A GPU Operations Framework for WattDB

Vítor Uwe Reus
vitor.reus@gmail.com

Daniel Schal
schall@cs.uni-kl.de

Databases and Information Systems Group
University of Kaiserslautern, Germany

Abstract. In the last decades, 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. WattDB is a energy proportional DBMS that aims to be energy proportional. In this work, I integrate the GPUs into WattDB, and for that, a Framework was developed to provide an efficient and easy to use platform for GPUs.

1 Introduction

In the last decades, energy consumption and production became one of the main problems of humanity for economic and environment 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 best way to solve this energy problem is to spend less energy in all areas, including DBMSs.

The first way to solve the 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 the exponential curve of energy consumption.

The second ways is to spend less energy, without harming performance. WattDB[1] is a distributed database system running on commodity hardware that tries to be energy-proportional[2] and thus, saving energy, compared to an off-the-shelf server. It is a database cluster of energy-efficient nodes. It dynamically turns nodes on and off, according to the overall cluster load.

Currently, WattDB just uses energy-efficient CPUs in the nodes. The energy efficiency of the nodes can be increased with graphical processor units (GPUs). GPUs can acquire much more energy efficiency than CPUs for heavy parallel computations.

The higher energy efficiency of GPU is possible due to their simpler architecture. Since it is a heavily parallel architecture, as we can see in Fig. 1, it devotes more silicon area to several ALUs. Less area is dedicated to control and cache, and that way its possible to achieve much more FLOPS with not too much energy.

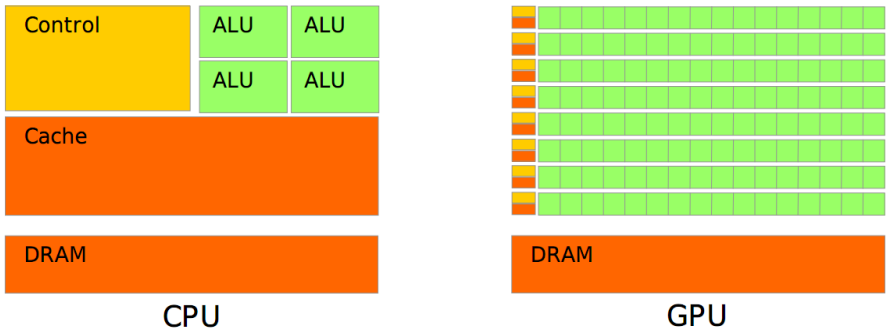


Figure 1. CPU and GPU Hardware architecture

In this work, I integrate the energy-efficiency advantage of GPU in WattDB. The problem is that it is not possible just 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 requires special designed code, and since the processor is not identical to a CPU, a special compiler is needed.

A lot of effort has been done before to use GPU in databases, such as using OpenGL[3], creating CUDA enabled SQL procedures [4], or changing specific database operators to work alone with CUDA [5]. Althought they show good results alone, these are all very different approaches, that cover small cases, and does not have a portable architecture that can be reused in any database system. To be able to use GPU on WattDB, parallel database operators where created, as well as GPU memory management operators, creating a GPU framework that can easily be reused in other systems.

1.1 Energy Proportionality

Energy proportionality is the ideal behavior of spending just the required energy amount, proportionally to the system load. Today's Servers does not achieve this, because even when iddle, the system needs energy to be on, mainly due to memory refresh intervals and HDD drive spins.

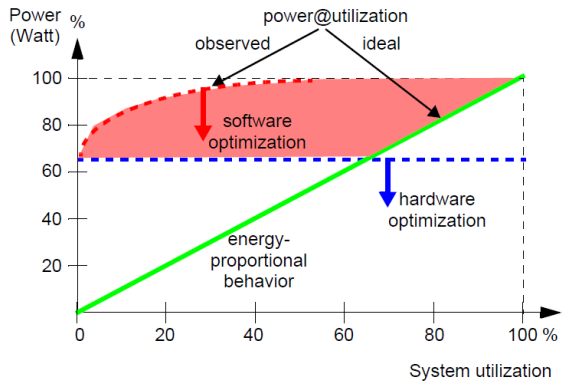


Figure 2. Energy proportionality

Fig. 2 illustrates the lack of energy proportional behavior observed in today's

hardware. Using the system at a high load is currently the only way that it spends the proportional amount of energy.

1.2 WattDB

WattDB is a distributed energy-proportional database cluster of wimpy nodes. The idea is to create an energy-proportional behavior using several nodes 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 removed, in a way that all the nodes will always have a high load. With that, an approximation of the energy proportional line is created.

The WattDB project aims to reproduce the energy-proportional behavior on a database cluster of wimpy, shared nothing computing nodes – replacing the one power DB server machine. The cluster is centered in one single node, which can attach further nodes while handling DB processing uninterrupted. In this manner, the cluster can scale up to n nodes, being able to smoothly grow and shrink dynamically. Cluster nodes can be seen as processors cores – increasing system utilization proportional to power consumption as needed. Using a single computing node we would never come close to the ideal characteristics of energy proportionality, as seen in REF2; however, WattDB will globally operate energetic proportional, assuring by software that the power consumption of the cluster will be proportional to its utilization.

1.3 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 points 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 pourpose computation on GPU, Nvidia introduced in February of 2007 the CUDA SDK, which is currently a widely adopted parallel computing architecture for GPU computing.

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[15], and Nvidia recently announced that it will open up the CUDA platform by releasing source code for the CUDA Compiler[8], 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's 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[6], because data is always being copied from one memory to the other.

To compute over the data stored in GPU, we normaly 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's why CUDA is a SIMD architecture.

A kernel is a C function defined with the special word *__global__*:

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

__global__ is a CUDA extension of C, and makes the nvcc compile this function to GPU entry-point code. This mean we can call this functino from the CPU, and then it will start running in GPU. This kind of function is called kernel.

To call this function, we first need to understand the threads hierarchy of CUDA. As we can see in Fig. 3, threads are organized in 2 levels of groups. The first level is called blocks, and they are a group of threads. The second level is a grid, which is a group of blocks.

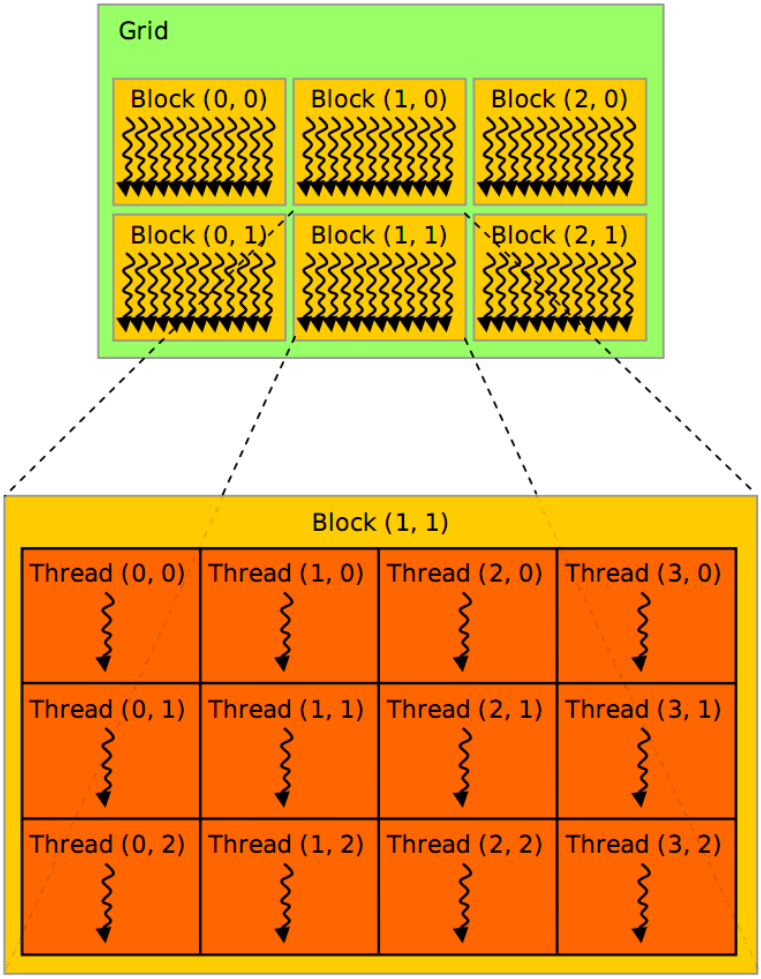


Figure 3. Thread Block and Grid.

When calling kernles, 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);
```

Note that this extensions will only work with the CUDA compiler.

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, I believe that CUDA can help WattDB to become more energy efficient.

In CUDA context, *device* is the GPU and his memory, and *host*, the CPU and his memory. I am going to follow this terminology.

2 GPU framework

Because of the major structural differences of the GPU, a framework is needed to wrap the GPU into a easy to use interface. The main differences are the separate memory of the GPU, and the programming language of GPU, in our case, *C for CUDA*. This programing language is a C with extensions for CUDA with some restrictions.

The dedicated GPU memory model makes necessary to use data copy. Data stored in GPU memory cannot be accessed by the CPU, and vice-versa, so the only way to make data available in both processing units is doing data copies.

The data copies needs to be done in a easy way to the programmer, to provide a good GPGPU framework for WattDB. CUDA does not have any automated copy facilities. Any time that data from CPU is needed in the GPU, a *cudaMemcpy* function of type *cudaMemcpyHostToDevice* needs to be called by the CPU. The framework will take care of this memory management, using two new databases operations: *CopyTo* and *CopyBack*.

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.

The programmer can now write his GPU database operators supposing the data is already in the correct memory. WattDB uses a vectorized Volcano Query Evaluation System[13] implementation. It has volcano-style operators interface. This operators will be extended to allow GPU operators. GPU operators will automatically use GPU memory, and the CPU operators, the CPU memory.

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 Fig. 4.

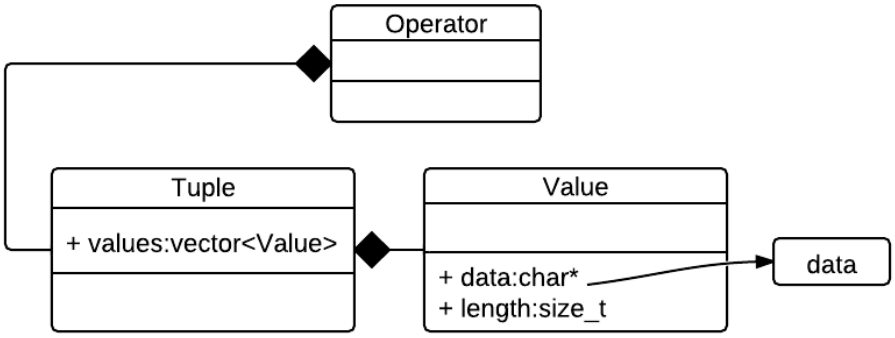


Figure 4. Current Tuple, Value and data implementation in WattDB

Since now we use the GPU memory, *DeviceOperators*, *DeviceTuples* and *DeviceValues* are added. 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.

The CPU operators will continue to exists, the new device classes will just be added to WattDB, as in Fig. 5.

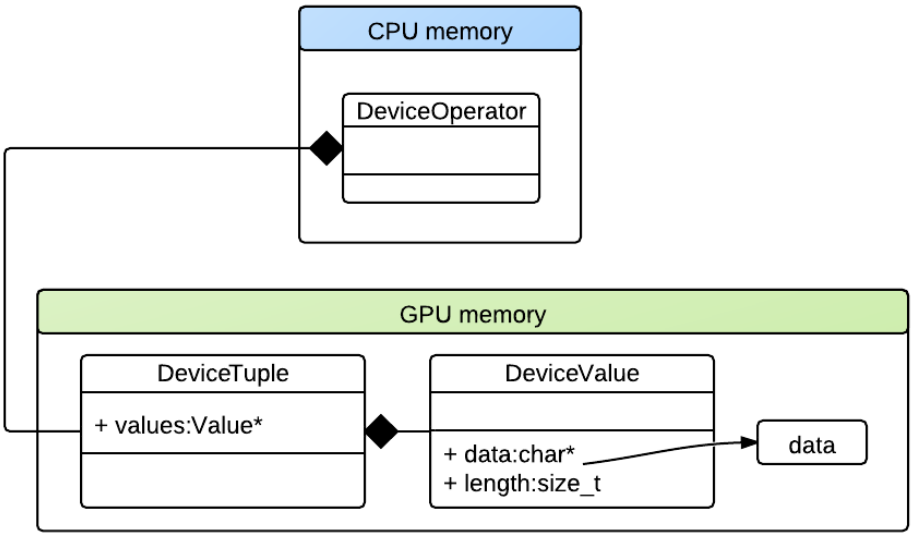


Figure 5. New device classes

Note that the DeviceOperators will be allocated on CPU, since it will be the bridge between CPU and GPU. It is alocated on CPU, because the query plan needs to have acces to the operators. DeviceOperators will hold their tuples in GPU, and is responsible to allocate them.

2.1 Copy operators and DeviceTuples

The copy operators are responsible to copy data between the CPU and GPU. This operators defines the plan mode, alternating between host and device mode, as seen in Fig. 3. 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.

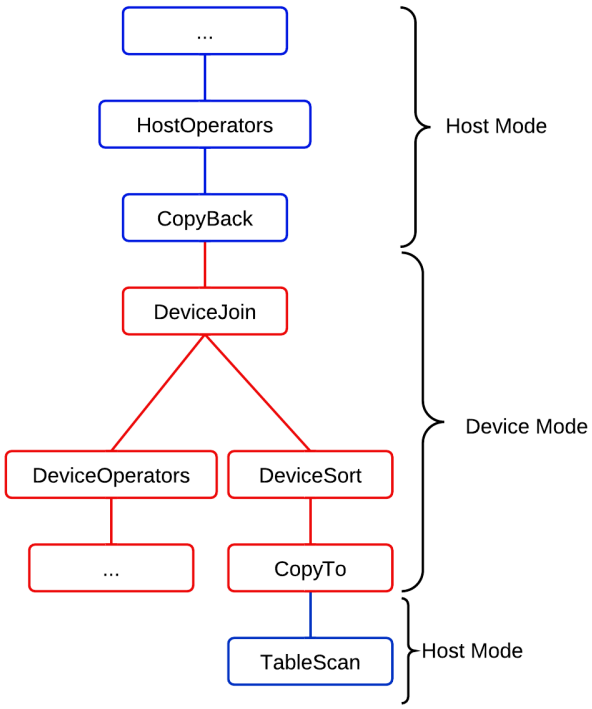


Figure 6. Query plan modes

To allow the execution of the ordinary 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.

In wattDB, the query plans are created with the ResultOperator objects. ResultOperator can hold their child operators, and fetches the data from them in form of Tuples using the next() method, as in Fig. 7 The new DeviceOperator, DeviceTuple and DeviceValues where needed because not all original code is compatible with CUDA, and the DeviceOperator needs to allocate the Tuples in GPU memory.

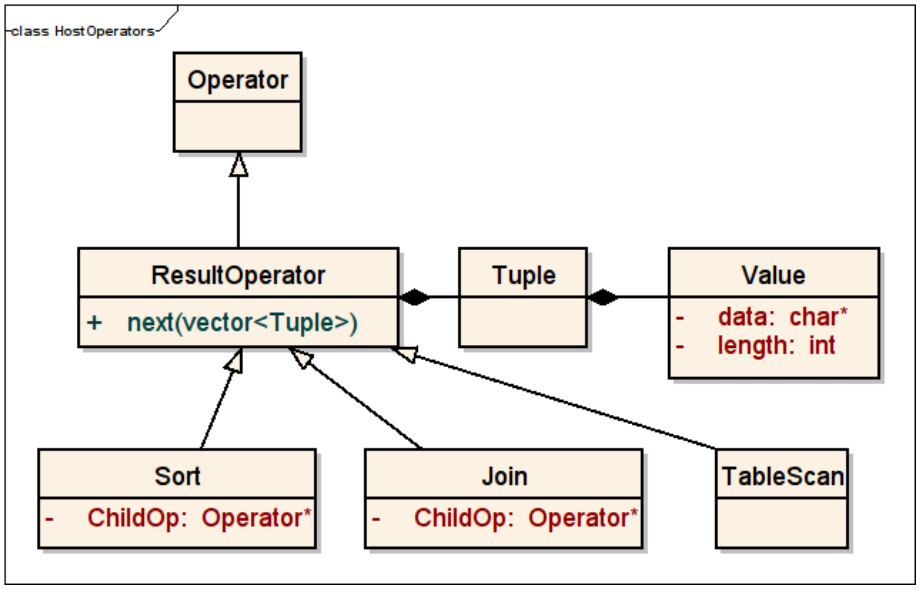


Figure 7. Operators, Tuple and Value

Both kind of tuples are present in the database, and the complete design can be seen in Fig. 8. The green classes are the ones inserted to be used by the framework.

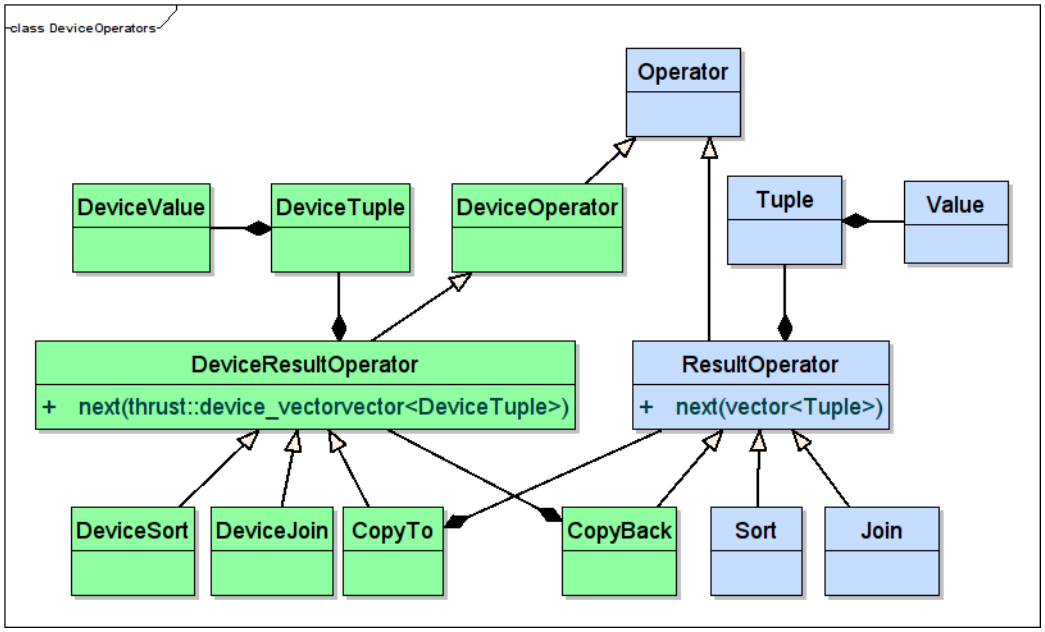


Figure 8. Adding DeviceTuples, DeviceValues and DeviceOperators to WattDB

It's interesting to note that CopyTo is a DeviceOperator, but CopyBack is an ordinary host Operator. This is not obvious in first look, but it's that way because CopyBack returns ordinary host Tuples, thus, needs the next interface of an ordinary host Operator. We can see this also in Fig 6. where next interface can be seen as the lines that connect the operators.

The new CUDA Fermi architecture present on new generation Nvidia cards allows the straightforward translation of ordinary Tuples to DeviceTuples. This architecture allows us to use free and malloc functions inside device code. In older

architectures, all memory management should be done on host side, using *cuda_malloc* and *cuda_free* functions. That 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 side memory management.

2.2 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 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.

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 Fig. 9.

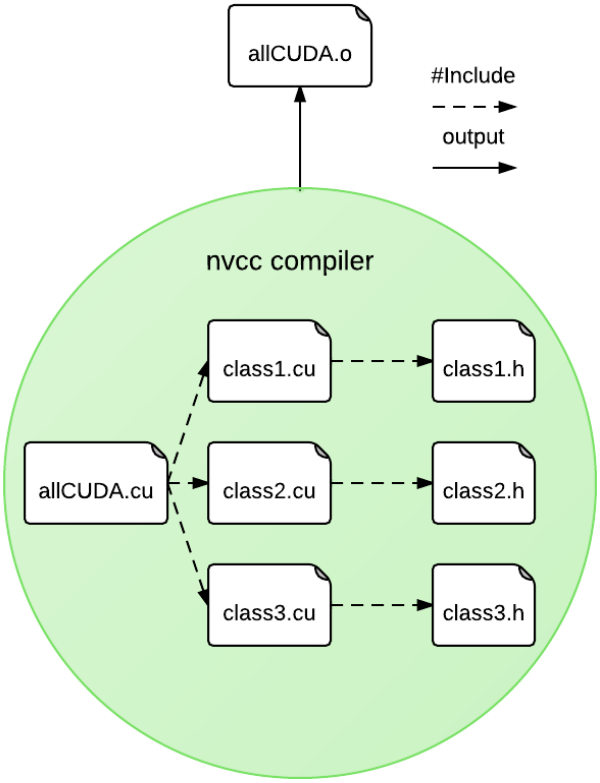


Figure 9: Using one file to be able to call device functions between different files.

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 avaliable in CPU.

```
class DeviceResultOperator {
 public:
   #ifdef __CUDACC__
    virtual bool next(
thrust::device_vector<wattdb::driver::access::DeviceTuple>&
outDT ,
thrust::device_vector<wattdb::driver::access::DeviceTuple*>&
outDTR
    ) = 0;
   #endif
};
```

The next example show how to hide the boost::spirit[14] 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 Fig. 10.

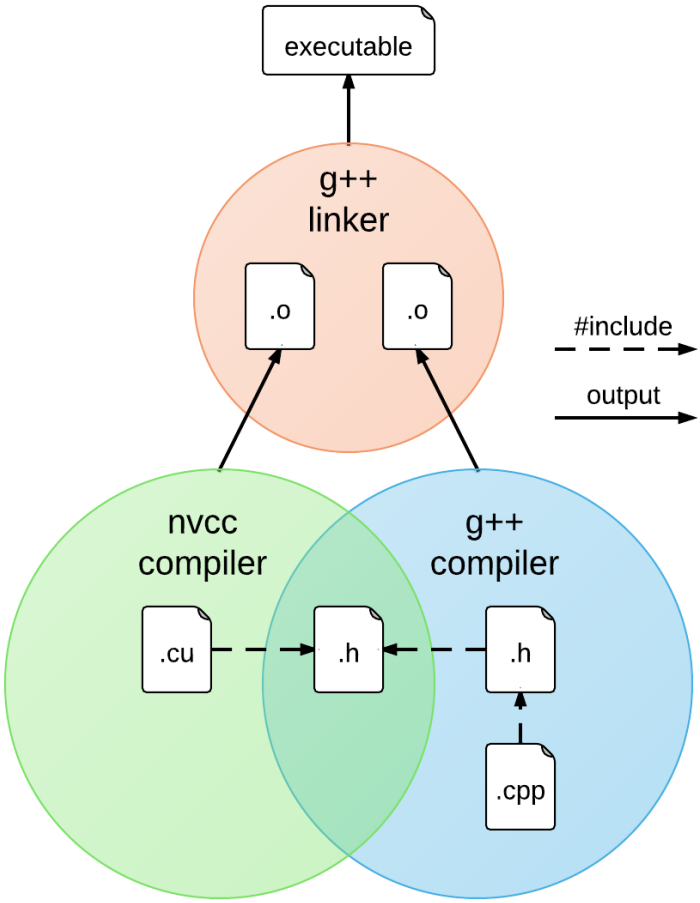


Figure 10: Using two compilers in a build process to integrate CUDA in an existing application.

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 Fig. 11.

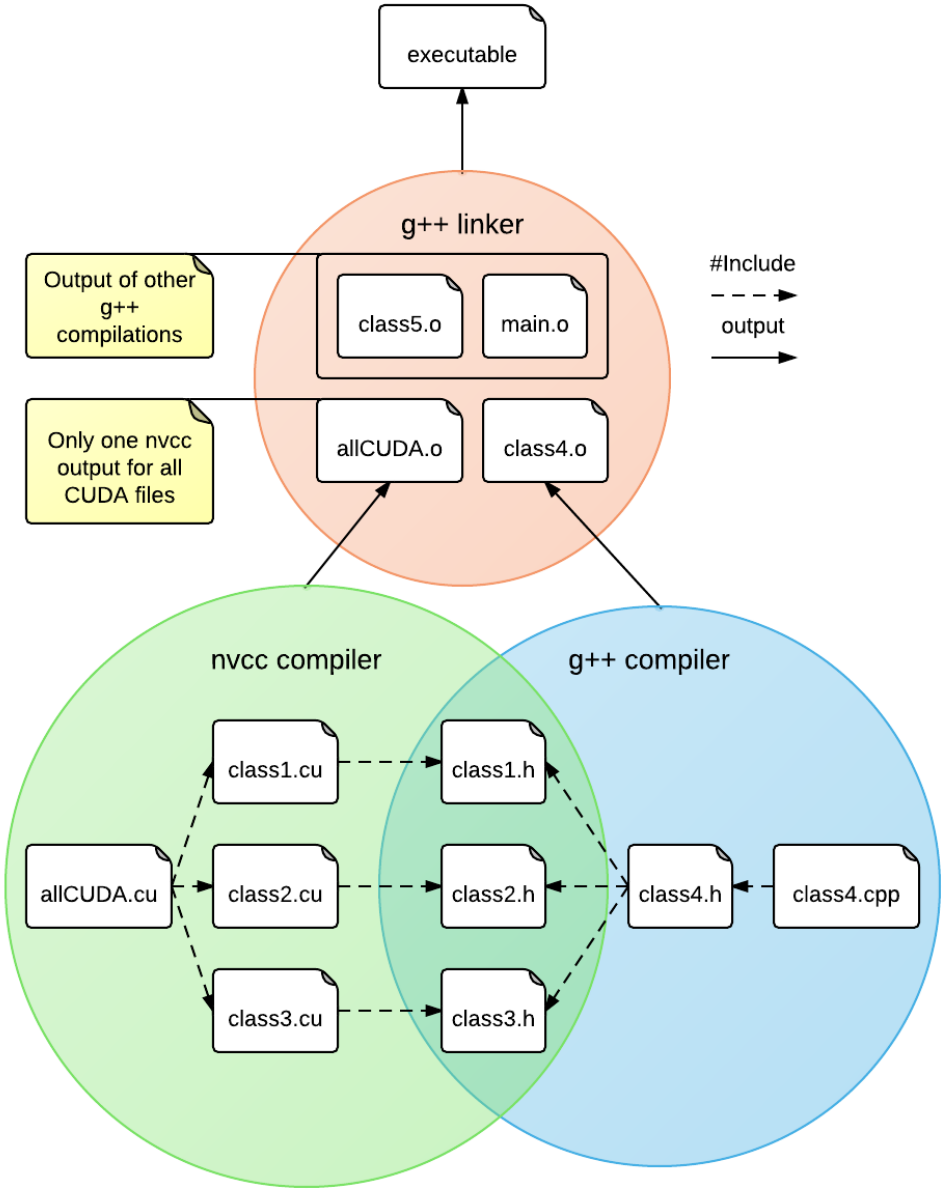


Figure 11: Resulting process to integrate CUDA in WattDB

2.3 DeviceOperators

With the Copy and C for CUDA limitations solved, it is now possible to start using the DeviceOperators. This Operators are like normal operators, but holds the Device tuples. DeviceOperators needs to be rewritten to use the parallel advantages of the GPU. This can be done manually translating already existing CPU operators. This will be 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 DeviceOperator does not aims to replace CPU operators, both of them are

available in WattDB. A query plan can be build with DeviceOperator or CPU operators. GPU operators will normaly be more energy efficient with huge sets of data, so WattDB will have the option to use both of them.

To decide between CPU and Device operators, energy consumption of the GPU should be taken into account in the cost prediction calculation of a query tree. GPU operations 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.

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 sequences of DeviceOperators in the device mode query plan, and reusing data that is already in the GPU memory.

Not all operations are highly parallelizable. Just by installing a GPU in a system makes it spend more energy. An heterogeneous configuration of the nodes of WattDB could be usefull, 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[7].

3 Device Sort operator

To test and verify if the framework is easy to use, and provides an efficient GPU plataform to WattDB, a GPU sort operator was implemented. I have implemented the device sort operator to prove that my framwork works. Althought we have to rewrite every operator to deal correctly with the DeviceTuples, this translation is stright-forward for all the operators, because the tuple structure of device tuples are very similar to host tuples.

I used the thrust[11] library to quickly implement the device sort operator, with the thrust::sort method. Thrust is a C++ template library for CUDA based on the Standard Template Library. It provides some usefull methods like thrust::transform (map), thrust::reduce or thrust::sort. It also has a device_vector implementation, with is equal to the std::vector, but using device memory. The DeviceTuples are stored using thrust::device_vector.

The DeviceSort operator can deal with different types of DeviceValue types of data, thanks to the use of a comparator functor. For each type of data, a different comparator is instantiated, allowing a smaller and clearer code.

Since using large datasets in GPU in WattDB is not possible now, I have measured the sorting times in raw sort, using a quick-sort implementation[16]. The energy consumption a Intel(R) Core(TM)2 Duo CPU T8300, when equipped with a GeForce Gt430 is the following:

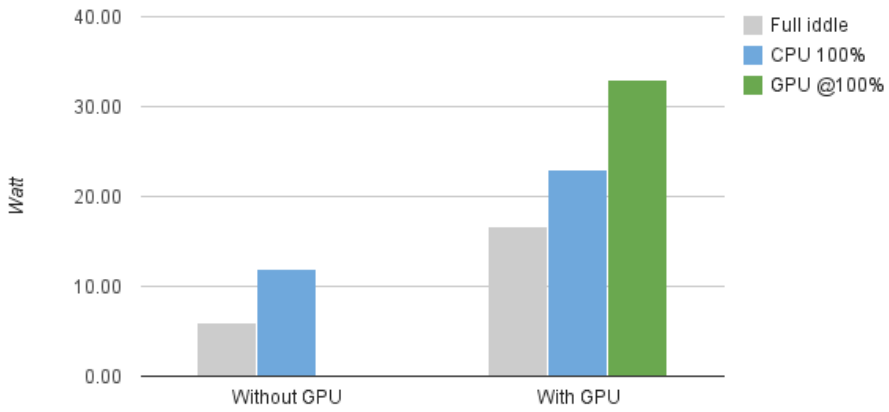
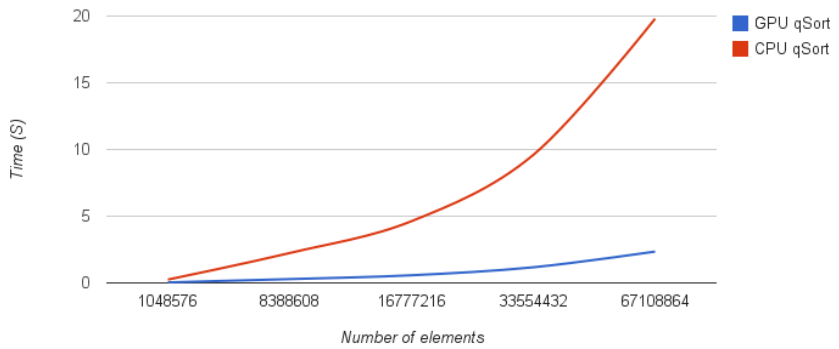


Figure 12. Energy consumption with and without a GPU installed in a system.

In Fig. 12, its possible to see that comparing CPU at 100% and GPU at 100%, energy consumption is around 3 times higher, but the speed of sorting on GPU is 10 times faster, as it can be seen in Fig. 13.



13. Sorting times in CPU and GPU

Than means that the energy efficiency of sorting in GPU is still around 3 times higher than CPU.

4 Conclusion

After developing this framework and testing it implementing a sort operator, i have concluded that the frameworks really creates a comfortable platform to for GPU operators, solving all the major problems when programming CUDA.

Because of the interface of volcano tuples, the translation between CPU and Device tuples are straightforward. The programmer does not need to worry about copying the data between the two memories since the copy operators already do this.

The distributed architecture of WattDB allow us to have heterogeneous configurations of the nodes, equipping just some of them with GPUs, since not all operations are highly parallelizable. Turning the GPU-enabled nodes can help achieve energy proportionality since the GPU is just highly energy efficient at high load.

At least, sorting on GPU shows that the GPU can decrease sorting time more times than the energy increase, making it more energy efficient.

[1] WattDB: An Energy-Proportional Cluster of Wimpy Nodes
[2] Energy Eciency is not Enough, Energy Proportionality is Needed!
[3] GPU-based Sorting in PostgreSQL, by Naju Mancheril of. Carnegie Mellon University
[4] PGOpenCL
[5] Accelerating SQL Database Operations on a GPU with CUDA, Peter Bakkum and Kevin Skadron Department of Computer Science University of Virginia, Charlottesville, VA 22904
[6] Nvidia Cuda programming guide v.4.0
[7] Pedro Dusso: A monitoring system for WattDB
[8] http://developer.nvidia.com/content/cuda-platform-source-release
[11] Thrust: A Parallel Template Library, Jared Hoberock and Nathan Bell
[13] G. Graefe. Volcano - an extensible and parallel query evaluation system
[14] Boost C++ Librarie - http://www.boost.org/
[15] Swan: http://www.multiscalelab.org/swan
[16]Daniel Cederman and Philippas Tsigas: GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors