές των δεδομένων, έχουμε αναπτύξει τον δικό μας διαχειριστή μνήμης με τον οποίο τα δεδομένα ο οποίος αποθηκεύει τα δεδομένα με τέτοιο τρόπο έτσι ώστε να χρειάζεται μια μεταφορά. Οι διασυνδεδεμένες δομές, όμως, έχουν στοιχεία που χρησιμοποιούν δείκτες, οι οποίοι δείχνουν σε μη έγκυρες θέσεις μνήμης όταν μεταφέρονται από το έναν συστατικό μέρος στο άλλο. Ο διαχειριστής μνήμης διευκολύνει τη διαδικασία μετατροπής των δεικτών ώστε να δείχνουν σε ώστε να δείχνουν σε έγκυρες θέσεις μνήμης λόγω του τρόπου που αποθηκεύει τα δεδομένα, Όσον αφορά την ίδια αναπαράσταση των δεδομένων μεταξύ των συστατικών μερών, έχουμε αναπτύξει συναρτήσεις που εγγυώνται την ίδια αναπαράσταση για δεδομένα που ανήκουν στον τύπο δομή. Επίσης προτείνουμε μια μέθοδο για την υποστήριξη διασυνδεδεμένων δομών σε ετερογενή συστήματα που αποτελείται από συστατικά μέρη με διαφορετικό πλάτος bit του εικονικού χώρου διεύθυνσης. Ως μελέτη περίπτωσης έχουμε χρησιμοποιήσει την εφαρμογή προσομοίωσης κίνησης υγρών Gerris που χρησιμοποιεί ως δομή δεδομένων ένα δέντρο

το οποίο μεταφέραμε σε μια κάρτα γραφικών. Στο κυρίως μέρος της διπλωματικής περιγράφουμε τους στόχους, τα προβλήματα που αντιμετωπίσαμε και τις τελικές λύσεις.

Contents

Chapter 1 - Introduction

Raw performance was the ultimate objective over the past decade. However, due to the increase of transistor density, each generation of processors grew smaller, faster, dissipated more heat and consumed more power. Therefore, there is a tendency to boost performance with relatively low cost and power dissipation. There is a good reason to believe that in a world where maximizing performance per watt is essential, we can expect systems to increasingly depend on many cores with specialized silicon whenever practical, since it has been proven that the more specialized the core the more power-efficient it is[]. For instance, general graphic processors units rely on specialized silicon and have become increasingly attractive for general purpose operations for addressing data parallel programming tasks.

Since the applications usually present a mix of characteristics, there is a transition to heterogeneous systems consisting of general processors and specialized accelerators. For example, the data-parallel workloads are executed on GPU, while the sequential code is executed on CPU. Therefore, we need a hardware which is a hybrid combination of components. Heterogeneous parallel architectures include one central processor and a range of hardware accelerators such as GPUs, DSPs and FPGAs e.t.c.

Although heterogeneous systems offer high performance with relatively low power dissipation, it is difficult to support complex linked data structures on heterogeneous systems. The fact that the components of the system belong to separate memory spaces and have different architectures hinders the support of complex linked data structures on such systems. However, most realistic applications use linked data structures. The purpose of this thesis is the transparent support of linked data structures on heterogeneous parallel systems with low-overhead and no extensive changes of code.

Linked data structures require many transfers between components of the system. Since transfers cost due to latency, we have developed a method minimizing the transfers among the components of the system. In particular, we have developed our own memory manager with which the data scattered in the heap are now stored in segregated heaps in a continuous memory area so as one transfer is needed. However, the linked data structures have as elements pointers but since the components of a heterogeneous system have different architectures, when the pointers are transferred to another component, they will be invalid. Since we use continuous memory areas for storing the nodes of linked data structures, we can rewrite the pointers to point to valid address space very easily. As for the same representation of data among the components, we have developed functions that enforce the same representation for structures. As heterogeneous system, we use one of two components with different virtual address space bit-width, hence the pointers of the components have different sizes causing false mapping of the data from one component to another and other problems. We propose a solution which faces these problems.

As case study, we have used the application Gerris[1][2][3] simulating fluid flow and its primary data structure is a tree which we transfer to a graphical processing unit.

In chapter 2 and 3, we describe the necessary background. More specifically, in chapter 2 we briefly present the application Gerris and the data structures transferred to the GPU. In chapter 3, we discuss the transition from single core era to multicore era and from multicore era to heterogeneous era [9] and a multivendor open standard for general-purpose parallel programming of heterogeneous systems (OpenCL [11] [12]). In chapter 4, we discuss the memory manager we have developed which minimizes the transfers and enables the pointer rewriting so as pointers to be valid, how we keep the same representation of data on different architectures, the limitations of our implementation and how we verify its correctness. We discuss our final thoughts in chapter 5.

Chapter 2 - The Application Gerris

2.1 An introduction to Gerris

Gerris [1][2][3] Flow Solver is a software project for computational fluid dynamics. Named after the *Gerris remigis* or common water strider, Gerris is a program for the solution of the partial differential equations (incompressible Euler) describing fluid flow. The main objective of this project is the primitive variables velocity and pressure to be calculated. Gerris uses a numerical method for solving the incompressible Euler equations, combining a quad/octree discretization, a projection method, and a multilevel Poisson solver. The domain is spatially discretized using square (cubic in 3D) finite volumes organized hierarchically as a quadtree (octree in 3D). The primitive variables are all defined at the centre of squares. For concentrating the computational effort on the area where it is most needed, Gerris uses dynamic adaptive mesh refinement where the quadtree structure is discretized finer at the areas where there is more intensive fluid flow.

Gerris can deal with simple boundaries and solid boundaries. There are boundaries around each tree of domain. The fluid in the boundary layer is subjected to shearing forces. A range of velocities exists across the boundary layer from maximum to zero, provided the fluid is in contact with the surface. So, some boundary conditions are applied to the partial differential equations. Also, Gerris can deal with arbitrarily complex solid boundaries embedded in the quad/octree mesh.

2.2 The data structures of Gerris

2.2.1 The quad tree

The type of spatial discretisation used for the integration of the Euler equations is a quad tree for two dimensions (octree for three dimensions). A quadtree is a tree data structure in which each internal node has exactly four children, corresponding to four square subcells in two dimensions. The fundamental idea behind the quadtree is that any domain can be split into four quadtrants. Each quadrant may again be split into four quadrants e.t.c. The corresponding three-dimensional structure is an octree. The level of a cell is defined by starting from zero for the root cell and by adding one every time a descendant child is added. Each cell C may have a direct neighbor in each direction d (four in 2D, six in 3D). Each of these neighbors is accessed through a face of the cell. Also, cells cut by a solid boundary are defined as mixed cells. The cells and tree structure of the quad tree are illustrated in Figure 1 .

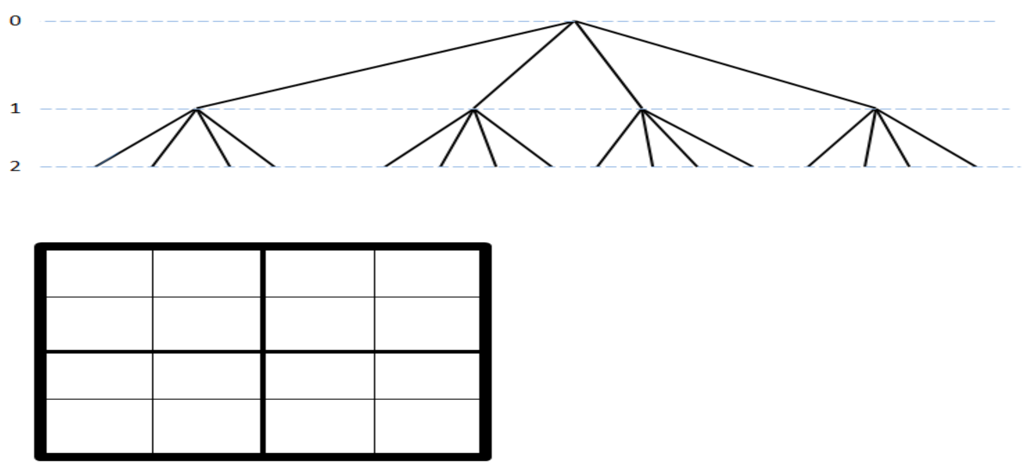


Figure 1

The quad tree

As for the total number of the cells contained in a quad tree, each k-th level contains 4*k squares, so the tree has a total of cells:

$$N = \sum_{k=0}^{n} 4^k$$

2.2.2 The full threaded tree

The full threaded tree [4] is used by Gerris as the implementation of the quad tree (and octree for 3D). All cells are organized in groups called octs. Each cell has a physical state vector U (vector U is the data at which each cell points at figure) associated with it, and a pointer to an oct which contains its children, if any, or a nil pointer. Each oct contains four cells in 2D (eight in 3D) and information. Each oct knows its level, which is equal to the level of the oct's cells and has a pointer to a parent cell and four pointers (six pointers in 3D) to parent cells of neighboring octs. This information is enough to find neighbors, children and parents of every cell without searching. The full threaded tree represents the intermediate cells of the tree and is illustrated in **Figure 2** .

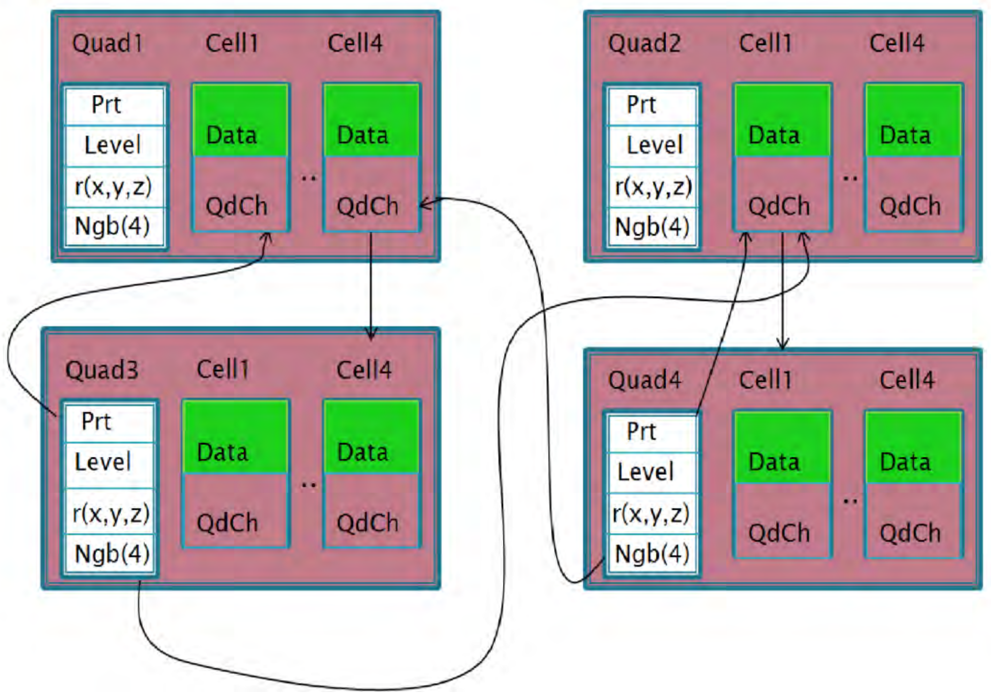


Figure 2

The full threaded tree

2.2.3 The whole domain

The whole domain may consist of more than one tree. The trees are connected with each other as neighbors. Apparently, the number of roots of domain is equal to the number of trees. An example of a domain with three trees is illustrated in **Figure 3**.

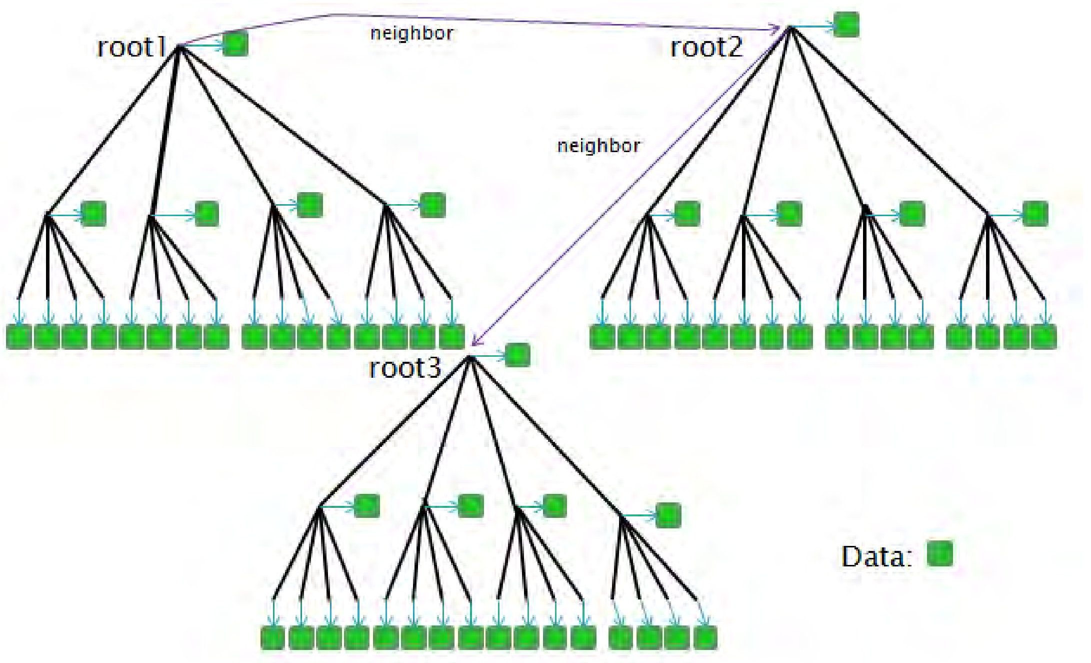


Figure 3

The domain is consisted of three trees connected with each other. Each cell of the tree points to a vector depicted as data. This vector represents all the variables used by each cell.

Each cell has neighbors in all directions (four in 2D). A neighbor can be a root, a cell, or a boundary. Cells on the boundary of the domain may not have neighbors in all directions.

Chapter 3 -Heterogeneous parallel systems

3.1 From single-core era to heterogeneous era

3.1.1 Single Core Era

Raw performance was the ultimate objective over the past decade. Squeezing more and more transistors onto a single die, increasing the clock rate and inventing architectural novelties were the trends for boosting the application performance.

In accordance with Moore's Law[8] (which predicts a doubling of transistors every 18 months) we have increased transistor density and we have enjoyed continuous performance improvement for four decades. However, as we allowed more and more transistors to be squeezed onto the same chip, the power density increased. Power converts to heat, and too much heat can destroy a chip in seconds. So it has been predicted that soon some of the transistors should be left unpowered or dark while the others will be working (known as dark silicon[5]). In addition, throughout the 1990's microprocessor frequency was synonymous with performance. Higher frequency meant a faster, more capable computer. Nevertheless, the ever-increasing transistor density and clock rate reached a plateau since they cause power consumption and temperature dissipation increase. In addition, it is difficult to extract more instruction-level-parallelism (ILP) from application codes. So each generation grew smaller, faster, and dissipated more heat, hence there was need for other innovative methods for high performance with relatively low power dissipation (see [8]).

3.1.2 Multicore era

It is irrefutable that "two heads are better than one" (see [8]). Multicore processors [8] are often clocked at lower frequencies, but have much better performance than a single-core processor. This is reasonable since if the work can be executed parallel, it maximizes resource utilization and thanks to lower frequencies we have lower power consumption. However, multicore processing posed some challenges. Regardless of chip organization and topology, multicore scaling will be fundamentally limited by power constraints. The percentage of chip that can be powered is decreasing so as the chip will not cross the limit of heat dissipation. The area of chip which cannot be powered is known as dark silicon. It is predicted that the amount of dark silicon will be more than 50% of the chip at 8nm. Therefore, there is a question in the point in scaling down and increasing the number of transistors per chip if we can't use them. Also, programmers are used to programming for a single thread and now they have to get used to finding the concurrency in their problem, and expressing that concurrency in their software. However, this shift to parallel programming is a daunting challenge for programmers since the streams of operations that will execute concurrently must be defined, the data they operate on associated with them, and the dependencies between them should be managed so as the same answer is produced with the single-threaded program.

Meanwhile, the multicore era see some interesting developments in GPUs. GPUs improve 3D graphics performance by offloading graphics from the CPU. GPUs support SIMD instructions that enable them to perform parallel operations on very large sets of data and they perform them at much lower power consumption relative to the serial processing of similar data sets on CPUs. So GPUs became increasingly attractive for general purpose operations for addressing data parallel programming tasks.

3.1.3 Heterogeneous Era

Due to performance and power scalability constraints, we should leave the single-core designs and move to heterogeneous computing. Heterogeneous computing refers to systems that use more than one kind of processor. We cannot continue with more aggressive general purpose processor designs as they cannot accelerate every kind of application without increasing complexity and power dissipation. There is a shift to multicore systems so as to gain performance not just by adding cores, but also by incorporating specialized processing capabilities to handle particular tasks with low power consumption. Processors specialized to a specific function have few wasted transistors because they include only these functional units required by their special function, while a general purpose processor must include a wide range of functional units to respond to any computational demand. Also, the specialized processors have simple control logic and smaller caches compared with the CPU's. Hence, the more specialized the core, the more power-efficient it is ([11] chapter 1, pages 4 -12). For instance, GPGPUs are specific-purpose processors since thanks to their vector processing capabilities; GPGPUs are used for data-parallel workloads. The applications usually present a mix of characteristics. There are parts being control-intensive, data-intensive, I/O intensive and compute-intensive. For instance mathematically intensive computations on very large data sets, which can be parallelized, should be executed on GPGPU, while the control-intensive part should be executed on CPU. It is apparent that we need a hardware being a hybrid of various components. Therefore, we need a mix of processors specialized in different tasks. Heterogeneous architectures include one central processor and a range of hardware accelerators such as GPUs, DSPs and FPGAs e.t.c. An example of a heterogeneous system is illustrated in **Figure 5**. The heterogeneous future is inevitable, a single application will exploit a number of processors that are specialized for different tasks and are different in location (on-die, from local to very remote e.t.c.).

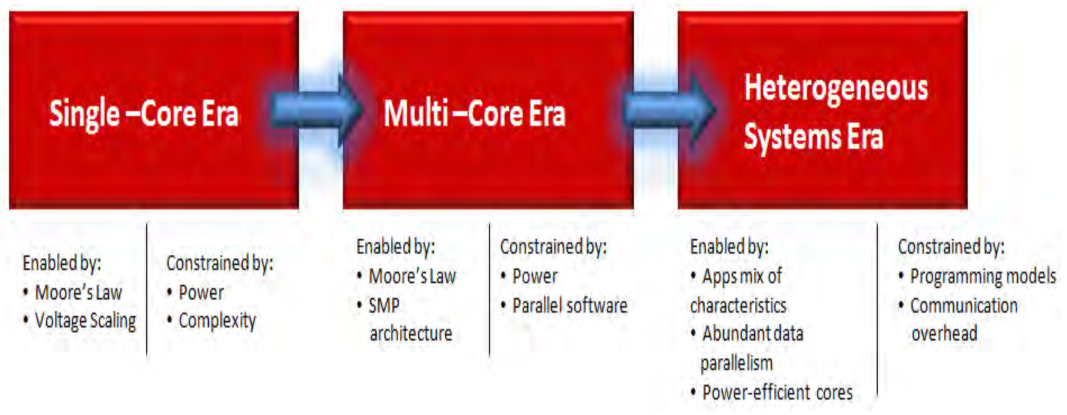


Figure 4

The figure [9] summarizes the previous analysis for the transition from single-core to multi-core and from multi-core to heterogeneous systems.

As discussed above, acceleration of application with low power consumption can only be attained through programming heterogeneous parallel platforms. We will refer some of heterogeneous systems' characteristics:

1) Different computational elements

The computational elements in the system may have different instruction sets and different architectures and may run at different speeds. An effective program must take into account these differences and appropriately map the parallel software onto the most suitable devices.

2) Each component may have different programming model

For instance, should we want to program CPU in parallel, we can use OpenMP or Pthreads, while programming Nvidia GPUs means using CUDA Programming model and GPUs from multiple vendors requires OpenCL programming model.

3) Deep complex memory hierarchy

Some components of heterogeneous systems may expose deep complex memory hierarchies. Usually, the architecture of components is NUMA (Non-Uniform Memory Access), where the memory access time depends on the memory location relative to a processor. The hardest part is that the memory hierarchy of some accelerators is not hardware-controlled. Therefore, in these cases the programmer should explicitly transfer the data to the region being nearest to the processor.

4) Separate memory spaces

The components of a heterogeneous system have separate memory spaces. That means that the components store separate copies of data in separate memory spaces. The latter implies that a pointer may have different "meanings" (and may be invalid) on different components of the system. So it is necessary the transfer of data between components. The CPU and GPU, for example, have separate memory spaces, as the GPU is connected with the CPU as peripheral via PCI-Express.

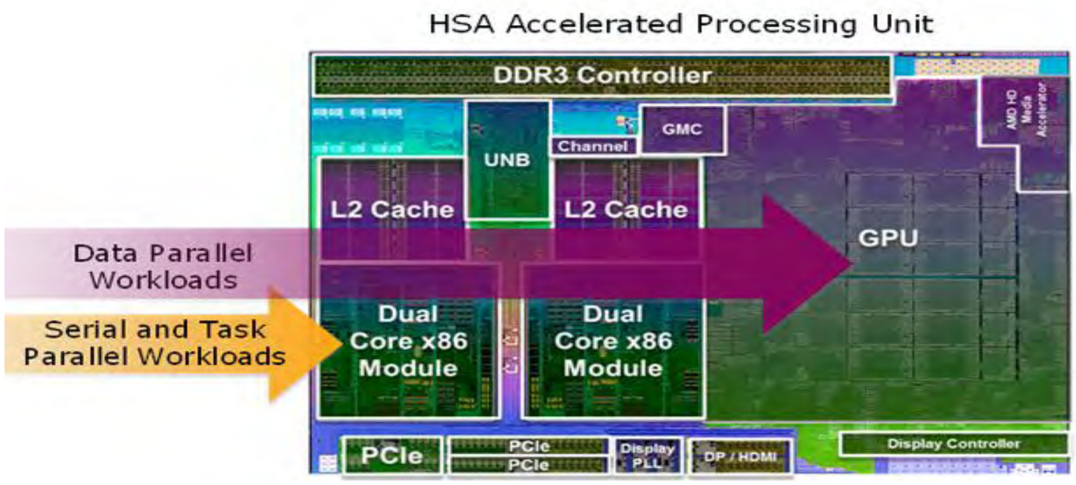


Figure 5

A heterogeneous systems architecture [12] accelerated processing unit with two dual core CPUs and one GPU. The serial and task parallel workloads are assigned to the CPUs, while the data parallel workloads are assigned to the GPU.

The heterogeneous systems of interest to high performance computing use an attached coprocessor or accelerator that is optimized for certain types of computation. These devices typically exhibit internal parallelism, and execute asynchronously and concurrently with the host processor Programming heterogeneous parallel platforms [], we have to deal not only with parallelism but with an attached asynchronous device as well, and with the complexity on parallel programming on that device. In particular, the program must manage the concurrent activities between the host and device, and manage data locality between the host and device. Hence, hardware heterogeneity is complicated and programmers have come to depend on high-level abstractions that hide the complexity of the hardware. OpenCL is a framework providing that high-level abstraction. Its key feature is functional portability across different heterogeneous which is achieved via its abstracted and execution model. More specifically, OpenCL implementation is correctly adjusted to suit to the target architecture, hence allows the programmer to make use of multiple execution devices present on a platform. Below we present a brief introduction to the OpenCL language.

3.2 OpenCL

OpenCL(OpenComputing Language) [11][12][16] is a multivendor open standard for general-purpose parallel programming of heterogeneous sytems that include CPUs, GPUs and other processors and is destined for data/task parallel computations. OpenCL promises functional portability. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance compute servers, desktop computer systems, and handheld devices. Hence, OpenCL has the potential to transform the software industry.

The OpenCL platform layer implements platform-specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices. An OpenCL platform always includes a single host and a collection of devices managed by OpenCL framework. The host is connected to one or more OpenCL devices. The device is where the parallel routine will be executed. The OpenCL application consists of the host program and one or more kernels. Kernels are the streams of instructions executing on the OpenCL devices. Therefore, the OpenCL device is often referred to as compute device. OpenCL supports multiple device classes namely CPUs, GPUs, DSPs, the IBM Cell and other processors. The OpenCL devices are further divided into compute units which are further divided into one or more processing elements. Computation on a device occurs within the processing elements. The host sends commands to devices for execution of code, for transfer of data or synchronization. The OpenCL platform model is illustrated in **Figure 6** .

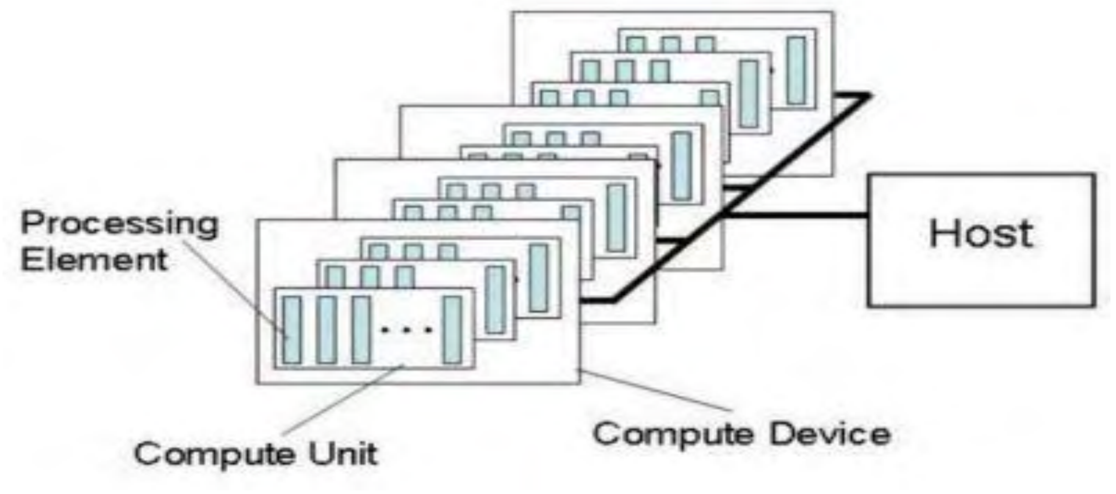


Figure 6

The OpenCL platform model with one host and one or more OpenCL devices. The host is connected to one or more OpenCL devices. Each OpenCL device has one or more compute units, each of which has one or more processing elements(see [11] chapter 1, page 12).

OpenCL provides functions to:

1) discover the components that makeup the heterogeneous system.
2) create contexts representing the devices to be used.
3) perform host-device transfers.
4) compile the kernel function which will be executed on chosen devices.
5) launch the kernel.
6) check for errors.
7) program the kernel functions.

Memory model

The OpenCL memory model defines a relaxed consistency model. In other words, the values seen in memory by an individual thread are not guaranteed to be consistent across the full set of threads at all times. A summary of the memory model in OpenCL and how the different memory regions interact with the platform model is illustrated in **Figure 7**. The host memory is a memory region being visible only by the host. The host and OpenCL device memory models are independent of each other.

Four types of memory are available on the OpenCL devices:

- **Global memory**
  The largest memory of the device with relatively high latency and visible by all threads.
- **Constant memory**
  Small, read-only memory with low-latency.
- **Local Memory**
  Accessible by multiple processing elements belonging to the same compute unit. It is much faster than global memory , since it is closer to the processing elements that global memory.
- **Private memory**
  Memory region accessible within each processing element. It is the fastest memory of the device since it is similar to registers in a CPU core.

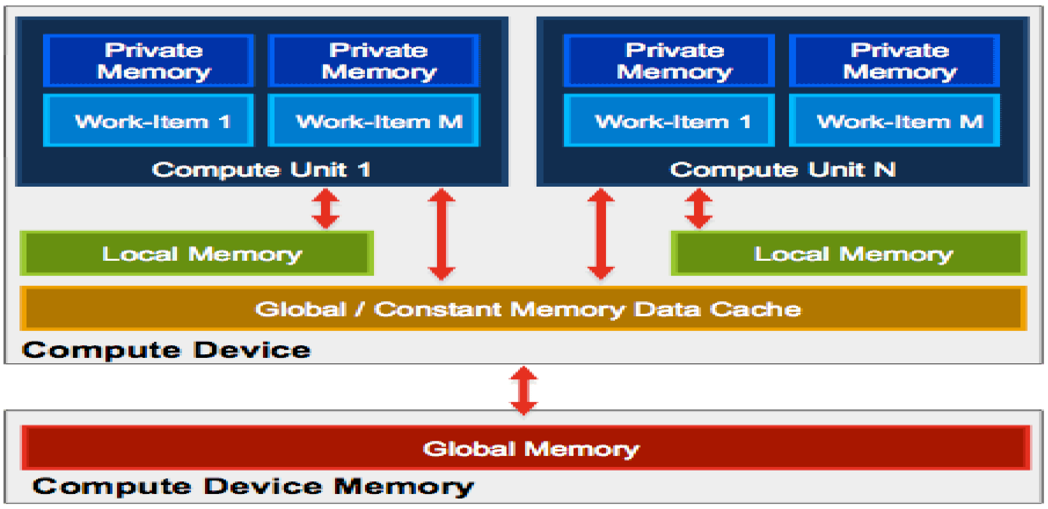


Figure 7

The OpenCL memory management is explicit. The data must be moved from host memory to global memory, from global memory to local memory and back.

Steps for creating an OpenCL application:

An OpenCL application must carry out the following steps (see **Figure 8**):

1) Discover the components that make up the heterogeneous system.
2) Probe the characteristics of these components and choose which are suitable for the application.
3) Create the kernels that will run on the platform.
4) Transfer data from host to devices.
5) Execute the kernels in the right order and on the right components of the system.
6) Transfer the final results from devices to the host.

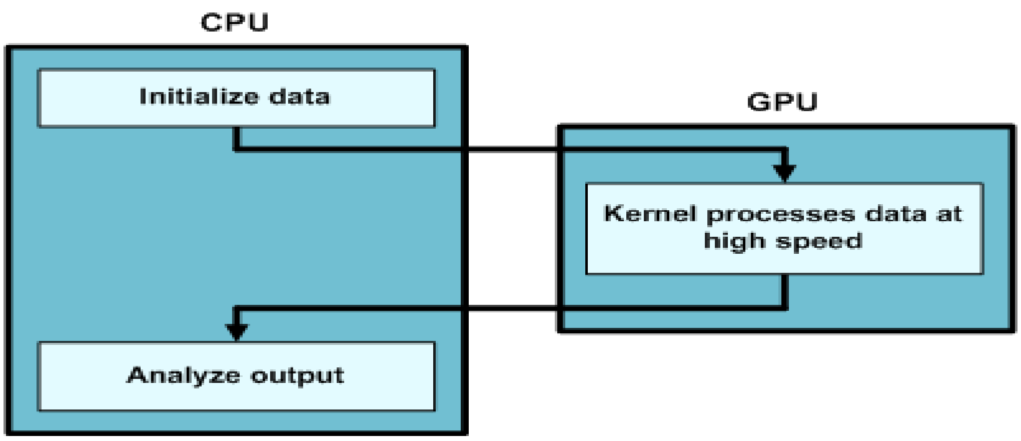


Figure 8

Steps for an OpenCL application

Chapter 4 - Transparent support of linked data structures on heterogeneous parallel systems

4.1 General

The current trend for using linked data structures on heterogeneous systems[7][9][10] is to be avoided. This happens since it is difficult the support of linked structures on heterogeneous systems. However, most realistic applications use linked data structures. In order to solve this problem, we have developed a transparent method supporting this kind of structures on heterogeneous systems. Our demands are low-overhead and no extensive changes of code.

Linked data structures cause many transfers of data between the host and the device since the nodes are scattered in the memory. Transfers between the components of the system cost due to latency (see [14]), thus we have developed a method minimizing these transfers and we discuss it at section

4.2 Memory transfers aggregation". At this section, we propose the use of a continuous memory area so as one transfer is needed. In addition, since the architecture of the device is different from the host's one, the pointers of the linked data structures on the device will be invalid, Hence, it is necessary to rewrite the pointers which is enabled by the fact that continuous memory areas are allocated. Also, the components of a heterogeneous system have different architectures, therefore the data may be represented in different way. This problem is more frequent when the form of the data is structure as padding is added differently due to the different compilers of the components. We have developed a method which guarantees the same representation of data between the components of the system and it is described at section"**4.3 Enforcing the same representation of data between components**". Finally we describe the "Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.**"** of our implementation and the "**4.5 Verification"** of its correctness.

4.2 Memory transfers aggregation

One of the trickiest things in programming heterogeneous systems is managing the transfer of data between devices. Transferring data takes time and the programmer must be careful so that the transfer time does not overpower any performance gains from parallelizing the algorithm. When it comes to transfer time, we usually think of it as having two components: the time due to latency and the time depending on the size of data being transferred. The last time is calculated as the size of data divided by the bandwidth, while the time due to latency is constant. So the latency is added cumulatively at each transfer, thus, we prefer as few transfers as possible so as not to have high overhead of latency.

Linked data structures have nodes which are scattered in the heap. We cannot sustain this form of data structure as we must transfer each node to the device implying many small data transfers. Thus the nodes must be stored in a large continuous memory area.

It is preferred one large transfer instead of many smaller since we pay the transfer latency one time and exploit the bandwidth better. However, one large transfer is not easy because the data are scattered in the memory. Therefore, we should do some extra work creating our own memory manager.

In order to avoid many small data transfers between host and device, the solution is the use of a continuous memory area. In particular, our technique is illustrated at **Figure 9** where the data are scattered in the first heap, and we save them in a continuous memory area in the second heap. We allocate initially a contiguous memory area and here in after we assign addresses from this area. When the first data segment is allocated, we put a pointer to point at the end of that segment. So, the second data segment is allocated next to the first, and generally every data segment is allocated next to the previous one. In this way, the nodes of the linked data structures are stored as segregated heaps in a continuous memory area so as one transfer is needed.

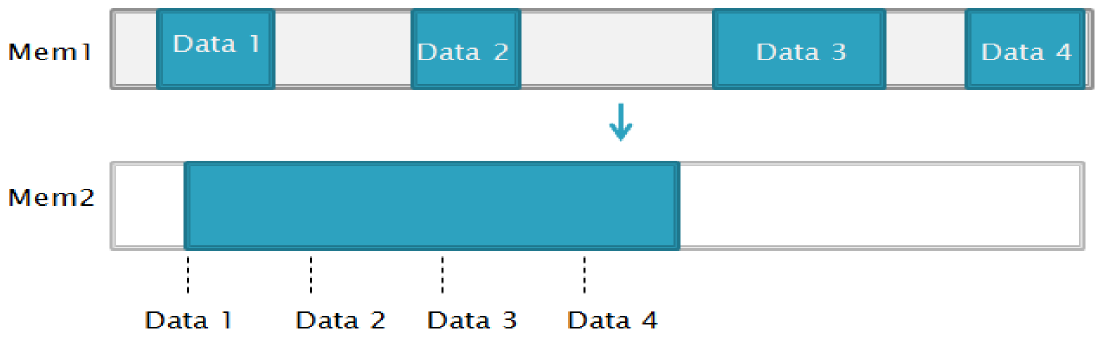


Figure 9

From data scattered in the heap to segregated heaps in a continuous memory area.

Memory manager:

We present the form of our memory manager.

Once the program starts to execute, a contiguous memory region is allocated:

contiguous_memory = my_malloc(sizeof contiguous memory region);

The memory manager intercepts the mallocs of the program calling a function with the following form.:

```
void *my_malloc(size) {

        …

        …

       returnPointer  = nextPointerFree

       nextPointerFree +=size;

       …

       return returnPointer;

}
```

Each time my_malloc is called, it returns an address of the contiguous memory region allocated at the beginning. At the end of my_malloc, the pointer *nextPointerFree*(see **Figure 10**) holds the address which will be returned at the next call. More precisely, nextPointerFree takes as value the address at which the previous data was allocated plus the size of the current data.

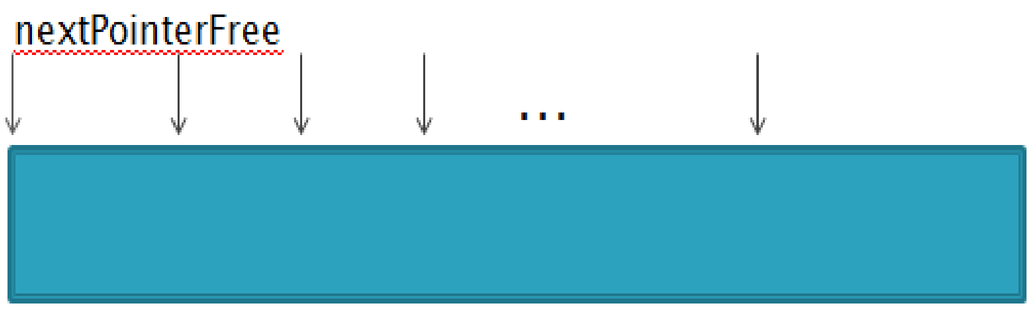


Figure 10

It is illustrated how the pointer nextPointerFree points to the next available address with each call.

The use of memory manager in Gerris

The memory manager was tested in application Gerris. The data structures of Gerris are examined at Chapter 2 - The Application Gerris. Its basic data structure used is a quad tree.

We have allocated continuous memory areas for:

- **Tree (all tree nodes except roots)**
  The full threaded tree (see **Figure 2**) represents this kind of nodes.
- **Roots**
  The roots of the trees (see **Figure 3**).
- **Boundaries**
  Cells cut by boundaries of the domain
- **Data**
  Each node points to a separated heap holding the data (see **Figure 3**). The data contains the variables (velocity, pressure and other variables) and elements for cells cut by solid boundaries.

The **Figure 11** depicts the data structures and some connections with pointers by the intervention of our own memory manager.

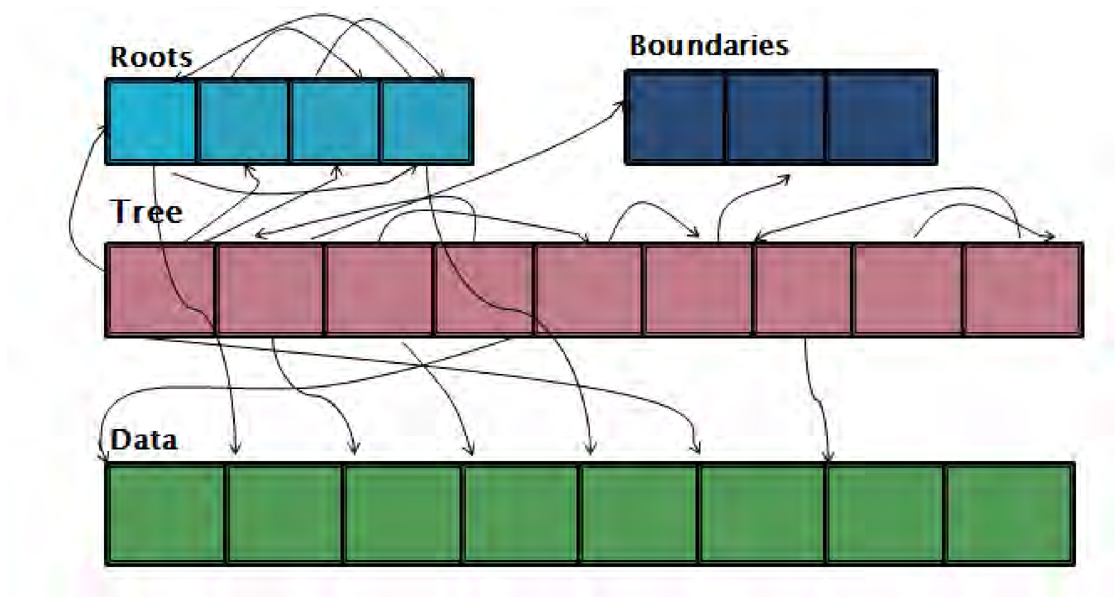


Figure 11

The figure illustrates the data structures of Gerris with the intervention of our own memory manager. We have allocated one continuous area for each object type.

4.2.1 Pointer rewriting

The linked data structures use structures with pointers. Once the pointers are transferred on the device, they will be invalid, as the host and the device are different architectures and have separate memory spaces (see Separate memory spaces). As it was referred at the previous section, all the nodes and the data used by the tree are scattered in the heap and therefore the tree is transferred to the device in segments. Therefore, the conversion of the device pointers so as to point to a valid address space is impossible.

The memory manager that we have developed solves the previous problem of the invalid device pointers. In particular, we allocate continuous memory areas for each data type. As the base addresses of the continuous memory areas allocated on the host and on the device are known, we can make the device pointers valid with pointer arithmetic.

At **Figure 12**, it is illustrated the procedure of pointer rewriting. Start is the base address of the host buffer and start ' is the base address of the device buffer. If a host pointer p points to the memory address start + offset, the corresponding device pointer p' will be rewritten as start'+ offset.

The expression for the conversion of the device pointers is:

devicePointer = deviceBaseAddress + HostPointer - - HostBaseAddress

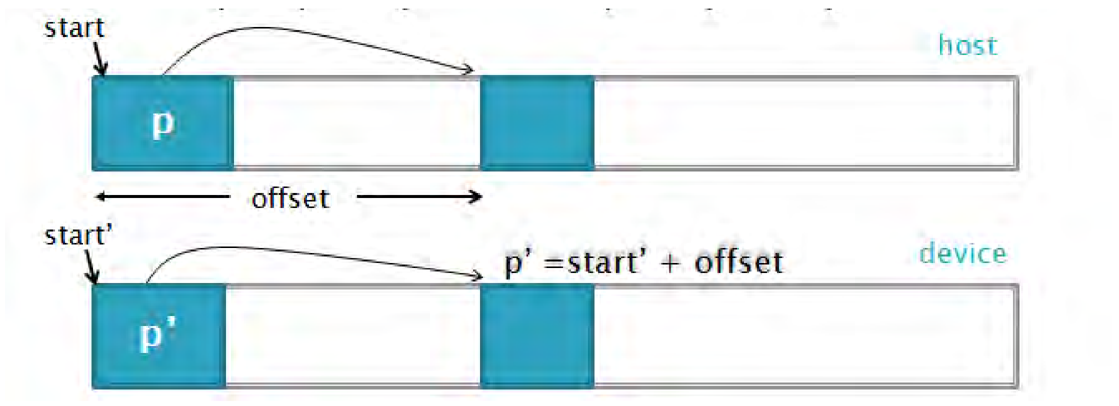


Figure 12

It is illustrated the procedure of pointer rewriting. Start is the base address of the host buffer, while start ' is the base address of the device buffer. Each host pointer p points at the memory address start + offset. Each device pointer p' will be rewritten as start' + offset.

As it is referred at Chapter 2 - The Application Gerris, the full threaded tree is used for the representation of the quad tree and consists of structures with pointers. The continuous memory areas allocated for each data type are illustrated at

We chose to use a heterogeneous system of a CPU and a GPU. We transfer the continuous memory areas (each for a data type, see **Figure 11**.) of the data structures of Gerris to the GPU. The pointers of the structures of the tree will point to invalid memory addresses. Thus, we rewrite the pointers so as to point to valid memory space with the technique illustrated at **Figure 12**.

In order to have low overhead, the pointer rewriting has been performed totally parallel on GPU. Each thread is responsible for rewriting the pointers of one element of the continuous memory areas Tree and Roots. Therefore we need these continuous memory areas to be arrays with elements of a specific type, so as each thread to be responsible for one element. For having continuous memory areas of a specific type, we allocate separate continuous areas instead of one large continuous area. However, we do not need the continuous memory area Data to be an array of a specific type, as each thread accesses it via the arrays Tree and Roots.

The positive thing is that the connections of the tree does not change on device, hence the tree is transferred to the device only once. If the connections of the tree were changed on the GPU, we would transfer the tree back to the CPU and update the connections. In this case, additional calculations would be added. Nonetheless, the data at which each node of the tree points are changed, thus we transfer the continuous memory area Data from CPU to GPU and inversely at each kernel call so as the updated data are used from the CPU and the GPU.

One alternative solution for keeping the connections among the nodes of the tree is to replace the pointers of structures with indices in the arrays. However, this is not such a good approach as we should change the code which is complex. Also, there are pointers pointing to more than one array. For instance a neighbor can be a root, a node, or a boundary. One last reason for not using indices is that the type of elements of the contiguous memory space Data is not specific (Variables and solids are stored in this space).

4.3 Enforcing the same representation of data between components

The size of a struct between a CPU and GPU may differ. Compilers add padding between elements if necessary in order to align the data address on a specific boundary. Due to the fact that CPU and GPU have different compilers, padding [13] can be placed differently. If CPU padding differs from GPU padding, there will be problem to the mapping of data from CPU to GPU.

As it is referred at Chapter 2 - The Application Gerris, all the data structures of Gerris that we transfer to the GPU consist of structures with basic type elements and pointers. Thus, we have faced this problem of different structure sizes between host and device.

4.3.1 Data alignment and padding

Data alignment [13] helps CPU to fetch data to memory effectively. Given the ultimate objective is high performance, CPU does not read one byte at a time but in 2, 4, 8 e.t.c. byte chunks at a time. Hence, the address of each data object is aligned at a specific boundary.

Most compilers align variables on their natural length boundaries [13]. However, the struct member variables must be aligned according to their next neighboring elements' alignment rules in order to prevent performance penalties. Therefore, the compiler pads bytes between the elements of a struct. An example is illustrated at **Figure 13**.

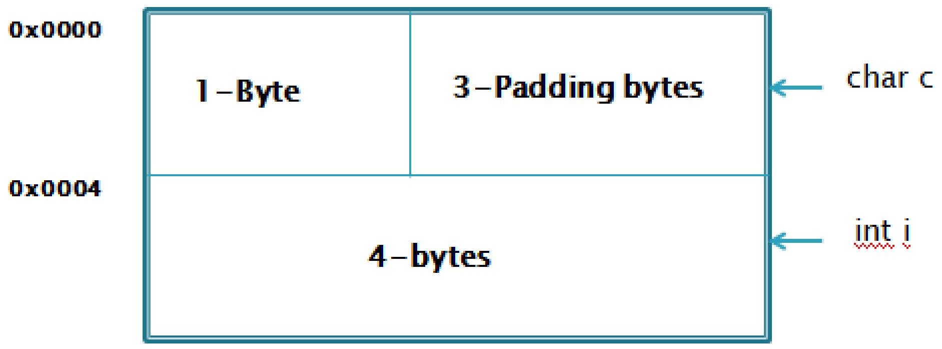


Figure 13

We have a struct with one char and one int. The compiler will pad three bytes between them so as the int will be aligned on a 4-byte boundary. Therefore, the size of struct will be 8 bytes instead of 5 bytes.

4.3.2 Same representation of structs between CPU and GPU

As heterogeneous programming language we have used OpenCL [11][12][16]. OpenCL is derivative of C99 rules, thus compilers are free to insert padding between struct members and at the end of the struct. If we want to pass structs to kernels it makes sense to specify alignment attributes [1] rather than attempting to guess what the particular OpenCL compiler we have installed is doing. Inserting hand-crafted padding members means that your program may not work correctly in a different computer.

If CPU padding and GPU padding differ, the size of structs at CPU and GPU differ too. Hence, the data will not be mapped correctly from CPU to GPU, and as a result the wrong data will be accessed (see **Figure 14**).

We have developed functions for each type of struct that we transfer to the GPU. These functions compare the structs of CPU and GPU, and specify how many bytes should be pad. Subsequently, the user should add the appropriate aligned attributes, and then run the program for verifying that sizes of structs are the same between host and device.

[1] The aligned attribute forces the compiler to align a variable on a specific boundary.

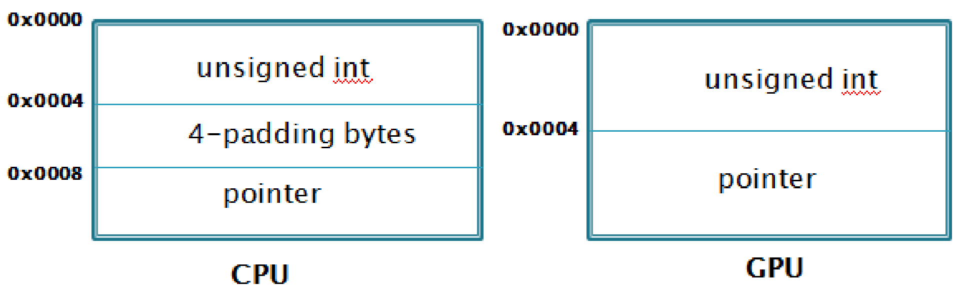


Figure 14

Provided that we have a struct with an unsigned int and a pointer, the figure depicts different padding between CPU and GPU. CPU pads four bytes between the two elements of the struct, while GPU adds no padding. If we attempt to read the pointer at GPU, instead of the pointer we will read the bytes padded by CPU. It is apparent the mapping of data from CPU to GPU is not correct.

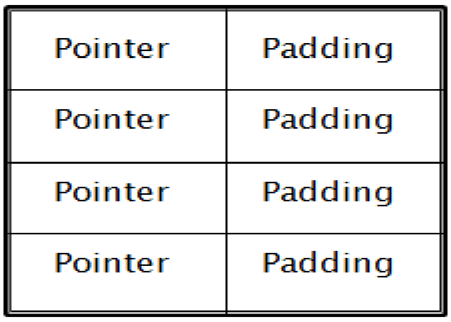
We have as host a 64-bit CPU and as device a 32-bit GPU. Therefore, the CPU pointer has the size of eight bytes, while the GPU pointer has the size of four bytes. As the data structures of Gerris are structs with pointers, the sizes of structs between CPU and GPU are different, and as a result the mapping of data from CPU to GPU is not correct.

One solution to this problem is to pad four bytes after each GPU pointer. Gerris have some arrays of pointers. However, we cannot pad bytes in between elements of an array. Since array is a contiguous block of memory, C does not allow padding between the elements of an array. Nonetheless, an array of structures may have padding added to each element as part of the struct itself.

Two dimensional array

We can solve this problem by a two-dimensional array (see **Figure 15**). For instance, if we have an array void * a[4] and convert it to void *a[4][2], four bytes padding is added after each pointer since GPU pointer's size is four bytes.

```
void * a[4][2];
```



Pointer	Padding
Pointer	Padding
Pointer	Padding
Pointer	Padding

Figure 15

We convert one dimensional array of pointers to two dimensional in order to have four bytes padded after each pointer (because GPU pointer's size is four bytes).

Union

We should leave the solution of the two-dimensional array since there is a problem at the **4.2.1 Pointer rewriting**. The GPU pointer has the size of four bytes, while the CPU pointer has the size of eight bytes. Hence, during the procedure of pointer rewriting, the CPU pointers are read half.

The expression for the pointer rewriting is:

GPU Pointer = GPU Base Address + CPU Pointer – CPU Base Address (1)

For example, If CPU pointer is 0x7f036d799014, we read 0x799014 at GPU.

The solution is the replacement of pointers with unions. All the elements of union share the same memory space, thus the size of union equals the highest size of all the elements.

We replace each pointer with the following type (see **Figure 16**):

```
union customPointer {
        global void * ptr;
        unsigned long ptrFull
    }
```

For the pointer rewriting of the Gerris tree, we read the ptrFull pointer (8 bytes) for getting the CPU Pointer, and after performing the expression (1) we assign the valid GPU address to pointer ptr (4bytes). In this way, we read the whole CPU pointer, and mapping of data is correct since now the pointer type is 8 bytes regardless of the GPU architecture.

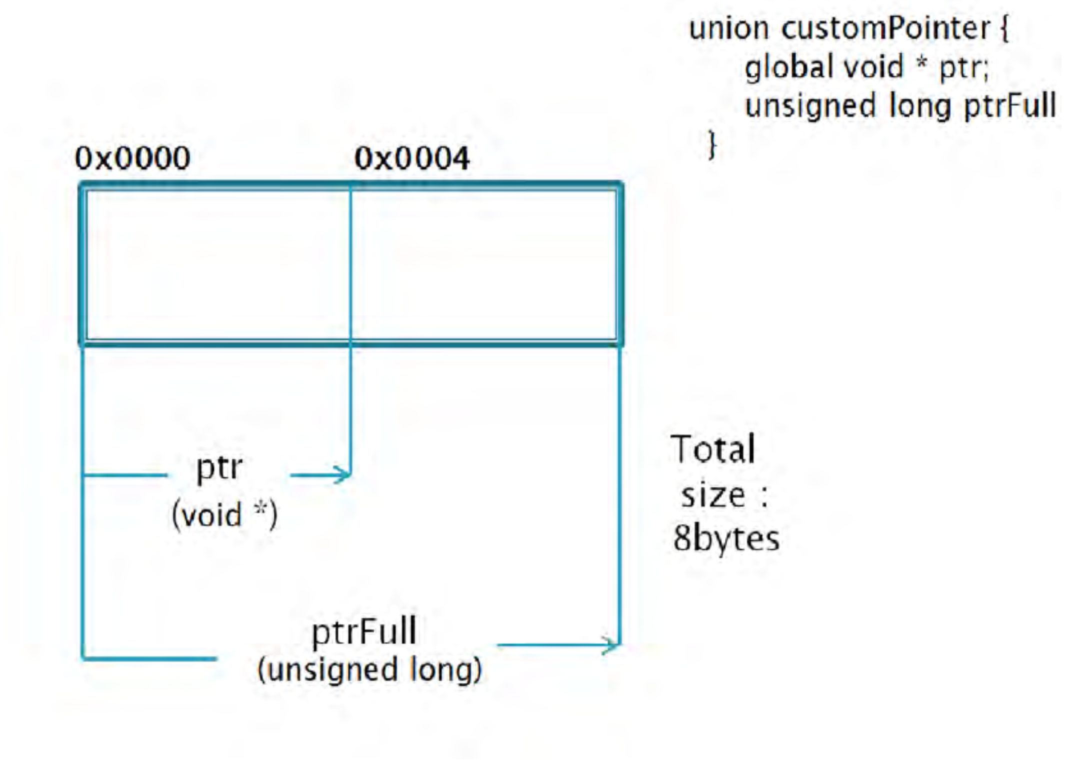


Figure 16

Union has the same syntax with struct, but differs at the storage allocated. Variables of a structure are stored at different memory areas, while variables of a union share the same memory area. Union’s size is equal to the maximum memory required by the members of the union. In this way, we read the whole CPU pointer and padding is automatically added after each pointer. With unions we have solved the problem of adding padding at arrays of pointers. So, the mapping of data from CPU to GPU is correct.

4.3.3 The same virtual address bit-width of the components

The question is why we make the situation so complicated as we can use components with the same virtual address bit-width leaving the complex solution of unions. As far as CPU is concerned, we can set it so as to be 32 bit, but in this case we do not use data more than 4GB. As for the GPU, we can use a 64-bit one, but we prefer one of 32-bit. There are not GPUs with DRAM memory more than 4GB. So if we have a GPU 64 bit, we cannot take advantage of it. Also, if we have a GPU 64 bit, we need more registers comparing with a GPU 32 bit. However, registers are limited resources [15] (e.g. the Fermi architecture has 63 registers per thread) but applications need registers because kernels store their local variables in registers. Having 64 bit pointers implies need for more registers, and as a result the possibility of register spilling[2] [15] is increased. Register spilling brings more memory transfers, and consequently deterioration in performance. Lastly, with 64 bit pointers, each thread needs more registers, therefore less threads can run concurrently and again there will be performance decrease.

4.4 Limitations

There are some limitations of our implementation. Firstly, the replacement of pointers with unions is drawback since we must change the code. Also, If an application changed the connections of the linked data structure on the device, each time we would transfer the pointers from device to host, we would make the pointers to point to the host's address space. Apparently, our implementation is not appropriate for this kind of applications as it adds many transfers and calculations. The last limitation is related to the memory footprint of an application. Lastly, if the memory footprint of an application does not increase almost continuously meaning that the program frees constantly memory, our approach is not appropriate. In particular, "holes" are created in the continuous memory area and as a result unnecessary data are transferred between host and device.

[2] Register spilling definition: when the code overcomes the maximum number of registers per thread then some variables are transferred or "spilled" to local memory.

4.5 Verification

We have transferred a function traversing the trees so as to verify that the changes of the code do not affect the correctness of the application. The traversal of the tree cells is recursive. The cells are visited recursively, and it is checked if they are at a specific level. If yes, the node calls a function. The traversal of the tree is called with an argument defining a specific level varying from zero level to a maximum desired level. However, the GPU does not support recursion. As the cells call the function per level, there is independence at the same level, and thus the cells can call the function in parallel. Therefore, we do not traverse the tree, but we traverse the arrays Roots and Tree (see **Figure 11**) which contain all the cells of the tree. Each thread checks one element of either Roots (one cell) array or Tree array (four cells). If the cells are at the desired level, a function is called.

Chapter 5 - Conclusion

The heterogeneous future is inevitable, a single application will exploit a "jungle" of enormous numbers of cores that are increasingly different in kind (specialized for different tasks) and different in location (on-die, from local to very remote e.t.c.). Although we harness performance with relatively low power dissipation of heterogeneous systems [7][8][9][10] , the current trend for using linked data structures at these systems is to be avoided. However, most applications use complex structures. The components of the heterogeneous systems have usually separate memory spaces and have different architectures. Therefore, it is the latency of transfers and the different representation of data among components that make the support of linked data structures difficult. The goal of this undergraduate thesis is the transparent support of linked data structures on heterogeneous systems. We have developed a memory manager that minimizes the transfers of the nodes of the linked data structures from the host to the device and a method ensuring that the data representation is the same on different architectures so as the mapping from host to device is correct. Our demands are low-overhead and not extensive changes of code. As heterogeneous system, we have used a CPU and a GPU. We program this system with OpenCL[11][12][16] and as case study we have used a fluid simulation application named Gerris [1][2][3]whose basic data structure is a tree.

Bibliography

[1] Gerris page: http://gfs.sourceforge.net/wiki/index.php/Main_Page

[2] Stephane Popinet: "Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries", National Institute of Water and Atmospheric Research, PO Box 14-901 Kilbirnie, Wellington, New Zealand

[3]Gerris reference:
http://src.gnu-darwin.org/ports/science/gerris/work/gerris-0.6.0/doc/html/book1.html

[4] A.M. Khokhlov, Fully Threaded Tree for Adaptive Mesh Refinement Fluid Dynamic Simulations Washington, DC 220375, (5-7)

[5] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, Doug Burger, "Dark Silicon and the End of Multicore Scaling" on ISCA'11

[6] The Heterogeneous Programming Jungle:
http://www.hpcwire.com/hpcwire/2012-03-19/the_heterogeneous_programming_jungle.html?page=1

[7] Heterogeneous Processing: a Strategy for Augmenting Moore's Law:
http://www.linuxjournal.com/article/8368

[8] A brief history of microprocessors:
http://www.csa.com/discoveryguides/multicore/review2.php

[9] Graphics and CPUs to gether: Are the Heterogeneous Processors the Future of Computing?
http://forwardthinking.pcmag.com/none/282278-graphics-and-cpus-together-are-heterogeneous-processors-the-future-of-computing

[10] What is Heterogeneous Systems Architecture (HSA)?
http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/

[11] OpenCL Programming Guide – A. Munshi, et al., (Pearson, 2012)
BBS

[12] The OpenCL specification Khronos Group

[13] Data alignment
http://www.songho.ca/misc/alignment/dataalign.html

[14] A look at GPU memory transfer:
http://blog.theincredibleholk.org/blog/2012/11/29/a-look-at-gpu-memory-transfer/

[15] Lecture “Local Memory and Register Spilling”, Paulius Micikevicius NVIDIA

[16] Khronos Forums: http://www.khronos.org/message_boards/