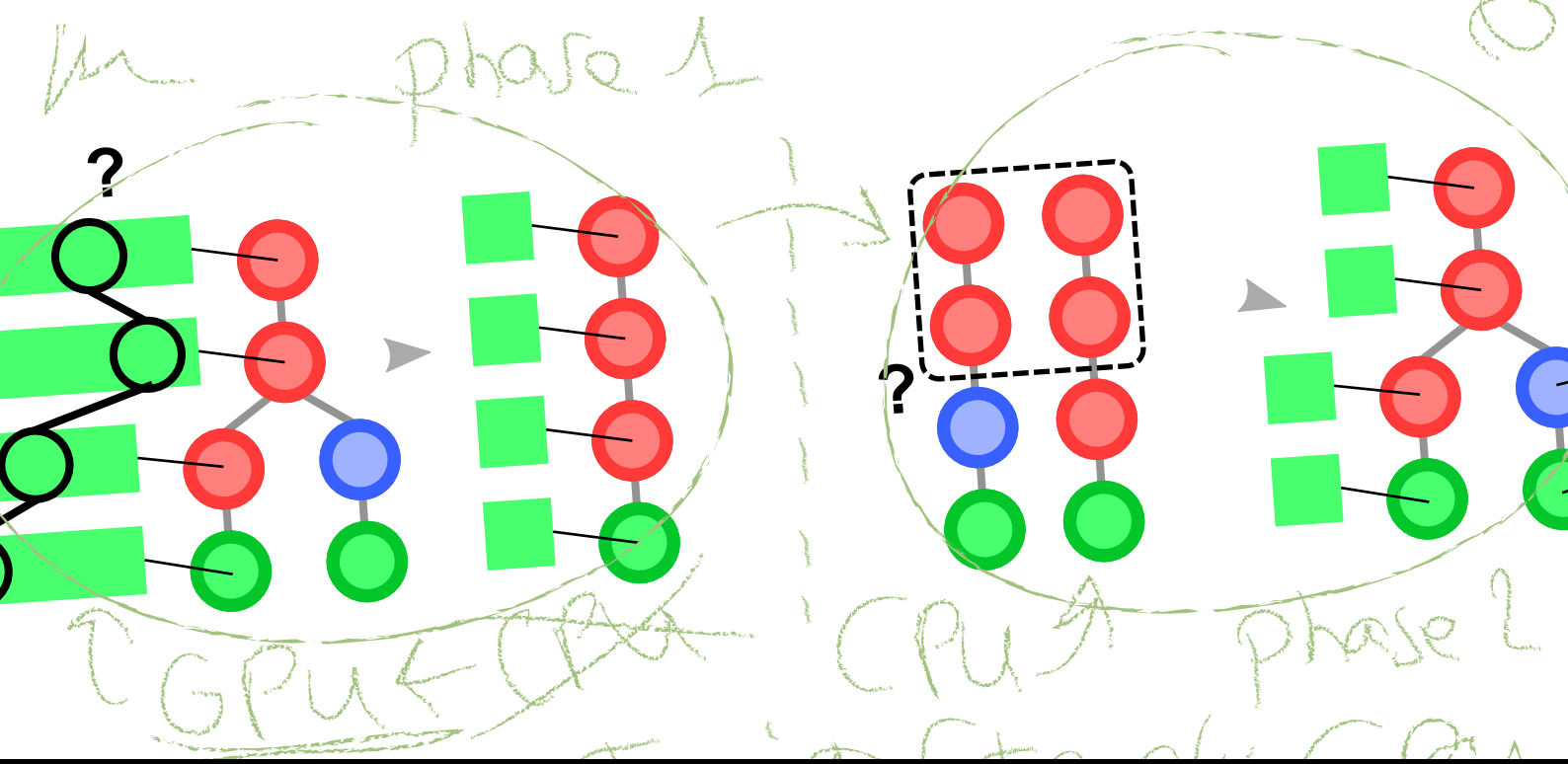


UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLIARD

XML query processing using GPGPU

ST50&AHPM internship report - P2010

JORDAN Vincent

Computer Science Department
Software and knowledge engineering
High efficiency algorithms and modelisation

Kitagawa Data Engineering laboratory
University of Tsukuba
Tennoudai 1-1-1, Tsukuba, Ibaraki, Japan 305-8573
www.kde.cs.tsukuba.ac.jp

Advisor of University of Tsukuba
KITAGAWA Hiroyuki

Advisor of UTBM
LAURI Fabrice

University of Tsukuba

utbm université de technologie Belfort-Montbéliard

Acknowledgements

First and foremost I would like to thank Professor Hiroyuki Kitagawa, my internship adviser. He accepted me into his laboratory and enabled this internship opportunity at Tsukuba University. Although Professor Kitagawa is extremely busy with his own research schedule, he has never hesitated to assist me with any questions or problems I've had over the duration of my internship.
I am grateful to Professor Toshiyuki Amagasa for suggesting my current research topic. He trusted my abilities at solving my project's numerous implementation issues. Along with his support on my over all project, Professor Amagasa has helped me improve this report.

I cannot over look Takahiro Komamizu for his continuous support both inside and outside of the laboratory. KDE laboratory would not have its superior environment without him. He does every task with a smile such as, network wires, teaching Japanese, and presentation reviews. He spends so much time helping everybody, including me for this report.
I thank Djelloul Boukhelef for his comments and clever debugging ideas… in French.
I also want to thank Mariko Kamie, Maria Alejandra Quirós and Sherry Morgenstern for their interest and friendship during the whole six months of my internship. Sherry Morgenstern gave smart advices that improved the English of this report.

You cannot fully enjoy the experience of living in Japan without the Japanese language. I would like to thank my five Japanese teachers from the International Student Center for their motivation to teach Japanese to beginners like me: Professor Tanaka (Mon.), Miyazaki (Tue.), Seki (Wed.), Onodera (Thu.) and Imai (Fri.).

I cannot forget to thank my Japanese teacher at my French university, Keiko Jimbo and Mireille Jacquot who manages administrative tasks for Computer Science internships. Japanese and French universities schedules are not synchronized, but I never had to care about it thanks to their efforts.

Introduction

This final project assignment is the last step of the curriculum at the University of Technology of Belfort-Montbéliard. This 6 months internship validates my software engineering ability as well as my research master studies. Therefore it had to include both of these aspects of Computer Science. The purpose of this training period is to apply and further improve my skills learned at the university. It is also a way to prepare students, such as myself, for their future jobs, by already having experience. Thus the choice of the institution where to do this internship is crucial.

In order to achieve this, I decided to do my internship at the Kitagawa Data Engineering laboratory of the University of Tsukuba, Japan. The internship took place from April 5th to September 17th 2010. The initial theme was "*XML query processing using GPGPU*" and involved CUDA language for its implementation.

This report is divided into four sections. The first part will introduce the University of Tsukuba and the Kitagawa Data Engineering laboratory. It will also explain my expected work at KDE lab. and the schedule of the 6 months spent there. The two following sections are about the required knowledge that I had to gain before starting: *XML query algorithms* and *NVidia GPU architectures*. Finally the fourth part will explain the difficulties and applied solutions in order to use GPU for XML query processing.

Since this report was designed to validate both engineering and research studies, it is supposed to feature the content of two reports. The second and third sections will contain more details about research carried out in *semi-structured language* (XML) and *many-cores parallel processing* (GPGPU) while the last section will give more information on software engineering issues in development using the *CUDA toolkit*.

Table of content

About KDE laboratory and my internship

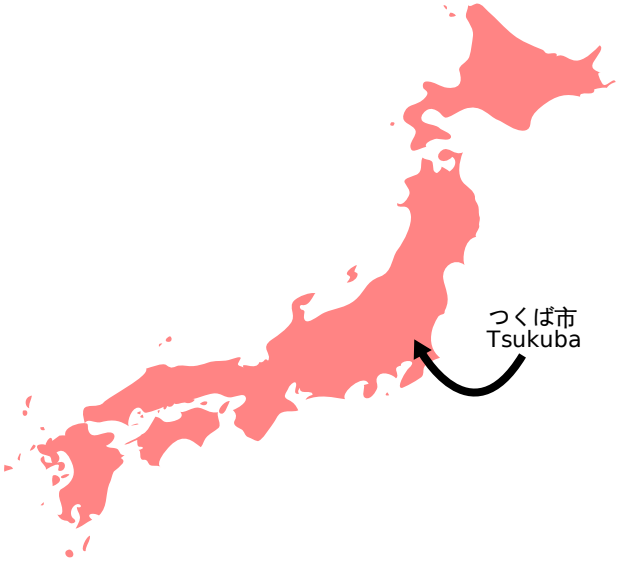
XML query processing algorithm: Parallel TwigStack

nVIDIA GPU architectures and the CUDA framework

Parallel TwigStack on GPU

1. About KDE laboratory

KDE laboratory belongs to the University of Tsukuba. Tsukuba city is located at the foot of mount Tsukuba in the Ibaraki prefecture. This prefecture is in the Kantō plain on the main Japanese island, Honshū.
The city is about 50 km northeast of Tōkyō and approximately 45 min by its dedicated *TsukubaExpress* train line. Tsukuba is a planned science city built in 1962 to relieve Tōkyō's overpopulation problem and to create the largest Japanese research center. The city hosts 60 national research institutes and more than 240 private research facilities. Some major research institutes such as the Japan Aerospace Exploration Agency (JAXA) and the High Energy Accelerator Research Organization (KEK) are in the city. According to [T-INFO], 19,000 researchers (40% of Japanese researchers) are working there. In order to create this science city, Japanese government spent close to 50% of the public research budget for several decades according to [W-TSUKUBA].



Location of Tsukuba city on main Japanese island

1.1 University of Tsukuba

Tsukuba Univ.

Thanks to its implementation in Tsukuba Science city, the University of Tsukuba, established in 1973, took advantage of its advanced research environment and became one of the major universities in Japan. In 2010, the University of Tsukuba was ranked 11th among universities in Japan, 20th among approximately 200 universities in Asia and 174th among more than 600 universities in the world (found in [QSTOP] chart).

In May 2010, 16,828 students were registered at Tsukuba University (source [T-UNIV]). This number includes 1,697 international students. Most of them are Chinese citizen (763 students). Since Tsukuba city was designed as an international science city, its university hosts many foreign students. *Global30* is a government project which aims at "*establishing core universities for internationalization*". Only 13 Japanese universities (including Tsukuba) participate in this project.
With 2.58 Km^2, the campus of Tsukuba University is the largest single campus in Japan. Despite its location in city center, the campus features many green areas, sports fields, lakes and little forests. Approximately 4000 dormitory rooms are provided by the university. The institution also includes a university hospital with 800 beds.
According to [T-CCS1] facilities description, University of Tsukuba has central position in the Tsukuba research computer network (20 Gbits/s) and in Japanese universities network (10 Gbits/s).

The university research system is constituted of 26 research institutes. The department of computer science belongs the Graduate School of Systems and Information Engineering (SIE) and is composed of 35 laboratories divided into five research groups. The department of Computer Science of the University of Tsukuba has the largest number of professor in Japan (the course has 61 faculties).

1.2 KDE laboratory

KDE Lab.

KDE acronym stands for *Kitagawa Data Engineering*. This research laboratory in computer science belongs to the "Software system & computer architecture" research group and focus on management issues of massive data (e.g. very large databases). It exists since 1993 (deduced from publications list found in [T-KDE].
The staff of KDE laboratory is composed of 44 people (as of September 2010). This number includes 3 professors, 2 post-graduate students in Doctoral's program, 28 graduate students in Master's program, 5 undergraduate students in Bachelor's program and 4 other students (for example: short-term internship).
Researches are carried out in three main domains related to data engineering. The laboratory is also involved in the creation of meteorological database for the *Global Environment and Biological Sciences* division. In the following subsections, each field comes with a relevant and recent research paper example published by the KDE laboratory.

A. Infrastructure for Information Integration

Laboratory investigates infrastructures, systems and applications to integrate heterogeneous databases and data sources. The research especially focuses on stream data integration (like GPS position and video stream from street cameras) into conventional relational databases.

StreamSpinner (see [STRSPIN]) is a project created at KDE lab. It's a data stream management system which employs

an architecture combining a stream processing engine and DBMS. The system is able to process both continuous queries and traditional one-shot queries. The system is based on an extensible framework and can cope with streaming video or audio as well. An example of extension is the analysis of video frames acquired through several surveillance camera.
Researches are also done about distributed stream processing since some processes can require heavy computational analysis (as, for example, streaming video frames). Distributed stream processing engines (DSPE) are built on the cooperation of several stream processing engines (SPE), thus a node failure can trigger a failure of whole system. The paper below suggests an adaptive strategy to overcome those unpredictable events.

A-SAS: An Adaptive High-Availability Scheme for Distributed Stream Processing Systems
Hiroaki Shiokawa, Hideyuki Kawashima and Hiroyuki Kitagawa
Proceedings of third International Workshop on Sensor Network Technologies for Information Explosion Era (SeNTIE 2010), May 2010

The laboratory also researches further in database infrastructure for time-series data obtained by sensors. The issue of real-world monitoring databases is the data insertion function since it has to be extremely fast. The DBMS also has to include data analysis functions and continual query support.
KRAFT is a sensing database infrastructure created at KDE lab. for that purpose.

Encrypted databases is another field studied at KDE lab. This research addresses especially web DBMSs of businesses, governments or even individuals because they have numerous entry points that can put database at risk. Privacy protection became a great challenge in our society of instant information communication. Encrypted database means that the data storage format is encrypted. It prevents data from being read even if someone gains access to the storage medium (using stolen hard drive or remote access to the server file system).
The paper below studies database security by cryptography techniques. It proposes a mixed cryptography database (MCDB). This framework aims to encrypt database over untrusted network while keeping querying efficiency.

MV-OPES: Multivalued-Order Preserving Encryption Scheme: A Novel Scheme for Encrypting Integer Value to Many Different Values
Hasan Kadhem, Toshiyuki Amagasa and Hiroyuki Kitagawa
IEICE Transactions on Information and Systems, September 2010

B. XML and Web Programming

XML is widely used language for machine-readable data representation. It is a recommendation of World Wide Web Consorsium (W3C) since 1998 for a better interoperability in network environments, therefore the amount of generated data using this language is huge and still increasing. Several issues of XML data management are studied at KDE lab.

XML Functional Dependency (XFD) is similar to functional dependency in relational database. It is a kind of constraint between two sets of attributes in a relation from a database. FD are important in normal forms definition for relational databases. A functional dependency between a set of attributes and another dependent attribute can denote redundancy in the DB content.
XFD enables the same, but for XML data: XML Normal Forms (XNF) but unlike relational databases, since XML is flexible and hierarchical, XFD definition is uneasy. The paper below discuss a scheme for efficient XFD detection based on OLAP-inspired algorithm.

Fast Detection of Functional Dependencies in XML Data
Hang Shi, Toshiyuki Amagasa and Hiroyuki Kitagawa
The 7th International XML Database Symposium (XSym2010), September 2010

XPath is a simple XML query language that is the base to more complex *SQL-like* query languages (like XQuery). Querying process on large XML data (of several gigabytes) can be a problem despite the creation of new query processing algorithms (such as TwigStack) optimized for XML and its hierarchical semi-structured data storage.
XML query algorithm optimization is expected to be done thanks to data parallel execution. Parallel XML query processing can take advantage of new multi-core architectures. The challenge of efficient partitioning has been studied at KDE lab. In the paper below, a partitioning model is presented so that several instances of the same query algorithm can be executed in parallel on different parts on the XML document.

Executing Parallel TwigStack Algorithm on a Multi-core System
Imam Machdi, Toshiyuki Amagasa and Hiroyuki Kitagawa
Proceedings 11th International Conference on Information Integration and Web-based Applications and Services (iiWAS2009), December 2009

C. Data Mining and Knowledge Discovery

Many data mining and knowledge discovery techniques were studied at KDE lab: outlier detection, association and ratio rule mining, information extraction from documents, time-series document clustering, topic detection, mobility histogram construction for mobile objects.

In the paper below, a way to measure the freshness of a web page is proposed. The freshness of a web page is not an easy criteria to evaluate since freshness depends on page's content. Defining the freshness as "*whether or not a page has been recently bookmarked*" is not enough because the lifetime of freshness is variable. News scripts pages and manual or reference pages have not the same lifetime (the first is short while the second is longer). This method uses social bookmarks (especially the *spread of bookmark timestamps*) to extract up-to-date pages among the huge content available on the internet.

A Ranking Method for Web Search Using Social Bookmarks
Tsubasa Takahashi and Hiroyuki Kitagawa
Proceedings International Conference on Database Systems for Advanced Applications (DASFAA 2009), April 2009

New topic detection needs emerged recently. Video-sharing services host a content which is difficult to categorize automatically (without any human intervention). The paper below introduces a system for topic extraction from a set of videos by making use of time data and author's diversity.

Topic-Based Awareness Computing Model for Video-Sharing Service
Mariko Kamie, Takako Hashimoto and Hiroyuki Kitagawa
Second International Symposium on Aware Computing (ISAC 2010), November 2010

D. Scientific Databases

KDE lab. created and manages the JRA-25 Archive (more information in [JRA25]). This is a meteorological database designed to store long-term analysis of global weather data provided by the Japan Meteorological Agency (JMA). The database currently contains 25 years of data (circa 700GB, in August 2007). Web Services have been implemented on top of this database to provide reanalysis maps through the internet (e.g. GoogleEarth).

Other kind of scientific databases research includes satellite DEM images (DEM = Digital Elevation Model). The paper below provides a method to match change between two DEM image of the same area and information found in Web content. For example, if a new shopping center is built, some news report will talk about it on the internet and the elevation model will change in the same time. By matching those two events, the location of the new building and the article featuring further information can be linked. A prototype has been evaluated on Tsukuba city and was able to output some good results about real buildings.

Provinding Constructed Buildings Information by ASTER Satellite DEM Images and Web Contents
Takashi Takagi, Hideyuki Kawashima, Toshiyuki Amagasa and Hiroyuki Kitagawa
Proceedings of Data Intensive eScience Workshop (DIEW 2010) (DASFAA2010 Workshop), April 2010

E. Collaboration with CCS as Computational Intelligence group

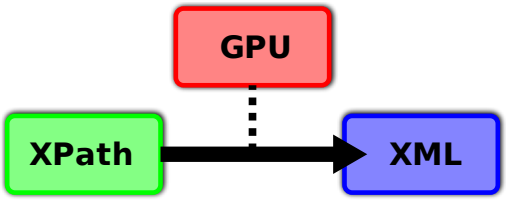
The Center for Computational Sciences or **CCS** is a framework for cooperative research in *Computational Science* which involves several laboratories in different fields.
"Computational science has shifted the paradigm of scientific research to include simulation as a fundamental method of science, along with experiment and theory" in [T-CCS2]. CCS has been created in 2004 as an inter-university research facility. Its main mission is to carry out large-scale simulation and data analyses in the following domains: *fundamental science* (physics of particles and astrophysics), *material science* and *life and environmental science*. CSS is the association of 32 professors and associate or assistant professors from different graduate schools.

The main features of CCS are high-performance computing systems and high-speed network infrastructure. The center has been designing massively parallel computers cluster since decades (from 1977). PACS-CS system, working since

2006, is divided into 2560 nodes connected by 20480 Gigabit Ethernet wires. With 10.35 Tflops, this system is ranked 34th (as of June 2006) in the famous [TOP500] of the most powerful known computer systems in the world. The system's network, named *Hypercrossbar*, is an original 3-dimensional interconnection web among computation nodes.

KDE lab. attempts to make its research achievements practical as much as possible in cooperation with other groups. Both *Computational Media group* and *Global Environmental Science group* also belong to CSS. Meteorological database design is a result of these cooperations.

2. Research topic and schedule



2.1 Research topic

Before arriving in Japan, my research topic was only defined as follows: *XML query processing using GPGPU*. In order to clearly introduce this topic, terms and acronyms will be defined first, then the core idea will be explained.

A. Definitions

eXtensible Markup Language (XML) is both human- and machine-readable, tree-based language widely used in computer world for the transmission and the storage of data. This simplification of the SGML language says [W-XML], was designed to ease and spread the usage of an interoperable language over the internet. This aim has been reached nowadays since the XML language is used by a huge amount of applications and most programming languages feature an XML parser. Actually XML is a meta-markup language and is used for creating other markup languages. XML tags are used to describe the contents of a document.

XML Path (XPath) language has been created to address parts of XML documents. XPath is required by both XPointer and XSL transformation standards, consequently of the broad XML usage, the need to locate specific data in XML documents became high and revealed a trade off: *Keep data in convenient XML format with slow querying or convert data into more efficient database format with faster querying?*
The biggest is the amount a XML data, the most time-consuming is the conversion option. My research is about finding a way to speedup XPath queries over big XML documents.

Graphic Processing Unit (GPU) is a processor architecture designed from the beginning for efficient and massive parallel execution. According to the increase of graphic rendering complexity, these chips gained a more general purpose design. GPGPU acronym refers to these new GPU architectures and stands for **General Purpose GPU**.
The strategy of Professor Amagasa (who suggested this idea) is to execute on GPU, several instances of the same query processing algorithm addressing different data partitions. This is data parallelism (opposed to task parallelism where one algorithm processes several data in the same time). Although this hardware is called "general purpose", several limitations have to be overcome.

"**Stream processing** is a computer programming paradigm" [W-STRPROC]. This paradigm was invented to create programs that use a limited form of parallel execution. A stream is a set of data and the stream processing consists in applying one or more hardware instructions to each element of the stream. That kind of parallel execution is limited since there is no control on the way this process is done (synchronization, communication, ...). The series of operations is the same for each data therefore it cannot use any conditional branching which depends on the data value.

Stream *S0*	4	18	12	5

Operations	`1 m = 10`
	`2 if S0[i] > m`
	`3  S1[i] += 2`
	`4 else`
	`5  S1[i] = 0`			

Stream *S1*	0	20	14	0

This is NOT stream processing

Stream *S0*	4	18	12	5

Operations	`1 m = 10`
	`2 if m > 5`
	`3  S1[i] += 2`
	`4 else`
	`5  S1[i] = 0`			

Stream *S1*	6	20	14	7

This is stream processing

Vector processing is a computer programming paradigm similar to stream processing paradigm. The major difference between vector and stream paradigm is the memory access pattern. Vector processing operators involve a memory read, an execution on an instruction over several data and finally a memory write. Memory access for each single-instruction leads to high bandwidth use.

B. Idea

The practical target of this research is to evaluate the opportunity of using GPU as XML query coprocessor for programs dealing with a big amount of XML data. In comparison with their parallel processing power, GPU are cheaper than most dedicated chips for parallel execution (such as DSP).
Making good use of GPU hardware, because it is still very graphic processing oriented, is a great challenge:

- optimize memory access since GPU design is aimed toward high floating point computation and low memory

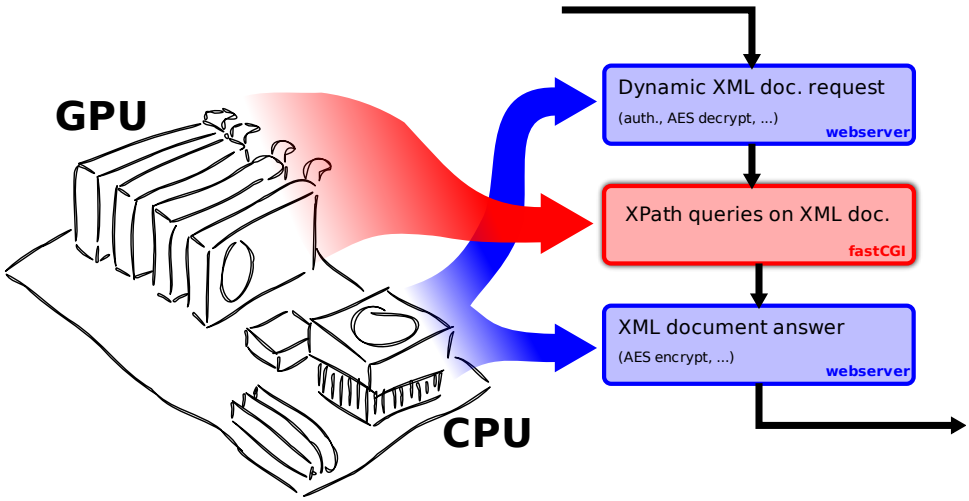
access performances
- hide the 'CPU to GPU memory' communication cost
- fit the GPU parallel architecture that is much more limited than multi-core CPU execution architecture

Research on parallel XML query processing was already done in the same laboratory by Imam Machdi. My main work was to create a GPU algorithm based on the results of his Ph.D. thesis. This task is challenging since current XML processing algorithms do not fit the stream processing paradigm recommended for GPGPU computation.
Two solutions are possible to solve this issue:

1. overcome GPU limitations in order to do more than stream processing
2. create a new algorithm that is "stream processing compliant"

During this internship, the first solution has been studied. This choice was made because of the tight schedule since creating a whole new algorithm would have been risky and may lead to "no result found" at the end of my internship.

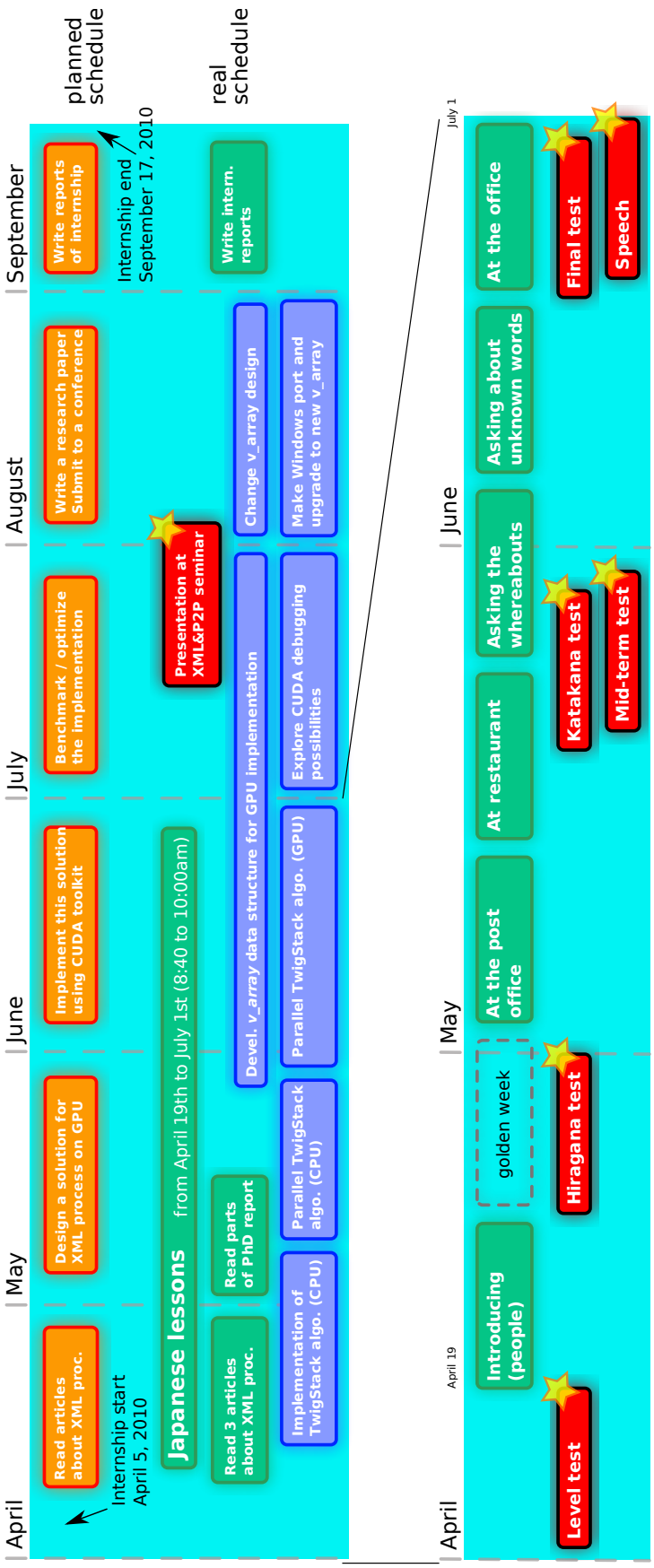
What is the practical purpose of this research?
My long-term plan is to build a hybrid webserver which makes use of CPU and GPU. This solution might be a cheaper alternative to multiprocessor configurations found nowadays. The main advantage of the hybrid solution would be its scalability since current motherboards can host from zero up to four discrete graphic cards (known as *4-way SLI* for nVIDIA, *4-way Crossfire* for AMD). At low request rate, CPU would handle the whole process and GPU devices would be added progressively while the rate increases.



The GPU-CPU hybrid webserver

2.2 Schedule

The planned schedule (in orange) was divided into one task per month. This expected schedule was flexible and no deadline has been set. It has not been provided by my professor, I created it myself to have a goal to reach.



Schedule of my 6 months internship at KDE laboratory
+ Japanese lessons schedule

After an adaptation week to my new life in Japan, I began with reading several research papers about XML query processing. Professor Amagasa suggested those three papers after an evaluation of my knowledge about XML. They

are the most relevant papers on XML query processing. This was intended to fill my lack of knowledge in that field. After this first step, I was able to understand more easily the section of the Ph.D. dissertation that I was expected to use in my implementation. I started documentation and implementation in parallel. Implementation helps understand the documentation.

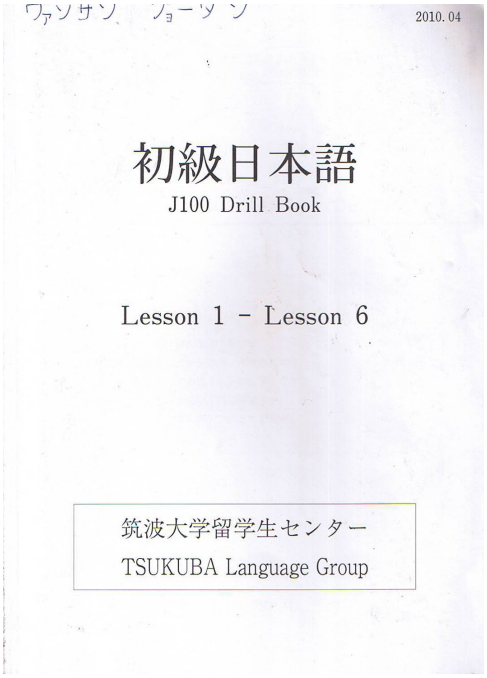
First a simple implementation has been done. The same algorithm as the one used by Imam Machdi was implemented from original research paper. This task was done to deeply understand the execution requirements. Consequently to the discovery of missing required features of the targeted nVIDIA toolkit, a data structure library has been designed and implemented for this framework. Meanwhile original and GPU versions of the algorithm have been upgraded in order to make use of this new library. The aim is to compare new solution with previous one. Unfortunately I did not manage to fix some execution errors of the GPU version, despite the several strategies tried.

By comparing planned and real schedule, it is obvious that until end of June, I was still on schedule, but from July to August, I have not been able to do what was planned: *benchmark* and *research paper*. Instead I tried to fix my implementation of parallel twigstack algorithm on GPU. For example, some parts of the project were ported to Windows in order to use another debugger (released in June 2010 only).

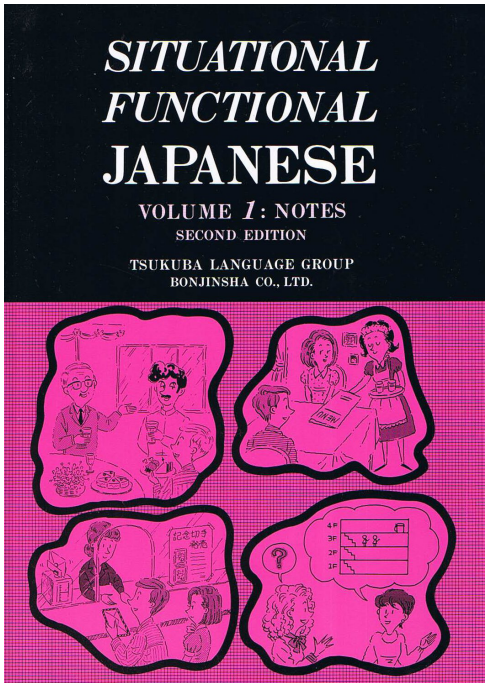
A. Japanese lesson

Because of its great community of foreign students, the international student center has important staff and facilities at Tsukuba University. The center offers a complete set of Japanese language courses for all levels of international students. According to [T-INTERSC], there are nine lecture levels (from J100 to J900) and six kanji levels (from K200 to K700). Japanese courses are not mandatory for short-term exchange research students like I was and they can be attended only if his academic adviser allows him to do so. Professor Kitagawa approved my request for Japanese course even if it was not useful for my current short internship.
At the UTBM, I attended "Japanese for real beginners" course (LJ00) but lessons were given using rōmaji transliteration. Japanese writing system is composed of three different scripts: Chinese-based ideographs (kanji), syllabic-based characters (hiragana and katakana). Since the Japanese level tests of Tsukuba University are not transliterated into rōmaji, I have not even been able to use my initial little knowledge of Japanese language and I got a grade close to zero. My level was J100 then, but fortunately I have been assigned to the right course since, despite its name, J100 is too fast for real beginners on my opinion.

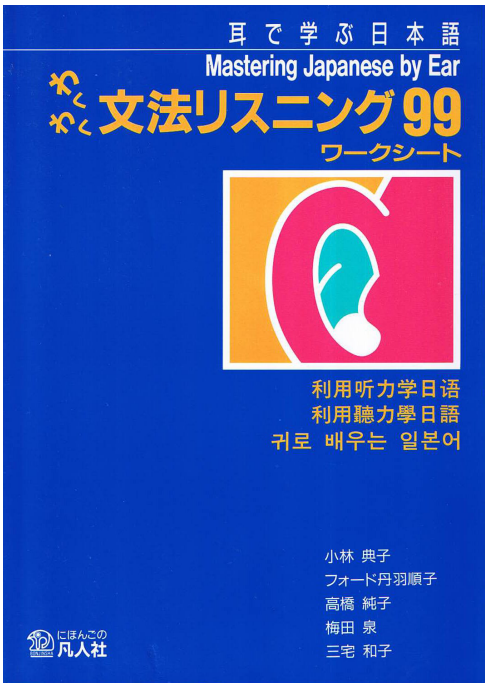
J100 is divided into six lessons. Course takes place every mornings (8:40-10:00) from Monday to Friday. Courses are free of charge. Students only have to purchase these three books: the J100 drill booklet, the "SFJ" book and the "Waku waku" book. The first contains structure drills, model conversation and conversation drills of the six lessons showed in the schedule above. The second contains grammar notes which goes outside the scope of J100. The third contains listening exercises.



Drill book



Situational Functional Japanese book



Waku waku book

3. XML query processing algorithms

In February 1998, XML 1.0 has been introduced by the W3C to be the new recommendation for data exchange on internet. At this time, the most simple querying solution was to store XML data into relational databases and to convert XML queries into SQL queries. Storing and querying became an important research domain because this solution leads to poor performances. This issue has been addressed by several research papers. The most famous is the TwigStack algorithm.
The first section makes an overview of the current solution for XML documents storage into well-known relational databases. The progress to TwigStack algorithm will be shown in the second section through two previously published papers on which TS is based.

3.1 Commercial relational database management systems

One of the first ideas when it comes to XML query processing is to store XML documents into relational database system and convert XPath queries into SQL queries. Using this solution, *twenty years of work on RDBMS query optimization, query execution, scalability, concurrency control and recovery immediately extend to* XML query processing. Commercial products such as Oracle, IBM DB2 and Microsoft SQL Server all provides specific features to store and query XML data.

RDMS	Oracle 11g	IBM DB2 9	Microsoft SQL Server 2008
Large object storage data type suitable for XML documents	XMLTYPE[1] store AS CLOB	XMLCLOB[2] or XMLVARCHAR[3] or XMLFILE[4]	[n]varchar(max) or varbinary(max)
Automatique mapping/shredding XML into relational storage	XMLTYPE[1] store AS OBJECT RELATIONAL	XML Collections	XML View[5]
Native XML storage data type	XMLTYPE[1] store AS BINARY XML	xml (pureXML)	xml

Storage options of an XML document into relational database management system

[1]XMLType cannot exceed 4GB.
[2]XMLCLOB cannot exceed 2Gb.
[3]XMLVARCHAR cannot exceed 32kb.
[4]More flexible but does not benefit from database-managed persistency and integrity.
[5]Schema cannot be recursive or the maximum recursion depth is known.

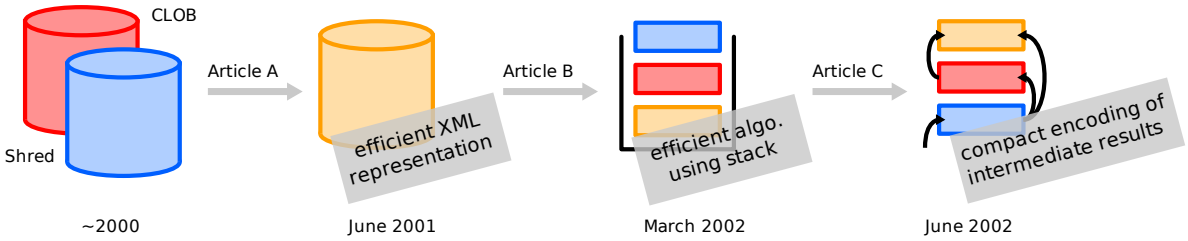
Most of the commercial relational database vendors claimed XML support in their products through XML extenders as soon as XML was released. Two storage options were available: plain-text storage as simple text string or shredding/mapping into standard relational tables. As it can be noticed in the table above, the three main vendors all finally included a *native XML* storage in the latest version of their product since XML data does not fit that well in relational databases: PureXML is a new feature of IBM DB2 9 [DB2-XML], (native) xml storage is a new feature of Microsoft SQL Server 2005 [SQLSRV-XML], BINARY XML storage is a new feature of Oracle 11g [GRALIKE10].

The main difference between relational and XML data model is that the first is structured and the second is semi-structured or unstructured. Relational data model is suitable for the storage of highly structured data having already well-known schema at database design. If the structure of the data is not known or if it may change significantly in the future, XML data model offers more flexibility. Due to its tree structure, XML data model excels at representing containment hierarchies (especially recursive ones), finally XML data model allows the creation of queries based on the structure of the data (rather than its value).

Even though XML solves many of the problems by providing a standard format for data interchange, there are other problems, such as storing the XML documents in a centralized repository, as well as the ability to quickly search for information or to trigger automatic data change when a particular action occurs. These kinds of issues can be addresses only by a database management system (DBMS). Those systems have a legitimacy for XML documents too, but they have to process them in a different way than relational tables. This is why some research in that field was required.

3.2 Research achievements on XML query processing

This section will introduce the evolution of XML query processing research through three papers and their respective algorithms. XML data preprocessing is presented first when explaining the first article. It will not be repeated since this task is the same and is required for the three algorithms. This task is also required for the GPU version explained in following chapters.



Research progress on XML query processing

This figure introduces the main improvement made by each article based on the work of the previous. Numerous successors exist to the last article of this figure, but do not introduce major changes.

A. MPMGJN algorithm

Utilization of the *Multi Predicate MerGe JoiN* algorithm for XML query is explained in the following research paper:

On Supporting Containment Queries in Relational Database Management Systems
Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo and Guy Lohman
SIGMOD 2001

From the observation that the inverted lists of Information Retrieval engines is well-suited to XML queries, this paper compares two possibilities of querying architecture for RDBMS: *separated "loosely-coupled" IR engine* or *native tables and SQL queries*. Because of the several advantages of native relational database storage and querying, the authors purpose was to achieve same or better performances using the second option.
The article shows that join algorithms and hardware cache utilization are not efficient when processing XML queries from relational tables.

The author predicted that *a substantial amount of* XML documents would be stored in relational databases in the future. Nine years later, major DBMS vendors included a native XML storage option in their product. In retrospect, the good idea of this article was not the join algorithm, but the efficient XML representation as an extended inverted index.

XML data preprocessing

The following simple example is a *well-formed* and *valid* XML document. The first line is the *prolog* of the document (a kind of header), the content goes from the second line to the end line. There are three kinds of token: *element* (opening and closing), *attribute* and *text*. The most common way to represent XML content is the tree data structure.

```
<?xml version="1.0" encoding="UTF-8" ?>
<library>
        <category name="France">
                <book>
                        <title language="English">The Little Prince</title>
                </book>
        </category>
</library>
```

Example of XML document viewed as plain text

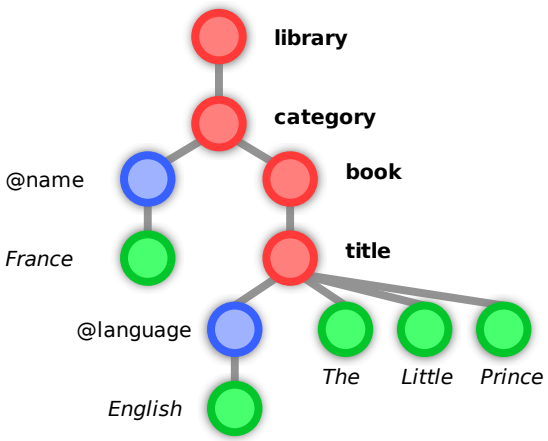
Same XML document viewed as a tree

An XML document can be formalized as a set of a vertice set, an edge set and a root vertice: (V, E, r) where $V = \{v_1, ..., v_n\}$ is the set of nodes which contains elements (XML tag, mandatory), strings (optional) and attributes (tag attributes, optional), $E = \{(v_j, v_k)\}$ is the set of edges between two tree nodes and $r \in V$ is the root node.

In order to simply the process of attributes, an XML document was formalized as a set of element node set, attribute node set, string node set, edge set and root element node: (El, At, St, Ed, r). El = {el_1, ..., el_n}, At = {at_1, ..., at_m} and St = {st_1, ..., st_p}. Ed = {(x, y)} where (x, y) ∉ {(at_j, at_k), (at_j, el_k), (st_j, st_k), (st_j, at_k), (st_j, el_k)}.

```
<?xml version="1.0" encoding="UTF-8" ?>
<library>
        <category>
                <@name>France</@name>
                <book>
                        <title>
                                <@language>English</@language>
                                The Little Prince
                        </title>
                </book>
        </category>
</library>
```

Example of XML document without element attributes



Same XML document viewed as a tree

XML document is not kept in its original tree structure from DOM parsers. The underlying structure is in the form of *streams*. Streams are sequences of XML nodes represented in a 3-ary tuple representation: *document number*, left and right *positions* and *depth*. The following example is still based on the same XML document and also feature a path column that is not mandatory.
The classic inverted index data structure maps words or phrases only. In order to store XML documents, it can be extended into: an element index (*E-index*), an attribute index (*A-index*) and a term index (*T-index*).

value	document number	left position	right position	depth	path
E-index (XML elements)					
library	1	1	17	0	/
category	1	2	16	1	/library/
book	1	6	15	2	/library/category/
title	1	7	14	3	/library/category/book/
A-index (attributes of XML elements)					
name	1	3	5	2	/library/category/
language	1	8	10	4	/library/category/book/title/
T-index (text words nested in XML elements)					
France	1	4	4	3	/library/category/name/
English	1	9	9	5	/library/category/book/title/language/
The	1	11	11	4	/library/category/book/title/
Little	1	12	12	4	/library/category/book/title/
Prince	1	13	13	4	/library/category/book/title/

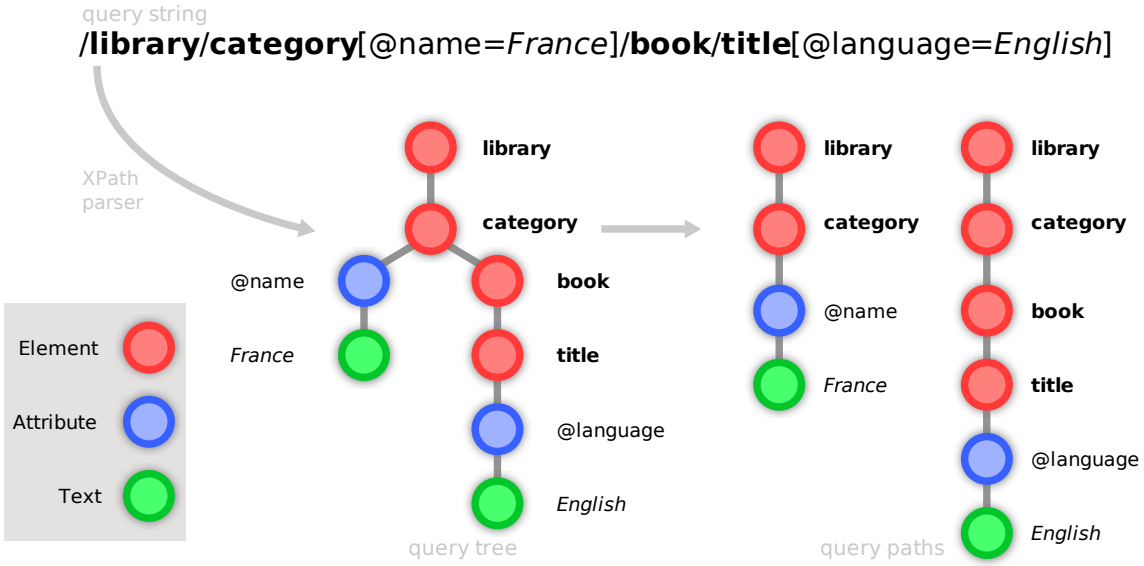
Example of the extended inverted index of XML data

This inverted indexes representation was chosen because it improves the discovery of containment properties (or structural relationship) on which XML queries are based. Paper use different terms. Containment properties between two nodes of the XML tree can be "*ascendant-descendant*" (indirect containment) or "*parent-child*" relationships (direct containment). I find the term between quotes easier to understand than the one between parentheses therefore I will use them.

Using the notation of the table example, "A is a *descendant* of B" is equal to this condition $doc\_no_A = doc\_no_B$ AND $left\_pos_A > left\_pos_B$ AND $right\_pos_A < right\_pos_B$. This condition matches all descendants. If only *children* have to be match, it only requires to append $depth_A = depth_B + 1$ to the previous condition. Because of the strict nesting structure of XML, $right\_pos_A < right\_pos_B$ can be omitted for child relationship.
A worth noting point about this representation of XML document is that checking an "*ascendant-descendant*" relationship is as easy as checking "*parent-child*" relationship. This is the main advantage of this representation over the tree representation.

Query parsing

An XML query can be seen as a set of *structural relationships*. Using XPath abbreviated syntax, '`/`' and '`//`' (abbr. of `/descendant-or-self::node()/`) represent *parent-child* and *ascendant-descendant* relationships.



Example of XPath query string to query paths conversion.
Please note that the tree representation of this figure always feature "*parent-child*" relationship between nodes.

The query of the figure means "*all English titles of books in categories named France of the library*". XPath query string is parsed in order to create a tree representation of it. The brackets allows to create *several* conditions in the query which means *several* branches in the tree representation. This query tree can also be viewed as a set of query paths.
Brackets can also have another role. Please compare the following queries:

1. `/library/category[@name=France]/book/title[@language=English]`
2. `/library[/category[@name=France]][/book/title[@language=English]]`

These two XPath queries do produce the same query tree and the same query paths set. The difference is only at display level. First query will output XML data between matching `<title>` while second query will output all data between `<library>` which contain a matching title (if library contains many matching titles, the whole library will be output as many times as the number of matching titles).

Algorithm

The MPMGJN algorithm is a variant of the well-known inverted list algorithm of Information Retrieval. For example, if the query to proceed is `B//"A"`, the inverted lists of the element `B` and the term `A` are retrieved into `List1` and `List2` (function input).

```
1  # List1: Outer list, List2: Inner list
2  function containmentMerge(List1, List2)
3      set cursor1 at beginning of List1
4      set cursor2 at beginning of List2
5      while cursor1 != “end of List1” and cursor2 != “end of List2”
6              # join only on the same document
7              if cursor1.docno < cursor2.docno
8                      cursor1++
9              else if cursor2.docno < cursor1.docno
10                     cursor2++
11             else
12                     # mark contains the start of the record scan. mark = cursor2 since no scan yet
13                     mark = cursor2
```

```
                    while cursor2.position < cursor1.position and cursor2 != “end of List2”
                            cursor2++
                            # if no start record found before end of list
                            if cursor2 == “end of List2”
                                    cursor1++
                                    cursor2 = mark
                            # if start record found mark will remember this start while scanning
                            else if cursor1.val contains cursor2.val
                                    mark = cursor2
                                    do
                                            # ~output
                                            merge cursor1 and cursor2 values
                                            cursor2++
                                    # stop scanning if join no more possible ("stop record")
                                    while cursor1.val contains cursor2.val and cursor2 != “end of List2”
                                    # next outer will be proceeded
                                    cursor1++
                                    # will restart at "start record" on next seek
                                    cursor2 = mark
                            # end of scan
                            end if
                    # end of seek
                    end while
            # end of document search
            end if
    # end of join merge
    end while
end
```

The inverted list containment merging algorithm with comments

When applied to B and A streams, the algorithm above performs a join operation like the following SQL query.

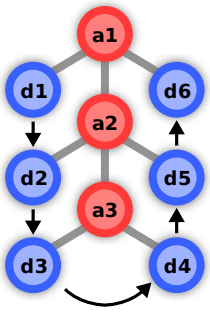
```
SELECT *
FROM elements e, texts t
WHERE e.value = 'B'
AND t.value = 'A'
AND e.doc_no = t.doc_no
AND e.left_pos < t.left_pos
AND e.right_pos > t.right_pos
```

Example of SQL query for B//"A"

The RDBMS query plan generator can choose between *index nested-loop join* and *merge join* algorithms to process the SQL query. The first choice is very close to the MPMGJN algorithm: *inner rows are selectively examined using start and stop keys in both*, but it still suffer from binary trees utilization. Binary search is efficient but performs unpredictable memory access. The improvement of MPMGJN is that the seeks are made serial in comparison with standard index nested loop, thus it has a better hardware cache usage and lower cache miss rate.
Despite its good cache usage, MPMGJN algorithm still has a big drawback: the worst case is quadratic.

```
<?xml version="1.0" encoding="UTF-8" ?>
<a1>
        <d1 />
        <a2>
                <d2 />
                <a3>
                        <d3 />
                        <d4 />
                </a3>
        </a2>
</a1>
```

Example of MPMGJN worst case, viewed as plain text



Example of MPMGJN worst case, viewed as tree

B. Stack-Tree algorithm

Stack-Tree algorithm is explained in the following research paper:

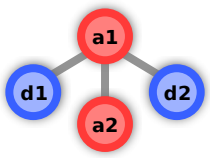
Structural Joins: A Primitive for Efficient XML Query Pattern Matching
Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, Yuqing Wu
ICDE 2002

This paper takes advantage of the inverted list representation of XML data while introducing a novel stack-featured algorithm. Stack-tree algorithm achieves linear worst-case complexity while MPMGJN algorithm was quadratic.

```
# AList: list of potential ancestors sorted
# DList: list of potential descendants sorted
function stackTreeDesc(AList, DList)
    a = AList.firstNode
    d = DList.firstNode
    outputList = NULL
    while “the input lists are not empty” or “the stack is not empty”
            if a.StartPos > stack.top.EndPos and d.StartPos > stack.top.EndPos
            # cannot contain any d or a: not a solution -> remove
            # example: </stackTopElem> ... <nextA> ... <nextD>
                    tuple = stack.pop()
            else if a.StartPos < d.StartPos
            # ...and a could be a descendant of top stack element if it isn't going
            # to be closed before d starts
                    stack.push(a)
                    a = a.nextNode
            # (a.StartPos > d.StartPos) but top stack element contains d
            # therefore d can't be nested more deeper -> return stack as result
            else
                    for a1=stack.bottom; a1!=NULL; a1=a1.up
                            append(a1,d) to outputList
                    end for
                    # d completed
                    d = d.nextNode
            end if
    end while
end
```

Stack-Tree algorithm with comments

This join algorithm provides for a more efficient *set-at-a-time* strategy performing no unnecessary comparisons while MPMG join algorithm used a *node-at-a-time* strategy (especially for *parent-child* relationship).



MPMGJN will perform unnecessary comparisons while Stack-tree will not.

The problem of the previous algorithm is solved since Stack-tree algorithm will not restart after examining `d1`, but will use the stack to go back later (for `d2`).
Thanks to its stack, Stack-tree algorithm can guarantee a linear worst-case complexity (in both CPU and I/O).
The paper presents an efficient join algorithm for binary relationships, but complex queries contains several binary relationships. Finding the optimal join ordering was outside the scope of this paper and will be addressed by the next one.

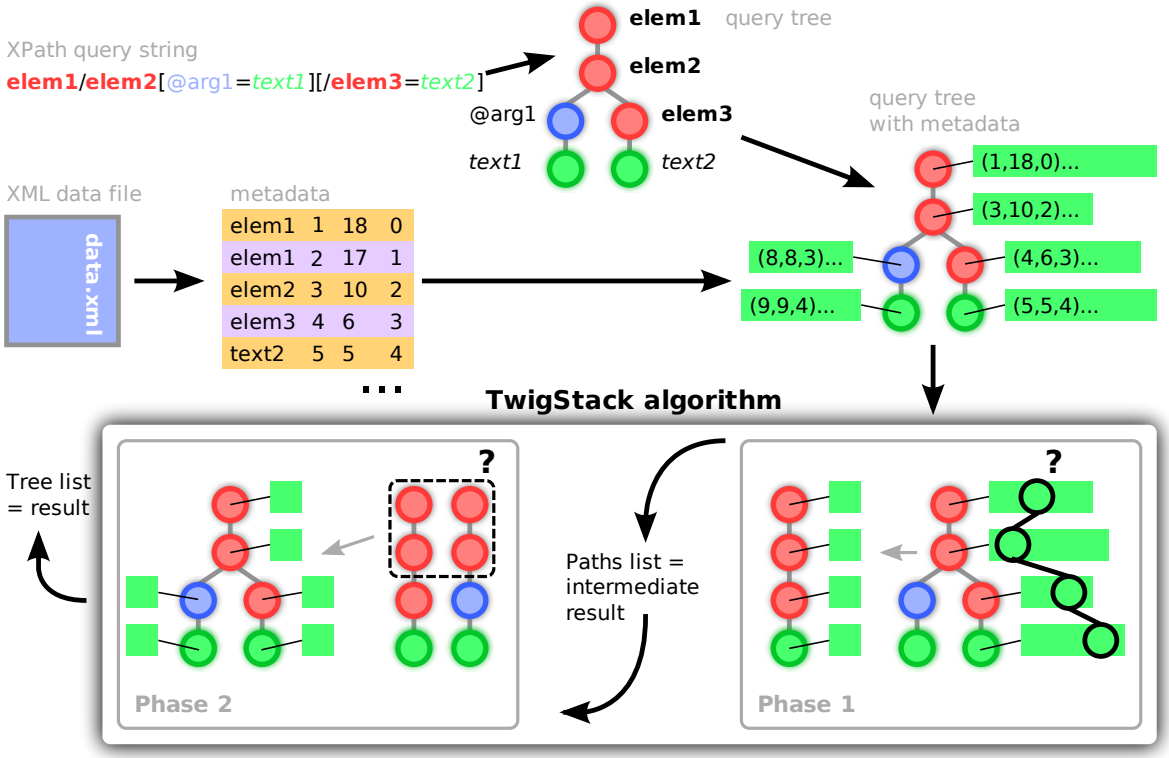
C. Holistic Twig Joins algorithms

Twig Joins is a family of algorithms for processing XML query patterns. They are refered as *holistic* because they allow to match structural relationships holistically (i.e., as a whole), thus reducing the number of not required temporary results. This strategy is opposed to previously presented join algorithms which only solved the problem of binary relationships while the join ordering of complex queries remained outside the scope of them.
Like Stack-tree, TwigStack algorithm uses a set-at-a-time strategy. The original TwigStack algorithm is explained in the following research paper and many other algorithms have followed such as Twig2Stack, TwigList or TwigMix.

Holistic Twig Joins: Optimal XML Pattern Matching
Nicolas Bruno, Nick Koudas and Divesh Srivastava
SIGMOD 2002

Algorithm overview

Like all algorithms presented in this section, TwigStack makes use of the extended inverted index representation of XML data. XML query string is also viewed as a tree, but unlike previous solutions, this tree is not divided into query root-to-leaf paths and binary relationships. The inverted list (refered as stream in the article) of each query node is retrieved and linked to the query tree.



TwigStack query processing algorithm overview

Algorithm execution is divided into two different steps: *phase 1* and *phase 2*.
In the first phase, query node streams are compared in order to find root-to-leaf paths. In the second phase, those paths are merged to create full query tree again.
As said before, the algorithm does not just solve all root-to-leaf path matching a query path because it would lead to many intermediate results which may not be part of the final answer. The `getNext()` function ensures that when query twig pattern has only ancestor-descendant edges, *each solution to each individual query root-to-leaf path is guaranteed to be merge-joinable with at least one solution to each of the other root-to-leaf paths*. This function is key feature of the TwigStack algorithm.

```
# q: query twig pattern (q = root of the tree)
function twigStack(q)
    # PHASE1
    # End of first phase is reached when
    # ∀qi ∈ subtreeNodes(q) : isLeaf(qi) ⇒ eof(Tqi)
    while !end(q)
            # getNext() call ensures that before a node hq from stream Tq
            # is pushed on its stack:
            # - hq has a descendent h_q_i in each of the streams Tqi.
            # - each of the nodes hqi recursively satisfies this too.
            q_act = getNext(q)
            if !isRoot(q_act)
                    cleanStack(parent(q_act), nextL(q_act))
            end if
            if isRoot(q_act) or !empty("stack of parent(q_act)")
                    cleanStack(q_act, nextL(q_act))
                    moveStreamToStack("stream of q_act",
                                      "stack of q_act",
                                      "pointer to top("stack of parent(q_act)")")
                    if isLeaf(q_act)
                            showSolutionsWithBlocking("stack of q_act", 1)
                            pop("stack of q_act")
                    else
                            advance("stream of q_act")
                    end if
            end if
    end while
    # PHASE2
    mergeAllPathSolutions()
end
```

```
function getNext(q)
    if isLeaf(q)
            return q
    end if
    foreach q_i in children(q)
            # Recursive call
```

```
            n[i] = getNext(q_i)
            if n_i != q_i
                    return n_i
            end if
    end foreach
    # Please note that n is an array of query nodes (n=children of q)
    n_min = minNextL(n)
    n_max = maxNextL(n)
    while nextR(q) < nextL(n_max)
            advance(q)
    end while
    ifnextL(q) < nextL(n_min)
            return q
    else
            return n_min
    end if
end
```

```
# T_q: stream of q, S_q: stack of q, p: pointer to parent
function moveStreamToStack(T_q, S_q, p)
    push(S_q, couple(next(T_q), p))
    advance("stream of q")
end
```

```
# S: stack, actL: left position of the actual query node
function cleanStack(S, actL)
    while !empty(S) and topR(S) < actL
    pop(S)
end
```

Holistic TwigStack algorithm with comments

Phase 1

The holistic twig joins algorithm can perform multiple scans over stream inputs simultaneously while reducing redundant query root-to-leaf path solutions optimally and skipping stream nodes that do not contribute to the solutions. The function `showSolutionWithBlocking()` does not show the solutions actually (the name comes from the research article). The complete intermediate path solution(s) which are stored in compact stack encoding are built and appended to the list of intermediate path solutions found so far. The second phase will use this list as input to the merge-join process.

Phase 2

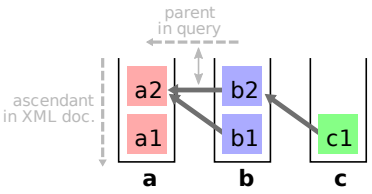
The second phase merge the intermediate root-to-leaf path solutions. Thanks to the *blocking* feature of the `showSolutionsWithBlocking()` function, path solutions are already sorted in root-to-leaf order. Since the input is sorted in order of the common prefix, the second phase is linear in the sum of its input and output. This is also a key feature of the efficiency of the TwigStack algorithm.

Compact stack encoding

The compact stack encoding is built in the first phase. `showSolutionsWithBlocking()` is a kind of *decoder*. During its execution, TwigStack algorithm might find a huge amount partial intermediate path that match the query. If all of those partial paths were stored individually, this intermediate storage could be bigger than input and output storage sizes.
The following example shows how a set of root-to-leaf path solution are encoded using stacks. In this example, the query is `//a//b//c`. A stack belongs to each node of the query. This stack stores the positional information retrieved from the inverted list (stream) corresponding to the node and a pointer to the parent stack element in the query.

```
<?xml version="1.0" encoding="UTF-8" ?>
<a>
        <a>
                <b>
                        <b>
                                <c />
                        </b>
                </b>
        </a>
</a>
```

XML document



Stack encoding of the result

01.	a1	b1	c1
02.	a1	b2	c1
03.	a2	b1	c1
04.	a2	b2	c1

Query result

4. Partitioning models for parallel XML query processing

Because of the *semistructured* nature of XML data, partitioning strategy for data parallelism is not an obvious process. At KDE laboratory, this issue has already been studied previously. The following thesis focuses on the metadata parallelization of XML documents in order to execute the Holistic Twig Joins algorithm in a cluster environment (shared-nothing memory). At the end, the thesis also evaluates multi-core CPU (shared memory) for parallel query processing.

A Study on Parallel Holistic Twig Joins for XML Query Processing
Imam Machdi
PhD thesis, March 2010

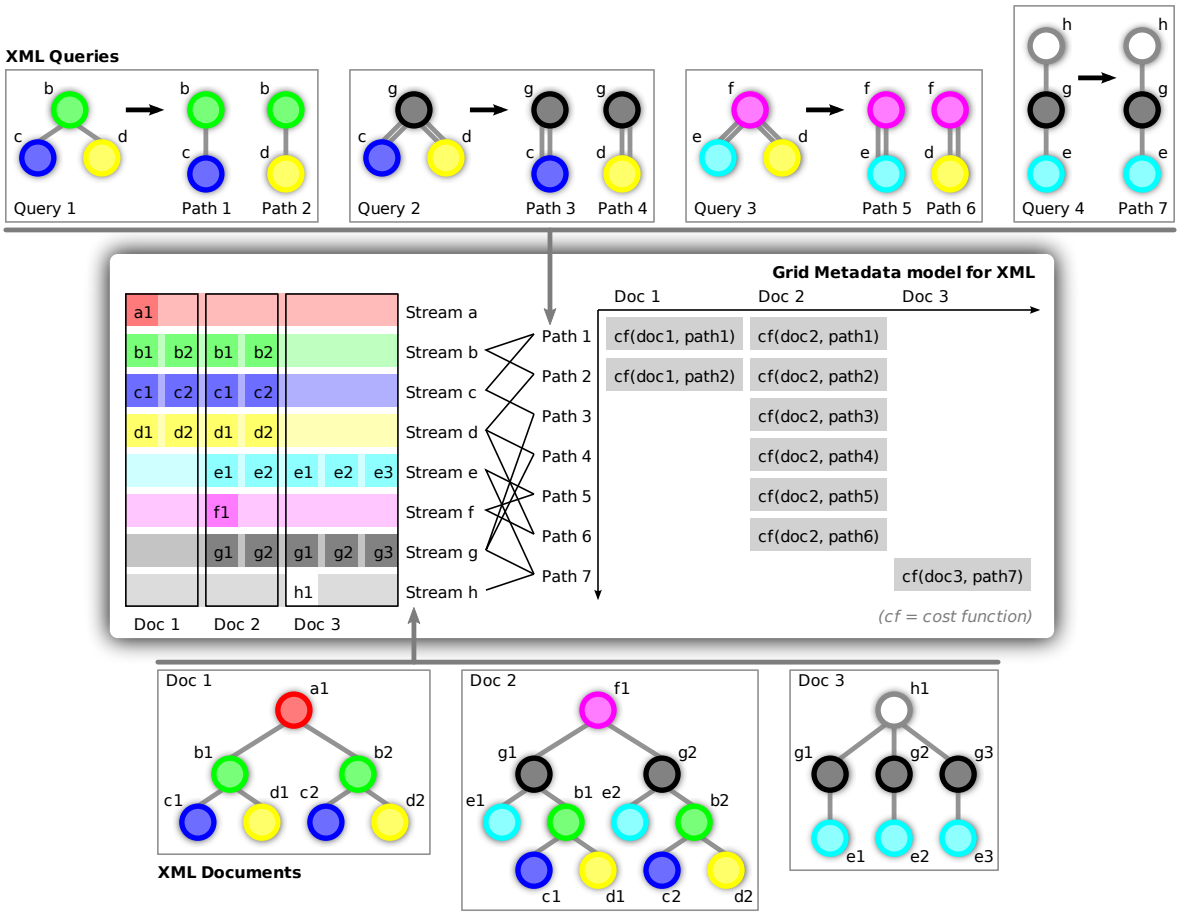
Two new models are introduced: the Grid Metadata model for XML (GMX, 4th chapter in the dissertation) and the Stream-based Partitioning for XML (SPX, 5th chapter in the dissertation). GMX model is refered as *static* partitioning while SPX model is refered as *dynamic* partitioning. This categorization may lead to misunderstanding if you do not keep in mind that static and dynamic refer to the cluster environment and not the query processing execution. None of the GMX and SPX models adapt partitioning *on-the-fly* while executing TwigStack algorithm.
One paper is named "*On-the-fly* Partitioning of XML Node Streams for Parallel Holistic Twig Joins". This paper introduces the SPX model as an *on-the-fly* process. Again, it should not be misunderstood: partitioning and query processing are not carried out in the same time *for the same query*. This process is *on-the-fly* from cluster node point of view since a node can partition and process a new incoming query without stopping the process of the whole set of queries (as opposed to GMX).
GMX model exploits the relationships between XML documents and query patterns to perform workload-aware partitioning of XML data. SPX model explores the structural relationships of query elements and a range-containment property of XML streams to generate partitions. Those both schemes were designed specifically for parallel holistic twig joins processing, thus streams of XML nodes are partitioned instead of XML documents.

Both proposed XML data partitioning schemes do not take into account the issue of XML data updates. In case of document change, whole partitioning has to be done again.
This section presents an overview of each model. The two models are complementary.

4.1 Grid Metadata model for XML

GMX model is divided into three main stages: generation of metadata, partitioning part and distribution part.

GMX model uses document and query metadata. Document metadata are generated from XML documents while query metadata are generated from query logs (i.e., previously executed queries). Document metadata consists of streams of XML nodes and query metadata consists of statistics: query occurence, estimated root-to-leaf output size and estimated final output size. The result of the GMX model is a 2-dimensional representation of the cost relationships between XML documents and queries.



Overview of constructing the grid metadata for XML (from GMX paper)

When metadata have been computed, partitioning stage performs clustering or refining of documents and queries according to the cost function and gathered metadata. The aim of making use of the cost function is to take into account the coherency between twig pattern queries and XML documents. The cost model is detailed in the thesis and will not be introduced in this document. Partitioning methods are inspired by OLAP-operations of relational data. They provide five different granularities.

The last stage is the distribution part. Partitions are allocated across the cluster nodes. Each node's workload should be balanced compared to others. The selected approach gives only sub-optimal workload balance. Actually, finding the optimal workload assignment is too costly. Heuristic functions are used to minimize workload variance. A threshold mechanism is implemented to find overloaded cluster nodes.

According to its author, *the [GMX] model has feature of reducing the workload variance significantly in cluster system, duplicating XML data necessarily to avoid data dependency among cluster nodes, and exploiting inter-query parallelism and intra-query parallelism.*

The thesis deals with *inter-* and *intra-* query parallelisms. In the example figure of the GMX model, we can see 4 queries on 3 documents. The 2-dimension division partitions the workload on a *per XML document* and a *per query path* basis. If the workload consists of a unique document and a unique query path, workload cannot be further partitioned through GMX model, then SPX model has to be used since this model allows better intra-query parallelism.

4.2 Streams-based Partitioning for XML

A workload imbalance can occur during query processing, despite GMX partitioning. SPX model has been introduced to cope with this issue. The aim here is to produce partitions each containing a subset of the query twig tree. The structure of the XML data drives the portioning process thus the method is also relevant for one XML document and one query only. Unlike GMX model, this model have no granularity limit. Like GMX model, this model produces partitions having no dependency among each other thereby duplicating some document metadata. Although the lack of granularity limit, there is still a trade-off to find because higher parallelism also creates bigger amount of redundant data which decreases the efficiency of each parallel process.
The SPX model is divided into two main stages: partitions generation and allocation of those partitions. Like for GMX model, the thesis provides an allocation plan based on a cost model. This part is not further detailed since it is dedicated to cluster environment that is outside the scope of this document.

A. Extension of the positional properties

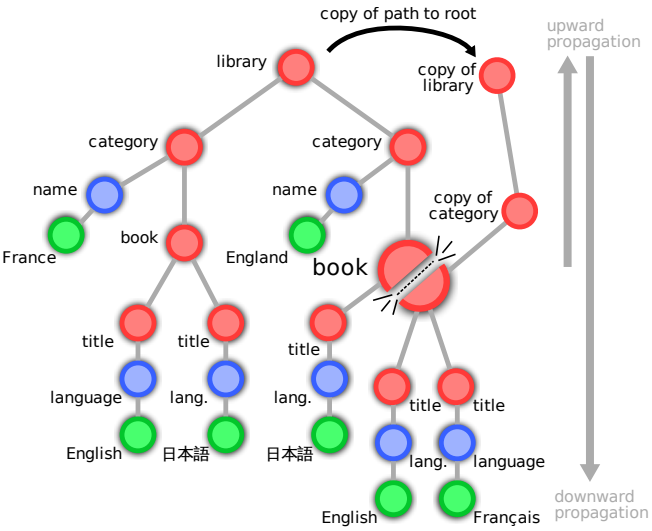
The containment properties defined in the work of Zhang et al. is extended with the notion of *left* and *right* containment. If D is a descendant of A, D is left contained in A is equal to this condition: $(doc\_no_A = doc\_no_D$ AND $left\_pos_A < left\_pos_D)$ OR $doc\_no_A < doc\_no_D$. A similar definition applies to right containment: $(doc\_no_A = doc\_no_D$ AND $right\_pos_A < right\_pos_D)$ OR $doc\_no_A < doc\_no_D$. Using those two properties, *most left* and *most right* containment can easily be defined. This is a key feature of the SPX partitioning model since it is used to respect range containment property among streams in the resulting partitions.

B. Overview of the model through an example

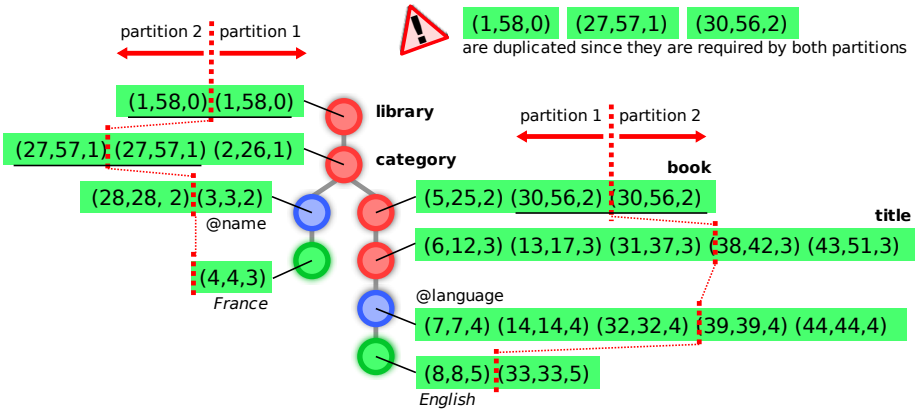
All the following figures are based on the same example. They show SPX partitioning using different representations.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- partition 1 -->
<library>
        <category name="France">
                <book>
                        <title language="English">The Little Prince</title>
                        <title language="日本語">星の王子さま</title>
                        <author>Antoine de Saint-Exupéry</author>
                </book>
        </category>
        <category name="England">
                <book>
                        <title language="日本語">ふしぎの国のアリス</title>
<!-- partition 2 -->
                        <title language="English">Alice in wonderland</title>
                        <title language="Français">Alice au pays des merveilles</title>
                        <author>Lewis Carroll</author>
                </book>
        </category>
</library>
```

SPX partitioning example: XML document as plain text



SPX partitioning example: XML document as a tree



SPX partitioning example: XML document as streams in the query

To start the partitioning, SPX model suggests beginning with the biggest stream. In the example, the two biggest streams are those bound to `title` and `@language` query nodes (they contain 5 elements). `title`'s stream was randomly chosen to undertake the initial split. Since we targeted two partitions, one contains three elements while the other contains two elements.
The resulting partitions are then propagated to all other streams of the query tree using the range containment property. For example, if the root node of the XML document is involved into the query, this node would be duplicated in all partitions (since there is always only one root element in an XML document, this is a restriction of the standard).

```
function upward(basePartition, stream)
    for streamPart in basePartition
            # look for the biggest range containment
            searchLeftContainment(first(streamPart), stream)
            mostLeft = nextMostL(stream)
            searchRightContainment(last(streamPart), stream)
            mostRight = nextMostR(stream)
            ancStreamPart = copyStream(stream, mostLeft, mostRight)
            ancStreamPartitions = ancStreamPartitions + ancStreamPart
    return ancStreamPartitions
end
```

```
function downward(basePartition, stream)
    if isDuplicate(basePartition)
            descStreamPartitions = equalPartitioning(stream, windowSize)
            return descStreamPartitions
    end if
    foreach streamPart in basePartition
            # look for the biggest range containment
            searchLeftContainment(first(streamPart), stream)
            mostLeft = nextMostL(stream)
            searchRightContainment(last(streamPart), stream)
            mostRight = nextMostR(stream)
            ancStreamPart = copyStream(stream, mostLeft, mostRight)
            ancStreamPartitions = ancStreamPartitions + ancStreamPart
    end foreach
    return ancStreamPartitions
end
```

Parts of the SPX algorithm

5. nVIDIA GPU architecture

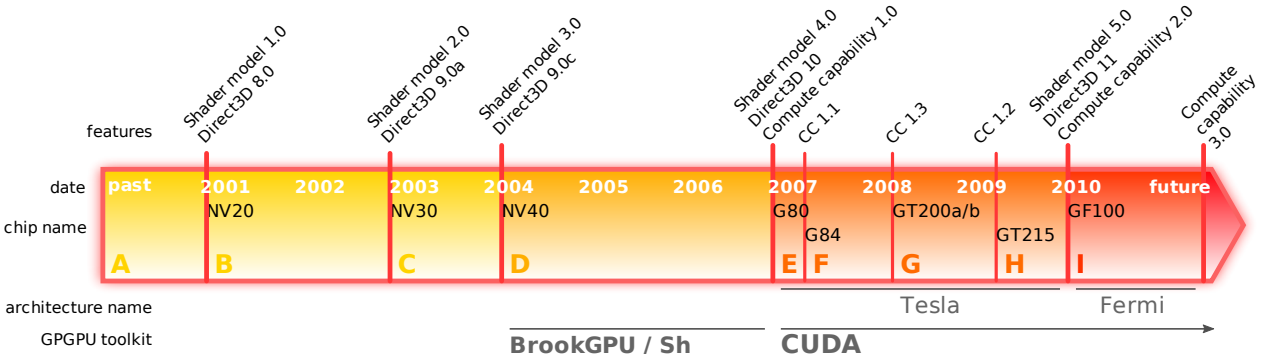
GPGPU (Many-Core) promises speedup that can reach an order of magnitude over current CPU (Multicore) architectures. GPU computing is as quite new phenomenon (as of 2010). Previously these processing units were dedicated to 2D/3D rendering and some specifically wired video acceleration. 3D rendering standard specifications (OpenGL and DirectX) included new features at each new version, especially about shaders capabilities, thus GPU became suitable for general purpose stream processing.

Memory accesses are among the slowest operations of a processor, due to the fact that Moore's law has increased instruction performance at a much greater rate than it has increased memory performance. This difference in performance increase rate means that memory operations have been getting expensive compared to simple register-to-register instructions. Modern CPUs sport large caches in order to reduce the overhead of these expensive memory accesses.
GPUs use another strategy so as to cope with this issue. Massive parallelism can "feed" the GPU with enough computational operations while waiting for pending memory operations to finish. This different execution strategy implies to look for new implementation ideas.
The GPU is especially well-suited to address problems that can be expressed as *data-parallel* computation (the same program is executed on many data elements in parallel) with high arithmetic intensity (the ratio of arithmetic operation to memory operations). This architecture was designed for image rendering (3D) and processing (video playback) but data-parallel processing can be also found in physics simulation, signal processing, computational finance or biology. An algorithm that is *data-parallel* is also referred as *embarrassingly parallel*. Those algorithms can be accelerated radically using GPU.

Since GPU implementation is still highly dependant to the underlying hardware, the purpose of this chapter is to show the evolution of the graphic hardware which explains its current limitations.

5.1 Evolution of the hardware architecture

This section will explain what happened to graphic processing units with an eye to become a suitable architecture for general purpose parallel processing.
The graphics adapter in PC compatibles computers started from a simple memory-mapped frame buffer. They became devices with 2D and 3D hardware acceleration. These devices contain their own memory for the frame buffer. This memory has two accesses: computer system can read/write picture into it and video out hardware reads it in order to create a signal for the display device(s).
A fast historical review is showed in the following timeline figure (evolution of Direct3D pipeline is based on [THOMSON06]):



Evolution of the NVidia GPU architecture to GPGPU

A. The first generations of nVIDIA GPU chips are based on a 3D graphics pipeline and a set of fixed-functions. Those fixed-functions can be selected by the host and are executed by hardware.
`NV1` is the first product of nVIDIA (released in 1995) and uses its own homemade graphic pipeline based on quadratic surfaces.
`NV3` is the second product of nVIDIA (1997) and uses a totally different pipeline based on polygon surfaces so that it can match Microsoft DirectX 5.0 requirements.
`NV10` is the third generation of nVIDIA GPU (1999). New fixed-functions have been added for DirectX 7.0 compliance, such as hardware Transform and Lighting (T&L) functions.

B. Each new version of DirectX introduces new fixed-functions requirements. The explosion of combinations of a fixed-function model became unwieldy therefore DirectX 8.0 introduced vertex and pixel shader models. This feature allows developers to create arbitrary programs to execute per-vertex or per-pixel. Since pixel shader (also called fragment shader) does not provide any branching or looping features in any version of the instruction set, this paragraph will focus on vertex shaders.

	1.1	2.0	2.x	3.0
instruction count	128	256	≥ 256	≥ 512

Maximum instruction count according to the shader model

A vertex shader is a program for "*a vector-oriented CPU with an instruction set and sets of registers used to carry out the instructions*". These tiny programs are written in a specific RISC-oriented assembly language. Some fixed-function elements can be implemented via vertex shader, but the purpose of this model is to produce results that are impossible to get through the fixed-function pipeline.
Vertex shader model 1.1 introduced by Direct3D 8.0 is the simplest architecture. The instruction sets contains operator for declarations, basic arithmetic, matrix arithmetic, simple comparison and basic lighting calculations. It provides no operators for conditional branching or flow control. There are several kind of registers: input, constant, output and temporary. Registers store 4 dimensional vector value of single precision floating-point number (approx. 6 decimal digits). Temporary registers provide a vertex shader with a small scratchpad for storing intermediate results.

C. The core runtime introduced by DirectX 9.0 contained significant enhancements to the shader models, beyond of what new GPU were capable at the time DirectX 9.0 was released. This may explain the a, b and c differentiation of this version.
Vertex shader model 2.0 allows integer and boolean 4 dimensional vector values in constant registers, but the main improvement is the addition of static flow control to the execution model. Subroutines, branching and looping instructions are appended to the instruction set, but the usage of these new instructions has many limitations such as static condition only. *In static flow control, all the conditional expressions for evaluating branch points refer to values that remain constant for the duration of the shader.* Additionally *loops execute a fixed number of times and conditional execution always follows the same path far all primitives drawn with the same set of constants. Different batches of primitives can have different flow control behavior by changing the constants between batches.* New constant registers (loop counter register) are provided for defining the constants used for the flow control.

	2.0	2.x	3.0
call nesting	1	1-4	4
static conditions	16	16	24
dynamic conditions	n/a	0-24	24
loop nesting	1	1-4	4
static flow count	16	16	∞

Flow control limitation according to the shader model

Vertex shader model 2.x is an intermediate execution architecture which provides some optional improvements. The most important new features from GPGPU point of view are predication, deeper nesting of static flow control instructions and dynamic flow control instructions. *Predication is a form of conditional execution that can be applied to individual instructions without branching*, for that purpose new predicate register is created for conditional flow control.

D. **Vertex shader model 3.0** relaxes architectural limits of previous models (especially looping possibilities). This model revision is still fully oriented toward graphic rendering. Some projects got through the numerous development difficulties and cheated the rendering pipeline so as to use the GPU as a stream processor. Brook and LibSH are most known examples of such projects [BUCK04], [LIBSH].
The Brook framework was developed since the beginning as a language for streaming processing. The language supports several dedicated streaming processors. When GPUs have been capable of stream processing too, the Brook language has been adapted to the capabilities of graphics hardware such as ATI Radeon X800XT and nVIDIA GeForce 6800 (NV40, DirectX 9.0c). Since the language maps to several streaming architectures, it is free of any explicit graphics constructs unlike other high-level shader languages such as Cg/HLSL and GLslang.

E. **Compute capability 1.0** refers to the generalization of the vertex shader as general purpose program of stream processing (this is an nVIDIA specific term). The nVIDIA CUDA toolkit was released simultaneously and made GPGPU computing more straightforward. From DirectX 10 point of view: this new revision introduces the unification of vertex, geometry and fragment processing into unified shader. DirectX 10 compliant GPU are definitely ready for general purpose parallel computation.

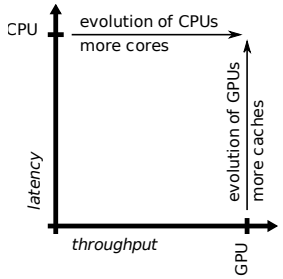
F. **Compute capability 1.1** and beyond are not coupled to DirectX requirements anymore. The main improvement is the availability of atomic functions. For instance, this feature is especially important for the histogram CUDA example.

G. **Compute capability 1.3** improves atomic functions possibilities and double-precision unit. Some of the limits of the Tesla architecture are revised (register memory, maximum number of warps and threads per SM).

H. **Compute capability 1.2** is not a mistake: CC1.3 was released before CC1.2. Actually the main difference between the two compute capabilities is the lack of double-precision unit in 1.2-capable GPU.

I. **Compute capability 2.0** corresponds to the Fermi architecture. GPGPU features of the nVIDIA GPU architectures undertook big efficiency/programmability changes. New threads synchronization functions are added.

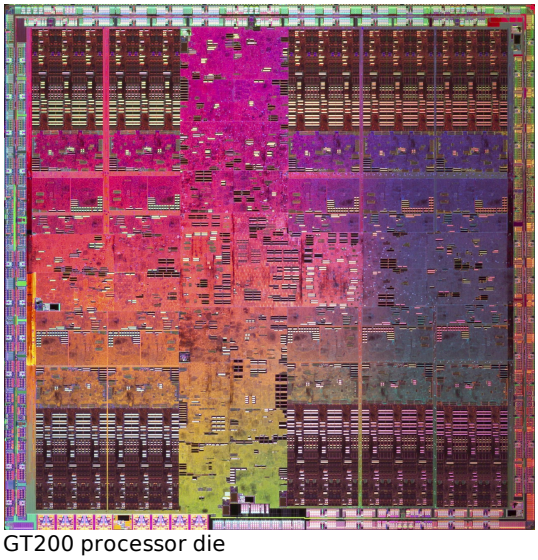
The "Little law" from Sylvain Collange: *data* = *throughput* × *latency*
The evolution of GPU capability toward more general purpose computation and the evolution of CPU capability toward more parallel and specific computation creates figure found beside: a convergence of CPU and GPU.
CPUs have good *latency* thanks to wide and coherent caches while GPUs have good *throughput* thanks to massive and hardware-handled parallelism.



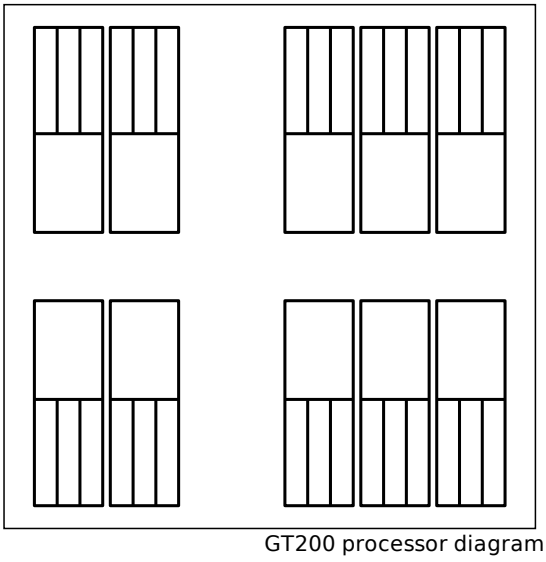
GPU-CPU convergence?

5.2 Tesla hardware architecture

Telsa is the product name of the first GPGPU-only device made by nVIDIA. It also became the name of the first CUDA-compliant GPU architecture. All GPU having compute capability from 1.0 to 1.3 belong to the Tesla architecture despite using other product name than Tesla (i.e. GeForce, Quadro).



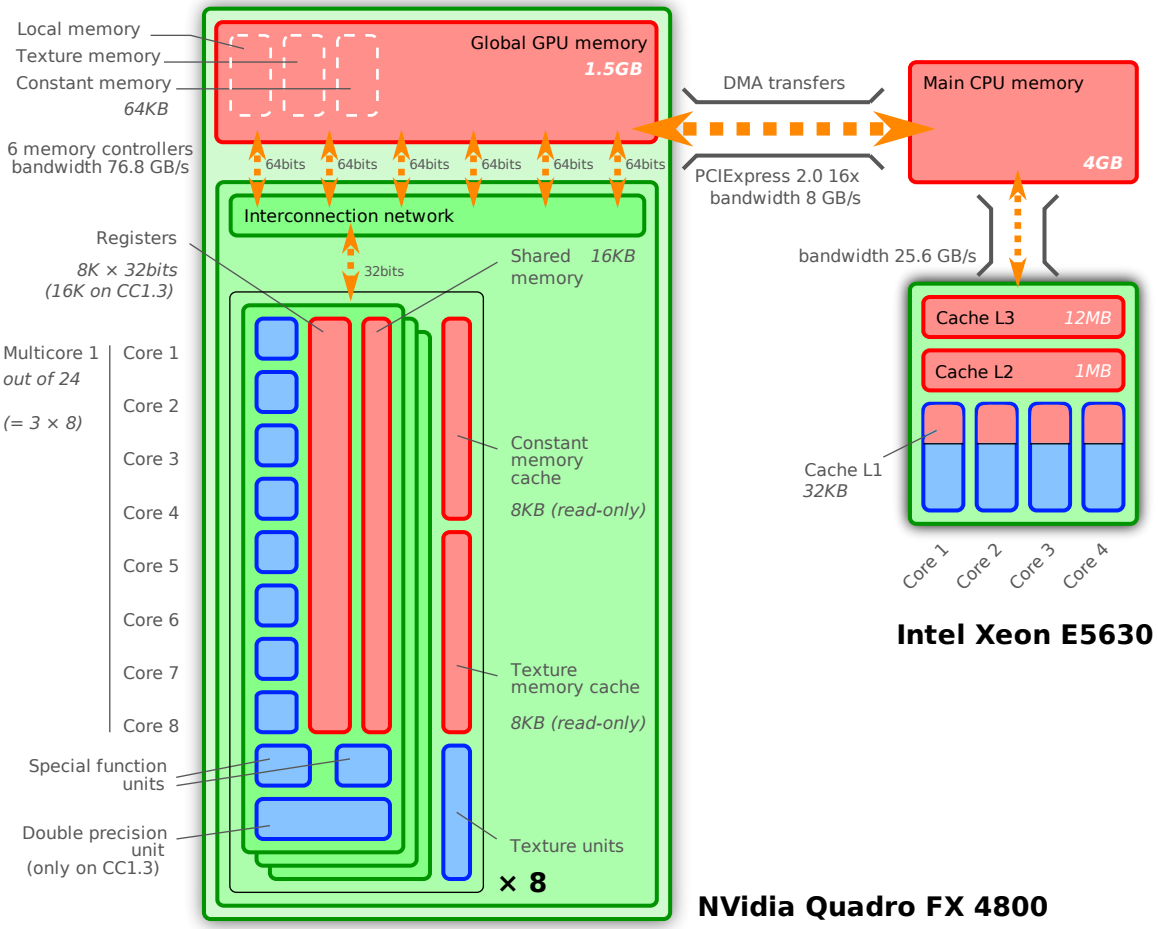
GT200 processor die



GT200 processor diagram

Hardware architecture of the GT200 chip

Unlike current CPU dies, GPU dies do not feature big cache memory areas. Most of the transistors are used for computation. Processing units are divided and nested into several groups and subgroups. From die point of view, the main division of the Tesla architecture is the *Thread Processing Clusters* (TPC) [WONG10]. Each TPC is linked to an interconnection network (probably of *crossbar* type according to [COLLANGE10]) which also links DRAM memory controllers. Memory is partitioned into 1 to 8 controllers. Memory chips are found outside the GPU chip. Each TPC contains 2 or 3 *stream processors* (SM, also named *multiprocessors*).



Example hardware of nVIDIA Tesla GPU architecture

Using the GPU from host for general purpose computation is generally performed through three steps: data copy from host to device memory, execution on device, data copy from device to host. This reveals the first issue of GPGPU computing: *the time wasted in memory transfers*.

A GPU is connected to the host through a high-speed bus such as PCI-Express 16x. This bus is mainly used for DMA transfers between CPU and GPU memories since none of them is able to directly address the memory space of the other.
There are several options for data transfer:

Paged memory
: When memory is allocated through `malloc()` function, the memory is paged. This mechanism allows the OS to allocate more memory than physically available. *Pages* of memory are swapped on-the-fly between hard drive and main host memory. When performing a memory copy between host and device, device will have to poll the CPU to know when the memory transfer has ended (source [FARIAS]).

Page-locked memory (aka pinned memory)
: PL memory is opposite of *pageable* memory. Disabling pageable memory feature lower the available CPU memory, but has some benefits: memory transfers can be performed concurrently with program execution on GPU (avoiding polling). Additionally, memory transfers are faster if the system uses a frontside bus (FSB) according to [CUDA32]. Please note that not all GPU support this feature.

Mapped memory (aka zero-copy memory)
: Since the compute capability 1.1, page-locked memory can be mapped into device address space. This memory region can be addressed by host and device using different addresses. This method is called *zero-copy* because data transfers between host and device are implicitly performed on-the-fly.

Write-combining
: WC improves host-to-device write performance. Individual small writes are merged into one greater write transaction (burst). Defining a memory region as WC memory has two main drawbacks: WC does not guarantee that the combination of writes and reads is performed in the correct order. Reading from WC memory from the host is much slower than *cacheable* memory. This comes from the fact that writes to WC memory are delayed in an internal buffer and this buffer is neither cached (i.e., slow read) nor snooped (i.e., no data coherency).
This option is available since Intel P6 family processor [WCOMB].

When data and program has been loaded into GPU memory, execution can start since the number of stream

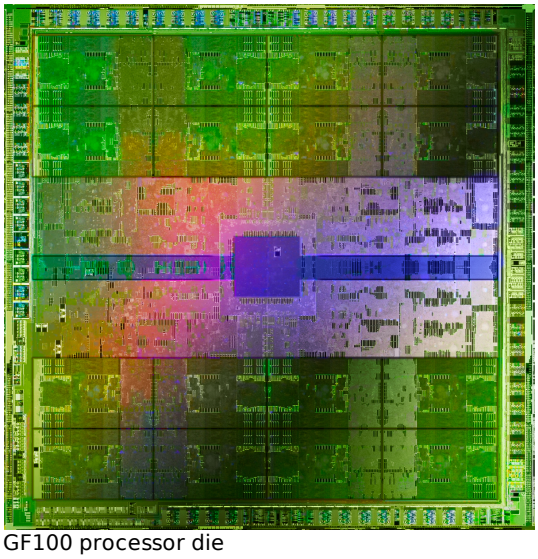
multiprocessors and thread per SM is decided in advance. On the Tesla architecture, a SM can have 768 or 1024 threads simultaneously active according to the chip. The number of SM depends on the GPU: entry-level GPU has a few SM while high-end GPU has a lot of them. The GPU of the example has 24 multiprocessors therefore it can deal with 24576 active threads. Of course, all threads are not executed in the same time since a multiprocessor has only eight cores, so-called CUDA cores. Following the same example, 192 threads are executed at each clock cycle. In the worst case, 128 cycles are required to execute one instruction of all threads. The GPU can perform zero-cost hardware-based context switching and immediately switch to another thread to process.

A big difference between CPU and GPU is the memory hierarchy. Registers and shared memory are extremely fast while global memory is several hundred times slower. It is worth noting that the shared memory is not a hardware cache, but is a *scratchpad* memory [W-SCRPAD]. Each SM has a such local high-speed internal memory. It can be seen as a L1 cache, but there are crucial differences: explicit instruction are required to move data from global memory to shared memory and there is no coherency among scratchpads and global memory. When used as a cache memory, if global memory changes scratchpad's content will not be updated. Shared memory is considered as fast as register memory as long as there are no *bank conflicts* among threads.
Global memory is linked to the GPU chip through a very large data path: up to 512-bits wide. Through a such bus width, sixteen consecutive 32-bits words can be fetched from global memory in a single cycle. The Quadro FX4800 of my running example has a 384-bits bus width and GDDR3 memory at 1.6 GHz allowing a theoretical 76.8 GB/s memory bandwidth. In real programs, this bandwidth is difficult to obtain since memory has to be aligned and threads accesses have to be *coalesced*. Coalescing allows to merge all independent thread memory access into one big access. Several thread access patterns are recognized by coalescing hardware. It also means there is severe bandwidth degradation for stridden accesses.

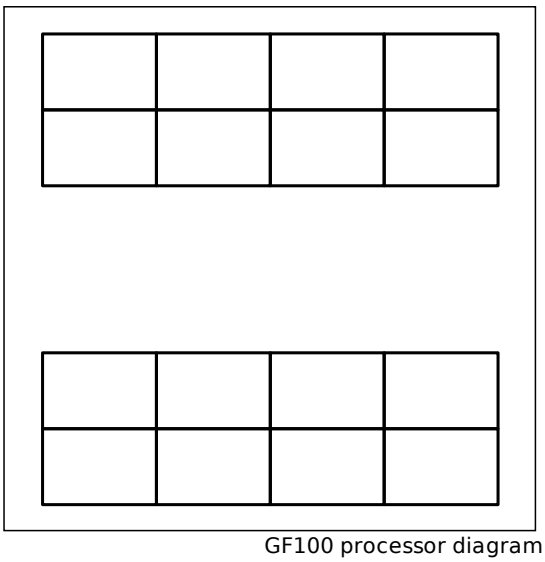
CUDA cores can handle integer and single-precision floating operations. When a thread encounters a transcendental and double precision operations, CUDA cores cannot be used. There are two *special function* units for transcendental operations and one double precision unit for double precision operation. Transcendental operation are, for example, sine, cosine or logarithm instructions. SFU also compute integer multiplication when 32-bits precision is required (instead of 24-bits). Obviously, in a such case, the SM cannot perform more than two or one operation per cycle. Tesla generation GPU which support double-precision are eight times slower in double- than single-precision [COLLANGE10]. According to nVIDIA documentation [GTX200], "*double-precision performance of all 10 TPCs of a GeForce GTX 280 GPU is roughly equivalent to an eight-core Intel Xeon CPU, yielding up to 78 gigaflops*".

5.3 Fermi hardware architecture

While the product name of CUDA-only device remained Tesla, its latest architecture's name changed to Fermi. All GPU having compute capability from 2.0 and higher (and below 3.0 probably) are based on the Fermi architecture.



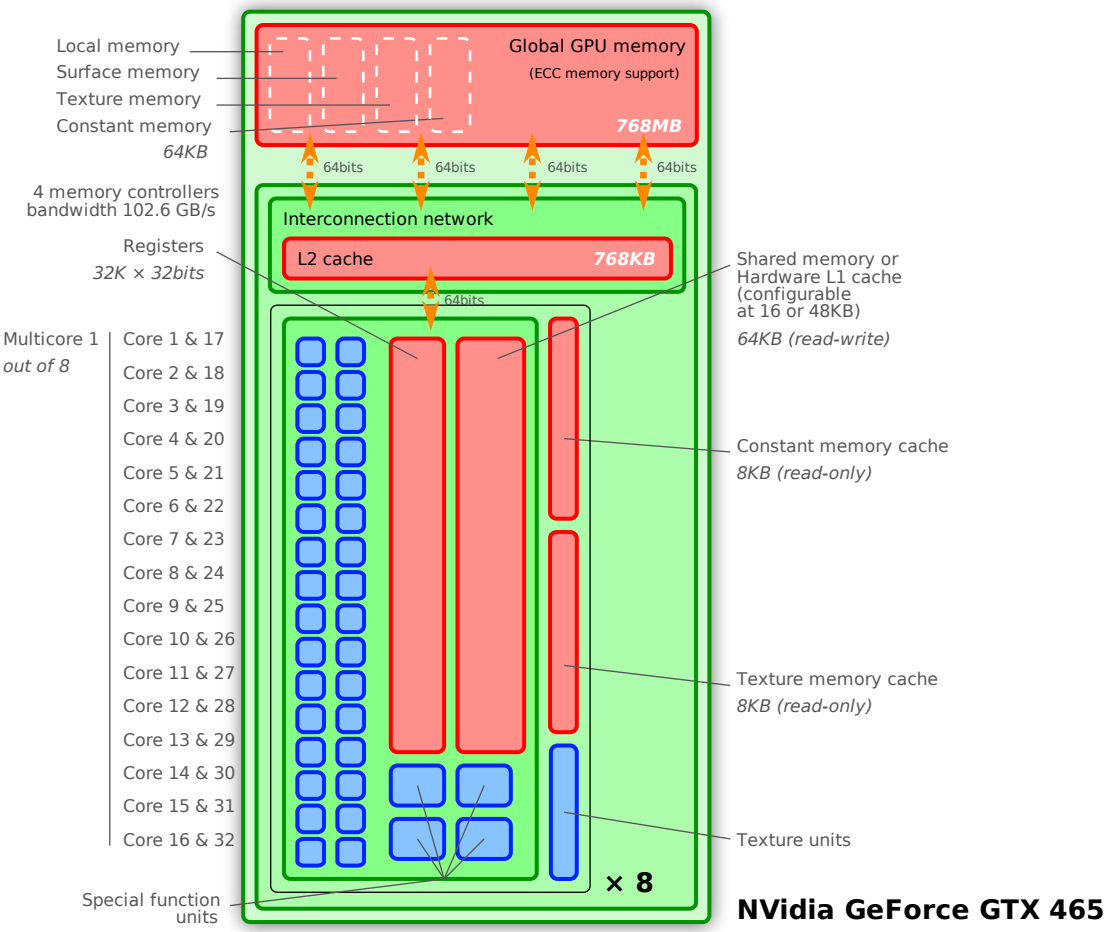
GF100 processor die



GF100 processor diagram

Hardware architecture of the GF100 chip

Fermi architecture introduces a 768KB fully coherent L2 cache common to all SM (referred as unified cache). This new cache has the same purpose as CPU L2 cache and is a significant change from previous architecture. This key feature shows a real general purpose orientation of the nVIDIA GPU. This cache does not accelerate graphics computation.
Another *non-graphical* feature is that all memories (from register to DRAM) can be protected by ECC (Error-Correcting Code).
Thread Processing Clusters division disappeared.



Hardware of nVIDIA Fermi GPU architecture

Unified address space enables full C++ support [FERMI]. In Tesla architecture, linear memory was addressed using a 32-bit address space. Load/store instructions were specific to different memory area. Fermi architecture uses a 40-bit unified address space. Load/store instructions can use 64-bit address width for future growth.

The previous shared memory space became a configurable scratchpad/L1 cache. Fermi architecture has two configuration option: 48 KB of shared memory and 16 KB of L1 cache or 16 KB of shared memory and 48 KB of cache.

This new architecture upgrades memory from GDDR3 to GDDR5. Despite its narrower memory bus width (256 *vs* 384-bit), the card of the figure has a theoretical memory bandwidth of 102.6 GB/s.

Devices of compute capability 2.x can perform a copy from page-locked host memory to device memory concurrently with a copy from device memory to page-locked host memory because GPUs of the Fermi architecture have 2 copy-engines. Since PCIexpess bus is duplex, total bandwidth is doubled in a such case.

5.4 Software mapping

CUDA stands for *Compute Unified Device Architecture*. CUDA is an extension of the C language for GPGPU computing. The programming model is tightly coupled with the architecture of nVIDIA graphic cards. Concept of the programming model can be mapped to hardware implementation.

The code example on the right will be used to show the mapping between CUDA programming model and GPU hardware execution model. This example also shows how a part of a software can be accelerated on GPGPU using the CUDA toolkit.
From line 1 to 5, there is the GPU code (aka. device). Everything else is executed on CPU (aka. host). `parallelAdd()` function is an ad hoc replacement of the `serialAdd()` function. `parallelAdd()`

```
1  /* GPU kernel */
2  __global__ void vecAdd(float* a, float* b, float* c) {
3      int i = blockDim.x * blockIdx.x + threadIdx.x;
4      c[i] = a[i] + b[i];
5  }
6
7  /* function relying on GPU */
8  void parallelAdd(int size, float* h_a, float* h_b, float* h_c) {
9      size_t vector_byte_size = size * sizeof(float);
10     float *d_a, *d_b, *d_c;
11
12     /* STEP 1: memory allocation for GPU exec. */
13     /* allocate some GPU memory and copy input vectors */
14     cudaMalloc(&d_a, vector_byte_size);
```

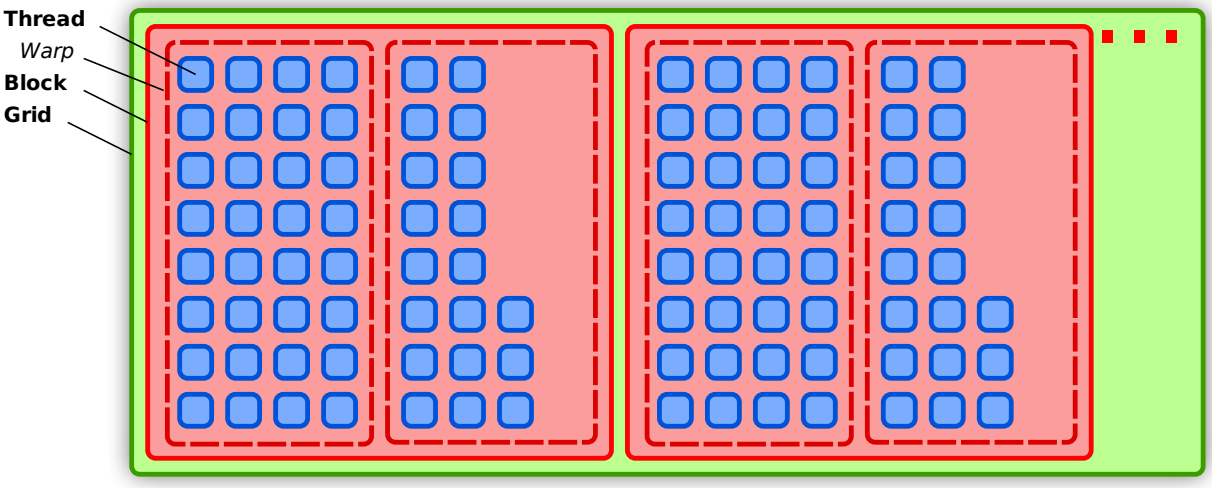
deals with all GPU specificities. As explained previously, the execution is performed through three steps. In the first step, `cudaMalloc()` and `cudaMemcpy()` allocate and load the required input data in GPU global memory. In this `parallelAdd()` function, there are both host (start with `h`) and device (start with `d`) pointers. The second step is *kernel* call. A kernel is a program executed on GPU. The calling syntax use a CUDA specific notation: `<<<2, size/2>>>` specifies the number of block (2) and the number of thread per block (`size/2`).

Like processing units in hardware, threads are also divided into groups. The following schema shows one grouping possibility. As for *Grid* and *Block* sizes, they are defined by developer at *Kernel* execution. *Warp* size is fixed by hardware and is 32 for both Tesla and Fermi, but this size might change in future architectures according to nVIDIA CUDA documentation.

Grouping possibilities for 102 threads are `<<<1, 102>>>`, `<<<2, 51>>>`, `<<<3, 34>>>` and `<<<6, 17>>>`. Since the basic unit of execution flow in a multiprocessor is a warp of 32 thread, it is useless to execute less that 32 threads in a block. Actually, if there was a condition so as to prevent from going out of bound of the vector arrays when the thread number exceed vector size, we could use any `<<<x, ⌈size/x⌉>>>`.

```
15      cudaMalloc(&d_b, vector_byte_size);
16      cudaMalloc(&d_c, vector_byte_size);
17      cudaMemcpy(d_a, h_a, vector_byte_size, cudaMemcpyHostToDevice);
18      cudaMemcpy(d_b, h_b, vector_byte_size, cudaMemcpyHostToDevice);
19
20      /* STEP 2: execution on device */
21      vecAdd<<<2,size/2>>>(d_a, d_b, d_c);
22
23      /* STEP 3: results retrieval */
24      /* copy GPU result to CPU memory and free GPU memory */
25      cudaMemcpy(h_c, d_c, vector_byte_size, cudaMemcpyDeviceToHost);
26      cudaFree(d_a);
27      cudaFree(d_b);
28      cudaFree(d_c);
29  }
30
31  /* function relying on CPU */
32  void serialAdd(int size, float* a, float* b, float* c) {
33      int i;
34      for(i=0; i<size; i++) {
35              c[i] = a[i] + b[i];
36      }
37  }
38
39  int main()
40  {
41      int vector_size = 102;
42      size_t vector_byte_size = vector_size * sizeof(float);
43      float *a, *b, *c;
44
45      a = (float*)malloc(vector_byte_size);
46      b = (float*)malloc(vector_byte_size);
47      c = (float*)malloc(vector_byte_size);
48
49      /* put some values into a and b vectors */
50      [...]
51
52      /* serialAdd(vector_size, a, b, c); */
53      parallelAdd(vector_size, a, b, c);
54
55      /* now c = a + b and c can be used */
56      [...]
57
58      free(a);
59      free(b);
60      free(c);
61
62      return 0;
63  }
```

Example of CUDA integration



Example of 102 threads divided into 2 blocks

The main difference between Tesla and Fermi architecture is in the way they execute warps of threads.
On a Tesla, the eight cores execute a warp in four clock cycles (8 by 8). A Fermi multiprocessor, each group of 16 cores execute a different warp. Therefore, two warps are executed in two clock cycles (2×16 by 2×16).

A question about *divergent* threads rises from this execution strategy: *What happens when threads do not execute the same code in a warp?*.
In the following example, execution path depends on the thread id. This case is handled differently on Tesla and Fermi.

```
1  /* GPU kernel */
2  __global__ void vecAdd(float* a, float* b, float* c) {
3      int i = blockDim.x * blockIdx.x + threadIdx.x;
4
5      if(i & 1) { /* if i is odd */
6              c[i] = a[i] + b[i];
7      } else {
8              c[i] = a[i] - b[i];
9      }
10 }
```

A kernel with divergent threads

On the Tesla architecture, each conditional branch is serialized. According to [WONG10], "else clause" is always executed first while other clauses are disabled, then "if clause" is executed (and "else clause" disabled).
Fermi architecture improves this issue because each multiprocessor features a dual warp scheduler. Each group of 16 cores can execute a different conditional branch. Our divergent thread example would be executed in parallel on Fermi architecture only. Of course, it works for half-warps only.

CUDA compiler allocates registers memory to threads. If the threads requires too many registers, *local* memory is used. Actually local memory does not exist in the hardware. Local memory is the name given to some global memory which was made private to a thread. This memory is extremely slow compared to register or shared memory, thus exceeding the maximum register memory lead to dramatically slow performances.

On devices of compute capability 2.x, function call is available. The size of the call stack can be queried using `cudaThreadGetLimit()` and set using `cudaThreadSetLimit()`.

A. CPU thread *vs* GPU thread

Despite using the same name, the word *thread* has a different definition on CPU or on GPU and can lead to misunderstanding. Unlike in CPU, GPU threads are managed by hardware. Classical thread programming techniques do not match GPU thread design. Translation from POSIX thread (Linux) or Windows thread models to CUDA thread is not obvious nor recommended. I quote this analogy since I think it helped me get the difference more clearly "the definition that applies to CUDA is *threads in a fabric running lengthwise*" [PPBLOG]. In other words, CUDA threads should not diverge for optimal performances. Divergent threads are not impossible to implement, but they can dramatically lower the performances.

Single-Instruction Multiple-Thread (SIMT) is the name given by nVIDIA to this execution strategy. Every thread in the same warp execute the same instruction in lockstep, but all threads can branch separately although it would lead to extremely bad performances even on the Fermi architecture.

B. What about *memory consistency*?

Both Tesla and Fermi architectures have several memory areas having different memory consistency models. CUDA toolkit is very *system-centric* (according to [GHARACH95] definition) and memory management can quickly become a real nightmare for programmers.

A *state space* is a storage area with particular characteristics.

name	addressable	access	cost	lifetime	description
.reg	No	R/W	0	thread	registers
.sreg	No	RO	0	thread	special registers (platform-specific)
.const	Yes	RO	0*	application	constant memory
.global	Yes	R/W	> 100	application	global memory
.local	Yes	R/W	> 100	thread	local memory (private to each thread)
.param	Yes	RO	0		user parameters for a program
.shared	Yes	R/W	0	block	shared memory
.surf	via surface instruction	R/W	> 100	application	surface memory (global memory)
.tex	via texture instruction	RO	> 100	application	texture memory (global memory)

State spaces as defined in PTX 1.4 [PTX14]
* constant memory cost is amortized

Each addressable state space has its own address space. It means that memory load instruction is different according to the memory space. For each dereferencing pointer code, nVIDIA compiler infers at compilation time which memory space is the right space. This process may fail like in the example below.

```
 5  __device__ void myCopyFunction(void* dest_ptr,
void* src_ptr, size_t size) {
 6      char* dest = (char*)dest_ptr;
 7      char* src = (char*)src_ptr;
 8
 9      while(size-- > 0) {
10              *dest++ = *src++;
11      }
12  }
13
14  __global__ void testKernel1(int* arg, void* mem)
15  {
16      int temp = 20;
17      *(void**)mem = (void*)&temp;
18  }
19
20  __global__ void testKernel2(int* arg, void* mem)
21  {
22      myCopyFunction(arg, *(void**)mem, sizeof(int));
23  }
```

```
46  $LDWbegin__Z11testKernel2PiPv:
47      .loc    17      6       0
48      ld.param.u64    %rd1,
[__cudaparm__Z11testKernel2PiPv_arg];
49      .loc    17      7       0
50      ld.param.u64    %rd2,
[__cudaparm__Z11testKernel2PiPv_mem];
51      ld.global.u64   %rd3, [%rd2+0];
52      mov.s64         %rd4, 3;
53  $Lt_1_1794:
54   //<loop> Loop body line 7, nesting depth: 1,
 estimated iterations: unknown
55      .loc    17      10      0
56      add.u64         %rd3, %rd3, 1;
57      add.u64         %rd1, %rd1, 1;
58      ld.global.s8    %rh1, [%rd3+-1];
59      st.global.s8    [%rd1+-1], %rh1;
60      .loc    17      9       0
61      sub.u64         %rd4, %rd4, 1;
62      mov.u64         %rd5, -1;
63      setp.ne.u64     %p1, %rd4, %rd5;
64      @%p1 bra        $Lt_1_1794;
65      .loc    17      23      0
66      exit;
67  $LDWend__Z11testKernel2PiPv:
68      } // _Z11testKernel2PiPv
```

Comparison between CUDA source and PTX 1.4

The following warning is generated at compilation time:
`./test.cu(10): Warning: Cannot tell what pointer points to, assuming global memory space`
The `$Lt_1_1794` token at line 53 of test.ptx shows the beginning of the while loop of the inlined `myCopyFunction()` function. From line 56 to 59, there is the body of the loop and from 61 to 63 there is the looping condition. `@%p1 bra $Lt_1_1794;` is a conditional branch instruction because `@%p1` denotes a condition on the register `%p1`.

The faulty instruction is at line 58: `ld.global.s8`. As the compiler warned, it assumed global memory. The good instruction would have been `ld.local.s8`. This problem is solved in the Fermi architecture thanks to unified pointers. The following example is the same as previous one except the PTX2 target.

```
 5  __device__ void myCopyFunction(void* dest_ptr,
void* src_ptr, size_t size) {
 6      char* dest = (char*)dest_ptr;
 7      char* src = (char*)src_ptr;
 8
 9      while(size-- > 0) {
10              *dest++ = *src++;
11      }
12  }
13
14  __global__ void testKernel1(int* arg, void* mem)
15  {
16      int temp = 20;
17      *(void**)mem = (void*)&temp;
18  }
19
20  __global__ void testKernel2(int* arg, void* mem)
21  {
22      myCopyFunction(arg, *(void**)mem, sizeof(int));
23  }
```

```
 88  $LDWbegin__Z11testKernel2PiPv:
 89      .loc    17      6       0
 90      ld.param.u64    %rd1,
[__cudaparm__Z11testKernel2PiPv_arg];
 91      .loc    17      7       0
 92      ld.param.u64    %rd2,
[__cudaparm__Z11testKernel2PiPv_mem];
 93      ldu.global.u64  %rd3, [%rd2+0];
 94      mov.s64         %rd4, 3;
 95  $Lt_2_1794:
 96   //<loop> Loop body line 7, nesting depth: 1,
 estimated iterations: unknown
 97      .loc    17      10      0
 98      add.u64         %rd3, %rd3, 1;
 99      add.u64         %rd1, %rd1, 1;
100      ld.s8   %r1, [%rd3+-1];
101      st.global.s8    [%rd1+-1], %r1;
102      .loc    17      9       0
103      sub.u64         %rd4, %rd4, 1;
104      mov.u64         %rd5, -1;
105      setp.ne.u64     %p1, %rd4, %rd5;
106      @%p1 bra        $Lt_2_1794;
107      .loc    17      23      0
108      exit;
109  $LDWend__Z11testKernel2PiPv:
110      } // _Z11testKernel2PiPv
```

Comparison between CUDA source and PTX 2.1

In the PTX 2.1 version (see [PTX21]), at line 100, the previous `ld.global.s8` was changed into `ld.s8` and the compiler does not complain anymore about the pointer.

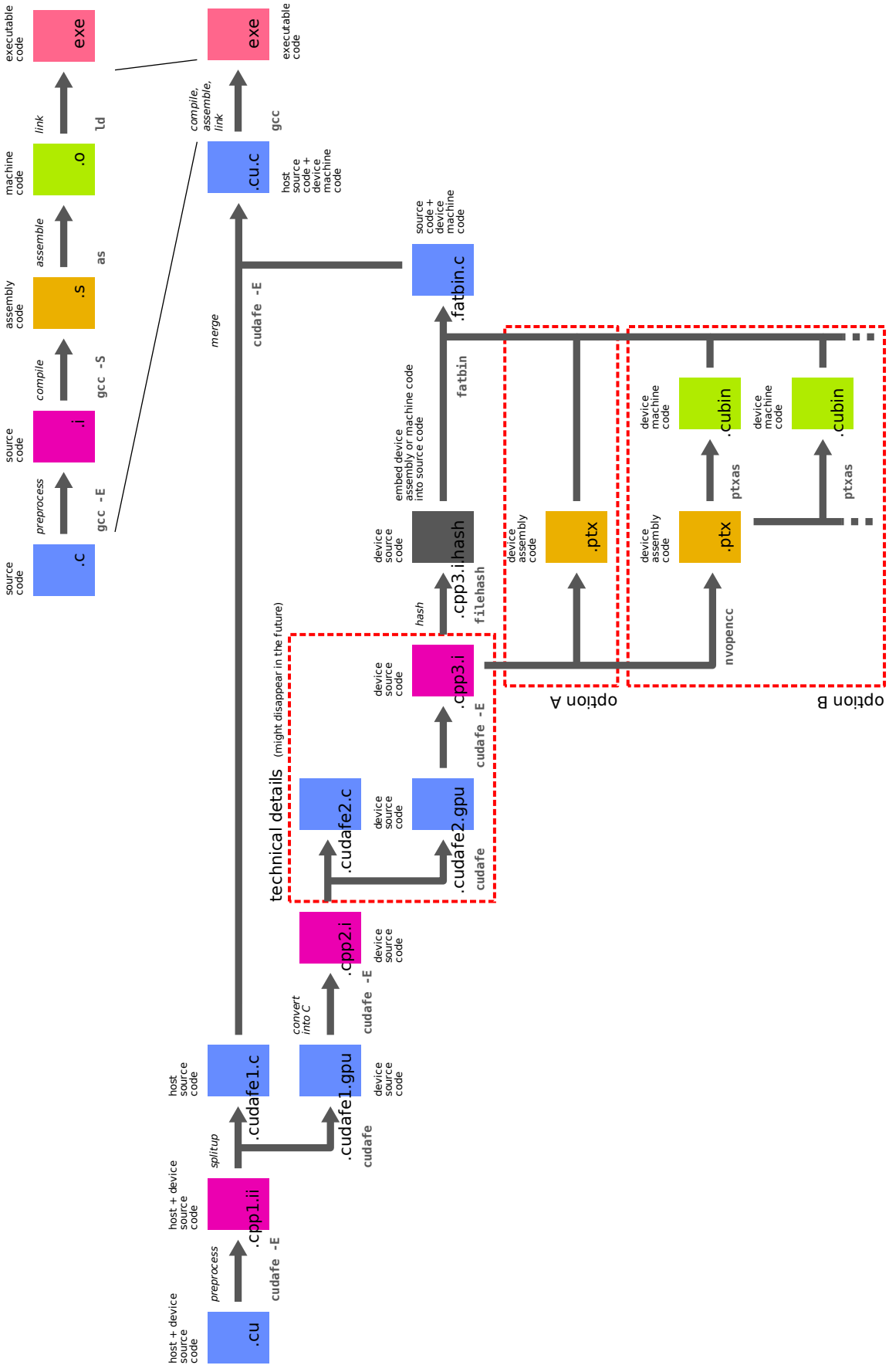
6. Development environment for multi platform CUDA software

The CUDA GPU computing development framework is available for three operating systems: *Microsoft Windows*, *GNU/Linux* and *MacOS X*. The last two share the same UNIX-like architecture thus CUDA toolkit is quite similar on both of them. The Nvidia CUDA buildchain falls back on the default compiler available from the operating system for host compilation (CPU). GNU Compiler Collection (`gcc`), Microsoft Visual Studio compiler (`cl`) or Intel C++ Compiler (`icc`) can be used by *nvcc*. The first CUDA SDK released to the public was the 1.1 Beta version in June 2007. At the time of writing, the latest version is 3.1, but the compilation workflow remained the same, despite the numerous improvements of the toolkit.

Before describing each step of building a CUDA software, let me remind the main stages of building any C/C++ software. To start off, preprocessing stage matches some text string and replaces them by others according to *macros* rules. Then, compilation stage translates the source code into assembly code. Next, assembly code is converted into machine code. Finally, the linking stage creates a connection to the operating system for primitives. This includes adding the runtime library, which mainly consists of memory management routines.
This process can apply to CUDA language as well since it takes place after a conversion of the C++ and CUDA extensions language into regular ANSI C language.

6.1 CUDA compilation workflow

The buildchain consists of several different tools. The following figure shows the complete compilation process and intermediate files of CUDA source file to the final executable file.
The first part is performed by `cudafe` which split up device (GPU) from host (CPU) code. The device code is then compiled by `nvopencc` into Parallel Thread eXecution (PTX) code, which is an intermediate assembly. This assembly code is then compiled into CUDA binary (Cubin) by the proprietary `ptxas` tool. Cubin format is the machine code of the targeted GPU instruction set. Cubin format is proprietary, undocumented and subject to change.



Overview of the CUDA compilation workflow (based on explanation found in [NVCC31])

A. CUDA front end

`cudafe` stands for CUDA frontend and has two purpose: preprocessing (with `-E` option) and CUDA source code analysis. This tool is based on gcc.

Unlike in the standard compilation scheme, preprocessing stage is performed three times. Please note that the `.ii` extension refers to C++ preprocessed files while `.i` refers to C preprocessed files.

The figure shows that CUDA frontend is invoked two times.

B. Nvidia compiler

Actually the compilation stage of the CUDA toolchain is divided into two parts: high-level and low-level compilations. The intermediate language between these two parts is the PTX assembly. Unlike well-known assembly codes (ARM, x86, ...), PTX is not just converted into Cubin (machine code) as a direct translation.
PTX defines a virtual machine and ISA (Instruction Set Architecture) for general purpose *Parallel Thread eXecution* (PTX). This compilation stage was introduced to provide a stable ISA that spans multiple GPU generations.
It is worth noting that `nvcc` is different from `nvopencc` since `nvcc` refer to the whole process of compilation (preprocessing and first-only or both compilation stages) while `nvopencc` refers only to the first stage of compilation producing PTX code.
There are several options for low-level compilation stage:

- **Option A** in the figure: `nvcc` can generate PTX code only. In that case `nvcc` = `cudafe` + `nvopencc`. PTX will be compiled *just-in-time* by the graphic driver. This solution is the most flexible since it allows the graphic driver to optimize the CUDA software for each GPU architecture (even future architectures, not yet known at development time).
- **Option B** in the figure: `nvcc` can generate one or several Cubin codes. In that case `nvcc` = `cudafe` + `nvopencc` + n*`ptxas`. This solution is more restrictive, but allows specific optimization for a specific GPU architecture.
- In both options, there is the possibility to store PTX/Cubin code inside the final binary file as a global initialized data array (using `fatbin`) or outside the final binary as a `.ptx` or `.cubin` external file. In the latter case, the host code will contain extra code needed to load and launch the most appropriate file. This feature is useful because Cubin files can be added, modified or removed just like any file. No need to recompile the host part.

`nvopencc` (high-level) and `ptxas`/*graphic driver* (low-level) both perform some compilation tasks.
`nvopencc` is a fork of a subset of the open-source Open64 compiler [OPEN64] developed by the Computer Architecture and Paralllel Systems Laboratory (CAPSL) of the University of Delaware. According to Mike Murphy from Nvidia in [MURPHY08], *Open64 was chosen for the strength of its optimizations* over GCC.
The high-level compiled only uses a subset of Open64 because its input is always C language. Another simplification is that `nvopencc` does not do any cross-file inter-procedural analysis (IPA) therefore whole kernel source code has to be included into one source file only (this might change in the future).

Low-level compilation is made by Nvidia proprietary Optimized Code Generator (OCG). PTX provides a virtual machine model and is independent of the underlying processor. OCG allocates registers and schedules the instruction according to the targeted GPU chip (Cubin format). *Decuda*/*cudasm* tools [DECUDA] can disassemble/assemble these files for G8X and G9X architectures even if it is not supported by nVIDIA. Those tools were created using reverse engineering.

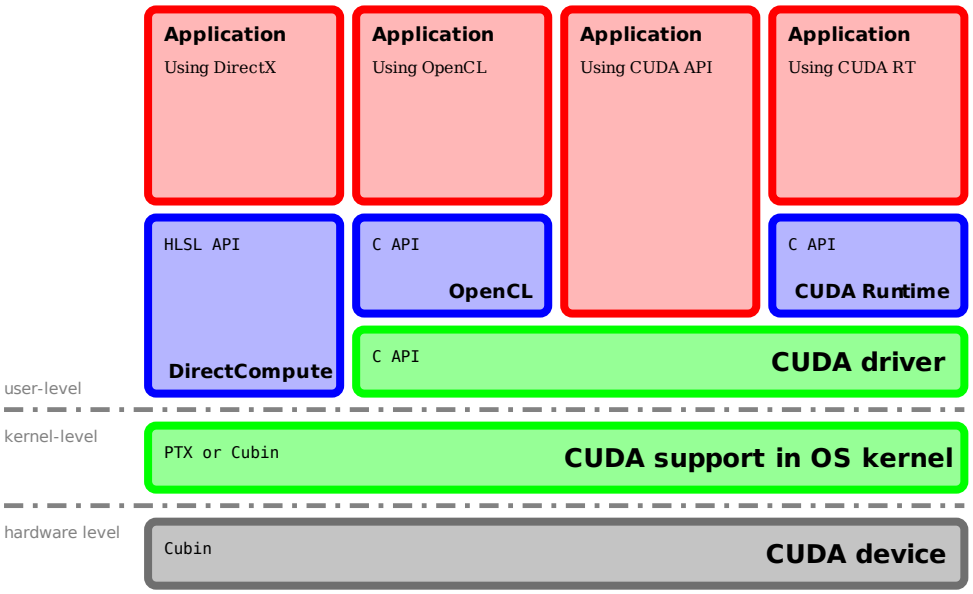
7. Parallel TwigStack algorithm on GPU

The main challenge of implementing XML query processing on GPU is to overcome GPU design for stream processing and its non-divergent threads multiprocessor model because current query processing algorithms do not fit in this paradigm.
Two solutions are possible to solve this issue:

- overcome GPU limitations so as to do more than stream processing
- create a new algorithm that is stream processing compliant

Given the limited time (6 months internship) and my initial lack of knowledge in XML query processing, the first possibility was chosen. First solution seemed having a more progressive learning curve while the second one promises more efficient processing and better results.

There are several available frameworks for General Purpose GPU computing. CUDA was chosen because it is the most stable and documented framework. CUDA features a hardware debugger. OpenCL support was experimental at the beginning of my internship, thus my implementation depends directly on CUDA library. The price of this choice is being bound to nVIDIA products while an OpenCL-based implementation would have been compatible with many more execution platforms (nVIDIA, ATI and VIA GPU, high-end x86 CPU and more) since OpenCL is supported by several processor makers.
CUDA and OpenCL are very close thereby switching from one to the other is not a such difficult task.



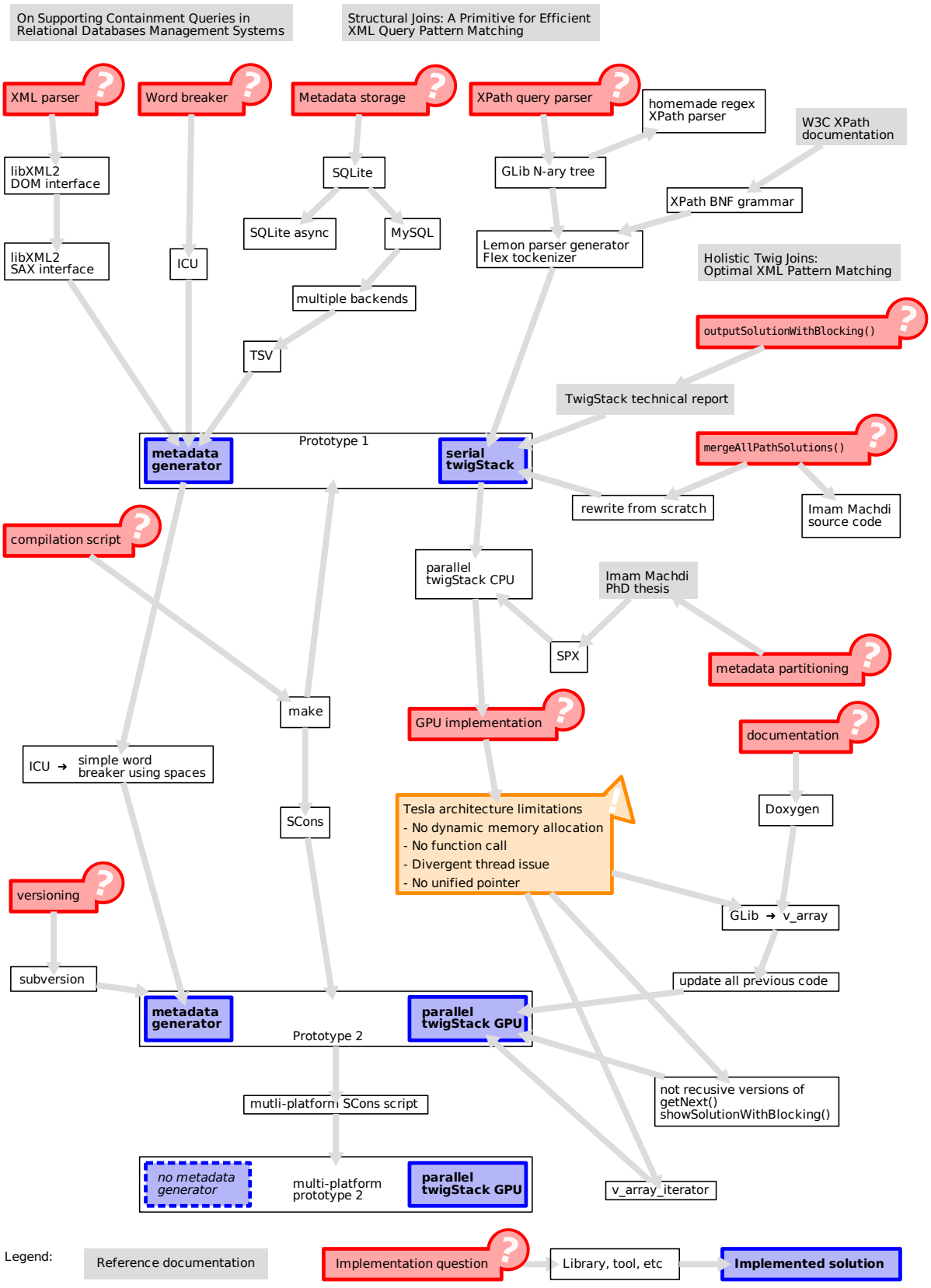
CUDA architecture from [CUDA]

As regards CUDA library choice (driver API or runtime), runtime API is easier to use. Since latest CUDA toolkit, both API (`cu*`) and runtime (`cuda*`) functions can be mixed in the same application. This convergence was probably made to allow developers to start with runtime, which is easier. When they require more low-level control, they can use driver API without having to upgrade the whole software like they would have to do previously.

The new Fermi architecture was released during my internship as well as several big changes in the CUDA toolkit. My implementation undertook several upgrade because of these toolkit updates, but the version at the time of writing is designed for Tesla architecture. Fermi is not compatible with Tesla at binary level, but video card drivers can compile PTX intermediate language just-in-time for both nVIDIA architectures.

7.1 Architecture

Software architecture undertook several deep changes and several prototypes have been developed. The following figure shows the evolution toward the current prototype. This figure was created from my daily and linear schedule of the whole internship.



Implementation design overview

The unlinked grey blocks are the three initial research articles. The red blocks featuring an interrogation mark point out implementation questions that rise from research articles. Of course, research articles are not implementation handbooks. Furthermore, none of the documents address implementation issue in GPGPU environment.
Implementation process is divided into two main prototypes represented by blue blocks. The big orange block shows the discovery of software design problems.

A. First prototype

The first prototype was aimed to improve my knowledge of the TwigStack algorithm and was designed for CPU only. It used the GLib (especially its doubly linked list implementation: GList). LibICU library was used for its Unicode word breaker engine. LibXML2 and its DOM interface was used as XML parser. SQLite3 was used as database back-end. A homemade XPath parser has been made using regular expression matching. The metadata generator fed directly the database.

Direct implementation of the TwigStack algorithm was not obvious since the full algorithm pseudocode is not given and what is given, is written using mathematical-style pseudocode. I especially had a misunderstanding problem with the differentiation of vector and scalar variables.

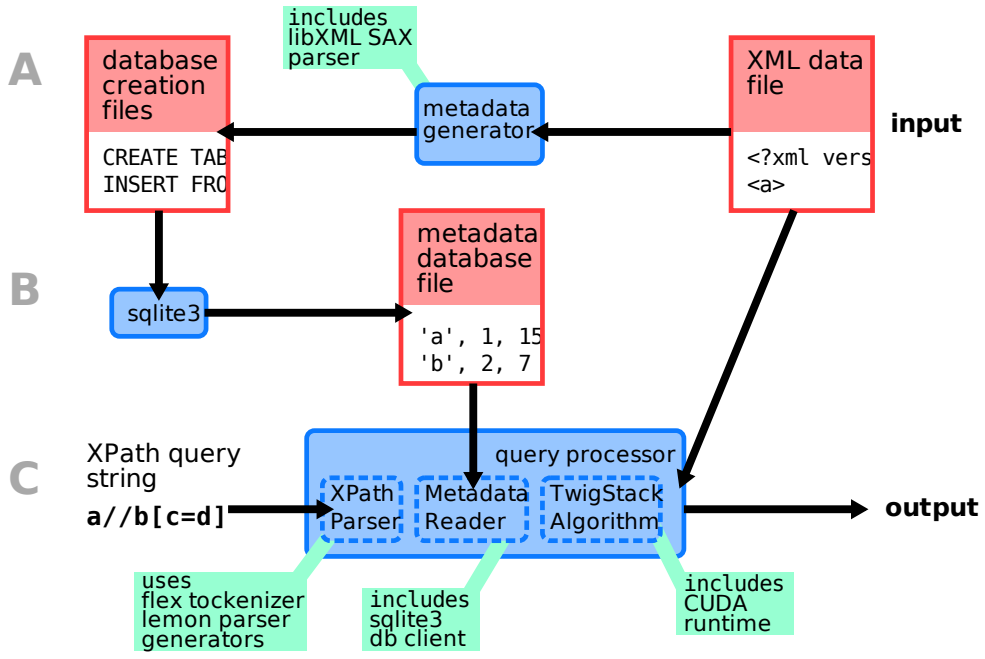
From this very first version, several implementation flaws has been discovered. The generation of the metadata database was very slow and the development of a parser from regular expression is difficult to maintain. Even if the parser is very simple, adding a new parsing feature is not easy.

B. First prototype (v1.1)

After having tried MySQL and SQLite in asynchronous mode [SQLITE-ASYNC] without success, it was decided to split the process into two phases. On a suggestion of professor Amagasa, the first step generates simple TSV files (TSV format is similar to CSV and stands for *Tab Separated Values*) while the second step creates database from loading TSV files. Using this new strategy, performances were dramatically improved.
I have also been asked to introduce XML attribute support in XPath query. Adding new syntax feature to my homemade parser was too much time-consuming therefore the XPath parser was restarted from scratch using tokenizer and parser generators.

C. Second prototype

Since Unicode word breaker slowed down overall performances, it was replaced by a simple homemade word breaker. This implementation only uses space, tab or newline characters for word breaking and is not suitable for all languages (e.g. Japanese).



Final implementation overview

Query processing from XML file is done in three steps:

A. A standalone program reads the XML document and make use of libXML SAX parser in order to generate metadata. Metadata are recorded in plain-text file as TSV files: one file for *XML elements* (tags), one file for *XML attribute* and one file for *text*.
B. `sqlite3` command line tool reads database creation scripts: a new SQLite database file is created. This base contains three tables: ELEMENTS, ATTRIBUTES and TEXT, then the metadata of each table is loaded from the corresponding TSV file.
C. *query processor* is the main part of the architecture. Its contains an automatically generated parser. According to the query tree, relevant metadata streams are loaded from the database, then the query is performed by TwigStack algorithm and matching part of the XML document are read back from the original file. Another

solution would have been to regenerate XML document from metadata information, but they do not contain everything such comments or indentation information.

D. Windows port

Official CUDA samples from nVIDIA are built using GNU make building tool on GNU/Linux and Visual Studio project files on Microsoft Windows.

SCons tool makes multiplatform building scripts easier to write. Host compilers (GNU compiler and Visual Studio compiler) do not use the same syntax for compilation options. SCons was created from the beginning for platform-depend command generation and provides an efficient and modular framework to achieve this. I succeeded in adding simple CUDA support to the build process for both Windows and GNU/Linux.

Furthermore, I created Visual Studio project files which fallback on SCons building scripts for compilation. In my opinion, this is the most comportable development process: centralized building script, but specific development tools according to the platform.

7.2 Metadata generator

The W3C specification provides a modified *Extended Backus–Naur Form* grammar (EBNF) but does not contain any information about how to process XML documents. Several kinds of XML parsers exist. XML parser categories are often represented by their programming interfaces (API) such as *Simple API for XML* (SAX) and *Document Object Model* (DOM). SAX is stream oriented while DOM is tree oriented.

SAX API of the XML parser is a much better choice for metadata generation because this process does not require to keep XML document in its original tree structure. DOM API builds the whole XML tree and can exceed available memory when processing big XML documents of several gigabytes. Big XML documents (e.g., Wikipedia XML export) is the main goal of this project.
Metadata generation is a simple pre-order tree traversal. Since XML elements are already stored in the same ordering, metadata generation using SAX is straightforward: XML elements get position is the same order as they are in the document. This is also an advantage of the inverted list representation. A stack of opening tag is built so as to match ending tag when found. Opening (left) and ending (right) positions are stored into the metadata base. When a text string is encountered, word breaker function is called and each word gets an unique position. This process comes from Information Retrieval tradition. It helps to find the distance between words more efficiently.

7.3 Query processor

A. XPath parser

XPath parser has been implemented using well-known tools: lex-compliant tokenizer generator [FLEX] and yacc-compliant parser generator [LEMON].

Flex is a tool for generating tokenizers (also known as scanners or lexer). A tokenizer matches lexical patterns in text (token). Another way to match lexical patterns is to use regular expressions, but a specifically generated tokenizer is far more efficient. By default, flex generated tokenizers are not unicode-compliant, but using UTF-8 encoding, the source language code can be looked as a byte stream that is compatible with ASCII encoding.

Input data of the query processor are the XML document and the XPath query. They have to be parsed first. XML parse is done by *metadata generator*. XPath query is parsed by *query processor*.
As for XPath parser, W3C specification provides a modified EBNF grammar. This context-free grammar can be used by a parser generator such as well-known *Look-Ahead Left-to-right Rightmost derivation* parsers (i.e., LALR(1)) generated by GNU Bison, for example.

XPath query can use abbreviated or unabbreviated syntax. For example, the following query:
`title[@language=français]` is written in abbreviated syntax. Unabbreviated syntax of the same query is:
`child::title[attribute::language=français]`. Since TwigStack algorithm supports only simple queries containing *parent-child* and *ancestor-descendant*, abbreviated syntax is enough, thus the implemented XPath grammar is based on a subset of abbreviated syntax only.

B. TwigStack algorithm

TwigStack algorithm is divided in two phases: the first phase matches root-to-leaf paths and the second phase merges those paths into trees according to the query. The first phase only was parallelized on GPU because it fits a data-parallel model. The second phase is performed on CPU. This phase could be parallelized as well, but following a task-parallel model that is not easy to implement using GPU.

Stream partitioning is based on SPX model. SPX model was created for allowing a thread to process both first and second part of the algorithm, therefore XML document has to be partitioned in a way that preserves whole query tree into partitions. Since this feature is not useful in my implementation, SPX partitioning was simplified and only

preserves root-to-leaf paths.

TwigStack algorithm makes an intensive use of stacks. This issue was a main point of the implementation: *How to handle stack data-structure on GPU?*
This question led to the *v_array* library. The next section presents this library on which is based the TwigStack implementation.

7.4 Expected performances

Because of the choice to implement on GPU the same Parallel TwigStack as made on multi-core CPU, each thread is fully divergent and executes its own TwigStack algorithm on its own partition. This is not a problem on CPU, but it is on GPU. CPU and GPU thread definitions are not the same and a fully divergent cannot be implemented using CUDA. Warps can be independent therefore a whole warp of 32 GPU threads is seen as one CPU thread.
In the current implementation, the first thread only of a warp executes the TwigStack algorithm and the others do nothing. This is indeed a huge waste of GPU resources.

Tesla and Fermi GPU have different warp execution strategies. On Tesla, since one multiprocessor executes one warp at a time, TwigStack parallelism only exists among multiprocessors. 7 cores out of 8 are unused.
On Fermi, a multiprocessor can execute two warps in the same time but 15 cores out of 16 are wasted.
As a consequence of the different core grouping, Tesla GPU has more multiprocessors than Fermi GPU while each one contains less cores. If n is the number of CUDA cores, Tesla can execute $n/8$ fully divergent threads while Fermi can execute $(n/32)*2 = n/16$.
High-end Tesla GPU cards have 30 multiprocessors (GeForce GTX 285, 240 cores) and high-end Fermi cards have 15 multiprocessors (GeForce GTX 480, 480 cores). Since a Fermi MP can process two divergent threads, the most advanced Tesla and Fermi cards can finally execute the same number of divergent threads in parallel (30) despite the big difference of available cores.
Fermi architecture introduces other improvements such as more cache and faster thread contexts swap therefore it is still expected to perform better than Tesla.

What about GPU vs CPU?
This is the main question of my thesis and unfortunately, I did not manage to execute my GPU implementation. All design issues have been overcome, but some bugs remain. As explained in the introduction, even GPU is not faster than CPU, it could still be used as an "XML query coprocessor" and relieve CPU workload.

Non-uniform parallelism using CUDA has already been studied in [LERNER08]. The last slide of the presentation is a list of wishes for an improved CUDA toolkit (especially about debugging support). Meanwhile CUDA improved but non-uniform parallelism is still not obvious to implement.

8. The *v_array* data-structure

8.1 Introduction

v_array is an implementation of **unrolled doubly-linked list** data-structure written in C language. This variation of the standard linked list is actually list of arrays since Each node contains several data elements. The data-structure was created for dynamic memory allocation. Dynamic memory allocation allows to implement and experiment using CUDA, algorithms that do not fit the current programming model.
It was designed to be very generic and to be used for CUDA development. Its main purpose is to enable developers to create, from same source code, several versions that use different kind of memory access (e.g. CPU or GPU main memory).

Why a new data-structure? The library has been created because of the limitations of nVIDIA GPU with compute capability of 1.1 → 1.3 (aka. Tesla architecture).

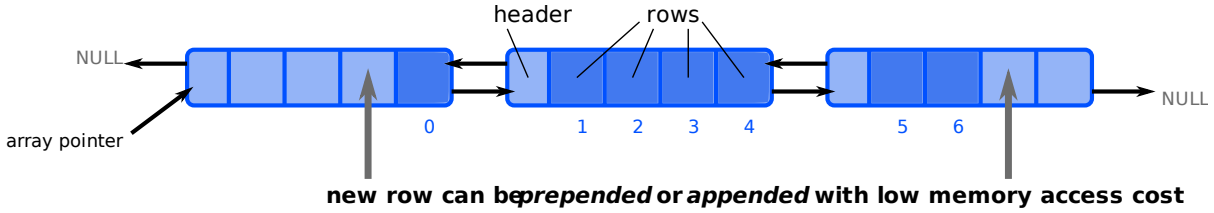
- ✓ No function pointers required (not supported by GPU of the Tesla architecture)
  a lot of macro used instead
- ✓ No dynamic memory allocation required (not supported by GPU of the Tesla architecture)
  homemade memory manager found in v_mem_manager.h
- ✓ Can use offset instead of pointers
  easier CPU and GPU results comparison (offset are the same on both sides whereas pointers are not)
- ✓ Source can be included directly
  can be used in CUDA kernels

8.2 Features

One of the main issue about a data-structure is its complexity. The numerous different data-structures available in computer science shows that a trade-off between flexibility and efficiency is always required. Each solution has strengths and weaknesses. The purpose of this section is to explain why *v_array* can be a better trade-off than others (this is not restricted to GPU processing).

A. A good trade-off between indexing and insertion costs

The most well-known solutions to store a dynamic sequence of data are the *linked list* and the *dynamic array*. As explained previously, unrolled linked list is a variation of the standard linked list. It is not a mix between linked list and dynamic array since nodes of the list do not double each time the previous array is full.



v_array memory representation

CPU processor architectures are optimized for loops on linear array. Branch prediction is designed to prevent from emptying the processor pipeline at each loop iteration. Cache memory is designed to prevent from fetching from the main memory when accessing the row of an array. Random memory access lowers the performances since cache miss rate is high.
Although GPU architectures do not rely on cache to optimize memory access, random memory access remains poorly efficient since it prevents memory coalescing.
The following table sums up advantages and drawbacks of several memory structure according to four criteria:

Indexing costs is defined by the number of memory access in order to reach the n^{th} row of the data-structure. Some memory structures such as linear array allows to compute the pointer of any row from the base pointer of the array. Some other structures are scattered in memory.

Inserting costs is defined by the number of memory allocation in order to insert n row (one-by-one) in the data-structure. For example, each time a row is inserted into a linked list, there is a new memory allocation. This is not the case for dynamic array since additional free rows are allocated in prevision. The value of the table is not the cost of insertion in the middle of the sequence.

Memory costs is defined by the number of memory unit which does not contain row data. For example, each node of a doubly-linked list has to store two pointers.

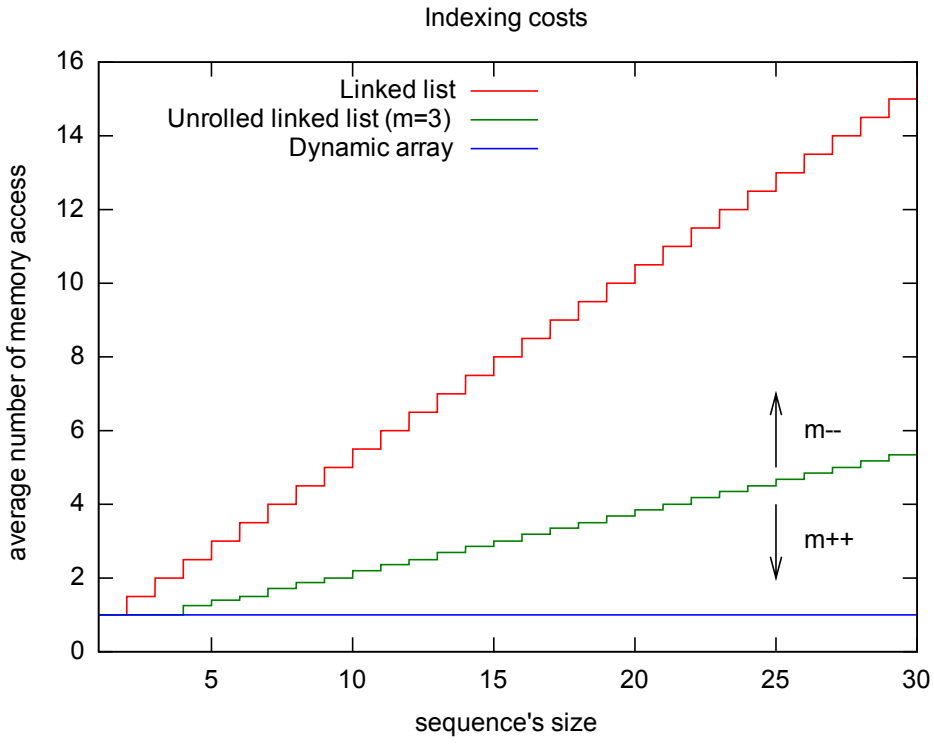
Cache miss costs is defined by moment when next memory access is not located just beside the previous one (spatial

locality). Memory structures having high data scattering have also higher cache miss rate.

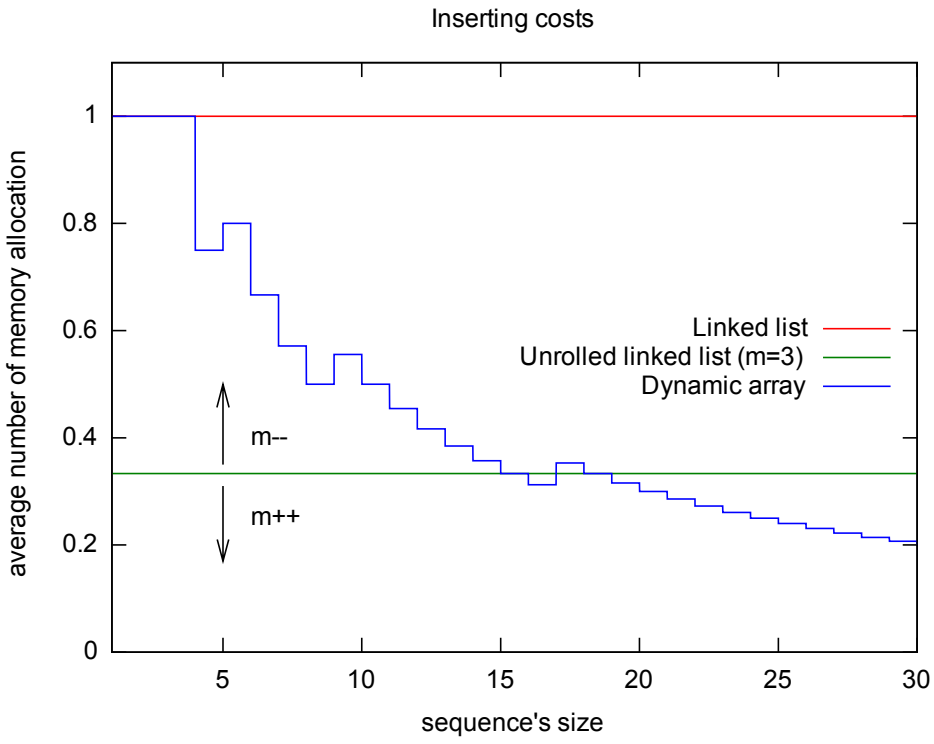
	indexing (average number of memory access)	insertion (average number of memory allocation)	additional memory space (per row)	cache miss (when?)
linked list	$\frac{n+1}{2}$	1	$2 * \text{sizeof}(\text{pointer})$	at each row
unrolled linked list	$\left\lceil \frac{n(n+m)}{2m} \right\rceil \frac{1}{n}$	$\frac{1}{m}$	$\frac{\left\lceil \frac{n}{m} \right\rceil * 2 * \text{sizeof}(\text{pointer}) + \left(\left\lceil \frac{n}{m} \right\rceil * m - n\right) * \text{sizeof}(\text{row})}{n}$	at each node (m rows)
dynamic array	1	$\frac{\left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil + 1}{n}$	$\frac{\left(2^{\left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil} - n\right) * \text{sizeof}(\text{row})}{n}$	at each array reallocation
array	1	N/A	0	at first array access only

Complexity of different data-structures. n is the number of rows and m is the number of row in each node of the unrolled linked list.

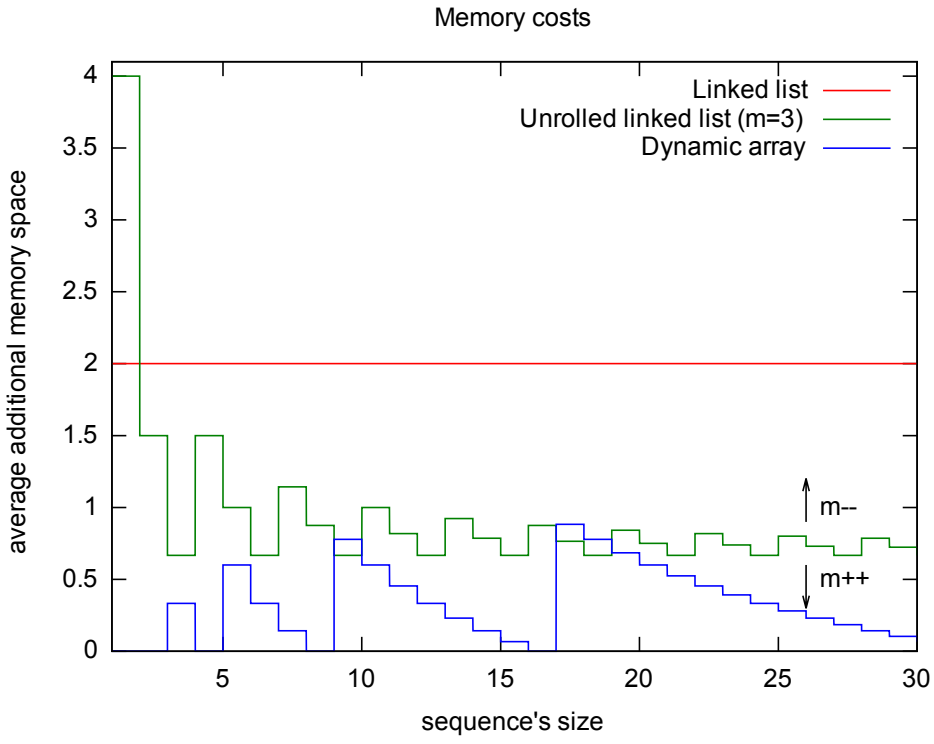
The following plots are a graphical representation of the table above. Unfortunately, I was unable to draw discontinuous plot using gnuplot. At each step, the relevant value is the right one (in the right corner of the step). Please keep in mind that the represented values are the average costs.



Indexing costs comparison



Inserting costs comparison

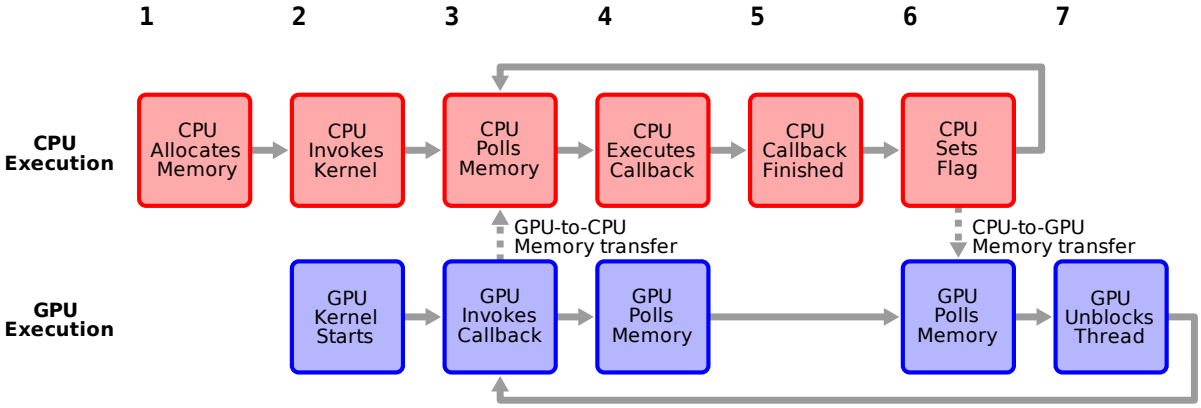


Memory costs comparison

Dynamic array is famous for its amortized inserting cost at the price of increasing memory cost. This data-structure is not easy to implement on a simple memory manager because of memory fragmentation and massive reallocations. On nVIDIA GPU, unrolled linked list is likely to have $m = 32$ since it allows to perform computation on a node of the list using a whole GPU warp. Memory allocations are always of the same size, thus enabling a very simple memory allocator. Unrolled linked list also makes easier the creation of a manual paging system in shared memory of the GPU (list node = memory page).

B. Dynamic memory structure

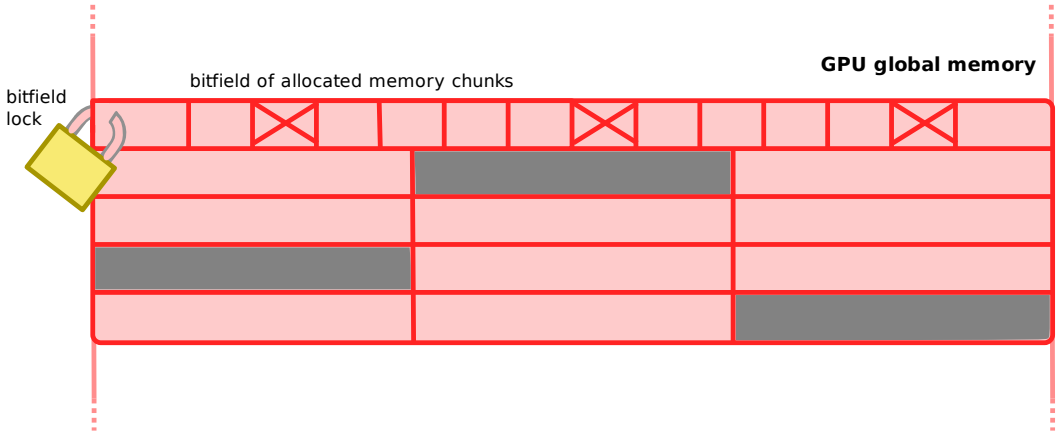
On the Tesla architecture, memory cannot be allocated dynamically. Allocating memory in advance is one solution. Another solution is using a CPU callback when GPU memory allocation is required.



GPU to CPU callback

GPU-to-CPU callbacks were experimented using *pools of zero-copy memory and polling*. This method is designed so that GPU can issue any CPU function callbacks. When GPU need to execute a CPU function, it writes function parameters into the zero-copy memory and sets a flag. Since CPU is polling this memory area meanwhile, the flag setting triggers the execution of the appropriate CPU function and then unset the flag. The same flag polling is performed on the GPU while the CPU executes the callback function. The flag unsetting triggers the GPU thread resume.
CPU callback would be an elegant solution for dynamic memory allocation if polling was replaced by signal/interrupts (sleep-base mechanism) [STUART10]. Unfortunately, Tesla architecture (or Fermi) does not support a such feature.

v_array contains a basic memory manager in order to simulate dynamic memory allocation on GPU using the other option (i.e., big memory allocation). The implementation was made straightforward: memory allocated size is fixed and should match the node's size of unrolled linked list. The header of the memory pool is a simple bitfield which stores memory chunk states: free or allocated.

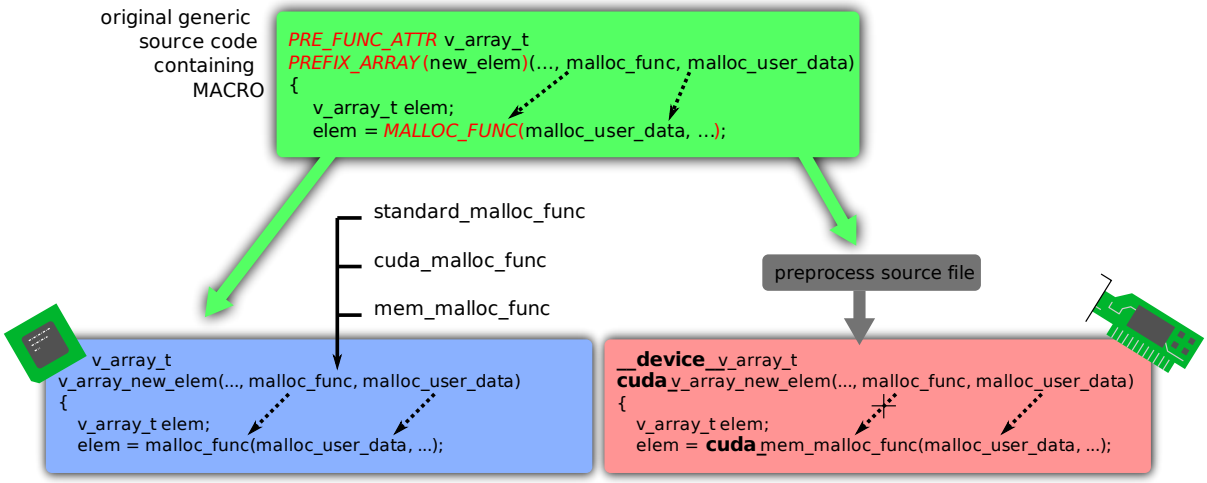


Memory pool for dynamic memory allocation

This simple solution was studied specifically in [HUANG10] and several efficiency improvements over this solution are shown.

C. Generic but function pointers not required

In order to allow both CPU and GPU usage, all memory accesses of the library (allocation, copy, unallocation) were made generic through function pointers. Unfortunately, GPU of the Tesla architecture cannot execute a such code since all function calls have to be inlined, thus memory accesses were also made generic through *macro* functions.



v_array memory access possibilities

v_array library allows to use 3 kinds of memory accesses in the same program:

standard_{malloc, memcpy, free}_func
: use `malloc()`, `memcpy()` and `free()` functions provided by *Operating System*. These functions can only be used in CPU code.

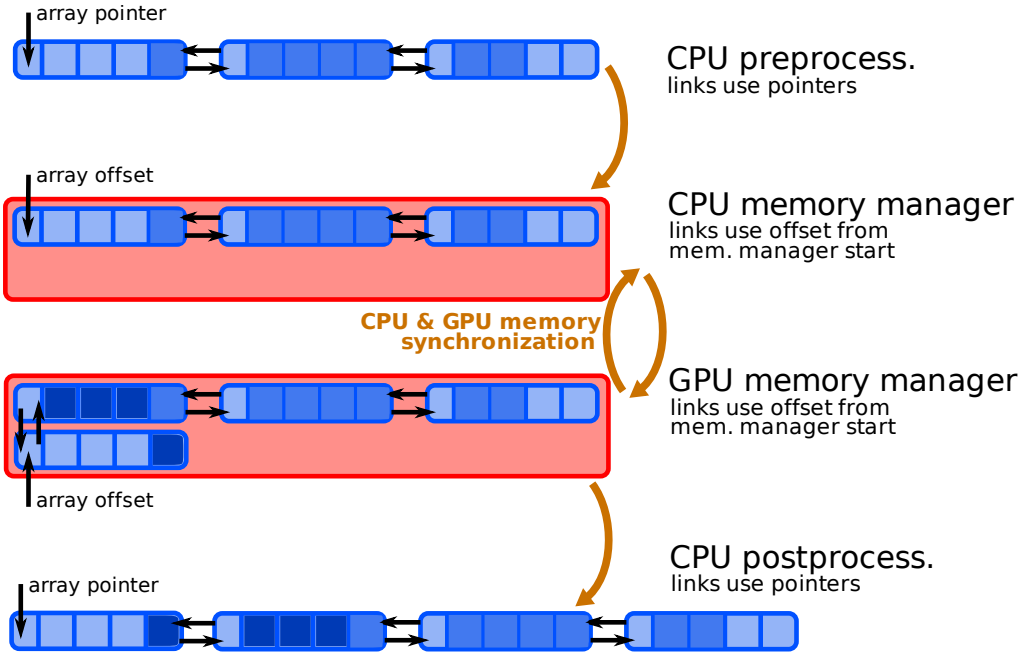
cuda_{malloc, memcpyH2D, memcpyD2H, free}_func
: use `cudaMalloc()`, `cudaMemcpy()` and `cudaFree()` functions provided by *CUDA runtime library*. These functions can only be used in CPU code.

(cuda_)mem_{malloc, memcpy, free}_func
: use `v_mem_manager_allocate_chunk()` and `v_mem_manager_free_chunk()` provided by *v_array library*. These functions can be used by CPU and GPU code (GPU code uses '`cuda_`' prefixed version).

D. Offsets instead of pointers

Even with *pinned memory*, GPU and CPU never share the same memory address space. In the case of stream processing, memory structures are static and do not likely contain any pointers. The case of a linked list is different since the links are made of pointers. Of course, dereferencing on the GPU a pointer to CPU memory will lead to crash, or worse, to read memory from an unexpected area (an issue that is hard to debug).
Since there is no memory protection, reading or writing to unallocated memory does not trigger any *memory segmentation fault*. If the GPU is used to display too, you can even accidentally scribble over video memory which produces artistic screen results and forces you to reboot the computer.

Using memory pool and offset from the base pointer of the pool instead of pointers has the advantage of creating identical memory space on both CPU and GPU, therefore memory area can easily be swapped in a single memory copy.

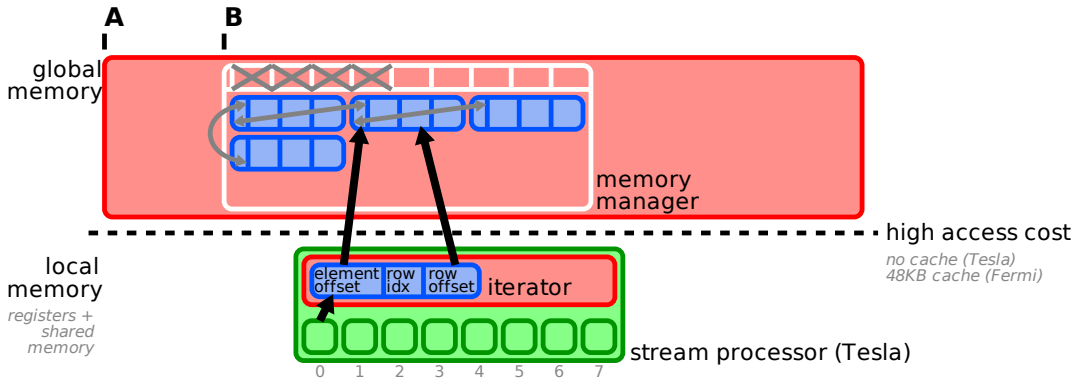


Recommended usage of *v_array*

In a such case, memory alignment has to be taken into account so as not to reduce GPU performances.

E. Memory spaces and iterators

Since the GPU architecture contains several addressable memory spaces having addressing costs with a difference of two orders of magnitude, it is important to avoid some memory accesses as much as possible. Another issue is related to compiler inference of pointer targets. Iterator structure has been introduced to address these issues.
Translation in the sequence of row is done relative to the iterator's current position instead of absolute to the sequence beginning or end. Since iterator is always stored in local memory, there are no possible confusion when passing it as argument to a function.



Memory fragmentation

Memory space confusion is specific to the Tesla architecture since the Fermi architecture solves this issue thanks to its unified pointer feature.

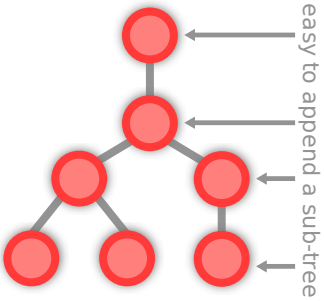
F. Other data-structures based on *v_array*

Stack

The *v_array* implementation already meets all requirements for an efficient double-ended queue data-structure (*deque* for short). A stack can be seen as a special case of a *deque* in which one end only is used.

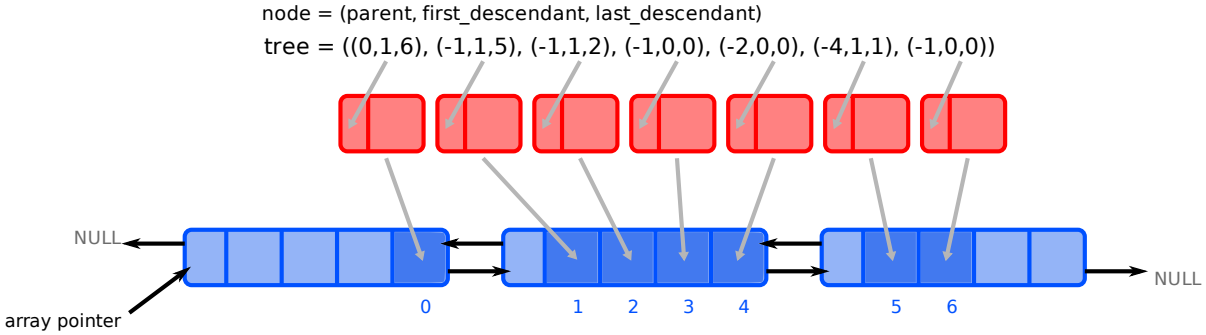
Tree

Implementing a tree data-structure into a v_array is less obvious than a stack. In order to keep the advantages of the *v_array*, some limitations to the *v_tree* operations have been decided. The insertion possibilities were limited to the right side of the tree since insertion/deletion at any position of the list is not implemented in *v_array*.



A tree example

On the other hand, some operations are very efficient like tree traversal (if the in-memory order is the same as the tree traversal order). Tree nodes are serialized into a sequence of nodes.



v_tree storage into *v_array*

8.3 API documentation

API documentation generated with Doxygen can be found at http://kde.cs.tsukuba.ac.jp/~vjordan/docs/v_array/api/. Doxygen generates documentation from comments in the source code thereby creating an always up-to-date documentation.

9. Debugging strategy

Since GPU software are uploaded to an external device, one has a very little feedback when something went wrong. If your GPU program gets more complex, the lack of GPU debugging solutions become a real issue.
At the beginning, the CUDA toolkit did not feature any GPU debugger. The nVIDIA solution was to create a minimalistic emulation layer enabled through a compiler option in order to generate CPU only executable from CUDA source code, then standard C language debugging tools can be used on those programs since they are not executed on GPU anymore. This solution is known as *device emulation* and could be activated with `-deviceemu` compiler option. This option is deprecated in 3.0 CUDA toolkit and is not available anymore since 3.2 toolkit. nVIDIA recommends to use the new CUDA hardware debugger. This debugger is very useful, but may crash. This section presents other available debugging solutions for CUDA.

9.1 "printf" debugging

A simple way to get more feedback from a software is to make it more verbose. This method is known as "printf" debugging because the `printf` C function is used to trace program execution. The developer often implements several levels of verbosity and enables them at compilation time through compilation parameters.
At the beginning of CUDA development, "printf" debugging was only available when using device emulation on CPU since GPU of the Tesla architecture were not able to perform any function call (unlike CPU). This solution was not efficient because GPU emulation was too straightforward and led to extremely slow execution (usually an order of magnitude). Atomic functions and race conditions among the thread of a warp were not emulated as well as global and local pointers incompatible address spaces.

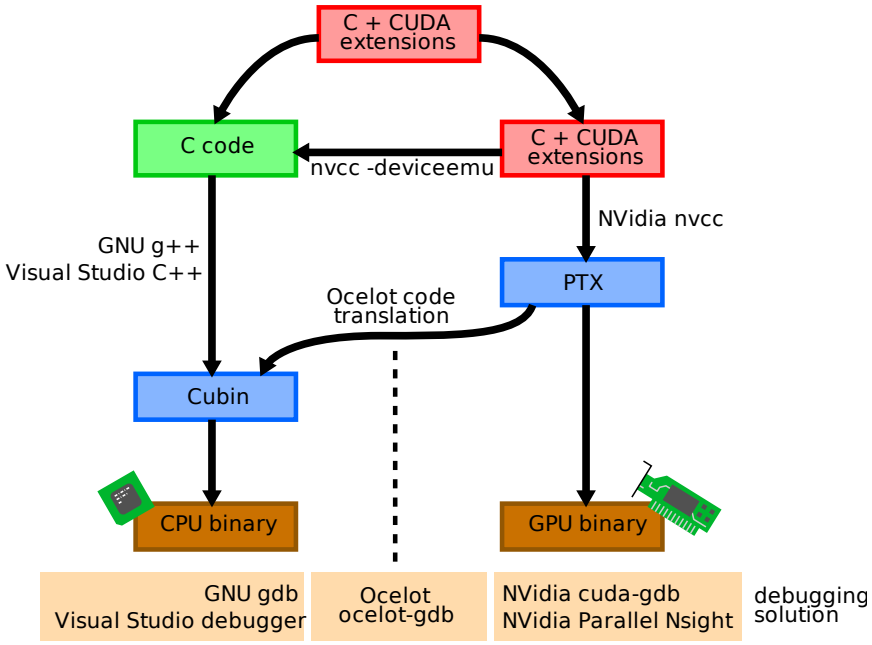
A nVIDIA employee created a hack to do a 'sort-of' `printf` while executing code on GPU, but this function has never been released in official CUDA toolkit. `cuPrintf` function writes strings in a fixed-size buffer in GPU global memory. When the program executes on CPU again, it reads this buffer and prints out all text strings using standard `printf`.
This library suffers from many limitations:

- Variadic functions are not allowed on GPU. `cuPrintf` is limited to 10 arguments.
- Output buffer has to be allocated prior GPU execution therefore it has a fixed size. Too many `cuPrintf` calls can lead to overflow.
- Outputs are asynchronous. Everything is printed at the same time *after* GPU execution.

The main limitation comes from its asynchronous behavior because output buffer is printed only when GPU kernel went until the end. When anything wrong happened, nothing is displayed at all.
These problems have been solved in Fermi architecture. Function call feature of this architecture allows system calls like `printf`.

9.2 CUDA Debugging softwares

A real CUDA hardware debugger was introduced with 2.2 toolkit (cuda-gdb in Linux/Unix toolkit) and was really usable from 3.0 toolkit (in March 2010). As for Visual Studio, Parallel Nsight debugger was made publicly available in July 2010.
The following figure sums up different debugging solutions.



Debugging possibilities for CUDA code

As explained previously, CUDA kernels can be debugged while executed on GPU using CUDA hardware debuggers for Linux or Windows. Those kernels can be debugged while implemented on CPU (but this feature is available anymore) using standard debuggers. There is a third solution: *Ocelot code translation*.

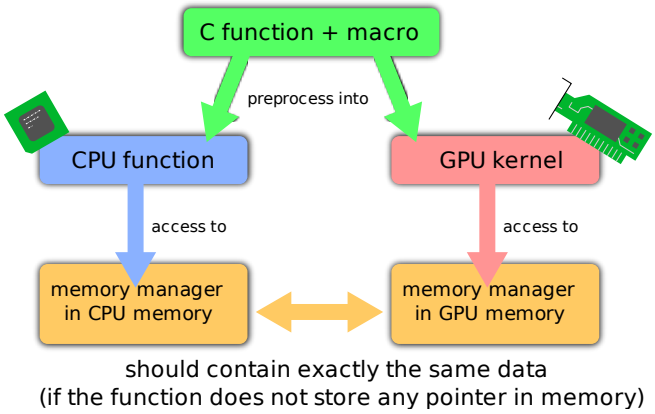
Translating GPU binaries to tiered SIMD architectures with Ocelot
Gregory Diamos, Andrew Kerr, Mukil Kesavan
Georgia Institute of Technology, Technical Report, January 2009

PTX kernels can be emulated or translated just-in-time to CPU target. Since Ocelot infrastructure performs a deep analysis of CUDA kernel, it does an accurate "bug-to-bug" emulation and enables efficient debugging [DIAMOS10].
Ocelot is the result of a research project about *dynamic compilation framework for heterogeneous systems* (quoted from Ocelot project website [OCELOT]). Several back-end were created, but the ones linked to CUDA are the most active.
Other research project addressed the same target, such as the Barra project [BARA], but Ocelot is the only remaining active project. At the time of writing, PTX 2.0 is still not yet fully supported.

The main limitation of this solution is that it works at PTX level. `ocelot-gdb` tool cannot read `cuda-gdb` debugging symbols. Matching the faulty CUDA code from PTX is not an easy task and it supposes a deep knowledge of PTX assembly.

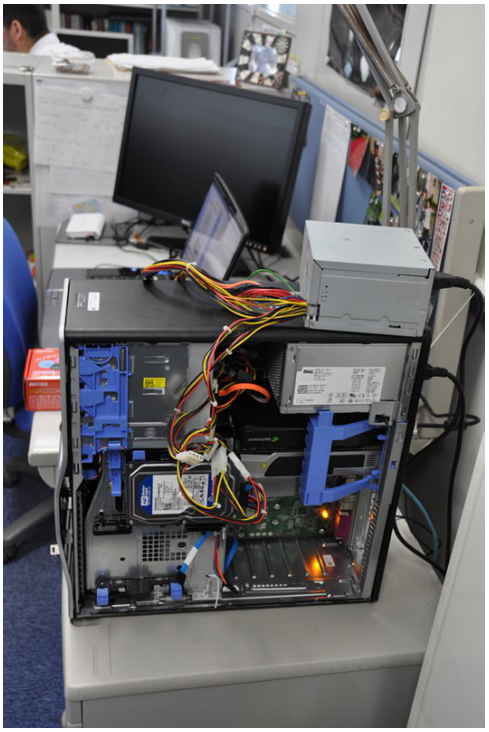
9.3 Another solution

The *v_array* library was designed to allow another debugging opportunity. Since CUDA language is mainly C language with a small extension, it can be translated into plain C language using preprocessor macro. The idea is to preprocess the *same* code for CPU (plain C) and GPU (extended C). This possibility is available according to the CUDA project. Since both versions can be executed in one executable, it becomes possible to compare in-memory result of each version. Thanks to *v_array* memory manager which was designed to contain the same data on both sides, it is possible to compare the whole memory state at byte level.



Comparing memory manager content between CPU and GPU memory

This process can be automated into unit tests. Reference version for CPU (aka. gold version) can be automatically compared with GPU version. Using this process, developer is able to immediately make apart general bugs from GPU-specific bugs. Unit tests of the *v_array* library are written this way. SCons build script automatically preprocess two versions of the CUDA kernel function. Functions name are prefixed according to each version.


My CUDA workstation at KDE lab.

Conclusion

The purpose of this report is to finalize my last project assignment, as well as my studies at the UTBM. It is the suitable time for a personal reflection on the curriculum I have endured. Engineering studies at the UTBM include three mandatory internships, the first done over one month and remaining two, which were completed over a 6 month time period. The previous and second internship had been spent at Euro Airport (France/Swiss, from September 2007 to January 2008). The main goal of these internships was to gather more credible experience in software engineering and project management through the model of an airport environment. The Technical skills I learned were already related to databases, since most of my work was linked to Oracle DBMS.
As for the third and final internship, I oriented my studies at KDE laboratory toward three goals:

- Current research with a view toward PhD research track
- GPU processing developement skill
- Internationalization and English communication

The result of each point is discussed in the following paragraphs.

About *current research*, this training period has been worthy of validating my project and aspirations to start doctoral studies. Although I decided on this laboratory with little knowledge of it, Kitagawa Data Engineering laboratory was a sensible choice.

About *GPU processing*, this point is the most uncertain aspect of my studies since I failed to finish the planned schedule because of unforeseen implementation issues. Given that it is from mistakes that you learn the most, I would try to behave differently if I have to cope again with a still new technology as GPGPU was during this internship.

About *internationalization*, Japan is a place without equivalents in the world for French or European people to sharpen their communication skills. The mix of its very own unique Japanese culture, combined with both American and other Asian cultures creates a true challenge for understanding and being understood. In human terms, my internship was a great success. I discovered and applied the process of *enculturation*: "the process by which a person learns the requirements of the culture by which he or she is surrounded, and acquires values and behaviors that are appropriate or necessary in that culture".

The UTBM report writing guidelines suggest including in the conclusion section as an estimation of the *financial gains* enabled by the work done during the internship. As regards to the work made in research laboratory, this estimation is generally extremely difficult to achieve because those gains are expected at the end of a very the long-term process. This estimation is also made fuzzier because of the fact that the gains are of a human aspect before becoming financial.

To conclude, I think that two out of three goals have been reached with success. I cannot deny that I would have liked to finish on time. Unlike in engineering, the probability of the implementation issues is very high in research because of the use of cutting-edge technologies. This personal experience leverages the importance of the conception of a development methodology even if the project seems small at start.

My proposition to go further in using GPGPU for XML query processing by undertaking a PhD at the same place is supported by both Professors Kitagawa and Amagasa.

This report was written using XML-based document formats (XHTML, SVG, MathML). Thus it could have been generated on GPU using a software that includes the result of this research.

References

9.1 Research papers

[SHIOKAWA10]
Hiroaki Shiokawa, Hiroyuki Kitagawa, Hideyuki Kawashima. ***A-SAS: An Adaptive High-Availability Scheme for Distributed Stream Processing Systems***. Proc. of 3rd. International Workshop on Sensor Network Technologies for Information Explosion Era (SeNTIE 2010), Kansas City, Missouri, USA, pp. 413-418, May 23-26, 2010.

[KADHEM10]
Hasan Kadhem, Toshiyuki Amagasa, Hiroyuki Kitagawa. ***MV-OPES: Multivalued-Order Preserving Encryption Scheme: A Novel Scheme for Encrypting Integer Value to Many Different Values***. IEICE TRANSACTIONS on Information and Systems Vol.E93-D No.9 pp.2520-2533, Sept. 2010. ISSN: 1745-1361, 0916-8532

[SHI10]
Hang Shi, Toshiyuki Amagasa, Hiroyuki Kitagawa. ***Fast Detection of Functional Dependencies in XML Data***. The 7th International XML Database Symposium (XSym2010), Singapore, pp. 113-127, September 13-17, 2010. (to appear)

[MACHDI10]
Imam Machdi, Toshiyuki Amagasa, Hiroyuki Kitagawa. ***Executing parallel TwigStack algorithm on a multi-core system***. International Journal of Web Information Systems (IJWIS), Vol. 6, No. 2, pp. 149-177, 2010.

[TAKAHASHI10]
Tsubasa Takahashi, Hiroyuki Kitagawa. ***A Ranking Method for Web Search Using Social Bookmarks***. pp. 585-589, 2009.

[KAMIE10]
Mariko Kamie, Takako Hashimoto, Hiroyuki Kitagawa. ***Topic-Based Awareness Computing Model for Video-Sharing Service***. ISAC 2010-2nd International Symposium on Aware Computing, National Cheng Kung University, Tainan, Taiwan, November 1-4, 2010. (to appear)

[TAKAGI10]
Takashi Takagi, Hideyuki Kawashima, Toshiyuki Amagasa, Hiroyuki Kitagawa. ***Providing Constructed Buildings Information by ASTER Satellite DEM Images and Web Contents***. Proc. of Data Intensive eScience Workshop (DIEW 2010) (DASFAA2010 Workshop), LNCS 6193, pp. 81-92, April 2010. Springer Berlin / Heidelberg.

[BUCK04]
Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan. ***Brook for GPUs: Stream Computing on Graphics Hardware***. Computer Science Department, Stanford University [cited 2010/12/02].
PDF file, available at: <http://graphics.stanford.edu/papers/brookgpu/>

[WONG10]
Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, Andreas Moshovos. ***Demystifying GPU Microarchitecture through Microbenchmarking***. Department of Electrical and Computer Engineering, University of Toronto [cited 2010/12/02].
PDF file, available at: <http://www.eecg.toronto.edu/~myrto/gpuarch-ispass2010.pdf>

[COLLANGE10]
Sylvain Collange. ***Analyse de l'architecture GPU Tesla*** (French). DALI, ELIAUS, University of Perpignan [cited 2010/12/02].
PDF file, available at: <http://hal.archives-ouvertes.fr/docs/00/44/38/75/PDF/collange_tesla_tr.pdf>

[FARIAS]
Thiago S. M. C. Farias, João Marcelo N. X. Teixeira, Pedro J. S. Leite, Gabriel F. Almeida, Mozart W. S. Almeida, Veronica Teichrieb, Judith Kelner. ***High Performance Computing: CUDA as a Supporting Technology for Next Generation Augmented Reality Applications***. Centro de Informática, UFPE [cited 2010/12/02].
PDF file, available at: <http://seer.ufrgs.br/index.php/rita/article/viewPDFInterstitial/rita_v16_n1_p71/7287>

[GHARACH95]
Kourosh Gharachorloo. ***Memory Consistency Models for Shared-Memory Multiprocessors***. WRL Research Report 95/9, Western Research Laboratory [cited 2010/12/02].
PDF file, available at: <http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-9.pdf>

[MURPHY08]
Mike Murphy. ***NVIDIA's Experience with Open64***. Nvidia corporation [cited 2010/12/02].
DOC file, available at: <http://www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc>

[STUART10]
Jeff Stuart and John D. Owens, Michael Cox. ***GPU-to-CPU callbacks*** (poster). University of California and Nvidia corporation [cited 2010/12/02].
PDF file, available at: <http://www.nvidia.com/content/GTC/posters/2010/P01-GPU-to-CPU-Callbacks.pdf>

[DIAMOS10]
Gregory Diamos, Andrew Kerr, and Sudhakar Yalamanchili. ***Dynamic Compiler for PTX***. Computer Architecture and Systems laboratory, Georgia Institute of Technology [cited 2010/12/02].

Available at: <http://www.gdiamos.net/papers/ocelot-nvidia-research.pdf>
[COLLANGE09]
Sylvain Collange, David Defour, David Parello. ***Barra, a Parallel Functional GPGPU Simulator***. ELIAUS, University of Perpignan [cited 2010/11/02].
Available at: <http://hal.archives-ouvertes.fr/hal-00359342>
[DIAMOS09]
G. Diamos, A. Kerr, and M. Kesavan. ***Translating GPU binaries to tiered SIMD architectures with Ocelot***. Georgia Institute of Technology, Technical Report GIT-CERCS-09-01, January 2009.
PDF file, Available at: <http://www.cercs.gatech.edu/tech-reports/tr2009/git-cercs-09-01.pdf>
[LERNER08]
Benjamin Lerner, Trevor Jim, Yitzhak Mandelbaum. ***Experiences coding non-uniform parallelism using the CUDA GPGPU architecture*** (slides). Computer Science and Engineering, University of Washington and AT&T Research.
PDF file, Available at: <http://www.cs.washington.edu/homes/blerner/files/njpls.pdf>
[HUANG10]
Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, Wen-mei Hwu. ***XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines***. IMPACT Research group, University of Illinois, NVIDIA Corporation.
PDF file, Available at: <http://impact.crhc.illinois.edu/ftp/conference/malloc.pdf>

9.2 Standards references

[XML]
Extensible Markup Language (XML) 1.0. World Wide Web Consorsium [cited 2010/09/15].
Available at: <http://www.w3.org/TR/xml/>
[XPATH]
XML Path Language (XPath) Version 1.0. World Wide Web Consorsium [cited 2010/09/15].
Available at: <http://www.w3.org/TR/xpath/>
[CUDA32]
CUDA 3.2 C programming guide. Nvidia Corporation [cited 2010/12/02].
PDF file, available at: <http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf>
[PTX14]
CUDA PTX 1.4 official documentation. Nvidia corporation [cited 2010/12/02].
PDF file, available at: <http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf>
[PTX21]
CUDA PTX 2.1 official documentation. Nvidia corporation [cited 2010/12/02].
PDF file, available at: <http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ptx_isa_2.1.pdf>

9.3 Project's homepages

[STRSPIN]
StreamSpinner. Kitagawa Data Engineering laboratory, University of Tsukuba [cited 2010/09/14].
Available at: <http://www.streamspinner.org>.
[LIBSH]
LibSH. Intel Corporation [cited 2010/12/02].
Available at: <http://libsh.org/>
[OPEN64]
Open64. Computer Architecture and Parallel Systems Laboratory, University of Delaware [cited 2010/12/02].
DOC file, available at: <http://www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc>
[DECUDA]
Decuda/Cudasm. [cited 2010/12/02].
Available at: <https://github.com/laanwj/decuda/wiki>
[OCELOT]
GPUOcelot. Computer Architecture and Systems laboratory, Georgia Institute of Technology [cited 2010/12/02].
Available from: <http://code.google.com/p/gpuocelot/>
[BARA]
Bara. University of Perpignan [cited 2010/12/02].
Available at: <http://gpgpu.univ-perp.fr/index.php/Barra>
[SQLITE-ASYNC]
An Asynchronous I/O Module For SQLite. SQLite project [cited 2010/12/02].
Available at: <http://www.sqlite.org/asyncvfs.html>
[FLEX]
Flex: The Fast Lexical Analyzer. [cited 2010/12/02].
Available at: <http://flex.sourceforge.net/>
[LEMON]

LEMON Parser Generator. SQLite project [cited 2010/12/02].
Available at: <http://www.hwaci.com/sw/lemon/>

9.4 Official documentations

[SQLSRV-XML]
Understanding XML in SQL Server. Microsoft TechNet [cited 2010/12/02].
Available at: <http://technet.microsoft.com/en-us/library/bb522493%28SQL.100%29.aspx>
[CUDA]
CUDA Architecture Overview. Nvidia corporation [cited 2010/12/02].
PDF file, available at: <http://developer.download.nvidia.com/compute/cuda/docs
/CUDA_Architecture_Overview.pdf>
[FERMI]
Fermi Compute Architecture Whitepaper. Nvidia corporation [cited 2010/12/04].
PDF file, available at: <http://www.nvidia.com/content/PDF/fermi_white_papers
/NVIDIAFermiComputeArchitectureWhitepaper.pdf>
[NVCC31]
The CUDA Compiler Driver NVCC (3.1). Nvidia corporation [cited 2010/12/02].
PDF file, available from: CUDA 3.1 toolkit
[GTX200]
GeForce GTX 200 GPU Technical Brief. Nvidia Corporation [cited 2010/12/02].
PDF file, available at: <http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf>
[WCOMB]
Intel Write Combining Memory Implementation Guidelines. Intel Corporation [cited 2010/12/02].
PDF file, available at: <ftp://download.intel.com/design/PentiumII/applnots/24442201.pdf>

9.5 Books

[THOMSON06]
Richard Thomson. ***The Direct3D Graphics Pipeline*** (draft). Richard Thomson's homepage [cited 2010/12/02].
PDF file, available at: <http://www.xmission.com/~legalize/book/download/>
[GRUSEC07]
Joan E. Grusec, Paul D. Hastings. ***Handbook of Socialization: Theory and Research***. Guilford Press, 2007, p. 547. ISBN 1593853327, 9781593853327.
Quoted through English Wikipedia: <http://en.wikipedia.org/wiki/Enculturation>

9.6 Wikipedia articles

[W-TSUKUBA]
Tsukuba, Ibaraki. English Wikipedia [cited 2010/09/14].
Available at: <http://en.wikipedia.org/wiki/Tsukuba,_Ibaraki>.
[W-XML]
XML. English Wikipedia [cited 2010/09/14].
Available at: <http://en.wikipedia.org/wiki/XML>
[W-XPATH]
XPath. English Wikipedia [cited 2010/09/14].
Available at: <http://en.wikipedia.org/wiki/XPath>
[W-GPGPU]
GPGPU. English Wikipedia [cited 2010/09/14].
Available at: <http://en.wikipedia.org/wiki/GPGPU>
[W-STRPROC]
Stream processing. English Wikipedia [cited 2010/09/14].
Available at: <http://en.wikipedia.org/wiki/Stream_processing>
[W-ISO690]
ISO 690. English Wikipedia [cited 2010/09/14].
Available at: <http://en.wikipedia.org/wiki/ISO_690>.
International Standard ISO 690:1987 bibliography standard
[W-SCRPAD]
Scratchpad. English Wikipedia [cited 2010/12/02].
Available at: <http://en.wikipedia.org/wiki/Scratchpad_RAM>

9.7 Information websites

[T-INFO]
Introduction To Tsukuba City, TsukubaCityInformation [cited 2010/09/14].
Available at: <http://tsukubainfo.jp/Main/IntroductionToTsukubaCity>.

[QSTOP]
QS World University Rankings Results 2010. QS Top Universities [cited 2010/09/14].
Available from WWW: <http://www.topuniversities.com/university-rankings/world-university-rankings/2010/results>.
[T-UNIV]
Outline of the University of Tsukuba. University of Tsukuba [cited 2010/09/14].
Available at: <http://www.tsukuba.ac.jp/english/public/booklets.html>.
[T-CCS1]
Facilities (English version). Center for Computational Sciences, University of Tsukuba [cited 2010/09/14].
Available at: <http://www.ccs.tsukuba.ac.jp/CCS/facilities-e.html>.
[T-CCS2]
Pamphlet (English version). Center for Computational Sciences, University of Tsukuba [cited 2010/09/14].
PDF file, available at: <http://www.ccs.tsukuba.ac.jp/CCS/files/pamphlet-E>.
[T-KDE]
Main Research Topics in KDE Laboratory (English version). Kitagawa Data Engineering laboratory [cited 2010/09/14].
Available at: <http://kde.cs.tsukuba.ac.jp/abstract_en.html>.
[T-INTERSC]
International Student Center . University of Tsukuba [cited 2010/09/14].
Available at: <http://www.japanese.intersc.tsukuba.ac.jp/>.
[JRA25]
JRA-25 Archive. JRA-25 Archive [cited 2010/09/14].
Available at: <http://gpvjma.ccs.hpcc.jp/~jra25/>
[TOP500]
TOP500 List - June 2006 (1-100). TOP500 Supercomputer sites [cited 2010/09/14].
Available at: <http://www.top500.org/list/2006/06/100>
[GRALIKE10]
Marco Gralike. ***Oracle 11g – XMLType Storage Options***. Personal blog [cited 2010/12/02].
Available at: <http://www.liberidu.com/blog/?p=203>
[DB2-XML]
Integrating XML with DB2 XML Extender and DB2 Text Extender,. IBM Redbook [cited 2010/12/02].
PDF file, available at: <http://www.redbooks.ibm.com/redbooks/pdfs/sg246130.pdf>
[PPBLOG]
Threads and blocks and grids, oh my!. /// Parallel Panorama /// blog [cited 2010/12/02].
Available at: <http://llpanorama.wordpress.com/2008/06/11/threads-and-blocks-and-grids-oh-my/>
[KENNEY05]
Paul E. McKenney. ***Memory Ordering in Modern Microprocessors, Part I***. Linux Journal #136, August 2005 [cited 2010/10/12].
Available at: <http://www.linuxjournal.com/article/8211>

Appendix A: CUDA to PTX1.4 full example

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

__device__ void myCopyFunction(void* dest_ptr,
void* src_ptr, size_t size) {
    char* dest = (char*)dest_ptr;
    char* src = (char*)src_ptr;

    while(size-- > 0) {
            *dest++ = *src++;
    }
}

__global__ void testKernel1(int* arg, void* mem)
{
    int temp = 20;
    *(void**)mem = (void*)&temp;
}

__global__ void testKernel2(int* arg, void* mem)
{
    myCopyFunction(arg, *(void**)mem, sizeof(int));
}

int main()
{
    int arg;
    size_t arg_size = sizeof(int);
    int *d_arg, *h_arg;
    void* d_mem;
    size_t mem_size = sizeof(void*);

    arg = 10;

    /* STEP 1: memory allocation for GPU exec. */
    /* alloc. GPU memory and copy 'arg' value */
    cudaMalloc(&d_arg, arg_size);
    cudaMalloc(&d_mem, mem_size);
    cudaMemcpy(d_arg, &arg, arg_size,
cudaMemcpyHostToDevice);

    /* alloc. CPU memory for the result */
    h_arg = (int*)malloc(arg_size);

    /* STEP 2: execution */
    /* exec GPU version */
    testKernel1<<<1,1>>>(d_arg, d_mem);
    testKernel2<<<1,1>>>(d_arg, d_mem);

    /* STEP 3: retrieve and check results */
    /* copy GPU result to CPU memory */
    cudaMemcpy(h_arg, d_arg, arg_size,
cudaMemcpyDeviceToHost);

    printf("%d\n", *h_arg);

    return 0;
}
```

```
    .version 1.4
    .target sm_10, map_f64_to_f32
    // compiled with /usr/lib//be
    // nvopencc 3.1 built on 2010-06-07

    //-----------------------------------------------------------
    // Compiling test.cpp3.i (/tmp/ccBI#.KR1vMQ)
    //-----------------------------------------------------------

    //-----------------------------------------------------------
    // Options:
    //-----------------------------------------------------------
    //  Target:ptx, ISA:sm_10, Endian:little, Pointer Size:64
    //  -O3 (Optimization level)
    //  -g0 (Debug level)
    //  -m2 (Report advisories)
    //-----------------------------------------------------------

    // ... content was removed here ...

    .entry _Z11testKernel1PiPv (
            .param .u64 __cudaparm__Z11testKernel1PiPv_arg,
            .param .u64 __cudaparm__Z11testKernel1PiPv_mem)
    {
    .reg .u64 %rd<4>;
    .local .s32 __cuda_local_var_22192_6_temp_0;
    .loc    17      14      0
$LDWbegin__Z11testKernel1PiPv:
    .loc    17      17      0
    mov.u64         %rd1, __cuda_local_var_22192_6_temp_0;
    ld.param.u64    %rd2, [__cudaparm__Z11testKernel1PiPv_mem];
    st.global.u64   [%rd2+0], %rd1;
    .loc    17      18      0
    exit;
$LDWend__Z11testKernel1PiPv:
    } // _Z11testKernel1PiPv

    .entry _Z11testKernel2PiPv (
            .param .u64 __cudaparm__Z11testKernel2PiPv_arg,
            .param .u64 __cudaparm__Z11testKernel2PiPv_mem)
    {
    .reg .u16 %rh<3>;
    .reg .u64 %rd<7>;
    .reg .pred %p<3>;
    .loc    17      20      0
$LDWbegin__Z11testKernel2PiPv:
    .loc    17      6       0
    ld.param.u64    %rd1,
[__cudaparm__Z11testKernel2PiPv_arg];
    .loc    17      7       0
    ld.param.u64    %rd2,
[__cudaparm__Z11testKernel2PiPv_mem];
    ld.global.u64   %rd3, [%rd2+0];
    mov.s64         %rd4, 3;
$Lt_1_1794:
 //<loop> Loop body line 7, nesting depth: 1,
estimated iterations: unknown
    .loc    17      10      0
    add.u64         %rd3, %rd3, 1;
    add.u64         %rd1, %rd1, 1;
    ld.global.s8    %rh1, [%rd3+-1];
    st.global.s8    [%rd1+-1], %rh1;
    .loc    17      9       0
    sub.u64         %rd4, %rd4, 1;
    mov.u64         %rd5, -1;
    setp.ne.u64     %p1, %rd4, %rd5;
    @%p1 bra        $Lt_1_1794;
    .loc    17      23      0
    exit;
$LDWend__Z11testKernel2PiPv:
    } // _Z11testKernel2PiPv
```

Appendix B: CUDA to PTX2.0 full example

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

__device__ void myCopyFunction(void* dest_ptr,
void* src_ptr, size_t size) {
    char* dest = (char*)dest_ptr;
    char* src = (char*)src_ptr;

    while(size-- > 0) {
            *dest++ = *src++;
    }
}

__global__ void testKernel1(int* arg, void* mem)
{
    int temp = 20;
    *(void**)mem = (void*)&temp;
}

__global__ void testKernel2(int* arg, void* mem)
{
    myCopyFunction(arg, *(void**)mem, sizeof(int));
}

int main()
{
    int arg;
    size_t arg_size = sizeof(int);
    int *d_arg, *h_arg;
    void* d_mem;
    size_t mem_size = sizeof(void*);

    arg = 10;

    /* STEP 1: memory allocation for GPU exec. */
    /* alloc. GPU memory and copy 'arg' value */
    cudaMalloc(&d_arg, arg_size);
    cudaMalloc(&d_mem, mem_size);
    cudaMemcpy(d_arg, &arg, arg_size,
cudaMemcpyHostToDevice);

    /* alloc. CPU memory for the result */
    h_arg = (int*)malloc(arg_size);

    /* STEP 2: execution */
    /* exec GPU version */
    testKernel1<<<1,1>>>(d_arg, d_mem);
    testKernel2<<<1,1>>>(d_arg, d_mem);

    /* STEP 3: retrieve and check results */
    /* copy GPU result to CPU memory */
    cudaMemcpy(h_arg, d_arg, arg_size,
cudaMemcpyDeviceToHost);

    printf("%d\n", *h_arg);

    return 0;
}
```

```
    .version 2.1
    .target sm_20
    // compiled with /usr/lib//be
    // nvopencc 3.1 built on 2010-06-07

    .visible .func _Z14myCopyFunctionPvS_m
(.param .u64 __cudaparmf1__Z14myCopyFunctionPvS_m,
 .param .u64 __cudaparmf2__Z14myCopyFunctionPvS_m,
 .param .u64 __cudaparmf3__Z14myCopyFunctionPvS_m)

    //--------------------------------------------------------
    // Compiling test.cpp3.i (/tmp/ccBI#.asnrJg)
    //--------------------------------------------------------

    //--------------------------------------------------------
    // Options:
    //--------------------------------------------------------
    //  Target:ptx, ISA:sm_20, Endian:little, Pointer Size:64
    //  -O3 (Optimization level)
    //  -g0 (Debug level)
    //  -m2 (Report advisories)
    //--------------------------------------------------------

    // ... content was removed here

    .visible .func _Z14myCopyFunctionPvS_m
 (.param .u64 __cudaparmf1__Z14myCopyFunctionPvS_m,
 .param .u64 __cudaparmf2__Z14myCopyFunctionPvS_m,
 .param .u64 __cudaparmf3__Z14myCopyFunctionPvS_m)
    {
    .reg .u32 %r<3>;
    .reg .u64 %rd<15>;
    .reg .pred %p<4>;
    .loc    17      5       0
$LDWbegin__Z14myCopyFunctionPvS_m:
    ld.param.u64    %rd1, [__cudaparmf1__Z14myCopyFunctionPvS_m];
    mov.s64         %rd2, %rd1;
    ld.param.u64    %rd3, [__cudaparmf2__Z14myCopyFunctionPvS_m];
    mov.s64         %rd4, %rd3;
    ld.param.u64    %rd5, [__cudaparmf3__Z14myCopyFunctionPvS_m];
    mov.s64         %rd6, %rd5;
    .loc    17      6       0
    mov.s64         %rd7, %rd2;
    .loc    17      7       0
    mov.s64         %rd8, %rd4;
    sub.u64         %rd9, %rd6, 1;
    mov.u64         %rd10, -1;
    setp.eq.u64     %p1, %rd9, %rd10;
    @%p1 bra        $Lt_0_1282;
    mov.s64         %rd11, %rd6;
    mov.s64         %rd12, %rd11;
$Lt_0_1794:
    //<loop> Loop body line 7, nesting depth: 1,
estimated iterations: unknown
    .loc    17      10      0
    add.u64         %rd8, %rd8, 1;
    add.u64         %rd7, %rd7, 1;
    ld.s8   %r1, [%rd8+-1];
    st.s8   [%rd7+-1], %r1;
    sub.u64         %rd9, %rd9, 1;
    mov.u64         %rd13, -1;
    setp.ne.u64     %p2, %rd9, %rd13;
    @%p2 bra        $Lt_0_1794;
$Lt_0_1282:
    .loc    17      12      0
    ret;
$LDWend__Z14myCopyFunctionPvS_m:
    } // _Z14myCopyFunctionPvS_m

    .entry _Z11testKernel1PiPv (
            .param .u64 __cudaparm__Z11testKernel1PiPv_arg,
            .param .u64 __cudaparm__Z11testKernel1PiPv_mem)
    {
    .reg .u64 %rd<4>;
    .local .s32 __cuda_local_var_24173_6_temp_0;
    .loc    17      14      0
$LDWbegin__Z11testKernel1PiPv:
    .loc    17      17      0
    mov.u64         %rd1, __cuda_local_var_24173_6_temp_0;
    ld.param.u64    %rd2, [__cudaparm__Z11testKernel1PiPv_mem];
    st.global.u64   [%rd2+0], %rd1;
    .loc    17      18      0
    exit;
$LDWend__Z11testKernel1PiPv:
    } // _Z11testKernel1PiPv

```

```
 80    .entry _Z11testKernel2PiPv (
 81            .param .u64 __cudaparm__Z11testKernel2PiPv_arg,
 82            .param .u64 __cudaparm__Z11testKernel2PiPv_mem)
 83    {
 84    .reg .u32 %r<3>;
 85    .reg .u64 %rd<7>;
 86    .reg .pred %p<3>;
 87    .loc    17      20      0
 88  $LDWbegin__Z11testKernel2PiPv:
 89    .loc    17      6       0
 90    ld.param.u64    %rd1,
[__cudaparm__Z11testKernel2PiPv_arg];
 91    .loc    17      7       0
 92    ld.param.u64    %rd2,
[__cudaparm__Z11testKernel2PiPv_mem];
 93    ldu.global.u64  %rd3, [%rd2+0];
 94    mov.s64         %rd4, 3;
 95  $Lt_2_1794:
 96   //<loop> Loop body line 7, nesting depth: 1,
estimated iterations: unknown
 97    .loc    17      10      0
 98    add.u64         %rd3, %rd3, 1;
 99    add.u64         %rd1, %rd1, 1;
100    ld.s8   %r1, [%rd3+-1];
101    st.global.s8    [%rd1+-1], %r1;
102    .loc    17      9       0
103    sub.u64         %rd4, %rd4, 1;
104    mov.u64         %rd5, -1;
105    setp.ne.u64     %p1, %rd4, %rd5;
106    @%p1 bra        $Lt_2_1794;
107    .loc    17      23      0
108    exit;
109  $LDWend__Z11testKernel2PiPv:
110    } // _Z11testKernel2PiPv
```

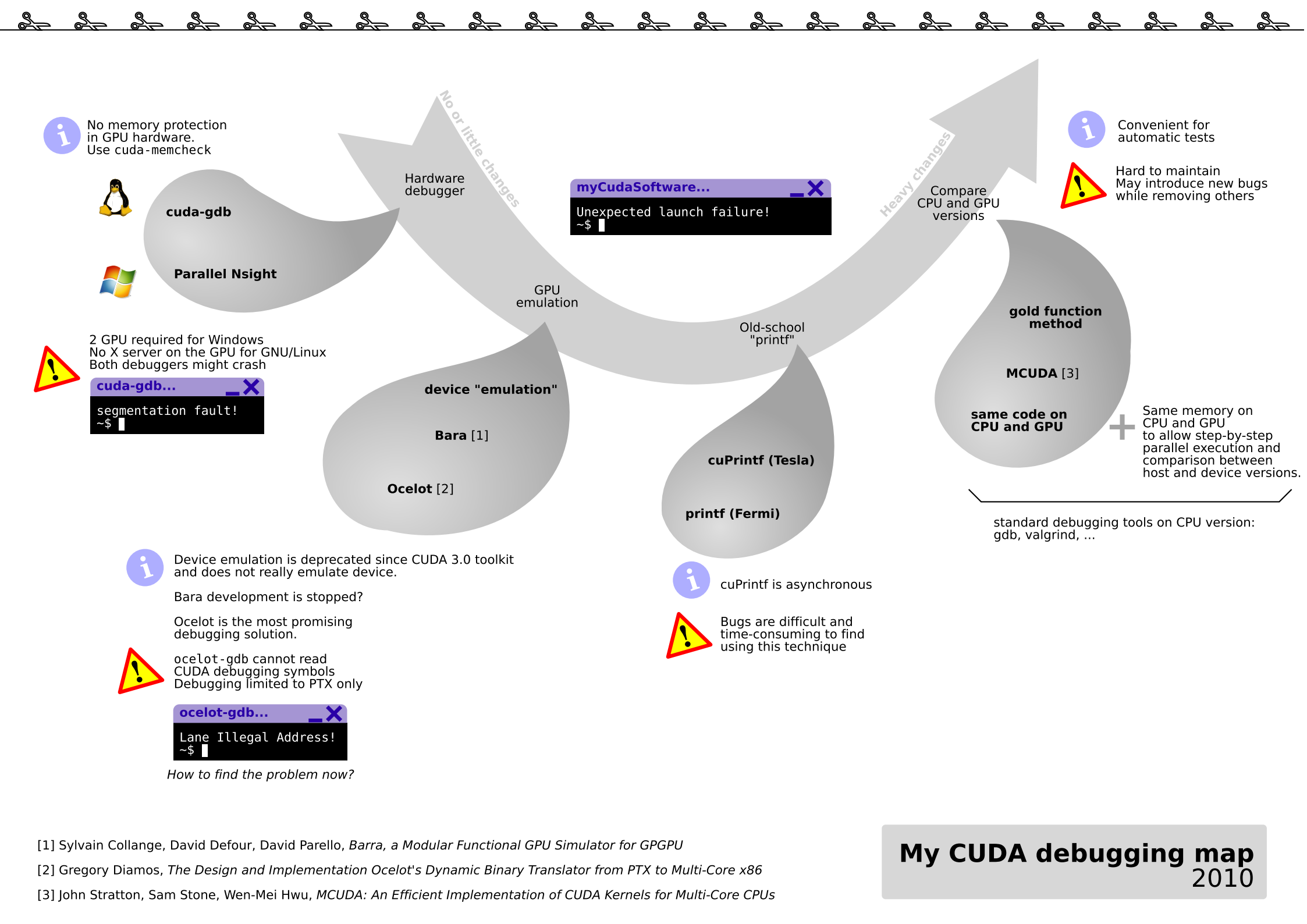
Appendix C: basic XPath grammar for lemon

```
/* primitive XPath BNF Grammar */

/* an xpath query 'xpath' is an expression 'expr' */
xpath(A) ::= expr(B).

/* an expression 'expr' can be:
 * - a relative path expression alone: /node_a/node_b/node_c
 * - an equality expression between a relative path expression and
 *   a primary expression: /node_a/node_b=value
 * - an equality between an attribute name and a primary expression
 *   (an attribute name can't be alone): @attr=value
 */
expr(A) ::= relativePathExpr(B).
expr(A) ::= relativePathExpr(B) EQUAL_SYMBOL primaryExpr(C).
expr(A) ::= attribute(B) EQUAL_SYMBOL primaryExpr(C).

/* a relative path expression can be:
 * - a single step (=just one XML tag name): node_a
 * - a child symbol and a single step: /node_a
 * - a descendant symbol and a single step://node_a
 * - a relative path and a child (= path/element_name)
 *   recursive declaration: /node_a//node_b/node_c
 * - a relative path and a descendant (= path//element_name)
 *   recursive declaration: /node_a//node_b//node_c
 */
relativePathExpr(A) ::= stepExpr(B).
relativePathExpr(A) ::= CHILD_SYMBOL stepExpr(B).
relativePathExpr(A) ::= DESC_SYMBOL stepExpr(B).
relativePathExpr(A) ::= relativePathExpr(B) CHILD_SYMBOL stepExpr(C).
relativePathExpr(A) ::= relativePathExpr(B) DESC_SYMBOL stepExpr(C).

/* each single step expression is can be:
 * - or a primary expression and a predicate list:
 *   node_a[predicate1][predicate2][...]
 * - just a primary expression: node_a
 */
stepExpr(A) ::= primaryExpr(B) predicateList(C).
stepExpr(A) ::= primaryExpr(B).

/* a list of predicate can be:
 * - just one predicate (= [predicate])
 * - more predicates (= [predicate1][predicate2][...])
 *   recursive declaration
 */
predicateList(A) ::= predicate(B).
predicateList(A) ::= predicateList(B) predicate(C).

/* a predicate is an expression between brackets (= [expr]) */
predicate(A) ::= BRACKET_L expr(B) BRACKET_R.

primaryExpr(A) ::= LITERAL_VALUE(B).

attribute(A) ::= ATTR_SYMBOL LITERAL_VALUE(B).
```

Appendix D: v_array full example

```
/*
// generate cuda_* version of all v_array functions
g++ -E -DCUDA_DEVICE_MODE "path_to_v_array"/v_mem_manager.c > cuda_v_mem_manager.c
g++ -E -DCUDA_DEVICE_MODE "path_to_v_array"/v_array.c > cuda_v_array.c
// compile
nvcc -arch sm_11 -Xcompiler "-fpermissive" -I"path_to_v_array" -L"path_to_v_array" -lv_array -o example example.cu
*/

/* for printf, malloc, memcpy and memcmp */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* for v_mem_manager_* and v_array_* functions */
#include <v_mem_manager.h>
#include <v_array.h>

/* host wrapper functions using OS alloc. func. */
__host__ void standard_malloc_func(void** ptr, size_t size, void* user_data) {
  *ptr = malloc(size);
}
__host__ void standard_memcpy_func(void* dest_ptr, void* src_ptr, size_t size) {
  memcpy(dest_ptr, src_ptr, size);
}
__host__ void standard_free_func(void* ptr, void* user_data) {
  free(ptr);
}

/* host wrapper functions using CUDA runtime */
__host__ void cuda_malloc_func(void** ptr, size_t size, void* user_data) {
  cudaMalloc(ptr, size);
}
__host__ void cuda_memcpyH2D_func(void* dest_ptr, void* src_ptr, size_t size) {
  cudaMemcpy(dest_ptr, src_ptr, size, cudaMemcpyHostToDevice);
}
__host__ void cuda_memcpyD2H_func(void* dest_ptr, void* src_ptr, size_t size) {
  cudaMemcpy(dest_ptr, src_ptr, size, cudaMemcpyDeviceToHost);
}
__host__ void cuda_free_func(void* ptr, void* user_data) {
  cudaFree(ptr);
}

__device__ void cuda_mem_malloc_func(void**, size_t, void*);
__device__ void cuda_mem_memcpy_func(void*, void*, size_t);
__device__ void cuda_mem_free_func(void*, void*);

/* The following workaround is used because CUDA code cannot call a __device__ function from another file */
#include "cuda_v_mem_manager.c"
#include "cuda_v_array.c"

/* device wrapper functions using memory manager */
__device__ void cuda_mem_malloc_func(void** ptr, size_t size, void* user_data) {
  *ptr = cuda_v_mem_manager_allocate_chunk((v_mem_manager_data_t*)user_data);
}
__device__ void cuda_mem_memcpy_func(void* dest_ptr, void* src_ptr, size_t size) {
  char* dest = (char*)dest_ptr;
  char* src = (char*)src_ptr;

  while(size-- > 0) {
        *dest++ = *src++;
  }
}
__device__ void cuda_mem_free_func(void* ptr, void* user_data) {
  cuda_v_mem_manager_free_chunk((v_mem_manager_data_t*)user_data, ptr);
}

/* CUDA kernel */
__global__ void exampleKernel(v_mem_manager_data_t* mem, unsigned int array_row_num, V_ARRAY_OFFSET_T* result)
{
  v_array_t a1, a2;
  V_ARRAY_OFFSET_T offset_a1, offset_a2;
  v_array_iter_t i1, i2;
  int i, value;

  /* multiple threads cannot safely access a critical section */
  if(threadIdx.x == 0) {
        /* create a new array elements */
        a1 = cuda_v_array_new_elem(1, V_ARRAY_FALSE, sizeof(int), array_row_num, NULL, mem);
        a2 = cuda_v_array_new_elem(2, V_ARRAY_FALSE, sizeof(int), array_row_num, NULL, mem);

        /* compute offsets from pointers */
        offset_a1 = cuda_v_array_get_offset(mem, a1);
        offset_a2 = cuda_v_array_get_offset(mem, a2);

        /* create iterators */
        i1.offset = offset_a1;
        i1.idx = 0;
        i2.offset = offset_a2;
        i2.idx = 0;

        /* insert some data */
```

```
        for(i=0; i<2; i++) {
                value = i + blockIdx.x;
                i1 = cuda_v_array_append_data(mem, i1, &value, NULL, NULL);
                i2 = cuda_v_array_append_data(mem, i2, &value, NULL, NULL);
        }

        /* append a2 to a1 */
        i1.offset = offset_a1;
        i1.idx = 0;
        cuda_v_array_append_copy(mem, i1, a2, NULL, NULL);

        /* store array offset for host */
        result[blockIdx.x] = offset_a1;
  }
}

int main()
{
  unsigned int i, block_num = 1, mem_chunk_num = 10, array_row_num = 3;
  size_t mem_chunk_size = sizeof(v_array_elem_t) + (sizeof(int)*array_row_num);
  size_t result_size = block_num * sizeof(V_ARRAY_OFFSET_T);
  v_mem_manager_data_t *d_mem;
  V_ARRAY_OFFSET_T *d_result, *h_result;
  v_array_iter_t iter;

  /* STEP 1: memory allocation */
  /* create GPU memory manager from CPU */
  d_mem = v_mem_manager_new_data(mem_chunk_num, mem_chunk_size, &cuda_malloc_func, NULL, &cuda_memcpyH2D_func);
  /* allocate GPU result array */
  cuda_malloc_func((void**)&d_result, result_size, NULL);
  /* allocate CPU result array */
  standard_malloc_func((void**)&h_result, result_size, NULL);

  /* STEP 2: execution */
  /* exec GPU version */
  exampleKernel<<<block_num, 1>>>(d_mem, array_row_num, d_result);

  /* STEP 3: retrieve results and display them */
  /* copy GPU result to CPU memory */
  cuda_memcpyD2H_func(h_result, d_result, result_size);
  /* print arrays */
  for(i=0; i<block_num; i++) {
          /* copy GPU array to CPU memory */
          v_array_t h_array = v_array_pre_copy(d_mem, NULL, h_result[i], standard_malloc_func, cuda_memcpyD2H_func);
          /* print array's content */
          iter.offset = (V_ARRAY_OFFSET_T)h_array;
          iter.idx = 0;
          printf("%d [", i);
          while(v_array_iter_inside_right(NULL, iter)) {
                  iter = v_array_get_row(NULL, iter);
                  printf("%d ", *(int*)iter.result.data);
                  iter = v_array_iter_next(iter);
          }
          printf("]\n");
          /* desallocate host copy */
          v_array_free(NULL, (V_ARRAY_OFFSET_T)h_array, standard_free_func, standard_memcpy_func);
  }

  /* STEP 4: memory desallocation */
  cuda_free_func(d_mem, NULL);
  cuda_free_func(d_result, NULL);
  standard_free_func(h_result, NULL);

  return 0;
}
```



[1] Sylvain Collange, David Defour, David Parello, *Barra, a Modular Functional GPU Simulator for GPGPU*

[2] Gregory Diamos, *The Design and Implementation Ocelot's Dynamic Binary Translator from PTX to Multi-Core x86*

[3] John Stratton, Sam Stone, Wen-Mei Hwu, *MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs*

My CUDA debugging map
2010

Keywords

20 - Public service
13 - Research
02 - Algorithms, 06 - Databases
03 - Software (data analysis)

JORDAN Vincent ST50&AHPM internship report - P2010

Abstract

XML is a machine-readable language designed to be both simple and extendable. It is widely used in computer world for data storage and communication therefore the need of querying XML data is high. The purpose of the research carried out during my stay at Kitagawa Data Engineering laboratory is to evaluate the possibility of using *General Purpose Graphic Processing Unit* (GPGPU) to handle this task.

GPGPU is a recent creation which gather research attention because of its large diffusion and cheap price. GPGPU is the result of the evolution of GPU coprocessors toward more flexibily and programable features in visual rendering. The latest features of these chips enable them to be used as stream processors.
Stream processing is a highly data parallalized task in which a series of operation is applied to each element of a data set (a stream).
Research on parallel XML query processing has already be done in the same laboratory by Imam Machdi. My main work is to create a GPU algorithm based on his PhD thesis results. This task is challenging since current XML processing algorithms do not fit in the stream processing paradigm.

Two solutions are possible to solve this issue:
- overcome GPU limitations in order to do more than stream processing
- create a new algorithm which is stream processing compliant

During this internship, the first solution has been studied.

Kitagawa Data Engineering laboratory
University of Tsukuba
Tennoudai 1-1-1, Tsukuba, Ibaraki, Japan 305-8573

University of Tsukuba