FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
INFORMATICS INSTITUTE
COMPUTER SCIENCE COURSE

VÍTOR UWE REUS

A GPU Operations Framework for WattDB

Graduation Thesis

Dipl.-Inf. Daniel Schall
Advisor

Prof. Dr. Renata Galante
Coadvisor

Porto Alegre, July 2012

CIP – CATALOGING-IN-PUBLICATION

Vítor Uwe Reus,

A GPU Operations Framework for WattDB /

Vítor Uwe Reus. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2012.

44 f.: il.

Graduation Thesis – Federal University of Rio Grande do Sul. Computer Science Course, Porto Alegre, BR–RS, 2012. Advisor: Daniel Schall; Coadvisor: Renata Galante.

1. GPU. 2. CUDA. 3. WattDB. 4. Database. 5. Framework. 6. GPGPU. 7. Energy-efficiency. 8. Energy-proportionality. I. Schall, Daniel. II. Galante, Renata. III. Título.

ACKNOWLEDGMENTS

First I would like to thank my family: my parents Margret Alice Reus and Paulo Nestor Reus, for all the care with which they treated me and for all the support they always gave me. I also would like to thank my grandmother Johanna Luise Voget, who was essential to my education, and was always an example of determination and good will.

Next I would like to deeply thank Prof. Dr. Theo Härder, for the opportunity to work in his research group and realize most of this thesis efforts under Technical University of Kaiserslautern. I also would like to thank Dipl.-Inf. Daniel Schall, which is advisor of this thesis, and Prof. Dr. Renata de Matos Galante for co-advising me in the rest of this work, both where unbelievably helpful along this writing process.

I also would like to thank all my friends and relatives, specially my girlfriend Paula, for the care and comprehension along this work.

Finally, to UFRGS as a whole, for the opportunity of studying in an university with excellence and international prestige; and to all others who have directly or indirectly helped me through this work.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic and logic unit
CUDA	Compute unified device architecture
DBMS	Database management system
FLOPS	FLoating-point operations per second
GPU	Graphics processing units
GPGPU	General-purpose computing on graphics processing units
IOPS	Input/output operations per second
SDK	Software development kit
SQL	Structured query language

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In the last decades, rising energy consumption and production became one of the main problems of humanity. Energy efficiency can help save energy. GPUs are an example of highly energy-efficient hardware. However, energy efficiency is not enough, energy proportionality is needed. The objective of this work is to create an entire platform that allows execution of GPU operators in an energy proportional DBMS, WattBD, and also a GPU Sort operator to prove that this new platform works. A different approach to integrate the GPU into the database has been used. Existing solutions to this problem aims to optimize specific areas of the DBMS, or provides extensions to the SQL language to specify GPU operation, thus, lacking flexibility to optimize all database operations, or provide transparency of the GPU execution to the user. This framework differs from existing strategies manipulating the creation and insertion of GPU operators directly into the query plan tree, allowing a more flexible and transparent framework to integrate new GPU-enabled operators. Results show that it was possible to easily develop a GPU sort operator with this framework. We believe that this framework will allow a new approach to integrate GPUs into existing databases, and therefore achieve more energy efficient DBMS.

Keywords: GPU, CUDA, WattDB, database, framework, GPGPU, energy-efficiency, energy-proportionality.

RESUMO

O aumento do consumo produção de energia elétrica tornou-se um dos principais problemas da humanidade nas últimas décadas. A eficiência energética pode ajudar a economizar energia. GPUs são um exemplo de hardware altamente eficientes em termos energéticos. No entanto, a eficiência energética não é suficiente, a proporcionalidade energética é necessária. O objetivo deste trabalho é criar uma plataforma completa que permite a execução de operadores em GPU para um sistema de gerência de banco de dados proporcionalmente energético, o WattBD, e também um operador de ordenamento em GPU para provar que essa nova plataforma funciona. Uma abordagem diferente para integrar a GPU no banco de dados foi utilizada. As soluções existentes para este problema tem como objetivo otimizar áreas específicas do sistema de gerência de banco de dados, ou fornecem extensões para a linguagem SQL que permitem especificar manualmente operações em GPU, resultando em pouca flexibilidade para otimizar todas as operações de banco de dados, ou garantir a transparência da execução em GPU para o usuário. Esse framework difere de estratégias existentes por manipular a criação e inserção de operadores em GPU diretamente no plano de execução, permitindo um framework mais flexível e transparente para integrar os novos operadores de GPU. Resultados mostram que este framwork possibilitou o desenvolvimento um operador em GPU de forma fácil. Acreditamos que este framework irá permitir uma nova abordagem para integrar GPUs em bases de dados existentes e, portanto, alcançar maior eficiência energética em sistemas de gerência de banco de dados.

Palavras-chave: GPU, CUDA, WattDB, database, framework, GPGPU, energy-efficiency, energy-proportionality.

1 INTRODUCTION

In the last decades, energy consumption and production became one of the main problems of humanity for economic and environmental reasons. Energy consumption is growing exponentially in all business areas. This trend is also followed in DBMSs, with constant growth of data that leads to bigger database systems. The first way to solve this problem is to generate always more energy. This solution is not perfect because the earth has a limited amount of energy to provide, and it will be impossible to follow an exponential curve of energy consumption growth. The second ways is to spend less energy, without harming performance. Energy efficiency can help save energy.

In this work, energy efficiency will be measured with Performance per Watt. Performance per Watt measures the rate of computation that can be delivered for every watt consumed. FLOPS per watt is a common measure of performance per watt. GPUs are an example of highly energy-efficient hardware because they have a high FLOPS per watt ratio. It is possible to achieve a higher performance per watt using general purpose computation on GPU because they are simpler than CPUs, and focus in massive parallel computation. Therefore, they can be good solution to spend less energy in a computing system.

Energy efficiency is, however, not enough. Energy proportionality is needed (Härder et al. 2011). The energy proportionality concept is to spend just the required amount of energy proportionally to system load. If a system uses 100 Watts at 100% load, it should use 50 Watts at 50%. This behavior is not followed in todays hardware, even in energy-efficient ones: a typical database server consumes about 60% of its peak energy consumption when idle (Tsirogiannis et al. 2010).

WattDB (Schall e Hudlet 2011) is a distributed database system running on commodity hardware that tries to be energy-proportional using a software strategy, and thus, saving energy, compared to an off-the-shelf server. WattDB is a database cluster of energy-efficient nodes. It dynamically turns nodes on and off, according to the overall cluster load. It is possible to achieve an approximation of the energy-proportional behavior with this architecture because every single node is supposed to be on a high load. Currently, WattDB just uses energy-efficient CPUs in the computing nodes.

The objective of this work is to create an entire platform that allows execution of GPU operators in WattBD, and also a GPU Sort operator to prove that this new platform works. This should increase the energy efficiency of WattDB. The main problem is that it is not possible to simply start running all code on GPU. GPUs have a separate memory, therefore constant memory copies between main memory and GPU memory needs to be done. The parallel architecture of the GPU also requires special designed code, and

since the processor is not identical to a CPU, a special compiler is needed. The proposed framework tries to solve this problems, and experiments show that it was possible to easily develop a GPU sort operator with this framework, and sorting on GPU is more energy efficient than sorting on CPU.

A lot of effort has been done before to use GPU in other DBMSs, such as using OpenGL (Mancheril 2011), creating CUDA enabled SQL procedures (PGOpenCL), or changing specific database operators to work alone with CUDA (Bakkum e Skadron 2010). Although they show good results alone, these are all very different approaches, that cover small cases, and does not have a portable and extensible architecture that can be reused in any database system. A different approach from existing ones has been used to integrate the GPU into the database, with the creation and insertion of GPU operators directly into the query plan tree.

In summary, the contribution of this work is a flexible and transparent framework to integrate new GPU enabled operators into a existing DBMS. It was possible to easily develop a GPU sort operator with this framework, showing that this architecture is extensible to new operators. We believe that this framework will allow a new approach to integrate GPUs into existing databases, and therefore achieve more energy efficient DBMS.

The rest of this work is organized as follows: Conceptual basis of energy proportionality, WattDB and GPU are presented in chapter 2. The GPU framework implementation details of the copy operations, how to overcome the CUDA limitations and the device operator model are presented in chapter 3. An experimental GPU sort operator is explained in chapter 4. Chapter 5 ends this work, showing the main contributions and presenting suggestions for future work.

2 CONCEPTUAL BASIS

The following topics are presented in this chapter: The concept of energy proportionality, why today's hardware does not achieve it and how a software solution can reach it. The WattDB project and how the several nodes works together. General purpose computation on GPUs and CUDA are presented, showing this architecture and why GPUs are more energy efficient than CPUs. In the end final thoughts are discussed.

Energy proportionality and WattDB are presented to understand the context of the framework. GPUs, CUDA and the limitations of the GPU architecture are shown to understand the need of a GPU framework in WattDB, which is confirmed by further implementation of a sort operator.

2.1 Energy proportionality

Ideally, power consumption of a computer system should be determined by its utilization. However, as seen in Figure 2.1, the observed power utilization of most of the systems does not behave like this. This leads to energy waste when the system is not at high utilization rates.

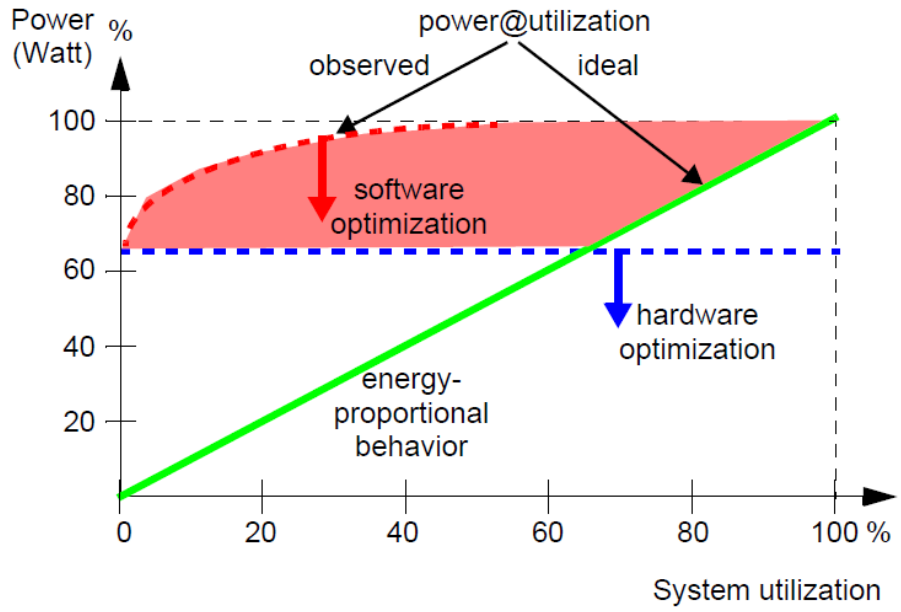

Figure 2.1: Lack of energy proportionality in today's hardware (Härder et al. 2011).

Energy proportionality is the ideal behavior of spending just the required energy amount, proportionally to the system load. Major hardware components are not energy proportional. The RAM, for instance, will always use the same amount of energy to maintain the data with refresh intervals, even if just a small percentage of the memory is being used. Although the energy consumption rises linearly when installing more memory chips, the consumption will remain the same even if just a small percentage of the memory is allocated. It is also not possible to just switch of RAM chips, especially in the course of DBMS processing, because they have to keep large portions of DB data close to the processor. The hard disk drive is another type of non energy proportional component. It needs to be constantly spinning, no matter how full it is, or how much IOPS are being performed.

CPU is one of the few energy proportional components in a computer system, but represent just a small slice of the consumption. The non-proportional behavior of most of the other hardware components makes the overall system non energy proportional. An energy proportional system would help companies to spare significant amounts of money from energy costs, while also helping reduce their carbon print. Google servers mostly reach an average CPU utilization of 30%, but often even less than 20% (Hoelzle e Barroso 2009). Figure 2.1 shows that at this load, power consumption is still around 90%, causing the system to use 300% more energy than needed.

2.2 WattDB

It is impossible to achieve energy proportionality with a single large server. To overcome the hardware limitation, a new server design is needed. WattDB is a distributed energy-proportional database cluster of wimpy nodes. The energy-proportional behavior is achieved using several small-scale server nodes (such as *wimpy nodes* (Andersen et al. 2009) or *Amdahl blades* (Szalay et al. 2010)) at high load, that can dynamically be turned on or off. When the overall cluster utilization rises, more nodes are attached to the cluster, and when it lowers, nodes are removed, in a way that all the nodes will always have a high load. Thus, an approximation of the energy proportional line is created as seen in Figure 2.2.

The cluster consists of 10 identical nodes. Two of them have 4 hard drives attached to provide persistent storage to the cluster, and are called data nodes. The other nodes, or compute nodes, can access the data through a 1Gb Ethernet switch. A dedicated master node is also attached to the Ethernet switch, and provides an SQL interface to the cluster. The overall cluster configuration can be seen in Figure 2.3

Each node can be considered an Amdahl-balanced (Szalay et al. 2010) node. They have 2GB of RAM, an Intel Atom D510, and the operating system is loaded from an USB stick. The data nodes have a low powered 250GB hard drive disk. This cluster configuration needs a software. WattDB manages node switching, with feedback information of each node, providing the proportional behavior. WattDB is also responsible to distribute the queries over the cluster, assuring a high load in each node.

WattDB is still under development, but is already able to process simple distributed queries over the nodes with an energy proportional behavior. The power consumption of WattDB is proven to be smaller than single oversized servers, and the performance of wimpy nodes is also better than a single server configuration (Schall e Hudlet 2011).

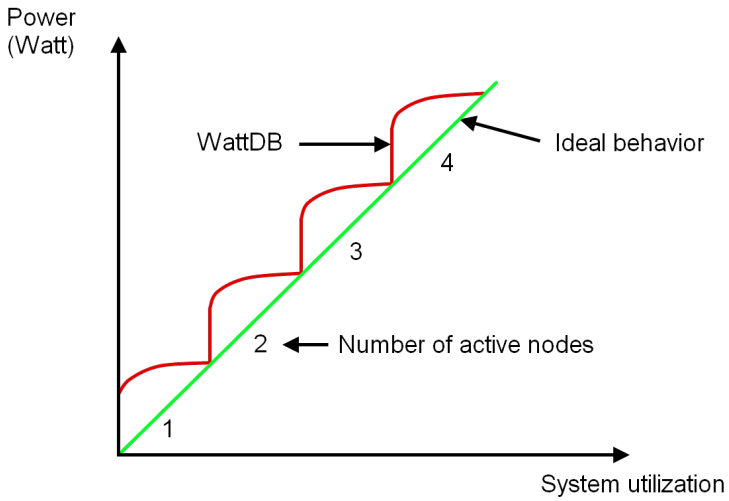

Figure 2.2: Approximate energy proportional behavior of WattDB.

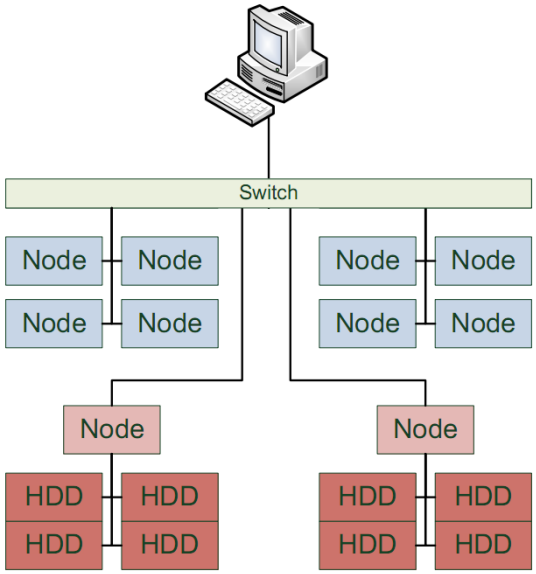

Figure 2.3: Cluster hardware layout (Schall e Hudlet 2011).

Therefore, WattDB is a proof that wimpy nodes are a good solution to achieve energy proportionality.

2.3 GPGPU and CUDA

The use of general purpose computation on GPU started when OpenGL, DirectX and similar APIs started to implement pixel shaders, which allows transformation of graphic textures with a short parallel program. Textures are matrices of floating point numbers that represent pixels. With shaders, floating point parallel computation capabilities was added to the GPU, and after some time, it made the GPU as flexible to program as CPUs. Seeing the interest of general purpose computation on GPU, NVIDIA[1] introduced in February of 2007 the CUDA SDK, which is currently a widely adopted parallel computing architec-

[1] www.nvidia.com

ture for GPU computing.

The higher energy efficiency of GPU is possible due to their simpler architecture design. Since the GPU is a heavily parallel architecture, as we can see in Figure 2.4, it devotes more silicon area to several parallel ALUs. Less area is dedicated to control and cache, and that way its possible to achieve much more FLOPS with much less energy consumption with a simpler architecture.

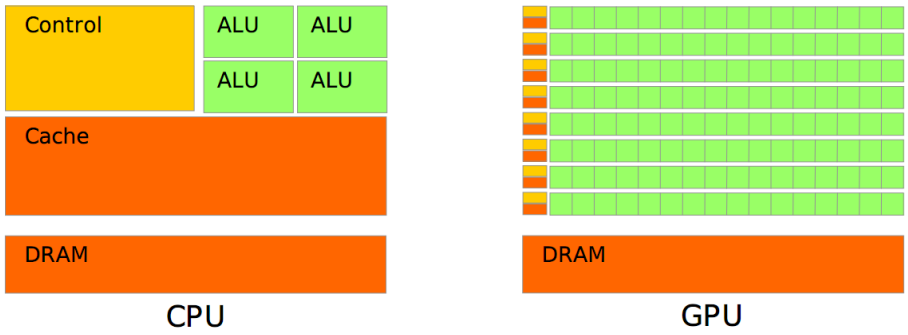

Figure 2.4: CPU and GPU Hardware architecture (NVIDIA 2008).

I will be using CUDA to develop in GPUs. It has the drawback of being compatible only with NVIDIA cards, but there exists some not perfect, tools that can convert CUDA code to OpenCL (Swan: A simple tool for porting CUDA to OpenCL 2012), and NVIDIA recently announced that it will open up the CUDA platform by releasing source code for the CUDA Compiler (CUDA Platform Source Release 2012), that turns the CUDA platform limitation problem less significant.

When the GPUs where designed, it was supposed that it would only do graphic operations. For this reason, it has a separate memory for storing graphics related data, like the frame buffer, sprites, textures or polygon meshes. This kind of data is usually loaded once, and then manipulated by the GPU, and does not matter the CPU.

The memory model of the GPU makes it impossible to access it is memory from the CPU, and the CPU memory from GPU. This is a problem for general purpose computations on GPU (GPGPU) since we want to be able to have the data available in both processing units. To solve this, constant data copies should be done between the two memories. This leads to memory copy bottlenecks (NVIDIA 2008), because data are always being copied from one memory to the other.

To compute over the data stored in GPU, we normally set one CUDA thread to be responsible of a single element of an array. So if there is a 1 million elements to be computed, each thread will be responsible for one of this element. That is why CUDA is a SIMD architecture.

CUDA proceadures are defined as special functions called kernels. This kind of functions are GPU entry-point functions. This mean we can call this function from the CPU, and then it will start running in GPU. To define a kernel we use the special word __global__, wich is a CUDA extension of C:

```
__global__ void myKernel(int data[], size_t length);
```

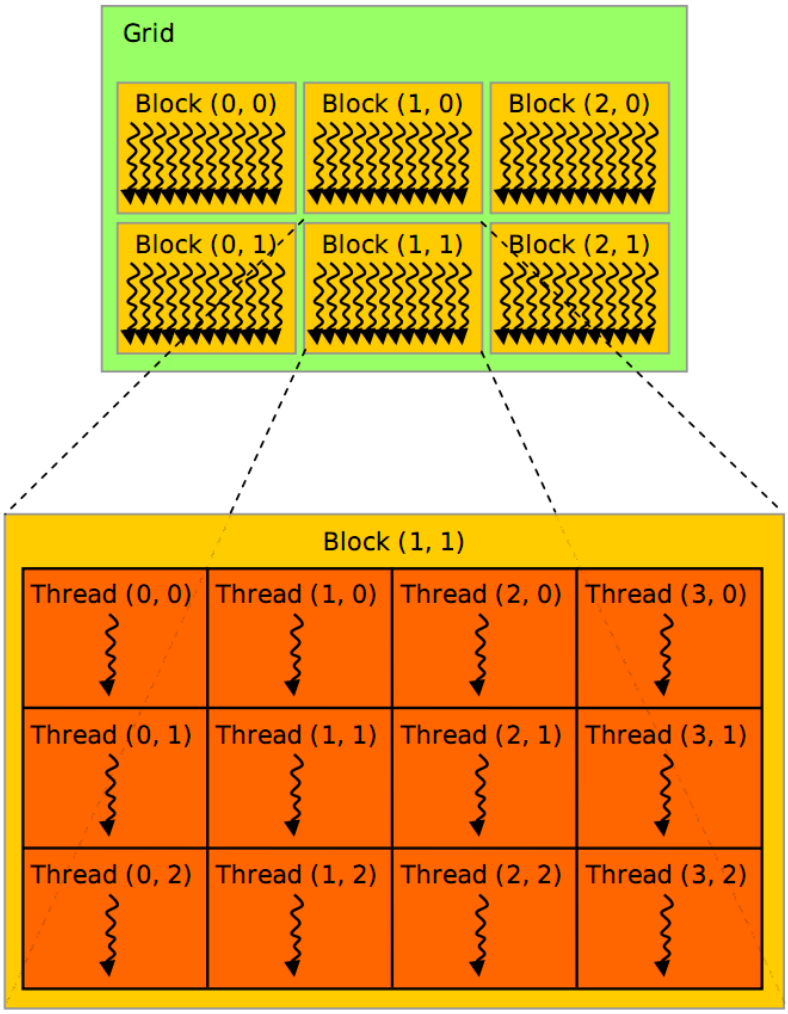

Figure 2.5: Thread Block and Grid (NVIDIA 2008).

To call this function, we first need to understand the threads hierarchy of CUDA. As we can see in Figure 2.5, threads are organized in 2 levels of groups. The first level is called Block, and they are a group of threads. The second level is a Grid, which is a group of Blocks.

When calling kernels, we can define the number of threads per block, and block per grid. Parallelism occurs in the grid. All cores will execute a different block at the same time. Therefore, all threads inside the same block will be executed sequentially by the same core.

To specify the number of threads that will run, we use the following CUDA extension:

```
myKernel<<<gridDimension,blockDimension>>>(myData, dataSize);
```

Where gridDimension and blockDimension, can be either an int or CUDA's dim3, a vector of size used to represent 2D or 3D Grids or Blocks. Note that this syntax extensions will only work with the CUDA compiler, and will throw a compilation error if compiled with gcc/g++.

Even with all the drawbacks of CUDA, that are introduced with its simpler architecture, it offers a great platform for parallel computation, and since it is proved that the

performance per watt of GPUs are higher that CPUs, we believe that CUDA can help WattDB to become more energy efficient.

The CUDA model assumes that the CUDA process executes on a physically separate *device* (GPU), that works as a co-processor of the *host* (CPU) that runs the C program. The CUDA model also assumes separate memory spaces, referred as *host memory* and *device memory*. This terms are important to understand the separate memory model of CUDA.

2.4 Final remarks

Today's servers does not achieve the proportionality, because most of the hardware components are not energy proportional. WattDB tries to become energy proportional by approximating the energy proportional behavior with multiple non energy proportional computing nodes.

In this work, GPGPU is integrated into WattDB. The objective is to provide parallel computation power to some parallelizable operations of the DBMS. The advantage of using the GPU in WattDB context is the high energy proportionality of the GPU. To be able to use the GPU in database systems, a GPU framework have been developed to allow the development of parallel operation in WattDB. The implementation is for WattDB, but design is not limited to it, and can be extended to other DBMS.

3 GPU FRAMEWORK

The objective of this chapter is to explain the GPU Framework architecture. Main problems related to the CUDA model and WattDB architecture are discussed and their solutions are presented. The next sections are organized as follows: An overview of the proposed framework is presented in section 3.1. Section 3.2 explains the volcano tuple architecture of WattDB and how it is implemented in GPU. Section 3.3 explains copy operators that converts normal tuples to GPU tuples. Section 3.4 explains the query plan modes, how the copy operators dictates them and why this is relevant to the new GPU operators. Section 3.5 discusses technical limitations of the CUDA programming language and how to overcome them. Concluding remarks are presented in section 3.6.

3.1 Overview

The objective of this framework is to create a flexible platform to develop GPU database operators in WattDB. Although one could manually port each operator to work with CUDA, the framework wraps the GPU into a easy to use interface hiding the major structural differences and low-level programming bureaucracy of the GPU and provides standard interfaces to the new GPU operators. That way, a easy to use environment is provided to create new operators, taking CUDA limitations and implementation details away from the programmer.

Figure 3.1 shows an overview of the framework architecture. The WattDB container represent part of existing WattDB architecture that matters to the framework, that is, the query plan, the operators and tuples. The new framework provides a platform that simplifies the use of the GPU. The new GPU database operators can be extended to user-defined operators, such as the GPU sort operator that we have implemented to validate the framework. The user-defined GPU operators should use user-defined functors to control the operator behavior. GPU operators use GPU tuples, which are similar to existing WattDB tuples, but are allocated in GPU memory space. At least, the framework provides copy operators, that convert WattDB tuples to GPU tuples.

In older CUDA architectures, all memory management should be made on host side, using cuda_malloc and cuda_free functions. This means that when code starts running on GPU, it could not allocate of deallocate memory. The new CUDA Fermi architecture allows the use of free and malloc functions inside device code. That allowed the straightforward translation of WattDB Tuples to DeviceTuples and allows a much bigger encapsulation, so the DeviceTuples are responsible for allocation of DeviceValues, and DeviceValues responsible for the actual data, which would be impossible without device

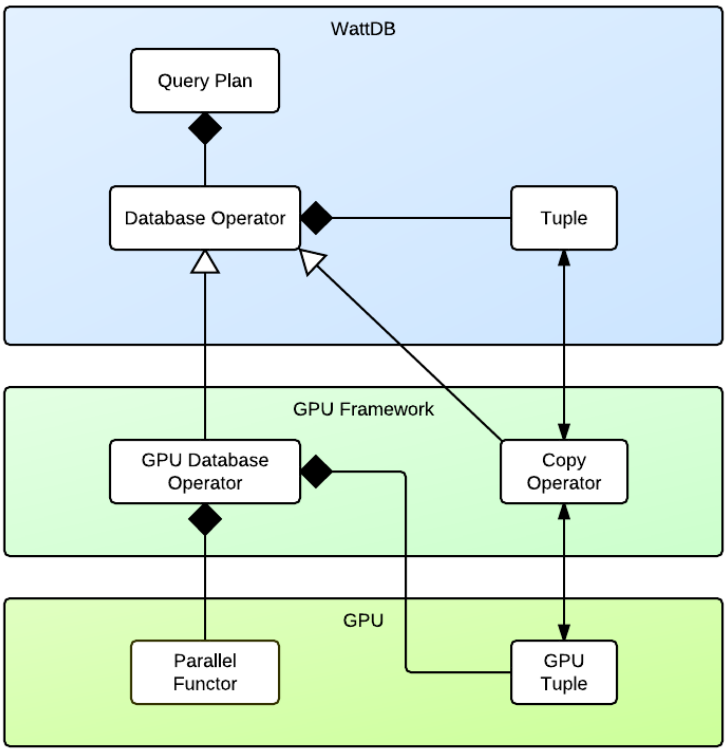

Figure 3.1: Framework architecture overview.

side memory management.

3.2 Device Tuples

WattDB represent database tuples using Volcano style Tuples (Graefe 1994). WattDB operators depend on the Tuples and Values implementation. This classes stores the data that will be used by the operators. Each Tuple holds a set of Values, and each Value have a reference to the memory area containing the actual data, as seen in Figure 3.2.

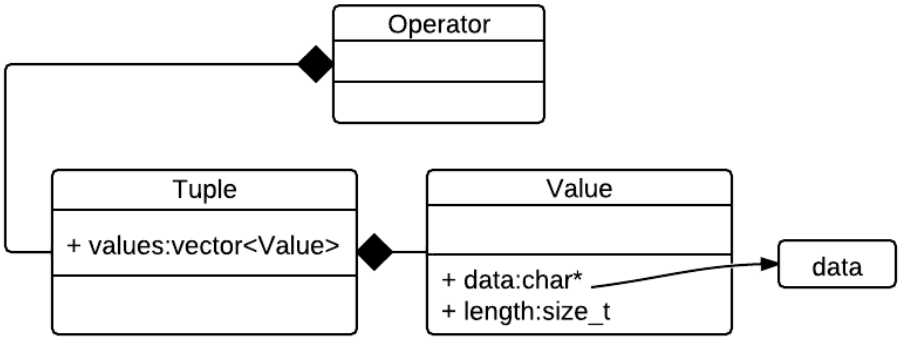

Figure 3.2: Current Tuple, Value and data implementation in WattDB.

Since now we use the GPU memory, *DeviceTuples* and *DeviceValues* will be added to WattDB. This classes will hold the data stored on GPU. This classes will need to be allocated on GPU memory, and are responsible for all memory management of the data. *DeviceOperators* will, although, be allocated on CPU memory because the query opti-

mizer needs to have access to them, in order to create the query plan. This is better explained in section 3.4. Figure 3.3 shows the DeviceOperator being allocated on CPU and holding reference to DeviceTuples and DeviceValues, being allocated in GPU.

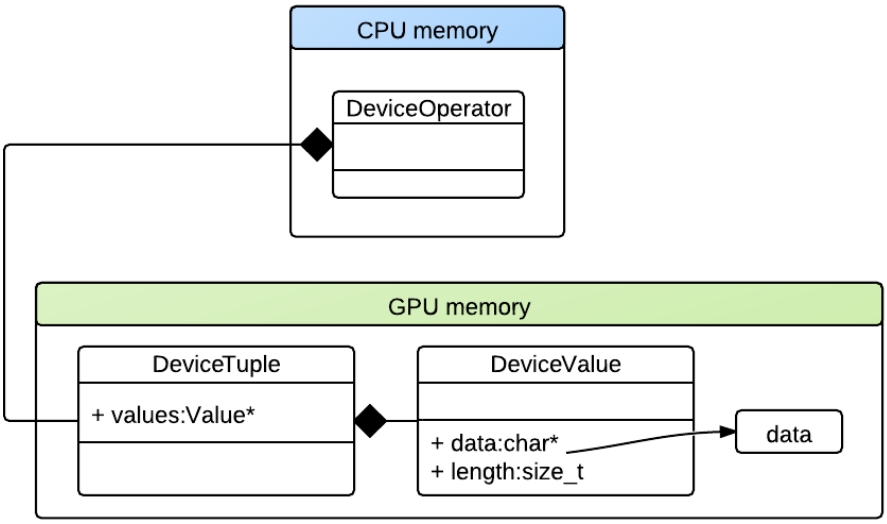

Figure 3.3: New Device Classes.

The new DeviceTuples does not replace the old WattDB tuples. Both kind of tuples are present in the database. WattDB will not be aware of the existence of the new DeviceTuples because the DeviceOperators encapsulate them. This allows minimum change to the existing code of WattDB, making the integration process very easy.

3.3 Copy operators

CUDA has a separate memory for the GPU. Because of hardware limitation, that is, no DMA access from device to the system memory, the device cannot access data stored on host memory, and vice-versa. The only way to make data available in both processing units is by doing data copies.

The separate CUDA memory model is the reason to create a second type of tuples in WattDB, the DeviceTuples. With two types of tuples present in the system, a mechanism is needed to transport data from WattDB tuples, to DeviceTuples. The data copies needs to be done in an easy way to the programmer, to provide a good GPU framework for WattDB.

In CUDA, to copy data between CPU and GPU, there is the cudaMemcpy function that is called called by the host:

```
cudaError_t cudaMemcpy( void *               dst,
                        const void *         src,
                        size_t               count,
                        enum cudaMemcpyKind  kind  );
```

Where dst is the destination pointer, src the data source pointer, count the number of bytes, and kind the kind of copy that is intended:

```
cudaMemcpyHostToHost       Host -> Host
cudaMemcpyHostToDevice     Host -> Device
cudaMemcpyDeviceToHost     Device -> Host
cudaMemcpyDeviceToDevice   Device -> Device
cudaMemcpyDefault          Default virtual address space
```

The framework will take care of the low-level CUDA memory management, using two new databases operators: CopyTo and CopyBack. CopyTo copies and convert an entire stream of tuples from Host to Device, and CopyBack from Device to Host.

The CopyTo and CopyBack operators will be inserted in the query plan to allow the execution of CUDA code, making the data available in the correct memory. The responsibility of copying the data is now taken away from the programmer, and given to the query plan optimizer.

3.4 Query Plan

In relational DBMS, the query plan is an execution plan of database operations to access or modify data. The query plan is normally represented as a tree. Some database operations are commutative and in other cases, some operations can be added or removed without changing the execution result. This leads to several different equivalent query plans with different performances. The objective of building a query tree is to heuristically find a good performance tree, that is, minimize the database response time and maximize throughput.

WattDB uses a vectorized Volcano Query Evaluation System (Graefe 1994) implementation. It has volcano-style operators interface. In wattDB, the query plans are created with the ResultOperator objects. ResultOperator can hold their child operators, and fetches Tuples from them using the next() method, as in Figure 3.4. This operators will be extended to allow GPU operators. The GPU framework adds the new DeviceOperator, which works almost the same way as the old WattDB ResultOperator, but holds DeviceTuples instead of the old WattDB Tuples.

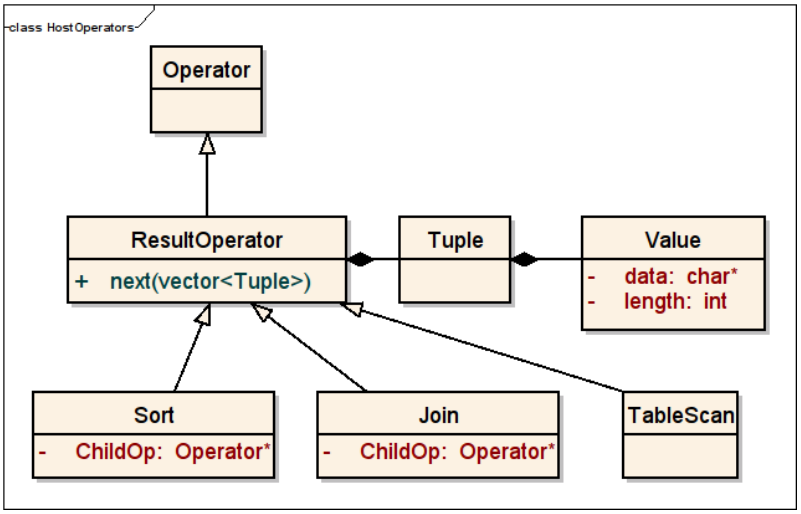

Figure 3.4: Operators, Tuple and Value.

The vectorized implementation of Volcano Tuples in GPU was made using the Thrust (Hoberock e Bell 2010) library. Thrust is a C++ template library for CUDA based on the Standard Template Library which is becoming a standard in CUDA programming, because it comes already installed with CUDA since version 4.0. It provides a device_vector implementation, with is equal to the std::vector, but using device memory. It also has some useful methods like thrust::transform (map), thrust::reduce or thrust::sort. The DeviceTuples are stored using thrust::device_vector.

The complete design can be seen in Figure 3.5. The green classes are the ones added by the framework. The CPU operators will continue to exists in WattDB, the new device classes will just be added to WattDB. The DeviceOperator does not aims to replace CPU operators, both of them are available in WattDB. A query plan can be build with DeviceOperator and CPU ResultOperator at the same time since both of them inherits from the common Operator class.

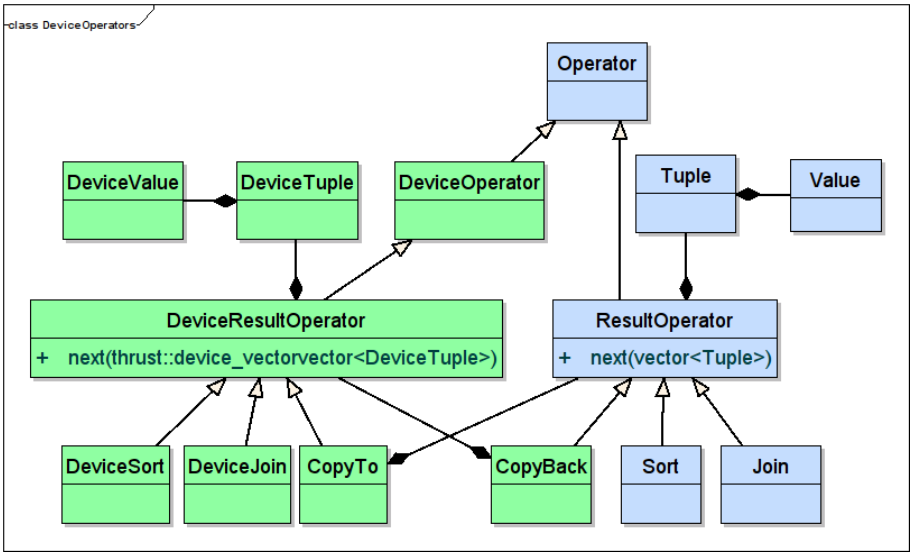

Figure 3.5: Adding DeviceTuples, DeviceValues and DeviceOperators to WattDB.

Note that the DeviceOperators will be allocated on CPU, as previously seen in Figure 3.3. DeviceOperators are allocated on Host memory, because the query plan needs to have access to the operators, just the DeviceTuples, DeviceValues and actual GPU data will be on device memory.

The copy operators are database operators that will be inserted into the query plan. They will copy and convert tuples to the right working memory. This strategy of placing copy operators in the query plan will define the *query plan mode*. The query plan mode alternates between host and device mode when placing copy operators, as seen in Figure 3.6. The query plan starts at host mode, usually with the Tablescan operator. If a CopyTo operator is inserted, the plan will be switched to device mode. If the plan is in device mode and a CopyBack operator is inserted, the query plan will be switched back to host mode. Other way to visualize the copy operators is interpreting them as a bridge for the CPU and GPU query execution flow, as seen in Figure 3.7. Note how the position of the Next() interface change between GPU and CPU, before and after the copy operators in Figure 3.7.

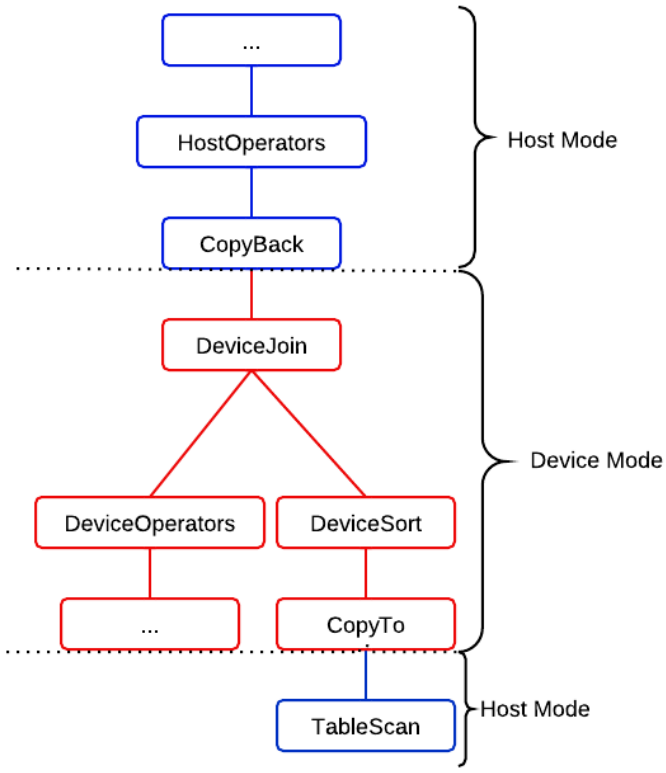

Figure 3.6: Query plan modes.

It is interesting to note that CopyTo is a DeviceOperator, as seen in 3.5, but CopyBack is a WattDB host Operator. This is not obvious in first sight, but it is that way because CopyBack returns WattDB host Tuples, thus, needs the next interface of an WattDB host Operator. We can see this also in Figure 3.7. where next interface can be seen as the lines that connect the operators.

To allow the execution of the WattDB host operators, the query plan must be on host mode, and to allow device operators, on device mode. This allows the coexistence of both host and device operators in one database, and provides a flexible architecture to make the choices between host and device.

With the query plan mode, the programmer can write GPU database operators supposing that the data is already in the correct memory. GPU operators will automatically use GPU memory, and the CPU operators, the CPU memory.

WattDB Operators needs to be rewritten as DeviceOperators to enable parallel execution advantage of the GPU. This translation needs to be done manually translating already existing WattDB Operators. This will be, although, a straightforward translation because both uses Volcano style Tuples, and when using the DeviceOperators the Tuples will be already on the right memory thanks to the Copy Operators and plan modes. The main difference is that the DeviceOperator will have a parallel CUDA functor defined in the GPU memory that can be accessed via the CPU using a map reduce strategy. This will be better explained in chapter 4, where a Device Sort Operator is implemented.

With a hybrid tree, there will be the possibility to choose between CPU or GPU operators. To decide between CPU or GPU operators, energy consumption of the GPU should be taken into account in the cost prediction calculation of a query tree. GPU operations

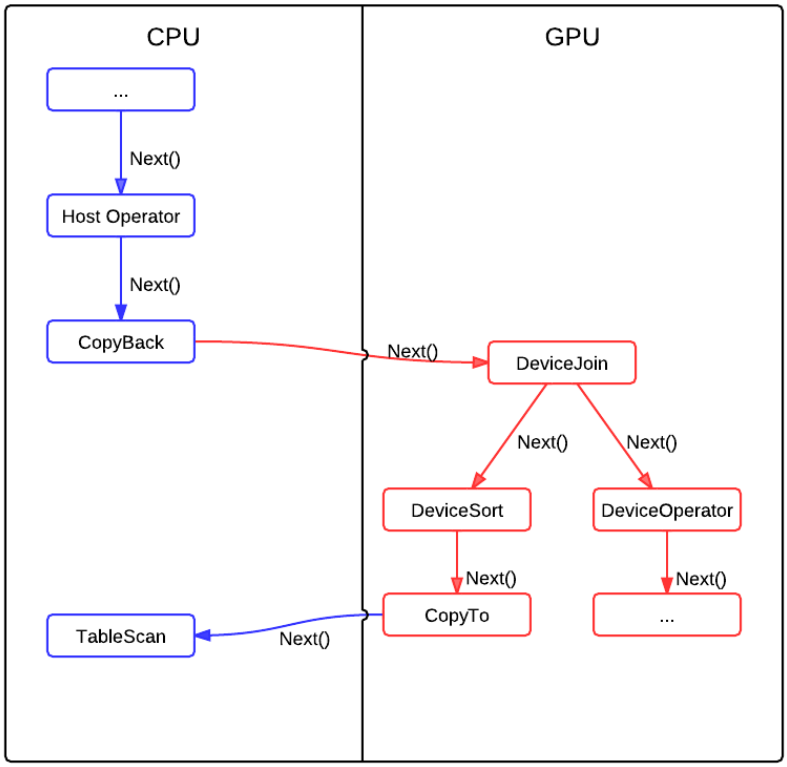

Figure 3.7: Example of a data flow in the framework.

normally spends more energy than CPU, but they normally run faster than CPU. The cost calculation could be changed to predicted_cost = energy_consumption * predicted_time in order to include the energy of the GPU, where energy_consumption is a different constant for CPU and Device operators.

At least, not all operations are highly parallelizable and also just by installing a GPU in a system makes it spend more energy. An heterogeneous configuration of the nodes of WattDB could be useful, meaning that just some of them will have a GPU installed. The nodes that have a GPU can be switched off when there are no parallelizable queries. The best way to do this is adding awareness of the GPU to the monitoring system of WattDB (Dusso 2012).

3.5 Dealing with C for CUDA limitations

The programing language for CUDA is Nvidia's "C for CUDA". It is C with Nvidia extensions and certain restrictions. CUDA source files have a .cu extension, and are processed with the Nvidia compiler nvcc. Not all standard C is compatible with nvcc, and there are some techniques to deal with this differences.

One big restriction of C for CUDA is that all device function calls are inlined. The first consequence of this is the inexistence of recursion. The second is not being able to call device functions between different CUDA source files.

The inexistence of recursion consequence can simply be fixed by transforming the recursive algorithms into iterative, and if needed, implementing a stack of arguments and context variables. There where no recursive code in the operators, tuples or values of

WattDB, therefore this technique was not used.

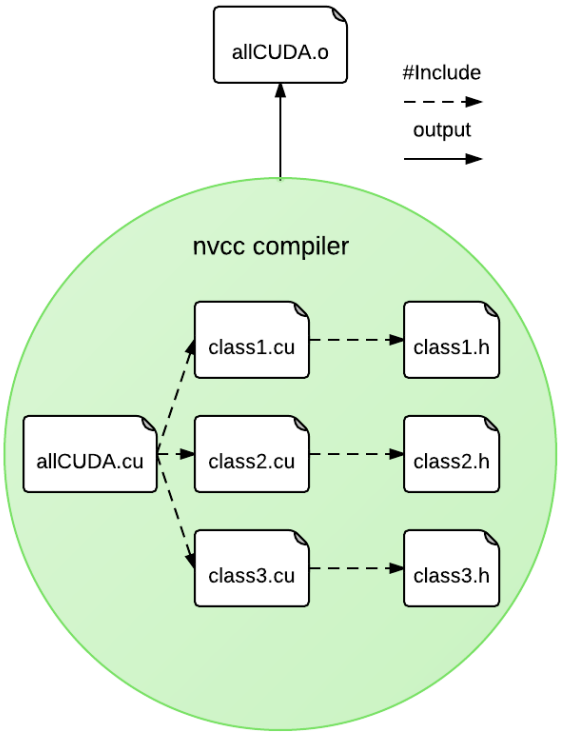

Figure 3.8: Using one file to be able to call device functions between different files.

The second consequence, the impossibility to call device functions between different CUDA source files, occurs because with all functions being inlined, a device side code linker is not needed. The resulting objects of the compilation of different CUDA source files would not be able to see each other. To overcome this restriction, a third file, that includes all other CUDA source files has been created, resulting just one big object, and one compilation step for all .cu files, as seen in Figure 3.8.

Another problem that needs to be solved is the use of two compilers. Since g++ can not compile CUDA, and nvcc does not compile all standard C, two compilers are really needed.

The advantage of nvcc is that it can produce an output object file that can be linked with g++.

The .h of the .cu file will also be compiled by the g++, therefore needs to be created in a way that can be processed by both compilers. Some CUDA extensions are not compatible with the C default syntax. This extensions includes, the definition of device functions, call of device function, device memory management functions. On the other hand, some default C can not be compiled with nvcc, like templates.

To create the compatible common file, #ifdef __CUDACC__ guards where used to hide code from one compiler or other. The example below shows the next interface of a DeviceOperator, that is invisible to the g++, because it uses DeviceTuples that are not available in CPU.

```
class DeviceResultOperator {
```

```
public:
  #ifdef __CUDACC__
   virtual bool next(
     thrust::device_vector<DeviceTuple>& outDT ,
     thrust::device_vector<DeviceTuple*>& outDTR
   ) = 0;
  #endif

};
```

The next example show how to hide the boost::spirit (Boost C++ Library) library, which fails to compile under nvcc:

```
#ifndef __CUDACC__
#include <boost/spirit/include/qi.hpp>
#endif
```

This code will create two different interfaces for the DeviceResultOperator class. One for the GPU, and other for the CPU. The common .h file, and all this process can be seen in Figure 3.9.

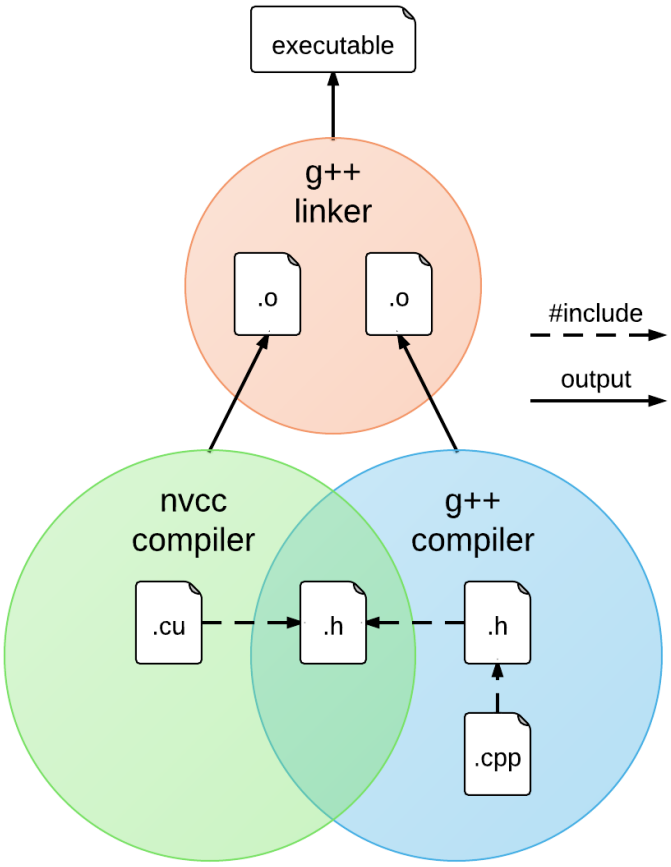

Figure 3.9: Using two compilers in a build process to integrate CUDA in an existing application.

Joining the two methods of creating a common .h file with two different interfaces, and using just one file to include all CUDA file, we achieve the resulting build process that can be seen in Figure 3.10.

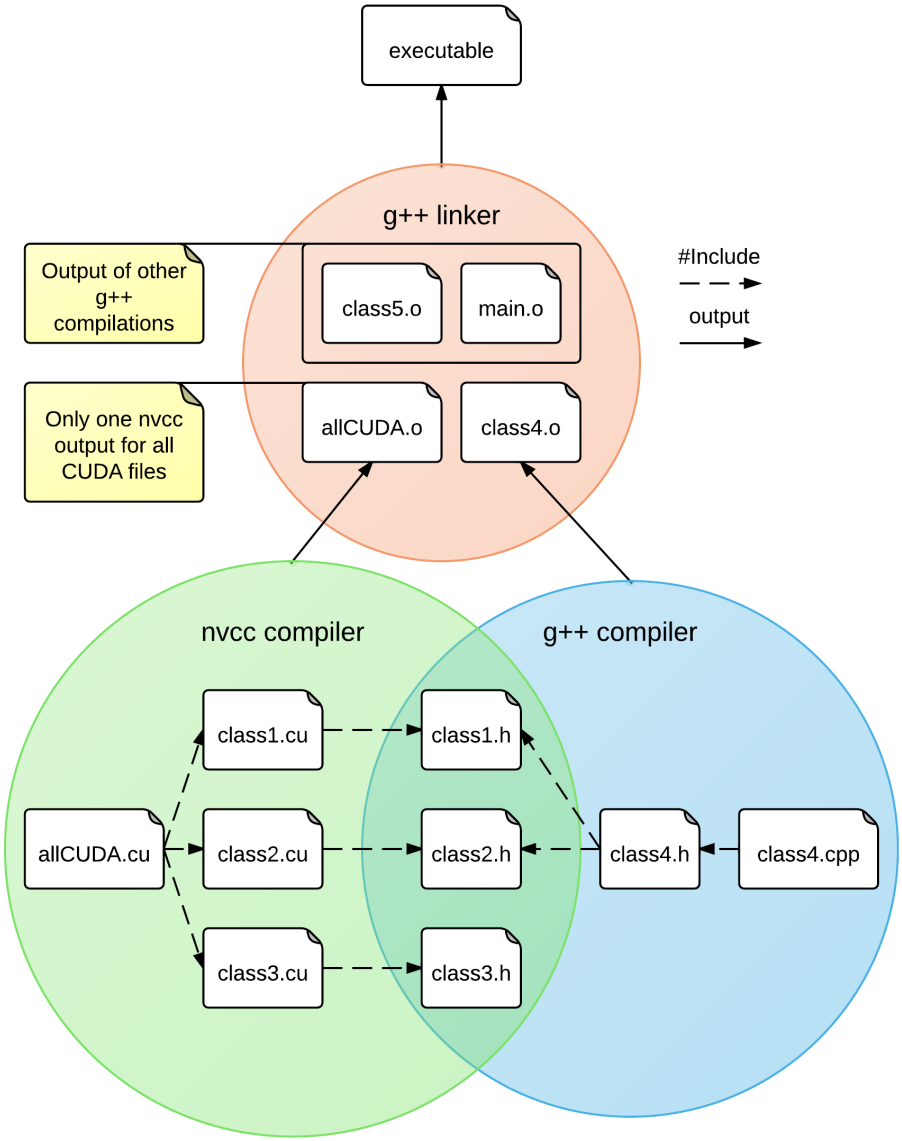

Figure 3.10: Resulting process to integrate CUDA in WattDB.

3.6 Concluding Remarks

In this chapter, the architecture of the Framework was presented, explaining how the copy operators alternate the query plan between host and device mode, allowing the execution of WattDB Operators or DeviceOperators. Technical workarounds for the CUDA programming language are also part of the framework. Everything together provides a comfortable environment to develop new DeviceOperators in WattDB.

The main contribution is that the framework solves a significant amount of the problems when programming in CUDA, allowing the programmer to focus only on the development of the DeviceOperator. Another contribution of this framework is that it does

not require the change of the existing architecture of WattDB, providing a modular and extensible architecture that can be used in other databases besides WattDB.

Other contribution is the data copy optimization. The copy of data is one of the bottlenecks of the GPU operators, but with the new cost prediction formula it is possible to determine the cost of the copy, and determine if it is worth running in GPU. Also, with copy being done just once, this bottleneck can be minimized creating a pipeline of DeviceOperators in the device mode query plan, and reusing data that has already been copied into the GPU memory.

The main limitation is the lack of a more automated translation from existing WattDB Operators into DeviceOperators. It is not as simple such as parallelizing loops with OpenMP, but easier when comparing to raw CUDA programming because of the automated memory copy management.

At last, we believe that this framework resolves major technical issues of the CUDA programming language, and provides an extensible platform to develop GPU operators with minimal architecture change for WattDB. Also, since this framework provides a modular architecture, it could be used in other DBMS as well.

4 EXPERIMENTAL EVALUATION

The objective of this chapter is to validate the GPU Framework, proving that it is easy to develop GPU operators with it and that GPU operations can be more energy proportional than CPU operations. The validation has been done implementing a DeviceSort operator. The next sections are organized as follows: The new DeviceSort operator is explained in section 4.1. Experiments configuration are presented in section 4.2 and experiments results are described in section 4.3.

4.1 Device Sort operator

In this section, the Device Sort operator is described. At the implementation time of this operator, WattDB did not had a host sort operator, so it was also developed to compare results, as shown in section4.3. After the development of the host sort operator, it was translated as a DeviceOperator. This operator fetches unordered tuples and returns them in order. Both implementations fetches all tuples from the query tree child operator, orders all tuples properly, stores the ordered result, and returns data to the parent operator on demand. This can be seen in Figure 4.1.

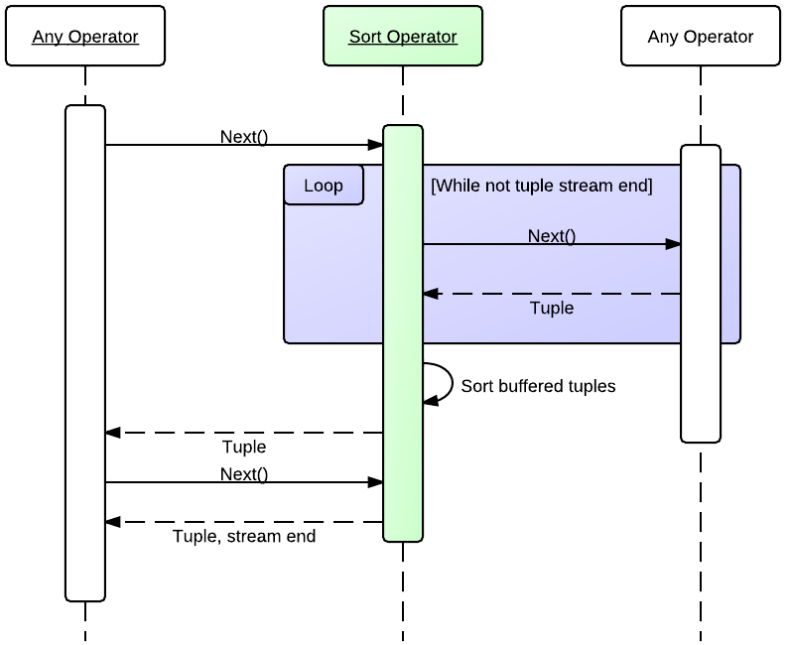

Figure 4.1: Sorting operator overview.

Since table columns are mapped as values, the description of the values that are going to be sorted needs to be passed when the sort operator is instantiated. That is because the SQL ORDER BY statement requires a list of columns to be sorted. The tuple description consists of the value id and value type.

To perform the sort on host operator, the Standard Template Library sort was used (std::sort). In order to use this function, a custom comparator functor is created using the value ordering data passed in the instantiation of the sort operator. This allows the sorting of any kind of value.

After the host sort operator was completely tested and working, the DeviceSort operator was implemented translating the host operator. The thrust (Hoberock e Bell 2010) library was used to quickly translate the device sort operator, using the thrust::sort method. The DeviceSort operator can deal with different types of DeviceValue types of data, the same way the host operator does, thanks to the use of a custom comparator functor, but this time, instantiated on device. These are the main differences between host and device sort: std::sort versus thrust::sort, standard functor versus device functor, and WattDB Tuples versus DeviceTuples. Since the interfaces of all this components are very similar, the translation was really easy.

Since the DeviceTuples of DeviceOperator are stored using Thrust::device_vector, it is highly encouraged to use thrust::transform (map) and thrust::reduce functions to code the device operators. Using map-reduce, the translation process of a host Operator becomes a simply definition of one or two device functors that will be used with a map-reduce strategy using the Thrust methods.

4.2 Experiments Configuration

The objective of the experiments is to validate the framework as correct, easy to use and energy efficient development platform for WattDB. The experiment has been done in two phases. The first is the implementation of the device sort, using qualitative measures to verify if the framework is easy to use. The ease of use is a very important objective of the framework because it encourages other team members to parallelize any operator, reducing development time. The second phase uses quantitative measures of the GPU sort versus CPU sort, comparing time and energy consumption. Time has been measured using a C++ timing library.

The sorting times where measured in raw sort, using a CUDA quick-sort implementation (Cederman e Tsigas 2009). The Host sort is std::sort. All tests were made in an Intel® Core™2 Duo T8300 CPU. The GPU is a GeForce Gt430. Operational system is Ubuntu Server 10.04 Kernel 2.6.32.

Energy consumption was measured using an energy measurement device designed for WattDB (Schall e Hudlet 2011), that measures overall system consumption instead of individual components. The measurement is made with a stream of real time data send over usb to the computer. 5 system configurations where measured. With no GPU installed: full idle and at 100% CPU load. With GPU: full iddle, at 100% CPU load and at 100% GPU load.

Since the query plan builder and cost prediction of WattDB was not completely implemented at the time of creation of the GPU Framework, a virtual query tree was manually

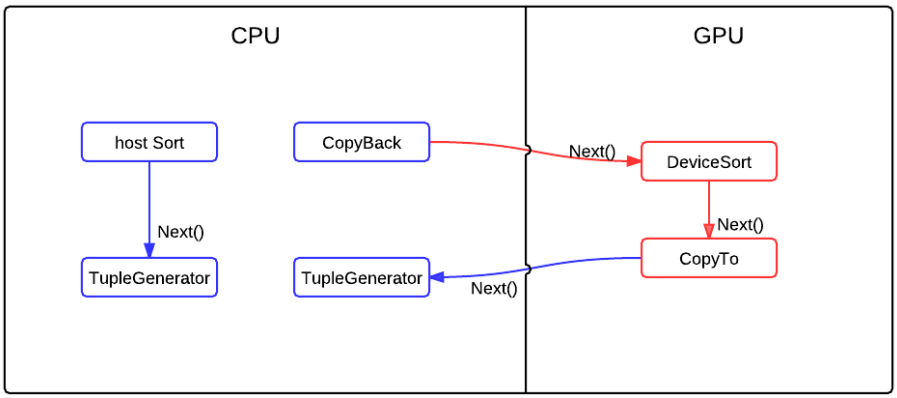

Figure 4.2: Host and Device Sorting query trees.

build to verify the correctness of the DeviceSort operator. This query plan includes a 1000 random tuple generator operator, a CopyTo operator, the DeviceSort and a CopyBack operator. The same was done with the host Sort operator and results where compared. This configuration is shown in Figure 4.2.

4.3 Experiments Results

The host Sort operator correctness was checked manually for small sets of data, and further checked with automated verification for large sets, that is, for a ascendant sort, the previous value is asserted to be smaller than the current one. No different values for the same initial sets where found comparing the result of the host Sort and DeviceSort operator. This proves that the DeviceSort is equivalent to host Sort, and since the host Sort is proven to be correct, DeviceSort is also correct. This *not-so-strict* correctness verification also proves that the copy operators are correct, that is why a validation of a single experimental DeviceOperator is so important to the overall framework correctness.

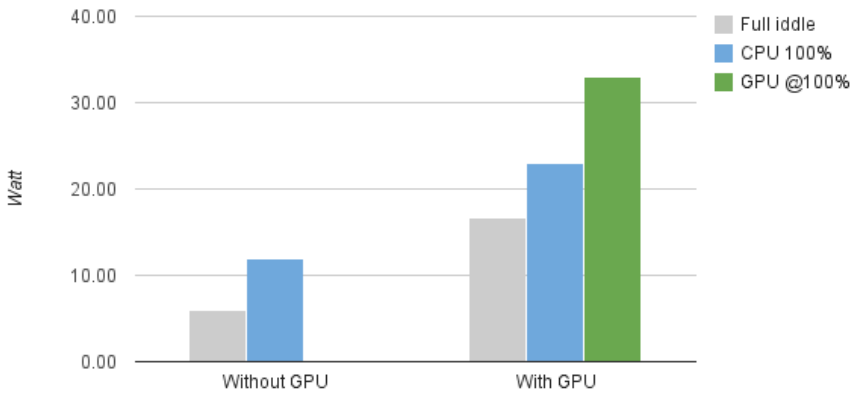

Figure 4.3: Energy consumption with and without a GPU installed in a system.

The energy efficiency measures where made comparing CPU operations with the same

GPU operations. The energy consumption was also measured with and without the GPU card installed. Figure 4.3 shows energy consumption growth when installing the GPU in the system. Installing the GPU gives a 11 Watt constant grow in energy consumption. Figure 4.3 also shows that comparing CPU at 100% and GPU at 100%, energy consumption is around 3 times higher.

When it comes to sorting times, the speed on GPU is 10 times faster as the CPU. Figure 4.4 shows the sorting times measures for different dataset sizes, including the data copy to and back from GPU. This means that the GPU is more energy efficient to sort than the GPU, because it sorts 10 times faster while using 3 times more energy, which gives $10 \div 3 \approx 3.3$ times more energy efficient than the CPU.

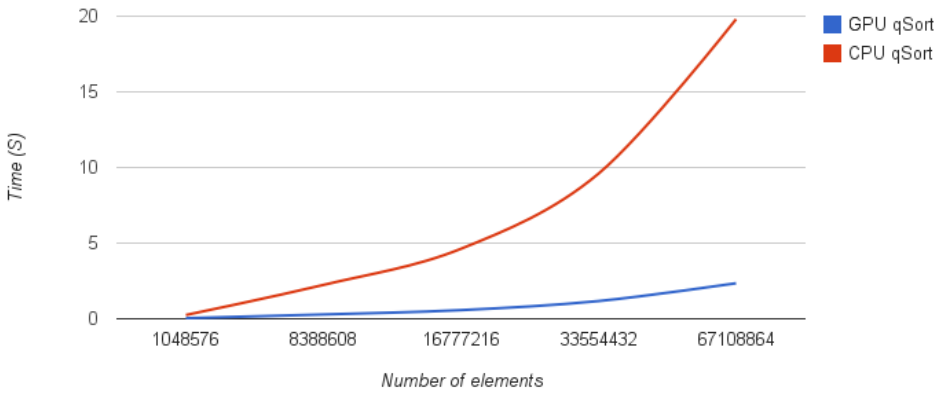

Figure 4.4: Sorting times in CPU and GPU.

The ease of use can be measured with the amount of code and architecture details that the programmer should worry about when programming in CUDA. Table 4.1 shows that the framework abstracts a lot of low level information, making the programming much easier and faster.

	GPU Framework	Raw CUDA
CUDA threads architecture		X
Memory Copy		X
CUDA object linking		X
CUDA memory allocation		X
Basic Thrust knowledge	X	

Table 4.1: Programmer responsibilities.

5 CONCLUSION

This work described a new GPU Framework architecture to develop GPU operators in WattDB using CUDA. Experiments show that the framework implementation works correctly, provides a comfortable platform to develop GPU operators and that the GPU is more energy efficient than the CPU when they are at full load.

Results show that it is possible to integrate GPGPU in existing DBMS with minimal architecture change of the existing database. The energy efficiency of the GPU is particularly interesting for the WattDB project, which needs energy-efficient nodes to provide an energy-proportional DBMS. The framework also proves that it is possible to provide an easy to use platform to develop GPU operators, taking away the lower level responsibilities from the programmer when programming CUDA.

Future work includes the development of more DeviceOperators for WattDB. Some optimizations could also be made on the copy operators, where projections could be added to copy just the columns needed by the GPU operators. In the SortOperator, for instance, the copy operators could be aware of the sorting columns, and just copy them to the GPU. When copying back, original data could be re-build using the unchanged host tuples.

More work can be done taking advantage of the the distributed architecture of WattDB allowing heterogeneous nodes configurations, equipping just some of them with GPUs, because not all database operations are highly parallelizable. This way, the GPU could be constantly at a high load, helping to achieve energy proportionality.

REFERENCES

[Andersen et al. 2009]ANDERSEN, D. G. et al. FAWN: A fast array of wimpy nodes. In: *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT: [s.n.], 2009.

[Bakkum e Skadron 2010]BAKKUM, P.; SKADRON, K. Accelerating SQL database operations on a GPU with CUDA. In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. New York, NY, USA: ACM, 2010. (GPGPU '10), p. 94–103. ISBN 978-1-60558-935-0. Available from Internet: <http://dx.doi.org/10.1145/1735688.1735706>.

[Boost C++ Library]BOOST C++ Library. Access in May 2012. Available from Internet: <http://www.boost.org/>.

[Cederman e Tsigas 2009]CEDERMAN, D.; TSIGAS, P. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *ACM Journal of Experimental Algorithmics*, v. 14, 2009. Available from Internet: <http://dblp.uni-trier.de/db/journals/jea/jea14.html#CedermanT09>.

[CUDA Platform Source Release 2012]CUDA Platform Source Release. 2012. Access in May 2012. Available from Internet: <http://developer.nvidia.com/content/cuda-platform-source-release>.

[Dusso 2012]DUSSO, P. M. *A Monitoring System for WattDB: An Energy-Proportional Databsae Cluster*. 1 2012.

[Graefe 1994]GRAEFE, G. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, v. 6, n. 1, p. 120–135, 1994.

[Härder et al. 2011]HÄRDER, T. et al. Energy efficiency is not enough, energy proportionality is needed! In: *DASFAA'11,1st Int. Workshop on FlashDB*. Springer, 2011. (LNCS, v. 6637), p. 226–239. Available from Internet: <fileadmin/publications/2011/DASFAA11_FlashDB_keynote01.pdf>.

[Hoberock e Bell 2010]HOBEROCK, J.; BELL, N. *Thrust: A Parallel Template Library*. 2010. Version 1.3.0. Available from Internet: <http://www.meganewtons.com/>.

[Hoelzle e Barroso 2009]HOELZLE, U.; BARROSO, L. A. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. [S.l.]: Morgan and Claypool Publishers, 2009. ISBN 159829556X, 9781598295566.

[Mancheril 2011]MANCHERIL, N. Gpu-based sorting in postgresql. In: *SIGMOD 2011 Conference*. [S.l.: s.n.], 2011. p. 1229–1232.

[NVIDIA 2008]NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[PGOpenCL]PGOPENCL. Access in April 2012. Available from Internet: <http://www.khronos.org/opencl/>.

[Schall e Hudlet 2011]SCHALL, D.; HUDLET, V. Wattdb: An energy-proportional cluster of wimpy nodes. In: *SIGMOD 2011 Conference*. [s.n.], 2011. p. 1229–1232. Available from Internet: <fileadmin/WattDB-2011.pdf>.

[Swan: A simple tool for porting CUDA to OpenCL 2012]SWAN: A simple tool for porting CUDA to OpenCL. 2012. Access in May 2012. Available from Internet: <http://www.multiscalelab.org/swan>.

[Szalay et al. 2010]SZALAY, A. S. et al. Low-power amdahl-balanced blades for data intensive computing. *Operating Systems Review*, v. 44, n. 1, p. 71–75, 2010.

[Tsirogiannis et al. 2010]TSIROGIANNIS, D. et al. Analyzing the energy efficiency of a database server. In: *Proceedings of the 2010 international conference on Management of data*. [s.n.], 2010. (SIGMOD '10), p. 231–242. ISBN 978-1-4503-0032-2. Available from Internet: <http://doi.acm.org/10.1145/1807167.1807194>.