# Data Parallel Acceleration of Decision Support Queries Using Cell/BE and GPUs

Pedro Trancoso*, Despo Othonos, and Artemakis Artemiou
Department of Computer Science
University of Cyprus, Cyprus
{pedro,cs01od,cs01aa2}@cs.ucy.ac.cy

## ABSTRACT

Decision Support System (DSS) workloads are known to be one of the most time-consuming database workloads that processes large data sets. Traditionally, DSS queries have been accelerated using large-scale multiprocessor.

The topic addressed in this work is to analyze the benefits of using high-performance/low-cost processors such as the GPUs and the Cell/BE to accelerate DSS query execution. In order to overcome the programming effort of developing code for different architectures, in this work we explore the use of a platform, Rapidmind, which offers the possibility of executing the same program on both Cell/BE and GPUs. To achieve this goal we propose data-parallel versions of the original database scan and join algorithms.

In our experimental results we compare the execution of three queries from the standard DSS benchmark TPC-H on two systems with two different GPU models, a system with the Cell/BE processor, and a system with dual quad-core Xeon processors. The results show that parallelism can be well exploited by the GPUs. The speedup values observed were up to 21× compared to a single processor system.

## Categories and Subject Descriptors

C.4 [**Computer Systems and Organization**]: Performance of Systems; C.1.2 [**Computer Systems and Organization**]: Processor Architectures—*Multiple Data Stream Architectures (Multiprocessors)*; H.2 [**Information Systems**]: Database Management

## General Terms

Algorithms, Performance

## Keywords

Cell/BE, Data-parallel model, Decision Support System, GPU, Performance Evaluation, Rapidmind

*P. Trancoso is a member of HiPEAC (EU FP7 program).


*CF'09,* May 18–20, 2009, Ischia, Italy.


## 1. INTRODUCTION

In the recent years we have observed a shift in the microprocessor design from the monolithic full-featured single-core processor to more simple cores arranged in multi-core processors. While this has helped in increasing performance at adequate levels of power consumption, exploiting this performance is a challenge. Namely, programming such processors is a much more difficult task than the traditional ones. As the number of cores in the processor increases, this issue becomes more serious.

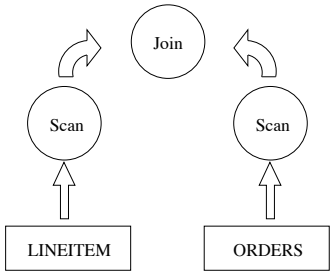
One driving force in producing processors that achieve increasing computation power is the computer gaming market. These are some of the most demanding applications for general-purpose systems. Consequently, many of the most advanced multi-core processors have been produced for that goal. Examples of such processors include the Graphics Processor Units (GPU) and the Cell/BE processor that can be found in the Sony Playstation 3. Because the main task of these processors is to execute certain classes of applications much faster than traditional hardware, we call them *accelerators*. Nevertheless, in addition to games, these processors can also be used for general-purpose applications. As such, a large interest has emerged in the community in order to exploit the computational power of these processors for non-graphical/gaming applications.

One obstacle in successfully taming this available power is the programmability of such systems. These large-scale multi-core processors usually support the Single Instruction Multiple Data (SIMD) or Single Program Multiple Data (SPMD) model of execution and are able to effectively improve the performance of data-parallel applications. Consequently, some new data-parallel platforms have been recently developed. Example of two such platforms include NVIDIA's CUDA [19] and Rapidmind [22, 23]. While the former was developed to allow the programming of general-purpose applications on the G80 GPU architecture, the latter allows for the development of applications for different multi-core processors such as the GPU, Cell/BE, or general-purpose multi-cores.

Technology improvements and application demands go hand-in-hand. In addition to games, we are observing an increasing demand for more accurate results and more efficient processing of larger data sets. Traditionally, one of the most demanding workloads are the database applications. In particular, the category of Decision Support Systems (DSS) database applications combines the processing of large data sets along with the computation of statistical information extracted from the data.

```
select
  l_shipdate
from
  orders,
  lineitem
where
  o_orderkey = l_orderkey
  and l_commitdate < l_receivedate
```

(a)




(b)

**Figure 1: TPC-H Query 12: (a) SQL query code and (b) Query plan.**

Therefore, our goal is to understand how the demanding DSS applications may benefit from the existing multi-core processors and how the performance on these systems compares to the performance on more traditional general-purpose multi-core systems. With this work we try to determine the benefit of using GPUs and Cell/BE processors in order to accelerate the execution of DSS queries. With this analysis we target to understand not only the effort required and benefit that can be achieved in today's systems but also what to expect in future large-scale and heterogeneous multi-core systems.

Another goal of this work is to explore the use of a common model to execute the same program on different platforms. Although it is well known that in order to fully exploit the features of a certain architecture it is necessary to develop the code dedicated to that architecture, with the constant changes in architecture and different products available, it is important to study if it is possible to use a single model and still achieve good results. As such, for this work we used the Rapidmind platform and the corresponding data-parallel model for the development of the code for the database algorithms. We therefore propose new data-parallel versions of the basic database algorithms.

To achieve our goal we analyze the performance of the basic database algorithms parallelized with Rapidmind for the Cell/BE and GPU systems and OpenMP for general-purpose multi-core systems. The algorithms were used for the execution of standard representative DSS queries taken from the TPC-H benchmark suite [28]. The experimental results show that the parallelism offered by the GPUs can be exploited for accelerating the algorithms of DSS queries. The larger the data set, for the data sets studied in this work, result in the best speedup compared to a modern single processor chip. It is obvious from the results that parallelism plays a significant role as the GPU with a larger number of stream processors always achieves a better speedup. The largest improvement was observed for the Data-Parallel Nested-Loop Join algorithm for a 2-join query where the best GPU achieved a speedup of more than 20× compared to the single processor system. Speedup is also observed for more efficient algorithms such as the Hash-Join. Compared to traditional general-purpose multi-core systems, the GPUs, given the larger degree of parallelism, may offer a better potential for performance improvement for DSS query execution.

This paper is organized as follows: Section 2 presents the database algorithms and their data-parallel implementation for Rapidmind, Section 3 describes the experimental setup, including the target architectures and application while Section 4 discusses the experimental results. Section 5 presents the relevant related work and Section 6 the conclusions to our work.

# 2. DATA-PARALLEL DATABASE QUERIES

## 2.1 Query Processing

In a *DataBase Management System* (DBMS), a query is submitted to the system in a high-level language (*SQL*), which describes the operations to be performed but does not impose any specifics on their implementation or execution. The DBMS can perform four basic operations: *Scan*, *Join*, *Order*, and *Aggregate*. For each of these operation the DBMS offers a set of algorithms, for example, a *Scan* may be performed using the *Sequential Scan* or the *Index Scan* algorithm. The responsibility of the DBMS *Query Optimizer* is to decide which algorithm to use for each operation and in which order to schedule the different operations so as to achieve the shortest execution time. The result of the optimization process is a *Query Plan* which can be represented as a tree, where the leaf nodes are *Scan* operations for a data *Table*. An example of a SQL query and its corresponding Query Plan is shown in Figure 1.

As mentioned before, the objective of this work is to analyze the potential of the emerging multi-core processors such as the Cell/BE or the GPU, using a single platform to develop and implement the algorithms which may then execute on the different systems. In order to achieve this we propose new data-parallel implementations of the basic database algorithms. In the future, the proposed algorithms should be plugged into a DBMS in order for the optimizer to also consider them as valid implementations for the supported operations.

## 2.2 Rapidmind Data-Parallel Platform

The Rapidming Development Platform has roots in the Sh platform [18]. Sh started as a research project at the University of Waterloo. Some years later, it became a commercial product and the Rapidmind Platform was released. The Rapidmind Platform is an extension to standard C++ and is designed in such a way that allows the developer

to create massively parallel applications, or extend existing applications to run on high-performance processors such as GPUs and the Cell/BE processor [12]. In the case of general-purpose processing, the Rapidmind Platform provides a computational model that can easily be mapped to computational resources in a system, including the GPU and the Cell/BE processor. Under the Rapidmind Platform, the multi-core processors operate based on the *Data-Parallel* or *Stream Programming Model*. Owens defines stream as an ordered set of data of the same data type [21]. This data type can vary from simple integers or floating-point numbers, to points or triangles. Computations can be performed on streams with the use of special *stream programs*.

The computational model of the Rapidmind Platform introduces new data types that can be used for undertaking the declaration and use of streams in the case of GPU or Cell/BE general-purpose programming. Besides that, the portion of source code that is targeted for execution on the GPU or Cell/BE, must be included in a special block of code declared as a *Rapidmind program*. This block of code must make use of the Rapidmind API. The execution proceeds as follows: the C++ code of the application is executed on the CPU and the Rapidmind *program* having as inputs and output the corresponding streams, is executed on the GPU or Cell/BE according to the selected *back-end*.

## 2.3 Mapping DSS Algorithms to Rapidmind

### 2.3.1 Mapping the Data

As mentioned before, the data to be processed by the multi-core under Rapidmind has to be formated as a *data streams*. *Data streams* are a specific data type from the programming environment, which logically resemble *data arrays* from regular high-level languages.

While DBMS systems may store the data in different ways, for this work the data is stored column-wise, *i.e.* all values of a particular attribute belonging to different records, are stored in the same *data stream*. As such, if we consider a Table $A$ with 3 attributes: *attr1*, *attr2*, and *attr3*; A *data stream* for each of these attributes is created in order to store the database data. Figure 2 depicts the logical and physical data organization for Table $A$.

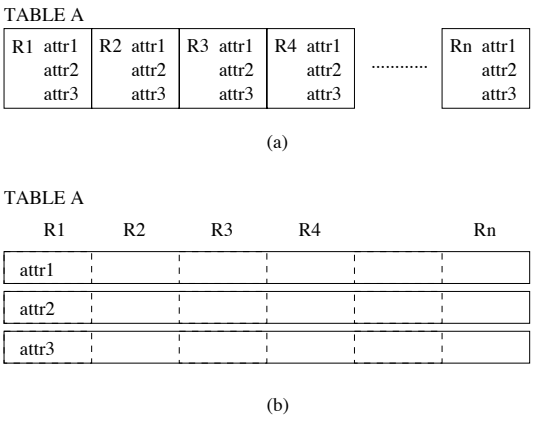



**Figure 2: Table $A$: (a) logical and (b) physical data organization.**

### 2.3.2 Mapping the Scan Operation: Data-Parallel Sequential Scan (DPSS)

As mentioned before, there are two basic implementations for the Scan operation: the Sequential and the Index Scan. Given the *streaming* or *data-parallel programming model* that is supported by Rapidmind, the most appropriate implementation is the *Sequential Scan*. In this algorithm, all the records are traversed and the record's attributes are checked against a certain condition. If the condition is satisfied then the record belongs to the results, otherwise it is filtered out. The condition may be a simple attribute comparison or a complex boolean function.

The mapping of this operation is done by implementing in the *Rapidmind program* the condition to be tested and by sending to this function as input parameters, all attribute data streams that are used to evaluate the scan condition. The result of this operation is also a data stream which has '-1' in the position of the records that are filtered out and '1' in the position of records that belong to the results. An example of a simple scan is shown in Figure 3-(a). In this example, the elements in the *attribute1* stream are checked using a condition that is satisfied if the elements are either 'Y' or 'Z'. If the condition is satisfied then the corresponding element in the *result* stream will have the value '1' or otherwise '-1'.

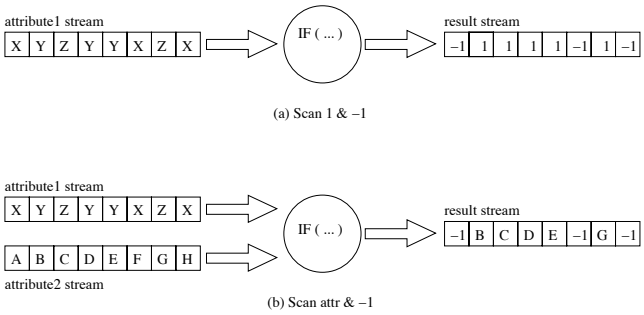



**Figure 3: Data-Parallel Sequential Scan with results in: (a) 1 & -1 format or (b) attr & -1 format.**

After this operation, the results may be either materialized, *i.e.* the records corresponding to the '1' positions may be copied to the results data structure, or they may stay as-is in order to be passed to the next operation. Notice that if the result of the Scan operation is to be used by another operation, e.g. an Aggregate or a Join operation, then the result stream will have '-1' in the position of the elements filtered out and the value of the attribute needed for the next operation in the valid record positions. This could be a key attribute value to be used by a Join operation. An example of this is shown in Figure 3-(b).

### 2.3.3 Mapping the Nested-Loop-Join Operation: Data-Parallel Nested-Loop Join (DPNLJ)

In the case of the data-parallel Join, the loops are executed by exploiting the streaming model. Assuming that we want to join Table $A$ with key $ka$ and Table $B$ with foreign key $ka$, the Rapidmind program that compares the key value against the foreign key value, is called with, for example, the key $A.ka$ and the stream of data composed of all foreign keys $B.ka$. Notice that adopting the *Nested Loop Join* as the basis for our implementation is not a limitation as this

algorithm is more efficient than the rest in some conditions and that is why it is widely available in all DBMS. It is interesting to offer a new, more efficient implementation of the simple algorithms for the *Query Optimizer* to consider it for the *Query Plan* execution.

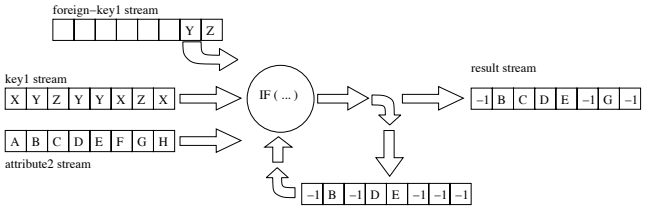



**Figure 4: Data-Parallel Nested-Loop Join**

Usually, a Join operation is performed as part of a larger Query Plan with operations before producing the inputs to the Join and operations after consuming the Join results. In these cases, the way the execution proceeds is relevant. As mentioned before, the Join is performed by checking a certain key value from one table against all the key values of the other table. As such, for every value of the first table the Join operation produces a subset of the results. The checks may be performed in a loop having the intermediate results passed as input to the next iteration in a *Feedback* way. Only when the operation has completed, the final result is passed to the next operation. Figure 4 presents a diagram for the DPNLJ algorithm. In this example, for each step, one element from the *foreign-key1* stream is picked and compared against all elements from the *key1* stream. If a match is found then the corresponding element from *attribute2* stream is copied to the *result* stream. Notice that the *result* stream is also fed into the join operation as to accumulate all the partial results. Also, the number of steps until the algorithm completes is equal to the number of elements in the *foreign-key1* stream.

### 2.3.4 Mapping the Hash-Join Operation: Data-Parallel Hash-Join (DPHJ)

Mapping the Hash-Join algorithm is not as trivial as the Nested-Loop Join. In the original Hash-Join implementation, when joining two table, a hash table is built using the tuples from one of the tables and then this hash table is probed using the tuples of the other table. The problem with this implementation is that the accesses to the hash table are random and therefore this algorithm can not be excuted in a data-parallel way. Instead, we need to use an implementation where two hash tables are built using the same hash function. After the hash-tables have been built, the result of the join can be determined by comparing the hash entries one-by-one. This operation can then easily be mapped to the data-parallel model. The only issue that needs to be addressed is that each entry of the hash table contains a pointer to an overflow list. In Rapidmind it is not possible to express this. Therefore, we implemented the overflow list by replicating the hash entry vector $n$ times. The major drawback compared to the original overflow list is the non-efficient use of the memory space required for the overflow. With this proposed solution, the Hash Join can be executed in a data-parallel way by passing to the parallel function $n_A$ hash vectors representing the hash table for table $A$ and $n_B$ hash vectors representing the hash table for table $B$. One problem faced with the Rapidmind implementation was the limited number of parameters used in the parallel function call. As such, we propose a solution that involves calling multiple time the *Rapidmind Program*, each time sending one hash vector from $A$ and one from $B$. Figure 5 depicts the Hash-Join algorithm as described above. In this example, for each step, each element of *Hash(foreign-key1)* stream is compared to the corresponding element of *Hash(key1)-0* stream. If a match is found then the corresponding element in the *attribute2* stream is copied to the *result* stream. As in the previous case, the *result* stream is also passed as input to the join operation. Nevertheless, In this case, the number of steps in order to complete the execution of the algorithm is equal to the size of the hash overflow ($n$).

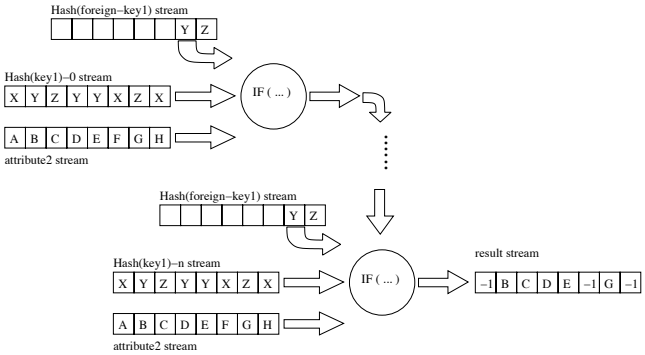



**Figure 5: Data-Parallel Hash Join**

## 3. EXPERIMENTAL SETUP

For this work we tested several systems. The *Baseline* system for our work is a workstation configured with an Intel Core 2 Duo Processor (E6420) clocked at 2.13GHz with 2GB of DDR2 RAM with maximum bandwidth 333MHz. Although the processor in this system has two cores, the baseline system is just a single core system. Therefore for all our experiments we have only used one of the cores of the system. Our target was to study the performance on the emerging multi-core architectures. As such we test the proposed algorithms using Rapidmind on one Cell/BE and two GPU based systems.

The Sony-Toshiba-IBM Cell/BE processor is composed of one host processor, the PowerPC Processor Element (PPE) and 8 Synergistic Processing Elements (SPEs). These cores are simple and do not support out-of-order execution. Also, each SPE has its own local user-managed memory sratchpad known as the Local Store (LS) of 256KB. For this work we used the Cell/BE processor that can be found in the Sony Playstation3 (*RM-Cell*). Unlike the original Cell/BE, this processor has only 6 SPEs available to the user and its main memory is limited to 256MB.

Regarding the GPUs, we have used two different graphics cards plugged into the same system. The host system in this case is the Baseline system previously described. Both GPUs are from the NVIDIA 8000 series but their characteristics are quite different. We have used a NVIDIA 8500GT (*RM-8500GT*) with a total memory capacity of 512MB, and a NVIDIA 8800GTS (*RM-8800GTS*)with 640MB of main memory. The former has 16 stream processors clocked at

Table 1: Specifications of Tested Systems.

| | Baseline | RM-Cell | RM-8500GT | RM-8800GTS | OMP-MC8 |
|---|---|---|---|---|---|
| System | Generic | Sony PS3 | NVIDIA 8500GT | NVIDIA 8800GTS | IBM x3650 |
| Cores | 1 Dual | 1+6 | 16 | 96 | 2 Quad |
| Model | Intel E6420 | PPE+SPE | Streaming | Streaming | Intel E5320 |
| Freq | 2.13GHz | 3.2GHz | 900MHz | 1.2GHz | 1.8GHz |
| Cache | 4MB | 512KB | 512MB | 640MB | 2x4MB |
| Mem | 2GB | 256MB | 2GB | 2GB | 48GB |

900MHz while the latter has 96 stream processors clocked at 1.2GHz.

Finally, for the sake of comparison with general-purpose multi-core systems, we have used a IBM x3650 system configured with two Intel Quad-core Xeon chips (E5320) clocked at 1.8GHz (*OMP-MC8*). In Table 1 we summarize the main hardware specifications for all tested systems.

The Operating System (OS) used for the Baseline and GPU experiments was Microsoft Windows XP with Microsoft DirectX 9.0c Application Programming Environment (API) for graphics rendering. For the Cell/BE and the multi-core systems we used Linux Operating System.

For this work we focused on the execution of the basic database algorithms and their parallelization. As such, the queries analyzed in this work were not executed on a full DBMS but instead were implemented as programs that executed the operations determined by the queries. Notice that the results of the queries were validated. For the Baseline implementation of the algorithms, we used the C programming language. All the CPU-based implementations were compiled using the GNU GCC compiler with the full optimization level "-O3". For the Cell/BE and GPU implementation, we have used the Developer Edition of the Rapidmind Platform [22] (version 2.1.0) through the C++ programming language. The code was compiled using the Microsoft Visual C++ 2005 Language and the Rapidmind Libraries. The C++ part of the code was optimized by using the "/O2" optimization provided by the Visual C++ compiler. The Rapidmind code was optimized through the use of special calls and techniques provided by the platform. Regarding the general-purpose multi-core system, we have used OpenMP compiler directives in order to parallelize the relevant loops. For the purposes of this work we use only a single degree of parallelism with this machine which is equal to the total number of cores available. Therefore the *OMP-MC8* multi-core executes the applications using eight threads.

```
select
  sum(l_extendedprice*l_discount) as revenue
from
  lineitem
where
  l_shipdate >= date '[DATE]'
  and l_shipdate < date '[DATE]' + interval '1' year
  and l_discount between [DISCOUNT] - 0.01 and
                         [DISCOUNT] + 0.01
  and l_quantity < [QUANTITY];
```

**Figure 6: TPC-H Q6. The parameters used were: DATE=2005, DISCOUNT=10, and QUANTITY=1000000.**

```
select
  sum(case when o_orderpriority ='1-URGENT'
  or o_orderpriority ='2-HIGH'
    then 1
    else 0 end) as high_line_count
from
  orders, lineitem
where
  o_orderkey = l_orderkey
  and l_shipmode in ('[SHIPMODE1]', '[SHIPMODE2]')
  and l_commitdate < l_receiptdate
  and l_shipdate < l_commitdate
  and l_receiptdate >= date '[DATE]'
  and l_receiptdate < date '[DATE]' + interval '1' year;
```

**Figure 7: Simplified version of TPC-H Q12. The parameters used were: SHIPMODE1=1, SHIPMODE2=2, and DATE=2009.**

```
select
  l_orderkey,
from
  customer, orders, lineitem
where
  c_mktsegment = '[SEGMENT]'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date '[DATE]'
  and l_shipdate > date '[DATE]';
```

**Figure 8: Simplified version of TPC-H Q3. The parameters used were: DATE=2007, SEGMENT=3.**

For timing the execution of the queries, the methodology included the execution of the computation for ten times, then discarding the minimum and maximum values and then calculating the average of the remaining values.

As for the workload, we have focused on the queries from the standard TPC-H [28] benchmark. The TPC benchmarks are widely used today as they provide objective performance metrics of computer systems. TPC-H is a Decision Support System (DSS) benchmark. The queries employed by this benchmark are relevant to industry decision support operations. The benchmark is used against large volumes of data and it is comprised of queries of high degree of complexity. The queries selected for this work were the following: *Q6*, *Q12*, and *Q3*. The reason for selecting these three queries is that they have increasing number of Join operations, therefore representing queries with different degree of complexity. The SQL code for the three queries is presented in Figures 6, 7, and 8. These queries are simplified versions of the original queries that are specified in the TPC-H Decision Support

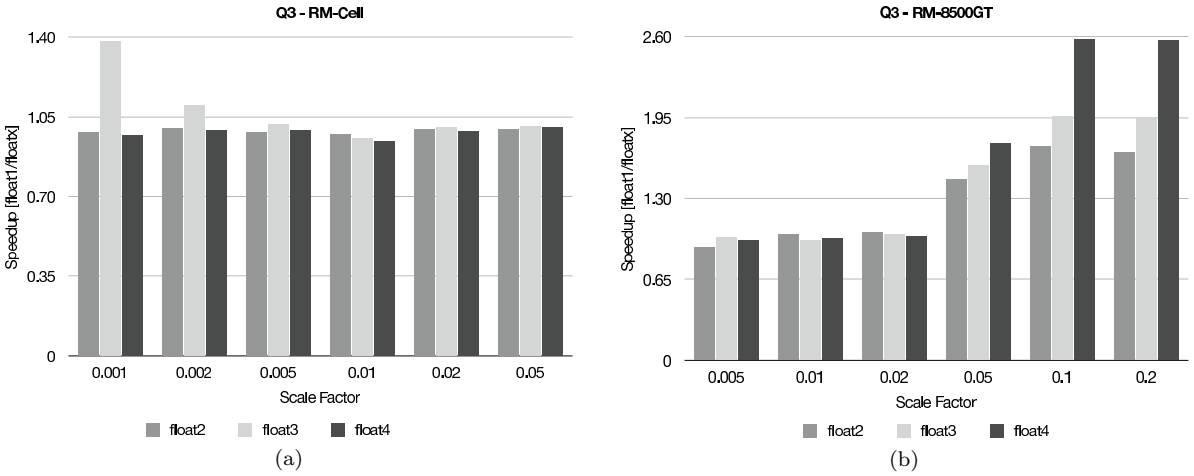



**Figure 9: Different Data Type for Q3: (a) RM-Cell and (b) RM-8500GT.**

Benchmark. Notice that the changes do not affect most of the execution as they cover the core operations.

The input data sets used were generated according to the benchmark specifications. The number of records in a table is related to a *scale factor* (SF). In Table 2 we present the different scale factors that were used in the experiments along with the size of the different tables, expressed in number of records, and the approximate size in MByte for the three tables.

**Table 2: Different data set sizes.**

| *Scale Factor* | *ORDERS* | *CUSTOMER* | *LINEITEM* | Approx Size [MB] |
|---|---|---|---|---|
| 0.01 | 15000 | 1500 | 60175 | 8.7 |
| 0.02 | 30000 | 3000 | 120515 | 17.7 |
| 0.05 | 75000 | 7500 | 299814 | 44.2 |
| 0.1 | 150000 | 15000 | 600572 | 89.3 |
| 0.2 | 300000 | 30000 | 1199969 | 179.6 |
| 0.5 | 750000 | 75000 | 2999671 | 454.0 |

Notice that in this work we report the *speedup* of the proposed algorithms compared to the execution of the same algorithm on a traditional CPU, in this case the serial execution on the Baseline system.

# 4. EXPERIMENTAL RESULTS

## 4.1 Data Type Analysis

As mentioned before, for the data-parallel implementation of the algorithms in Rapidmind, the platform supports *streams* of different data types. In particular, Rapidmind offers the possibility of using data types that are compounds of multiple original types. For example, the programmer may store the data in a stream of elements of type *Value3f*. This type corresponds to a structure with three variables of type float. The use of these data types helps to enable certain optimizations when compiling Rapidmind code. For example, the compiler will use a 4-way SIMD operation when the code includes an operation on data of type Value4f. In addition, because of the similarities of these compound data types with the graphics native data types, operations of *Value4f* data types will also exploit the benefits of the graphics processor operations. To illustrate the fact that there is a performance impact when choosing different data types to execute the same code, we execute the same code, in this case query *Q3* using the following data types: *Value1f*, *Value2f*, *Value3f*, and *Value4f* for both the *RM-Cell* and *RM-8500GT* systems. Figure 9 shows the speedup for the different data types, compared to the *Value1f* implementation.

From Figure 9 we can derive two facts. First, for the Cell/BE the performance seems stable across the different data types with the exception regarding the smaller data set sizes where the best performance is obtained for the *Value3f* data type. Second, for the GPU, the performance seems to depend considerably on the data type. As such, we prefer to use the *Value4f* for our algorithms as it is with this data type that it is possible to achieve the highest performance.

As a result from this study we implemented the Cell/BE codes using the *Value3f* data type and the GPU codes using the *Value4f* data type.

## 4.2 Data-Parallel Sequential Scan (DPSS): Q6

In this Section we present the results for the execution of TPC-H Query 6 (Q6) using the Data-Parallel Sequential Scan algorithm (DPSS). Figure 11 shows the speedup obtained by the different systems compared with the baseline system. These results are shown for six different scale factor data set sizes (0.01, 0.02, 0.05, 0.1, 0.2, and 0.5) and for each one of these we show four bars representing the different systems tested (*RM-Cell*, *RM-8500GT*, *RM-8800GTS*, and *OMP-MC8*).

From the results it is possible to observe that the GPUs considerably outperform the Cell/BE for the execution of this query. Nevertheless, the speedup values for both types of accelerator are quite small. It is only for the larger data set sizes that the 8800GTS is able to perform better than the baseline, achieving a maximum speedup of 1.35 for the 0.5 scale factor.

The observed results may be justified by the fact that DPSS is a highly parallel algorithm and therefore, the more cores there are available, the better the performance. Nevertheless, the parallelism suffers from the data transfer over-

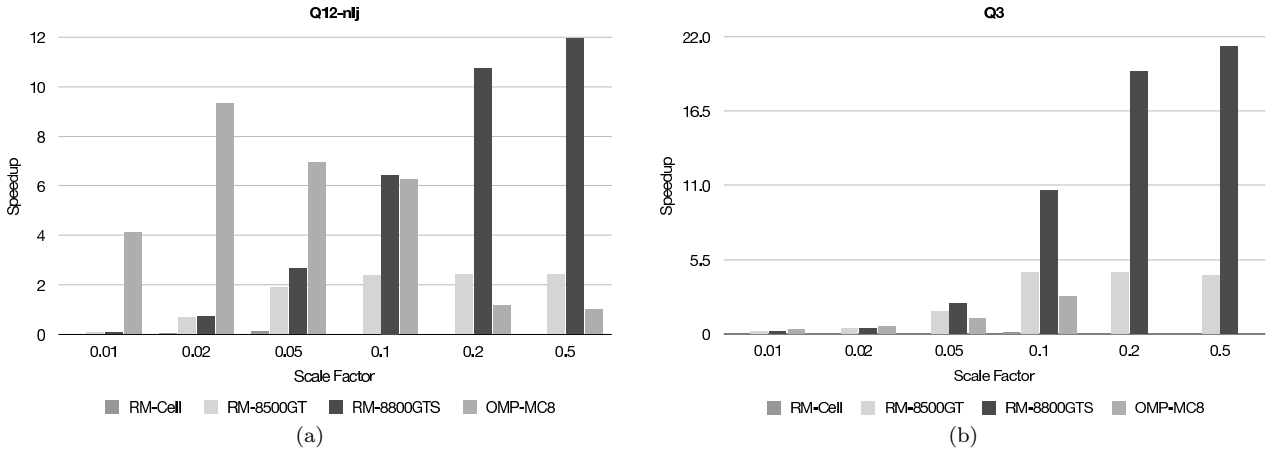



**Figure 10: DPNLJ: (a) Q12, and (b) Q3.**

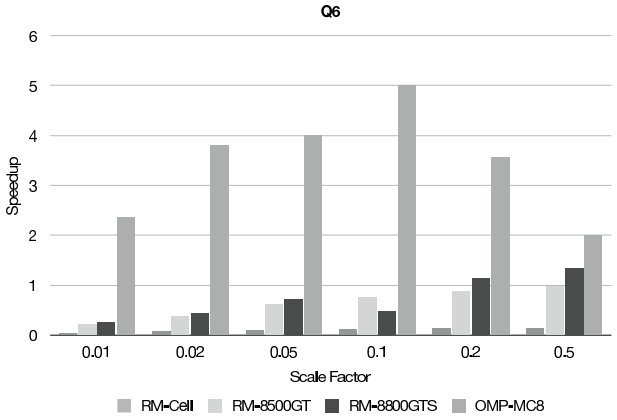



**Figure 11: DPSS - Q6.**

heads, which in this case plays a relevant role as the operation performed in the algorithm are very few. Consequently, the gains in parallelizing the computation can not overcome the penalty of the data transfers. In this case, this effect is more serious for the Cell/BE processor. While there are some studies that suggest some techniques such as overlapping data fetches with computation for the Cell/BE, the Rapidmind platform trades-off easy programmability for a control of some hardware features. Finally, we need to perform a more exhaustive study in order to fully understand the reason for the low performance results.

Notice that the general-purpose multi-core processor achieves a better speedup than the accelerators, for this case. A justification to this effect is that the data transfer overhead is not that large as for the accelerators. Nevertheless, it is relevant to notice that for scale factors larger than 0.1, the speedup actually starts to decrease. We attribute this trend to the fact that for larger data sets, the data is not able to fit in the local caches and therefore the execution is not as efficient.

## 4.3 Data-Parallel Nested-Loop Join (DPNLJ): Q12 and Q3

In this section we analyze the performance of the Data-Parallel Nested-Loop Join algorithm (DPNLJ) which is used in our case to implement the join operation in both TPC-H *Q12* and *Q3*.

Regarding the results for *Q12* shown in Figure 10-(a), as seen also in the previous results, the Cell/BE platform does not seem to offer any benefit for this query. On the one hand the speedup values are very small (less than 0.15 for the best case) and on the other hand its memory limitations do not allow for the execution of the query for a data set with scale factor larger than 0.05. The situation for the GPUs is different as it is possible to observe that both models show a speedup compared to the baseline system. While the *8500GT* level-off the speedup at approximately 2.4, the *8800GTS* reaches a speedup of 12 for the larger input data set (scale factor of 0.5).

Notice that for the Cell/BE we do not show results for data set sizes larger than the scale factor of 0.05 due to memory limitations for the execution of the algorithm under the Rapidmind platform.

It is interesting that, for this case, the traditional general-purpose multi-core is able to offer a speedup compared to the baseline system, but like in the previous case, this speedup decreases significantly for larger input data set sizes. In addition, the GPU shows a larger speedup than the general-purpose multi-core for data set sizes larger than the scale factor of 0.1.

As for the results for *Q3*, as shown in Figure 10-(b), they follow the results for *Q12* in a very similar way. The only difference is that the speedup values are much larger for *Q3*. This is due to the fact that while *Q12* has one join operation, *Q3* has two join operations. Given that the accelerator algorithm is much more efficient than the baseline one, combining a larger number of operations results in increased benefit in the overall final result. It is interesting to notice that, for this case, the *8800* is able to achieve a speedup larger than 21 for the larger input data set. This is a much larger speedup than any other tested system, justifying therefore the use of accelerators for this type of application.

Notice that there are some results missing in Figure 10-(a). As in the previous cases for large input sets the Cell/BE implementation can not execute. In addition, due to time limitations we are not able to show the results for the larger data set sizes for the general-purpose multi-core.

## 4.4 Data-Parallel Hash-Join (DPHJ): Q12

Finally we present the results for the Data-Parallel Hash Join (DPHJ) that is used for the implementation of TPC-H Query *Q12*. It is important to notice that the Hash-Join implementation is much more efficient than the original Nested-Loop Join (up to 659× better for the baseline system for the data sets studied). Even though it is much more efficient, it is not necessary that it may be applicable on every occasion. In some cases, it is not possible to build the required hash table. Therefore, it is important to study both Nested Loop and Hash Join implementations, propose optimizations to both and then let the *Query optimizer* to decide on which one to use for the execution of the query.

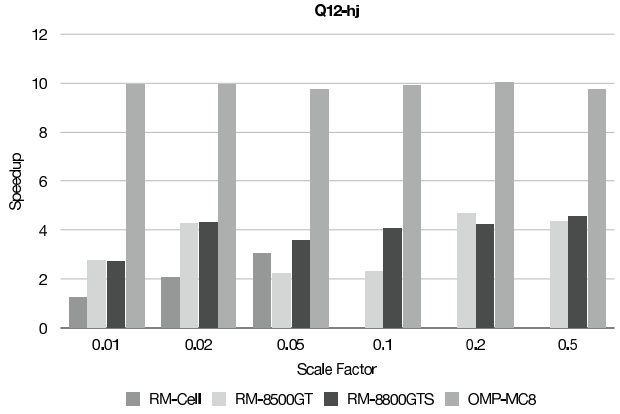



**Figure 12: DPHJ - Q12.**

Figure 12 shows us the speedup results for the different systems compared to the execution on the baseline system. Given that this algorithm is simpler, resulting in fewer calls to the kernel functions, the performance of the different accelerators is comparable, unlike in the previously observed cases. In particular, the speedup values for the accelerators range from 2 to 4. In this case and for the data input size with the scale factor of 0.05 the Cell/BE achieves a speedup of 3× that is close to the best obtained speedup obtained by the GPUs. The GPUs' performance seems more stable than the Cell/BE's and also there seems not to be a significant difference between the *8500* and the *8800*. The speedup values are up to 4 compared to the baseline, for the larger data set sizes.

It is relevant to notice that for this case, the general-purpose multi-core seems to take the advantage over the accelerators as it achieves a speedup value of 10, stable across the different input data set sizes. Nevertheless, we need to mention that the results reported here for the multi-core regard the parallelized version of the original hash-join algorithm, with a single hash table as opposed to the data-parallel hash-join algorithm used for the accelerators. An analysis of the results comparing the two hash-join implementations on the multi-core system shows that the original hash-join performs more than 2× better than the data-parallel one. This is therefore a case where in order to obtain speedup with the accelerators we need to revert to a not-so-efficient implementation of the same operation. If then we compared the same algorithm on both accelerators and multi-core systems we would be able to observe that the performance of the accelerators would be slightly better than the multi-core system.

## 4.5 Summary

Overall we can conclude that the large-degree of parallelism offered by the GPUs can be exploited for accelerating the algorithms of DSS queries. The larger the data set, for the data sets studied in this work, result in the best speedup compared to a modern single processor chip. It is obvious from the results that parallelism plays a significant role as the GPU with a larger number of stream processors almost always achieves a better speedup. This is also a determinant factor to the performance of the Cell/BE processor, which has only six processing elements, not to reach the performance of the GPUs.

The available local memory also seems to play a relevant role in the performance. The speedup observed for the general-purpose multi-core system decreases for the cases where the working set with larger data set sizes does not fit on the on-chip caches. A similar effect also influences the performance of the Cell/BE system which shows to have a limiting memory hierarchy for this type of application. The small local storage and system main memory are the two limiting factors for the Cell/BE.

Also, the speedups depend on the degree of complexity of the query. The more difficult the query is, the better the speedup. The largest improvement was observed for the Nested-Loop Join algorithm for *Q3*, a 2-join query where the best GPU achieved a speedup of more than 20× compared to the baseline system. It is relevant to notice that speedup is also observed for more efficient algorithms such as the Hash-Join.

Finally, the use of the data-parallel model for both Cell/BE and GPU seems to fit the latter but not so much the former. The high-level programming API does not allow the programmer to exploit the architectural features of the underlying system. This seems to affect mostly the Cell/BE as its performance is very dependent on how the programmer makes the management of the memory transfers, computation, and synchronization. It is relevant to notice though that if for the Cell/BE performance we considered the PPE processor as the baseline, therefore reducing the effects of the different architecture from the Intel baseline, we would be able to see speedup values up to 2.7× for the execution of Q3 using the DPNLJ algorithm. Given that it uses 6 SPE processors, its efficiency is up to 45%. Just for illustration purposes, the 8800GTS GPU achieves for the same query a maximum speedup of 21.3× using 96 streaming processors which translates into an efficiency of 22.2%. As such we could argue that the performance of the Cell/BE is being affected by the small number of available parallel resources.

Overall, compared to traditional general-purpose multi-core systems, the accelerators may offer a better potential for performance improvement for DSS query execution, given their larger degree of parallelism.

# 5. RELATED WORK

The wide availability, low cost, large degree of parallelism, and high raw compute power were the determining factors to trigger the interest of the community in using the new multi-core processors such as the Cell/BE and the GPUs for general-purpose applications.

In the recent years there is much work done in the area of GPGPU. An example of efficient GPU usage for general-purpose computing is the field of data mining. Based on the

streaming nature of the GPU it became possible to design fast GPU implementations for traditional data mining operators. An example of such implementation is the nearest neighbors algorithm [13, 15].

In addition, several other general-purpose applications have been mapped to the GPU such as: dense matrix multiply [17], linear algebra operations [16], sparse matrix solvers for conjugate gradient and multigrid [3], Fast Fourier Transform [14, 29], and Bioinformatics [4, 25]. A study exploring the capabilities of the GPU for GPGPU was also presented in [27].

Recent research has also focused on the use of the GPU for database applications [1, 8, 10, 26]. Bandi *et al.* [1] proposed the utilization of the GPU hardware architecture for accelerating spatial selections and joins. Govindaraju *et al.* [9], presented an algorithm for database and data mining computations using the GPU. This algorithm was used to accelerate the computation of equi-join and non-equi-join queries in databases and numerical statistic queries on data streams.

Sorting is also an operation of major relevance in a database system. In many cases, the execution time of database operations is fully dependent on the amount of time needed for the execution of the sorting algorithms. Greb and Zachmann [11] researched parallel sorting on stream processing architectures. The concept of their work was based on adaptive bitonic sorting [2]. Govindaraju *et al.* [7], presented *GPUTeraSort*, an algorithm used for sorting billions of records using the GPU. Their algorithm was executed against *nsort* [20], a commercial CPU-based sorting algorithm with high I/O performance. Their experimental results indicated that the overall performance of *GPUTeraSort* with a mid-range GPU is comparable to that of a high-end dual Xeon processor. Govindaraju *et al.* [8], have conducted research toward the acceleration of conjunctive selections, aggregations and semi-linear queries.

Regarding the Cell/BE, several researchers have ported different applications on top of Cell/BE-based system in order to improve the performance. Among others, Williams *et. al.* [30] present the potential of the Cell/BE processor for scientific computing. Servat *et. al.* [24] show how the Cell/BE can be used for Drug Design. And closer related to our study is the work by Gedik *et. al.* [6] which covers the topic of the implementation of join operations on the Cell/BE.

Also focusing on database workloads, Cieslewicz and Ross [5] have studied the database execution on modern hardware, in terms of cache usage, branch prediction, and multithreading, among others.

While much work has been done in this field, to the best of our knowledge, this work is the first that compares the performance of the Cell/BE and GPUs for the execution of real database queries under a common platform, Rapidmind.

# 6. CONCLUSIONS

In this work we proposed new data-parallel implementations of basic scan and join database algorithms in order to support complex Decision Support System (DSS) queries. The objective was to evaluate the applicability of emerging multi-core processors such as the Cell/BE and GPUs to accelerate traditionally complex and demanding DSS workloads. As the programmability of such architecture is a challenge, we decided to use a common platform, Rapidmind, to allow for the development of a single program and execution on diverse systems. In addition to evaluating the Cell/BE and GPU accelerators, we compared those results to a 8-way general-purpose multi-core system.

To test the proposed data-parallel algorithms we executed three representative queries from the standard TPC-H benchmark using real TPC-H data ranging from a scale factor of 0.01 up to 0.5. The experimental results showed that the large degree of parallelism offered by the GPUs allows for the efficient acceleration of the DSS queries. This is specially true for larger data sets and more complex queries. The speedup achieved for the data-parallel algorithms was up to 21× for a 2-join query. The benefits were observed, at a smaller degree, even across other more efficient algorithms such as the Hash Join. Other than for the Hash Join, the Cell/BE did not seem to be able to offer the same performance as the GPUs. The small degree of parallelism, memory limitations of the system, and lack of control from using the high-level data-parallel platform for its programming seem to be the dominant factors for these results. Also important is the fact that the speedup obtained for the GPU was almost all times better than the one obtained with the general-purpose multi-core system.

Overall, the results obtained are very encouraging and indicative that complex database queries may be accelerated with affordable hardware. As future work we are currently looking into the possibility of integrating these new algorithms into a real Database Management System. Also, we are pursuing a comparison between the performance of the execution of the high-level data-parallel code with code dedicated for the GPUs and Cell/BE, written in CUDA and Cell SDK, respectively.

# 7. ACKNOWLEDGMENTS

We would like to thank the rest of the CASPER research group for their comments on the work, and Rapidmind for providing the SDK for the development of this work.

# 8. REFERENCES

[1] N. Bandi, C. Sun, D. Agrawal, and A. E. Abbadi. Hardware Acceleration in Commercial Databases: A Case Study of Spatial Operations. Technical report, Computer Science Department, University of California, Santa Barbara, 2004.

[2] G. Bilardi and A. Nicolau. Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-memory Machines. *SIAM J. Comput.*, 18(2):216–228, April 1989.

[3] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[4] M. Charalambous, P. Trancoso, and A. Stamatakis. Initial Experiences Porting a Bioinformatics Application to a Graphics Processor. In *Proceedings of the 10th Panhellenic Conference on Informatics (PCI 2005)*, pages 415–425, November 2005.

[5] J. Cieslewicz and K. A. Ross. Database optimizations for modern hardware. *Proceedings of the IEEE*, 96(5):863–878, 2008.

[6] B. Gedik, P. Yu, and R. Bordawekar. Executing stream joins on the cell processor. In *VLDB '07:*

*Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 363–374, 2007.

[7] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High performance graphics o-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA, 2006. ACM.

[8] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2004. ACM.

[9] N. K. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha. A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors. Technical report, UNC, 2005.

[10] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 611–622, New York, NY, USA, 2005. ACM.

[11] A. Greb and G. Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In *Proceedings of the 20th IEEE IPDPS*, 2006.

[12] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[13] J. D. Hall and J. C. Hart. GPU Acceleration of Iterative Clustering. In *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors*, August 2004.

[14] T. Jansen, B. v. Rymon-Lipinski, N. Hanssen, and E. Keeve. Fourier Volume Rendering on the GPU using a Split-stream FFT. In *Proceedings of Vision, Modelling and Visualization*, 2004.

[15] E. Kenneth, I. Hoff, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast Computation of Generalized Voronoi Diagrams using Graphics Hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques (SIGGRAPH 1999)*, pages 277–286, 1999.

[16] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 908–916, New York, NY, USA, 2003. ACM.

[17] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55, New York, NY, USA, 2001. ACM.

[18] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[19] NVIDIA. NVIDIA CUDA Technology. http://developer.nvidia.com/object/cuda.html.

[20] Ordinal. Nsort: Fast Parallel Sorting. http://www.ordinal.com/.

[21] J. Owens. Streaming architectures and technology trends. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 9, New York, NY, USA, 2005. ACM.

[22] Rapidmind. Writing Applications for the GPU Using the RapidMind Development Platform. http://www.rapidmind.net.

[23] Rapidmind. Rapidmind Platform. http://www.rapidmind.net, 2007.

[24] H. Servat, C. Gonzalez, X. Aguilar, D. Cabrera, and D. Jimenez. Drug design on the cell broadband engine. In *PACT '07: Proceedings of the 16th INternational Conference on Parallel Architecture and Compilation Techniques*, page 425, 2007.

[25] A. Stamatakis. An Efficient Program for Phylogenetic Inference Using Simulated Annealing. In *Proceedings of IPDPS2005*, 2005.

[26] C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration for spatial selections and joins. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 455–466, New York, NY, USA, 2003. ACM.

[27] P. Trancoso and M. Charalambous. Exploring Graphics Processor Performance for General-Purpose Applications. In *Proceedings of the Euromicro Symposium on Digital System Design, Architectures, Methods and Tools (DSD 2005)*, pages 306–313, August 2005.

[28] Transaction Processing Council. *TPC Benchmark H (Decision Support) Standard Specification Revision 2.6.1.* June 2006.

[29] J. Wang, T.-T. Wong, P.-A. Heng, and C.-S. Leung. Discrete Wavelet Transform on GPU. http://www.cse.cuhk.edu.hk/ ttwong/demo/dwtgpu/ dwtgpu.html.

[30] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pages 9–20, 2006.