DOI:10.15774/PPKE.ITK.2014.009

RACER data stream based array processor and algorithm implementation methods as well as their applications for parallel, heterogeneous computing architectures

Ádám Rák

A thesis submitted for the degree of
Doctor of Philosophy

Scientific adviser:
György Cserey, Ph.D.

Faculty of Information Technology and Bionics
Péter Pázmány Catholic University

Budapest, 2014

DOI:10.15774/PPKE.ITK.2014.009

Acknowledgements

I thank Interdisciplinary Technical Sciences Doctoral School of Pázmány Péter Catholic University, Faculty of Information Technology and Bionics, and its senior masters, Professor Dr. Tamás Roska, and Professor Dr. Péter Szolgay for the encouragement and advisement during my studies.

I would like to thank György Cserey, for his support, encouragement, advices and inspiration.

I thank my colleagues for their advices and with whom I could discuss all my ideas: Gergely Balázs Soós, Gergely Feldhoffer,

Ákos Tar, József Veres, Balázs Jákli, Norbert Sárkány, Gábor Tornai, Miklós Koller, István Reguly, Csaba Józsa, András Horváth, Attila Stubendek, Domonkos Gergely, Mihály Radványi, Tamás Fülöp, Tamás Zsedrovits, Csaba Nemes and Gaurav Gandhi.

I thank Vida Tivadarné her patience and devoted work to make the administrative issues much easier, the help of PPCU's dean's office and the help of academic and financial department.

The support of grants Nos. TÁMOP-4.2.1/B-11/2-KMR-2011-0002 and TÁMOP-4.2.2/B-10/1-2010-0014 are gratefully acknowledged.

I am grateful to Anna for providing her friendship and our discussions even her always busy schedule.

I am very grateful to my mother and father and to my whole family who always believed in me and supported me in all possible ways.

DOI:10.15774/PPKE.ITK.2014.009

Abstract

This dissertation (i) describes polyhedron based algorithm optimization method for GPUs and other many core architectures, describes and illustrates the loops, data-flow dependencies and optimizations with polyhedrons, defines the memory access pattern, memory access efficiency ratio and absolute access pattern efficiency, and presents problem decomposition; (ii) introduces a new data stream based array processor architecture, called RACER and presents the details of the architecture from the programming principle to the applied pipeline processing; (iii) presents a new algorithmic approach developed to evaluate two-electron repulsion integrals based on contracted Gaussian basis functions in a parallel way, provides distinct SIMD (Single Instruction Multiple Data) optimized paths which symbolically transforms integral parameters into target integral algorithms.

DOI:10.15774/PPKE.ITK.2014.009

Contents

DOI:10.15774/PPKE.ITK.2014.009

DOI:10.15774/PPKE.ITK.2014.009

DOI:10.15774/PPKE.ITK.2014.009

List of Figures

DOI:10.15774/PPKE.ITK.2014.009

DOI:10.15774/PPKE.ITK.2014.009

DOI:10.15774/PPKE.ITK.2014.009

DOI:10.15774/PPKE.ITK.2014.009

DOI:10.15774/PPKE.ITK.2014.009

DOI:10.15774/PPKE.ITK.2014.009

DOI:10.15774/PPKE.ITK.2014.009

Chapter 1

Introduction

In recent years a new direction has started in the world of computing, which is based on increasing the number of cores and execution units rather than the clock frequency of the processors. This trend is manifested in all of the network devices, desktop computers and even in cell phones. The main reason for this can be traced back to physical laws, as the miniaturization of microchips and the increase of the clock frequency led to a much too long communication time between the remote parts of the processor. This delay is caused mostly by wiring and metal connections of the chip. The further increase of the clock frequency is therefore not only impeded by a limit determined by the silicon's switching speed, but it also increases the experienced value of the delay. Too much delay implicates more fragmentation of the architecture into several execution units and cores.

According to Moore's law, the manufacturing cost of digital integrated electronics per transistor is becoming cheaper. This will help the above mentioned direction further, as in a well-designed multiprocessor system, the increase of the number of cores is a simple task. This way not only more and more transistors, but more and more cores (or, raw computing power, increasing at the same rate as defined in Moore's law) are gained for the same price.

This trend is lead by graphics processing unit (GPU), which is achieved and even exceeded the number of 5760 cores per microchip in 2014. These multi-core or many-core systems are DSP, FPGA, CELL and GPU, but this trend encompasses the embedded, multimedia processors too.

DOI:10.15774/PPKE.ITK.2014.009

Besides the rapid development of the hardware, the question arises how these architectures can be programmed efficiently. Many-core processor systems show not only more variety than traditional predecessors, but require fundamentally new programming approach. In order to integrate as many cores as possible in a processor unit, the computational units were simplified as much as possible. Practically most of the results of the last twenty years had been thrown away from single processor core optimization. Which focused on a single processor core optimization. Thus the difference between a simple computational unit, e.g. Floating-Point Unit, and a core with full functionality is not clear. The functional differences between many-core and traditional processors are illustrated by that if a strict serial program has been executed on a many-core processor, then the running time is often 100 times slower than running it on a non-parallel CPU. The standard OpenCL programming language has been created to program such a new parallel systems. OpenCL is a low level C language, which pushes off the problem of parallelization of the algorithms to the programmers.

The efficient implementation of an algorithm requires the deep knowledge of the target architecture. Based on experiences, this knowledge is necessary even while using OpenCL language because the smallest optimization solutions can be speed up the program by a few orders of magnitude. The problem is even more complicated, because the manufacturer (NVIDIA, AMD-ATI) changes its architecture in every year and fundamentally redesigns it in every two years. Moreover it is common that the manufacturer has difficulties to understand its own product and exploit its advantages.

There is significant demand to have solutions that can automate the parallel implementation of algorithms with mathematically backed methods. This includes those methods too, where implementing algorithms efficiently on a new architecture is assisted by machine learning.

Considering these problems, my aim was to analyze the parallelization of general algorithm classes and demonstrate my results and methods on a few difficult algorithms.

The trend is obvious, the number of cores per processor will increase exponentially in the next five-ten years. However, the difference between each, following

DOI:10.15774/PPKE.ITK.2014.009

architectures is not only the number of processors, but the changes of architectures. This evolution leads not only to higher number of processing units, but to the more efficient and optimized operation and also to the increased computational power per area. If we examine the parallel architectures, we find that the objective is to maximize the general purpose computing power per unit area by employing trade-offs. These trade-offs and disadvantages at the most common architectures are the following:

The difficulties of memory reading and writing of CPUs are hidden by using traditional cache hierarchy. This solution, especially if we have more processor units, increases untenable the ratio between chip area of cache memory and chip area of pure computing. It is a good balance for the less computationally intensive tasks, but quite wasteful in case of scientific or graphical computations.

DSP: digital signal processor. These devices are very similar to CPUs, the difference is mainly between their parameters. DSPs are designed for running signal processing algorithms efficiently (FFT, matrix-vector operations) with low power consumption and competitive price. The chip area (ie. the cost of manufacturing) is much smaller than CPUs', because of the above reasons, DSPs has less cache memory. Therefore the system memory access patterns of DSPs is more restricted if we want to exploit the available bandwidth.

The vector based SIMD (single instruction multiply data) architecture of GPUs (graphical processing unit) introduces a very strong constraint on the implementation of threads. In a workgroup every thread has to do the same operation on different data, reading the data from adjacent memory. Therefore both the memory bandwidth and computing resource utilization of the silicon area are very high. But working with this architecture the programmer has to solve the efficient use of memory, contrary to the CPU, this system does not hide the architecture details and does not solve the related problems.

Cell BE (cell broadband engine): this is a hybrid architecture, which includes a classic PowerPC CPU processor connected to SPUs (synergistic processing units). The SPUs are very simplified vector processing units, which have relatively large local memory on chip. The programmers are responsible to solve even every tiny technical problems, from the appropriate feeding of the pipeline to organize the

DOI:10.15774/PPKE.ITK.2014.009

internal logic of the memory operations. This device has only indirect memory access via the local memory.

FPGA (field programmable gate array): On this architecture, arbitrary logic circuit can be implemented within certain broad limits. Usually the implemented circuit is relatively efficient, since the desired circuit is realized physically on the FPGA by connecting on-chip switches. Consequently the logic circuits of the FPGA can be adapted directly to the given task, therefore this architecture can exploit most efficiently the available processing units. However the cost of this enormous flexibility is the low density of the processing units on the chip surface, since the switching circuits and universal wiring need large chip area.

Systolic Array: this classical topological array processor architecture contains effectively only execution (computing) units, adder and multiplier circuits, which are usually solve some linear algebra operations in parallel. Its applicability is very limited, because its topology is specific for the executed algorithm. This architecture does not contain neither memory architecture, nor program control structure. These units should be provided by another system. The flexibility is sacrificed for efficiency, since the computing units utilized almost 100 percentage during operation and the surface of the silicon chip contains effectively only computing units.

CNN (cellular nonlinear/neural networks): this architecture is efficient at using local image processing operations (low resolution image processing algorithms on grayscale images) with extremely high speed and low power consumption. Every pixel is associated to a processing unit, the process is analog and there is only a very little analog memory. Accessing the global memory compared to the internal speed is very slow and also needs the digitalization of the pixels. This architecture is optimized for 2D topological computations with low memory.

Considering these problems, my aim was to design a computational architecture (RACER architecture), which is not limited by the disadvantages of the previous parallel architectures, Turing complete and fully general algorithms can be implemented efficiently on it, moreover its performance per area is maximized as much as possible.

The dissertation is organized as follows. Chapter 2 describes polyhedron based algorithm optimization method for GPUs and other many core architectures, de-

DOI:10.15774/PPKE.ITK.2014.009

scribes and illustrates the loops, data-flow dependencies and optimizations with polyhedrons, defines the memory access pattern, memory access efficiency ratio and absolute access pattern efficiency, and presents problem decomposition. Chapter 3 introduces a new data stream based array processor architecture, called RACER and presents the details of the architecture from the programming principle to the applied pipeline processing. Chapter 4 presents a new algorithmic approach developed to evaluate two-electron repulsion integrals based on contracted Gaussian basis functions in a parallel way, provides distinct SIMD (Single Instruction Multiple Data) optimized paths which symbolically transforms integral parameters into target integral algorithms. Chapter 5 summarizes the main results and highlights further potential applications, where the contributions of this dissertation could be efficiently exploited.

The author's publications and other publications connected to the dissertation can be found at the end of this document.

DOI:10.15774/PPKE.ITK.2014.009

Chapter 2

Polyhedron based algorithm optimization method for GPUs and other many core architectures

This chapter introduces a new method of a compiler application to many-core systems. In this method, the source code is transformed into a graph of polyhedrons, where memory access patterns and computations can be optimized and mapped to various many core architectures. General optimization techniques are summarized.

2.1 Introduction

The multi-core devices like GPUs (Graphics Processing Unit) are currently ubiquitous in the computer gaming market. Graphical Processing Units, as their name implies are used mostly for real-time 3D rendering in games which is not only a highly parallel computation, but also needs great amounts of computing resources. Originally, this need encouraged the development of massively parallel thousand core GPUs. However these systems can be used to implement not only computer games, but also other topological, highly parallel scientific computations. Manufacturers are recognizing this market demand, and they are giving more and more general access to their hardware, in order to aid the usage of GPU for general purpose computations. This trend has recently resulted in relatively

DOI:10.15774/PPKE.ITK.2014.009

cheap and very high performance computing hardwares, which opened up new prospects of computationally intensive algorithms.

New generation hardwares contain more and more processing cores, sometimes over a few thousand, and the trends show that these numbers will exponentially increase in the near future. The question is how developers could program these systems and may port already existing implementations on them. There is a huge need for this today as well as in the forthcoming period. This new approach of the automation of software development may change the future techniques of computing science.

The other significant issue is that GPUs and CPUs have started merging for the biggest vendors (Intel, NVIDIA, AMD). This means that developers will need to handle heterogeneous many core arrays, where the amount of processing power and architecture can be radically different between cores. There are no good methodologies for rethinking or optimizing algorithms on these architectures. Experience in this area is a hard gain, because there seems to be a very rapid ($\approx$3 year) cycle of architecture redesign.

Exploiting the advantages of the new architectures needs algorithm porting, which practically means the complete redesign of the algorithms. New parallel architectures can be reached by "specialized" languages (DirectCompute, CUDA, OpenCL, Verilog, VHDL, etc.). For successful implementations, programmers must know the fine details of the architecture. After a twenty years long evolution, efficient compiling for CPU does not need detailed knowledge about the architecture, the compiler can do most of the optimizations. The question is obvious: Can we develop as efficient GPU (or other parallel architecture) compilers as the CPU ones? Will it be a two decade long development period again or can we make it in less time?

Every algorithm can be seen as a solution to a mathematical problem. The specification of this problem describes a relationship from the input to the output. The most explicit and precise specification can be a working platform independent reference implementation, which actually transforms the input from the output. Consequently, we can see the (mostly) platform independent implementation, as a specification of the problem. This implicates that we can see the parallelization

DOI:10.15774/PPKE.ITK.2014.009

as a compiling problem, which transforms the inefficient platform independent representation into an efficient platform dependent one.

Parallelization must preserve the behavior in the aspect of specification to give the equivalent results, and should modify the behavior concerning the method of the implementation. Automated hardware utilization has to separate the source code (specification) and optimization techniques on parallel architectures [9].

The polyhedral optimization model [53] is designed for compile-time parallelization using loop transformations. Runtime parallelization approaches are based on TLS (Thread-Level Speculation) method [54], which allows parallel code execution without the knowledge of all dependencies. Researchers are interested in algorithmic skeletons [52] recentely. Usage of skeletons is effective if the parallel algorithms can be characterized by generic patterns [55]. Code patterns address runtime code optimizations too.

There are different trends and technical standards emerging. Without the claim of completeness, the most significant contributions are the following: OpenMP [16] - it supports multi-platform shared-memory parallel programming in C/C++ and FORTRAN, practically it adds pragmas for existing codes, which direct the compiler. OpenCL [66] - is an open, standard C-language extension for the parallel programming of heterogeneous systems, also handling memory hierarchy. Threading Building Blocks of Intel [17] - is a useful optimized block library for shared memory CPUs, which does not support automation. One of the automation supported solution providers is the PGI Accelerator Compiler [18] of The Portland Group Inc., but it does not support C++. There are application specific implementations on many-core architectures, one of them is a GPU boosted software platform under Matlab, called AccelerEyes' Jacket [20]. Overviewing the growing area, there are partially successful solutions, but there is no universal product and still there are a lot of unsolved problems.

2.1.1 Parallelization conjecture

My conjecture is that any algorithm can be parallelized, even if inefficiently. This statement disregards the size of the memory, which can be a limiting factor, but serial algorithms suffer from the same problem too, so this is still a fair

DOI:10.15774/PPKE.ITK.2014.009

comparison. In order to define this statement in a mathematically correct way, I need a simple definition of the parallelization potential of non-parallel programs. For easier mathematical treatment we can disregard the effect of limited global memory of the device.

Given $\forall P_1$ non-parallel program, with the time complexity $\Omega(f(n)) > \mathcal{O}(1)$, let us suppose that $\exists P_M$ efficient parallel implementation on M processors with the time complexity $\mathcal{O}(g(n))$ where:

$$\lim_{M\to\infty, n\to\infty} \frac{f(n)}{g(n)} = \infty \tag{2.1}$$

This means that given big enough inputs, where the input size is n, and an arbitrary huge number of core, we can achieve arbitrary big speedup over the serial implementation of the algorithm. From Equation 2.1 we can derive a practical measure of how well an algorithm can be implemented efficiently on a parallel system:

$$\eta(M) = \lim_{n\to\infty} \frac{f(n)}{g(n)} \quad \eta(M) \geq \mathcal{O}(1) \tag{2.2}$$

In case of arbitrary big input size n, we can achieve only a speedup limited by the number of cores. I call this speedup $\eta(M)$, which is a function of the number of cores the architecture has. This limit depends on both the implemented algorithm and the architecture, so $\eta(M)$ can be called parallelization efficiency. It can describe in an abstract way how much the implemented algorithm is parallelization friendly. It can be seen that achieving practical speedup is equivalent to $\eta(M) > \mathcal{O}(1)$. Consequently if $\eta(M) = \mathcal{O}(1)$ then the problem is not (efficiently) parallelizable so, Equation 2.1 does not hold.

Some brute-force algorithmic constructions for many-cores yield inefficient speedup, for example Equation 2.3, which means that the measured speedup of the algorithm is the square root of the number of parallel cores.

$$\eta(M) \geq \sqrt{M} \tag{2.3}$$

DOI:10.15774/PPKE.ITK.2014.009

This limit is very arbitrary, but in my opinion anything lower than $\sqrt{M}$ speedup is not economical, because this scales very well for small architectures, where the number of cores is below 16, like CPUs, but is very impractical for GPUs, where the number of cores is measured in thousands.

I can state that anything with a parallelization efficiency less than $\sqrt{M}$ is practically not parallelizable in the many-core world, however that is why I chose the lower limit of my conjecture to be $\sqrt{M}$.

2.1.2 Ambition

Parallelization is a very difficult and potentially time consuming job if done manually. Fully automating it is a computationally impossible task, but even partially automating it, on a set of algorithmic classes can greatly increase the productivity of the human programmer.

My aim was to describe algorithms using a general mathematical representation, which enables us to tackle the algorithm parallelization more formally, and hopefully more easily. I used polyhedral geometric structures to describe the loop structure of the program, which is used in modern optimizing compilers for high level optimizations. However, it was unable to represent more complicated dynamic control structures and dynamic dependencies inside loops, so I had to extend the theory to encompass a wider range of algorithmic classes.

In a static loop, we know everything about the control-flow behavior of the loop in compile time, this can be easily extended by allowing the loop bounds to be known just before the start of execution of the whole loop structure [22, 23]. A dynamic loop, or control-flow, in the other hand cannot be known in compile time, every decision in the code happens in run-time depending on the results of the ongoing computations.

While the classical polyhedral approach covers many useful algorithms, it can be very lacking in practice, because the lack of dynamic control handling. My observation is that by just including a few more very constrained algorithmic classes containing dynamic control can practically cover most useful cases, especially if we only consider GPU programming. This approach simplifies the original difficult problems into fitting the algorithms into these classes, after that we have

DOI:10.15774/PPKE.ITK.2014.009

recipes for implementing these classes on the given architectures.

2.2 Polyhedrons

Because computation time generally concentrates in loops, it is practical to depict the algorithm as a graph of loops, where the graph represents the data-flow between the different loops. These loops are the hot points of the algorithm and we can run them on parallel architectures for better performance. Hot point means that most of the execution time is concentrated inside them. This is a very general way of porting algorithms, but there are numerous issues with this approach that I aimed to mitigate.

When we execute an algorithm on a parallel architecture, it usually boils down scheduling loops on cores. This in practice means, that we try to map the parallel execution onto the parallel hardware units, in both space and time. In order to optimize and schedule them better, we can convert loops into polyhedrons. This allows a mathematical approach where each execution of the core of the loop is a single point of the polyhedron, because (well behaving) loops are strictly bounded iterative control structures. This mapping between discrete geometric bodies and loops is a straightforward transformation if possible.

Discrete polyhedra can be defined multiple ways with linear discrete algebra. I defined these loop polyhedra as generic filled geometric shapes in a discrete euclidean space bounded by flat faces. The exact definition can vary by context in the literature. My definition is as follows:

$$\begin{aligned} \text{Points: } & \bar{x} \in \mathbb{N}^d \\ \text{Bounding inequality system: } & \mathbf{M} \cdot \begin{bmatrix} \bar{x} \\ 1 \end{bmatrix} \geq \bar{0} \text{ where } \mathbf{M} \in \mathbb{R}^{(d+1)\mathbf{x}n} \\ \text{Points of the polyhedra:} & \mathbb{P} := \left\{ \bar{x} \middle| \mathbf{M} \cdot \begin{bmatrix} \bar{x} \\ 1 \end{bmatrix} \geq \bar{0} \right\} \\ & F^{\Pi}_{filter} := \mathbb{P} \rightarrow \{true, false\} \\ & K_{kernel} := \{\partial_{R1} ... \partial_{Rn}, \mathcal{S}, \partial_W\} \end{aligned} \tag{2.4}$$

Definition 2.4 defines the polyhedron corresponding to a loop structure. The polyhedron contains points of a d dimensional space of loop variables. The given

DOI:10.15774/PPKE.ITK.2014.009

matrix inequality defines a convex object in this space. All points of this convex object are part of the polyhedron, these points define set $\mathbb{P}$.

In the definition, $\bar{x}$ is a point of the polyhedron, matrix $\mathbf{M}$ defines the faces of the polyhedron. In the algorithm $\bar{x}$ corresponds to each execution of the core of the loop structure, and $\mathcal{M}$ corresponds to the bounds of the loops. These bounds can be linearly dependent on other loop indexes, which allows us to rotate and optimize the polyhedron.

It is desirable to allow minimal dynamic control-flow in this formalism, so I have defined F^{Π}_{filter}, which is a scheduling-time executable function to decide the subset of polyhedron nodes that we would like to execute. This means that this function is quasi static in the sense that it had to be decided just before we start to execute the loop structure, but we do not know the result of this function before that. If we run a scheduler before the loops, to map them into parallel execution units, we can use this F^{Π}_{filter} function to filter out the polyhedral nodes early. This way we can prevent the scheduling of empty work to the execution units. It is important to note that filtered non-functional parts should be in minority, otherwise the polyhedral representation is useless for optimization, and it is processing a sparsely indexed structure instead of an actual polyhedron.

The K_{kernel} represents the kernel operation to be executed in the nodes, this is the formal representation of the core of the loop structure. I treat this code as sequential data-flow, which has memory reads at the start, and writes at the end. In Definition 2.4 $\partial_{R1}...\partial_{Rn}$ are the memory reads, $\mathcal{S}$ is the sequential arithmetic, and ∂_W is the memory write. If we cannot fit the extra control-flow inside the kernel into our representation, we can treat the rest inside as data-flow function $\mathcal{S}$, but any side effects should be included in the memory operations ∂_R, ∂_W.

Because of the possibly overlapping memory operations, internal dependencies arise, and possibly between polyhedrons too. We depict dependency set as $\mathbb{D}$ with the following definition:

DOI:10.15774/PPKE.ITK.2014.009

$$\begin{aligned}
\mathbb{D} &:= \{f_i | f_i : \mathbb{P} \to \mathbb{P}\} \\
f_i(\bar{x}) &:= round\left(\mathbf{D}_i \cdot \begin{bmatrix}\bar{x} \\ 1\end{bmatrix}\right) \text{ where } \mathbf{D}_i \in \mathbb{Q}^{(d+1)\mathbf{x}d} \\
F^D_{filter} &:= \mathbb{D}x\mathbb{P} \to \{true, false\} \text{ Dependency condition} \\
f_i(\bar{x}) &\text{ can be evaluated at scheduling-time in } \mathcal{O}(1) \text{ time}
\end{aligned} \tag{2.5}$$

Dependency set is a function which maps from nodes of the polyhedron to nodes of the polyhedron. In practice, it defines that a given node of the polyhedron depends on which nodes must precede it in execution time. A dependency set can be defined as multiple maps. Most algorithmic cases can be covered by a linear dependency mapping, which can be represented as a linear transformation on the index vector.

Dependencies arise from the overlapping memory operation, mostly because one operation uses the result of another one, or less likely they update the same memory and one overwrites the output of another.

The domain of Matrix $\mathbb{D}$ is rational numbers, this allows more general transformations, so we can represent most dependencies by a linear transformation. In simple cases it is an offset, but it can be a rotation as well, e.g. the swap (or mirror) of the coordinates. Most of the time linear transformations should have enough power of representation, because the original algorithms are created by human intelligence, and these loops tend to have linear dependencies, otherwise it would be too difficult to understand.

In practice however, there are dependencies which are dynamic/conditional and can only be evaluated during execution time. I propose the formal treatment of dynamic dependencies, in the case where we can evaluate them no later than the start of the loop structure containing the mentioned dependency. It is possible since the dynamic dependencies, like the previously mentioned F^{Π}_{filter} filter function, are constraints on the scheduling, and we are doing scheduling just before executing the loops. This is the reason why the dynamic behavior of dependencies is treated as F^D_{filter} filter function, which is very similar to the F^{Π}_{filter} kernel filter function. This F^D_{filter} function represents the dynamic part, it is essentially a condition which states which dependencies on which nodes should be counted as valid.

DOI:10.15774/PPKE.ITK.2014.009

In the following I define the memory access patterns similarly to dependencies. The access pattern maps from the nodes of the polyhedron to the memory storage indexes.

A single memory access ∂ in the kernel is described in the following:

$$\begin{aligned} \partial_R, \partial_W &: \text{read and write operations} \\ g &: \mathbb{P} \to \mathbb{N}^n \\ g(\bar{x}) &:= round\left(\mathbf{A} \cdot \begin{bmatrix} \bar{x} \\ 1 \end{bmatrix}\right) \text{ where } \mathbf{A} \in \mathbb{Q}^{(d+1)\mathbf{x}n} \end{aligned} \tag{2.6}$$

Where a member of $\mathbb{N}^n$ is an n dimensional memory address, and $g(\bar{x})$ is the access pattern.

It is interesting to note that the memory can be n dimensional as well. Although the memory is usually handled as a serial one dimensional vector, in some cases it can be a 2D image mapped topologically into a 2D memory. While the memory chips are 2D arrays physically, they are addressed linearly from the processing units. However, GPUs have hardware supported 2D memory boosted by 2D coherent cache in order to help the speed of 3D and 2D image processing. Therefore it is useful to manage higher dimensional memory. It is even more practical from a mathematical point of view since, the polyhedra made from loops tend to be multidimensional, we can make the memory access pattern optimizations much easier if we can match the dimensionality of the memory indexing to the polyhedron.

The similarity between the definition of access patterns and dependencies is not a coincidence as I have mentioned previously, overlapping or connected memory operation can imply dependencies. However, the importance of access patterns does not stop here, because most architectures are sensitive to memory access. Some architectures can only do very restricted memory access patterns, like FPGA, or systolic arrays. Others are capable of general random access, like CPUs and GPUs, however the bandwidth difference between access patterns can be over an order of magnitude. It is very important to optimize also this practical aspect, because such bandwidth differences matter greatly in engineering practice.

DOI:10.15774/PPKE.ITK.2014.009

2.2.1 Handling dynamic polyhedra

It was mentioned in the previous section that the most important part of this work is formalizing the handling of dynamic polyhedra, which includes both conditional execution and conditional dependencies. When we process static loops we generate a walk of polyhedron off-line. This walk essentially implies an optimized loop structure and perhaps a mapping to parallel cores. Generating a walk however is not possible for dynamic polyhedra, because essential information is missing at compile time.

The obvious solution is generating the walk in runtime, which can have a performance drawback. In extreme cases this can take longer than the actual computation we want to optimize. My solution to mitigate this problem is that we do not need to process everything in execution time. All the static dependencies can be processed offline, and the dynamic dependencies can be regarded as worst cases in compile time. This in itself does not solve the original problem, but opens up the avenue to generate code for the scheduler. The job of this scheduler is to generate the walk during runtime, but due to the knowledge we statically know about the polyhedron, this scheduler code is another polyhedron itself. Our aim is to make the structure of the scheduler simpler, and hopefully less dynamic like the original problem, because otherwise this solution would be an infinite recursion.

In practice this means that the schedule should be somewhere halfway between an unstructured scheduler and a fully static walk. In order to avoid confusion with the static walk, I am calling the ”dynamic walk” **Plan**. This name is more befitting, because in the implementation the **Plan** contains only indexes, instructing which thread should process which part of the polyhedron.

The unstructured scheduler in this context means that the scheduler cannot effectively use the fact that it is processing a polyhedral representation. It is a simple algorithm which iterates through each point of the polyhedron, and schedules them when their dependency is satisfied. This can be greatly improved if the scheduler only looks at the possible cases, by using the static information.

Since the **Plan** governs the order of the execution, it determines the access patterns of the physical memory, this implicates that the formalism should include

DOI:10.15774/PPKE.ITK.2014.009

this relationship as well. The following **Plan** related notations and definitions are used in this paper: **Plan** (see Equation 2.8): $\mathcal{P}(\Pi)$ is an optimized dynamic walk of the polyhedron. It is the job of the scheduler to generate this before the execution of the loop structure, after that the loop structure is executed according to this **Plan**.

Plan efficiency (see Equation 2.9): $\eta(\mathcal{P})$ is closely related to parallelization efficiency. It well characterizes the hardware computing resources according to the plan, according to the pipeline and computing unit utilization. In other words if all the pipelines are optimally filled in all compute units (cores), we can say that this efficiency is 1. We can define this efficiency as a ratio of available computing resources and the computing resources used by when we execute the **Plan**.

Access pattern (see Equation 2.10): $\partial(\mathcal{P}, \Pi)$ depicts the memory access pattern of the polyhedron while it is using the **Plan**.

Access pattern efficiency: $\eta(\partial(\mathcal{P}, \Pi))$ characterizes the efficiency of the utilization of the hardware memory bandwidth. This is similar to the **Plan efficiency**, but instead of computing resource, we have memory bandwidth.

2.2.2 Loops and data-flow dependencies in polyhedra

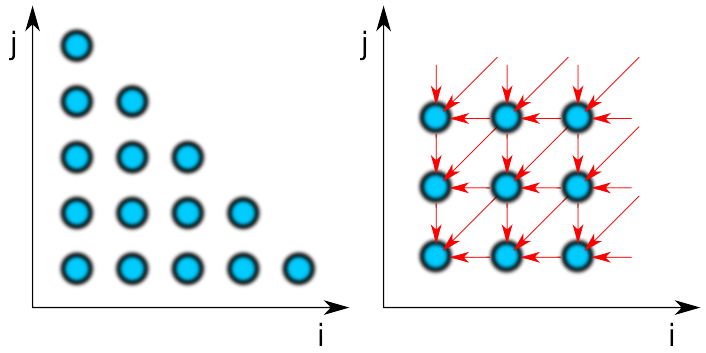
In this section I would like to present two examples in order to aid the understanding of the concepts behind polyhedral loop optimization. The first example is a loop with two index variables i, j, which can be represented by a discrete polyhedron depicted in Figure 2.1(a):

```
for(int i=0;i<5;i++)
for(int j=0;j<(5-i);j++)
{
  S1(i, j);
}
```

The polyhedron is two dimensional, because there are two loop variables and it is a triangle because the bound of the second variable j is dependent on the first i. There are no dependencies depicted, and the core of the loop is represented by an $S1(i, j)$ general placeholder function.

In the following I would like to present a more realistic example:

DOI:10.15774/PPKE.ITK.2014.009



(a) (b)

Figure 2.1: (a) Geometric polyhedron representation of loops in the dimensions of i, j loop variables. (b) Another example, 2D integral-image calculation, where data-flow dependencies are represented by red arrows between the nodes of the polyhedron. The dependency arrows point in the inverse direction compared to the data-flow.

```
for (int i = 0; i < n; i++)
for (int j = 0; j < m; j++)
{
  int sum = array[i][j];
  if (i > 0) sum += array[i-1][j];
  if (j > 0) sum += array[i][j-1];
  if (i > 0 and j > 0)
    sum -= array[i-1][j-1];
  array[i][j] = sum;
}
```

This example presents an integral-image calculation, where the algorithm calculates the integral of a 2D array or image in a memory-efficient way. The equivalent polyhedron can be seen in Figure 2.1(b) where the dependencies are depicted by arrows, which is practically the same as the indexing of the memory reads. It was chosen because this polyhedron has a complicated dependency and access pattern, which will serve as a good example optimization target later.

While the first example does not benefit from polyhedral optimization, as it can be easily parallelized anyways, the second example will be examined in depth in the next section.

DOI:10.15774/PPKE.ITK.2014.009

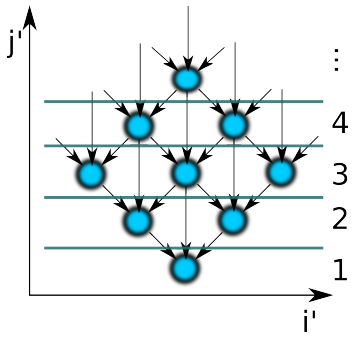


Figure 2.2: Polyhedrons can be partitioned into parallel slices, where parallel slices only contain independent parts of the polyhedron. For an example, rotating the polyhedron depicted in Figure 2.1(b), an optimized parallel version can be transformed.

2.2.3 Optimization with polyhedrons

The polyhedral representation of the loops provides a high level geometrical description where mathematical methods and tools can be used. Affine transformations can be applied on polyhedrons. These approaches can be used for multi-core optimizations. Polyhedrons can be partitioned into parallel slices, where parallel slices only contain independent parts of the polyhedron and they can be mapped to parallel execution units. For an example, rotating the polyhedron depicted in Figure 2.1(b), an optimized parallel version can be transformed, see Figure 2.2. These rotations can be easily done by polyhedral loop optimizer libraries [24], but the rotated/optimized code is not presented here because the resulting loop bound computations are of little interest for the human reader.

Geometric transformations can be used to optimize the access patterns too, not just the parallelization, because they transform the memory accesses along the polyhedron. These transformations should be done off-line (compile time) even for dynamic polyhedra, because there is usually no closed formula for getting them. However, in this optimization we search for an optimal affine transformation in the sense that it should allow the highest bandwidth while still obeying the dependencies and the parallelization. This is a general constrained discrete optimization problem, and thanks to the low dimensionality, we can easily afford to solve this by a genetic algorithm, or random trials, if we have an accurate

DOI:10.15774/PPKE.ITK.2014.009

model of the architecture.

If we want to handle dynamic polyhedra, we need to compute the worst case for the dependencies while we try to find a parallelization, and later on, in execution time, we use the transformed polyhedra in the scheduler. In this scenario, the complexity of the scheduler can be greatly reduced, because the problem has been reduced in the off-line optimizations. A practical example would be if the dependencies in Figure 2.2 were actually dynamic. In this case the parallelization is the same, but for some nodes the dependencies are missing. This is an optimization opportunity because we can possibly execute more nodes in parallel, so the scheduler can look to the next slice if there are available nodes. Thanks to the off-line transformations, the scheduler only needs to look at the polyhedra in (the compile time determined) parallel slices, which greatly simplifies the algorithm. Even more, as previously mentioned the structured way of the scheduling algorithm means that it is in itself a regular loop structure, which can optimized the same way.

2.2.4 Treating optimization problems

If we need to do high level polyhedral optimizations on real-world algorithms, we need to complete several steps depicted on Figure 2.3 and explained in more detail below. Most of these steps can be done automatically for well behaving algorithms, but some need human intelligence, especially to ensure that we chose the most well behaving realization of the algorithm to analyze.

I have identified the most common steps of this process:

1. Take the most natural, and the least optimized version of the algorithm
   This is very important to help the formal treatment of loop structure, because the more optimized the algorithm, the more complicated its loop structure tends to be, which can hide many aspects from the optimizations.

2. Consider the algorithm as a set of loops and find the computationally complex loops
   This step can be done semi-automatically, with static code analysis and benchmarking, most compilers already support this (GCC, LLVM)

DOI:10.15774/PPKE.ITK.2014.009

3. Represent the problem as mathematical structures (polyhedra)
   This is a bijective mapping between loop structures and polyhedra, a purely mathematical transformation.

4. Discover dependencies in-loop and between loops
   Loop dependency tracking is relatively simple after we formalize the loops, there is at least partial support for this already in GCC and LLVM.

5. Eliminate as many dependencies as possible
   Some of the trivial dependencies can be eliminated automatically, but most of them need a human hand. We need this step, because dependencies prevent parallelization and constrain the transformations, limiting out the ability to optimize.

6. Quantify $\Pi, \partial, \mathcal{P}$
   We formally quantify the shape of the polyhedra, the memory operations and their implied dependencies, and the possible ways for scheduling the loops.

7. Find the best transformation for parallelization
   This is a static polyhedral optimization (we disregard the dynamic parts), there are useful solutions in the literature [21, 25].

8. Estimate speed based on η
   We can estimate the speed by the simulation of the transformed algorithm, so we can guess our efficiency.

9. Optimize: transform $\Pi, \partial, \mathcal{P}$ to increase speed
   We apply geometric transformation to the polyhedra, in order to increase the efficiency of parallelization, and memory access.

2.3 Problems beyond polyhedrons

Of course, not every problem has a corresponding polyhedron representation, which can be transformed easily and automatically into parallelized form. There

DOI:10.15774/PPKE.ITK.2014.009

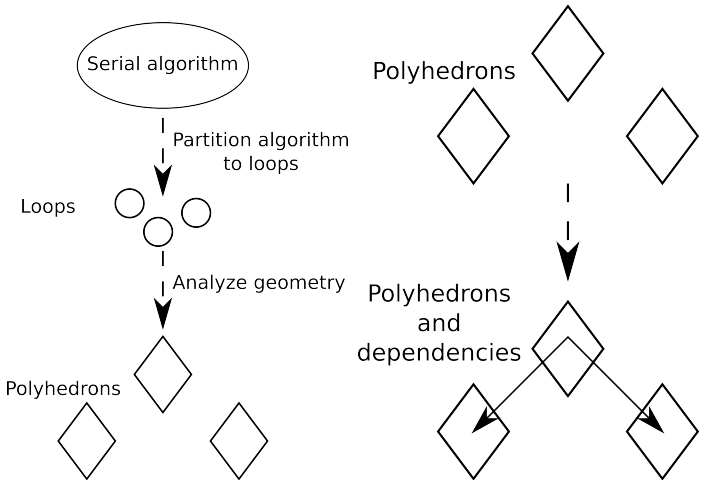


Figure 2.3: Schematic overview of the steps we need to take before we can apply geometric polyhedral transformation for optimization.

are a number of cases, which need human creativity to find appropriate solutions or sometimes we do not even know any effective solutions for them. In the following, two such examples are presented.

2.3.1 Dot-product, a simple example

Let us examine the simple dot-product example in the following code comparing to its polyhedron representation in Figure 2.4(a).

```
result = 0;
for (int i = 0; i < n; i++)
  result += vector1[i]*vector2[i];
```

Unfortunately, there is no usable polyhedral transformation available here. In this example, data-flow dependencies force a strict order of the execution. These dependencies are connected through an associative addition operator. So the solution of this problem here is to rearrange the order of the associative operators, see Figure 2.4(b), which will create the well known parallel reduction.

DOI:10.15774/PPKE.ITK.2014.009

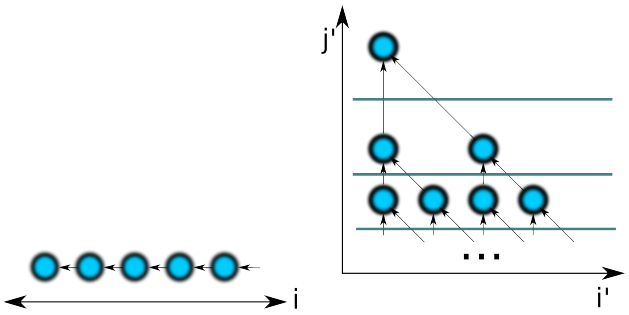


Figure 2.4: Polyhedron representation of the dot-product example (a). Rearranging the parentheses in associative chains gives a possible solution for its parallelization (b).

2.3.2 Irreducible reduction

We often encounter more complex algorithms, for example reduction, where the transformation is not trivial:

$$x_{n+1} = f(x_n, y_n) \tag{2.7}$$

Where the y_n is the input, and the final x_n is the output of this general reduction scheme. If the f function is not associative, then we cannot simply use the parallel reduction. In this case, we have to investigate the f function deeply, in order to convert the iteration into a parallelizable representation.

I can state, that my polyhedral transformation, and parallelization methods, can handle the complexity up to the associative operations. This means that if there are dependencies in the polyhedron, it is only possible to break them up, if they are connected by associative operators. More complex dependencies are not breakable, and they can possibly prevent the parallelization. However if the loop structure has enough dimensions, it still might be possible to find a parallelizable part of the algorithm.

DOI:10.15774/PPKE.ITK.2014.009

2.4 High-level hardware specific optimizations

Computational complexity usually concentrates in loops. Loops can be represented by polyhedrons, these are important building blocks of the program. Branching parts of the code can be reduced into a sequential code, or sometimes can be built into the polyhedrons, depending on the conditions. After eliminating the implicit side effects, the sequential code can be translated to pure data-flows. Applying these methods, only a bunch of polyhedrons - connected together in a data-flow graph - have to be optimized. As we have already seen, this is not trivial in itself, but after we have this representation, we can move forward to the optimization problem.

2.4.1 Kernel scheduling to threads

Kernel execution scheduling is a mapping of the polyhedron nodes to the symbolic plane of time $\times$ core id. Horizontal and vertical barriers can be defined on this plane. The horizontal barriers are synchronization points of the time axes, so they separate different parts of the execution in time and the vertical barriers separate the core groups. Dependencies cross horizontal barriers parallel with the vertical barriers (Figure 2.5). Actually, the most important task of this hardware aware scheduling is to place these barriers. After the barrier separation, the micro-scheduling is usually trivial inside the groups.

The vertical barriers usually represent a physical separation between the computing core groups. In the case of GPUs, it means that the multiprocessors are separated by the vertical lines. It is very important that the dependencies do not intersect the vertical lines, but intersect the horizontal lines. If these forbidden intersections actually happen, then on most architectures we lose the ability to ensure correctness of the dependency, because the hardware usually decides the exact timing of the parallel schedule.

In the groups separated by the barriers, the hardware specific micro-scheduling is executed, which is trivial because we do not have to take into account the dependencies.

DOI:10.15774/PPKE.ITK.2014.009

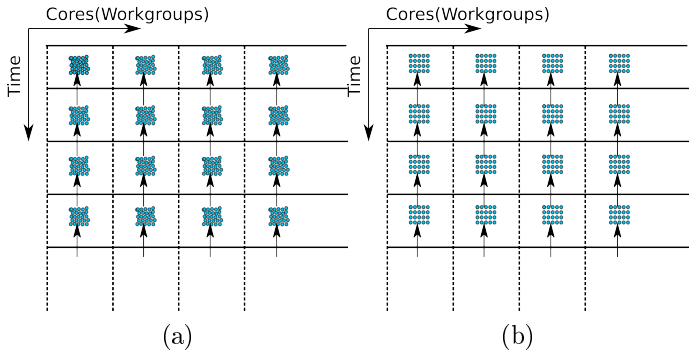


Figure 2.5: Scheduling polyhedron nodes to the time × core plain. In the first step we place horizontal and vertical barriers, which segment the plane into groups (a). The horizontal barriers are synchronization points of the time axes, so they separate different parts of the execution in time and the vertical barriers separate the core groups. In the next step microschedule the inside the groups to obtain the final structure (b).

A Plan is a parallel walk of the polyhedron, defined in the following way:

$$\mathcal{P}(\Pi) : \mathbb{N}x\mathbb{N} \rightarrow \begin{cases} \mathbb{P} \\ (\text{NOP}) \end{cases} \tag{2.8}$$

Indexing of the plan is : $\mathcal{P}(\Pi(t,i))$ where t is time and i is the core id.

The plan maps the polyhedron to the nodes of a polyhedron. It is possible that this set includes symbolic empty items too, because not all nodes have functional operations. The efficiency of a plan can be defined by simply dividing the number of polyhedron nodes by the area of the plane (which is equal with the number of nodes multiplied by time). Obviously the dependencies will limit the maximum efficiency.

Plan efficiency defines the effectiveness of the usage of computing resources according to the plan:

$$\begin{gathered} \mathcal{P}(t,i) \quad t \in [1;T] \quad i \in [1;I] \\ \eta(\mathcal{P}) = \frac{\sum |\mathbb{P}_i| t_{P_i}}{T \cdot I} \end{gathered} \tag{2.9}$$

DOI:10.15774/PPKE.ITK.2014.009

2.5 Memory access

On a multi-core architecture we need to keep the utilization of both the cores and memory bandwidth at optimal levels. Improving the core utilization has been discussed already in depth in previous sections. In this section the focus is on how to improve the memory bandwidth after we have achieved the parallelization and handled the dependencies. Many-core architectures are less reliant on traditional memory caching, because they cannot put enough cache memory into every core, due to the chip area constraints. Therefore the memory access of each core has to be coordinated in a way which is close to the preferred access pattern of the main memory. This memory is almost aways physically realized by DRAM technology, which prefers burst transfers. Burst transfers are continuous in address space, so when cores are accessing the memory, the parts of the memory accessed by different cores should be close to each other. Older GPUs [26] mandate that each thread should access the memory in a strict pattern dictated by their respective thread IDs, otherwise the memory bandwidth is an order of magnitude lower than optimal. Newer GPUs [34, 27] use relatively small cache memories for reordering the memory transfer in real-time, consequently we only need to keep the simultaneous memory accesses close together, but there is no dependence between the relative memory address and the thread IDs. These constraints on optimal memory access patterns underline the importance of the access pattern optimization, however even if we look at the traditional CPU cache coherency, we can find that there are optimizations possible too, if we wish to achieve the maximal performance, so these optimizations are important regardless of the architecture.

2.5.1 Access pattern ($\partial(\mathcal{P}, \Pi)$) and relative access pattern efficiency ($\eta(\partial(\mathcal{P}, \Pi))$)

I have formally defined the access pattern including its dependence on the runtime walk of the polyhedron, which is the plan. The access pattern can be seen as a product of the data storage pattern and the walk of the polyhedron. Together these two contain where and when the program accesses the memory.

So we can formally write:

DOI:10.15774/PPKE.ITK.2014.009

$$\partial(\mathcal{P}, \Pi) := \partial \circ \mathcal{P} \tag{2.10}$$

Where the $\partial(\mathcal{P}, \Pi)$ is the access pattern which depends on the parallelization.

2.5.2 Memory access efficiency ratio (θ)

We can write the memory bandwidth efficiency as η, which is the ratio of the full theoretical bandwidth and the achieved bandwidth. Usually achieving the theoretical maximum is unfeasible, so we can depict maximal achievable efficiency as η_{best}. Consequently we can depict the lowest possible bandwidth by η_{worst}, in this case we deliberately force the worst possible access pattern to try to lower the bandwidth of the memory access. We can define an interesting attribute:

$$\theta := \frac{\eta_{best}}{\eta_{worst}} \tag{2.11}$$

Where θ is the memory access efficiency ratio. This number can describe, in a limited way, the sensitivity of the architecture to the access pattern of the memory. Bigger θ usually means that the architecture is more sensitive to the memory access pattern, and we need to be more careful in the optimization. The important limitation of this number is that it does not tell us anything about the access patterns themselves.

2.5.3 Absolute access pattern efficiency ($\eta(\partial(\Pi))$)

If we want to optimize the access pattern, we can approach the problem from two sides. The first is to optimize the storage pattern of the data we want to access. This is constrained by the fact that we usually need to access the same data from different polyhedra, so the different storage patterns may be optimal for different places of accesses, but we can only choose one. The second way is to optimize the plans($\mathcal{P}$), which are runtime walk of the polyhedra. This is done independently from other polyhedra which access the same data, however we are constrained by the polyhedral structure, the parallelization and the internal polyhedral dependencies.

DOI:10.15774/PPKE.ITK.2014.009

For easier handling of the optimization, we wish to, at least formally, eliminate the dependence of the access pattern efficiency on the plan($\mathcal{P}$).

Let the absolute access pattern efficiency be:

$$\eta(\partial(\Pi)) \approx \max_{\mathcal{P}} \left(\eta(\partial(\mathcal{P}, \Pi))\right) \tag{2.12}$$

In other words, the absolute access pattern efficiency is the maximal achievable access pattern efficiency by only changing the plan($\mathcal{P}$). This eliminates the dependency on the plan, so the storage pattern optimization can take place.

This definition seems quite nonconstructive, since it implicitly assumes that we somehow know the best possible solution. However, the polyhedral optimization is a relatively low dimensional problem, and the dependencies also constrain it even more, which means that the plan has an even lower degree of freedom, so low that we can even perform exhaustive search. Very often this means searching in one degree of freedom. As a consequence it is affordable to compute $\eta(\partial(\Pi))$.

2.5.4 Coalescing

In GPU programming terminology memory access coalescing means that each thread of execution accesses memory in the same pattern as their IDs as depicted in Figure 2.6 and Figure 2.7a. This is usually true for the indexes of the processing cores as well. This coalescing criterion only has to hold locally, for example, on every group execution threads, but not between the groups. This minimal size of these groups is a hardware parameter.

On GPUs coalesced access is necessary for maximizing the memory bandwidth, however for modern GPUs [34, 27] the caches can do fast auto-coalescing. This means that the accesses should be close together, so the cache can collect them into a single burst transfer for optimal performance.

2.5.5 Simple data parallel access

This is the ideal data parallel access as depicted on Figure 2.7a, where every thread of execution reads and write only once, in other words there is a linear mapping between cores and memory. GPUs are principally optimized for this,

DOI:10.15774/PPKE.ITK.2014.009

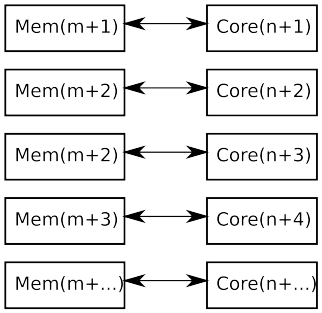


Figure 2.6: Typical coalescing pattern used on GPUs, where the core or thread IDs correspond to the accessed memory index

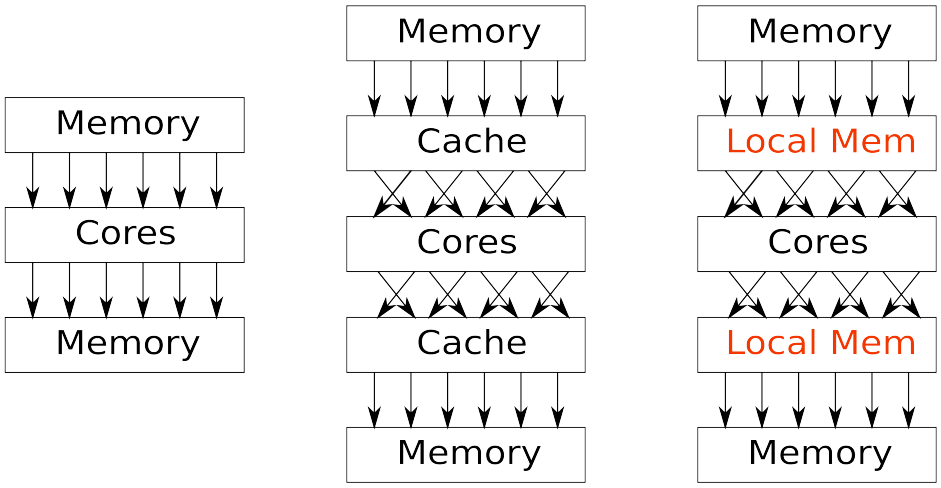
because this is very typical in some image processing tasks, e.g. pixel-shaders. This access pattern is highly coalesced by definition, this can achieve the highest bandwidth on GPUs.

2.5.6 Cached memory access

If the effects of caching are significant, mostly because they are big enough, we can optimize for cache locality. Consequently we achieve much higher bandwidth than the main memory has, because if our memory accesses are mostly local, and stay inside the cache, they do not trigger actual main memory transfers. However, highly spread-out memory accesses trigger main memory transfers, but due to the logical page structure of the memory, these transfers are even worse, because every transfer triggers a transfer of a whole page to/from the main memory.

The size and bandwidth of the various levels of the cache hierarchy are very important factors, sometimes even more important than the bandwidth of the main memory. All modern CPUs are optimized for this operation, and newer GPUs also contain enough cache, so this might be relevant for them too. This is depicted on Figure 2.7b.

DOI:10.15774/PPKE.ITK.2014.009



(a) (b) (c)

Figure 2.7: (a) A simple coalesced memory access pattern. (b) Random memory access aided by cache memory. (c) Explicitly using the local memory for shuffling the accesses in order to achieve the targeted memory access pattern.

2.5.7 Local memory rearranging for coalescing

Local memory rearranging is a GPU technique for achieving more coalesced memory access as depicted in Figure 2.7c. I would like to emphasize that this optimization can be automatized in my formal mathematical framework, which would offload a lot of work from the human programmer. Furthermore this is the most important step in linear algebra algorithms implemented on GPUs [28], because complex but regular access patterns routinely arise in these algorithms.

Essentially this is similar to caching, but thanks to the precise analysis based on the polyhedral model we know the exact access patterns. Therefore instead of using general heuristic caching algorithms, we can determine the storage pattern of the data inside the local memory, which would maximize the memory bandwidth. This would always perform significantly better than caching for representable problems in this framework. On some GPUs [34] there is enough caching for significantly speed up the non-coalesced access, this can be seen in Figure 2.8, where run times of an 8192×8192 matrix transposition algorithm are

DOI:10.15774/PPKE.ITK.2014.009

GPU name	Full-coalesced run time	Non-coalesced write run time	Non-coalesced read run time
Tesla C1060	35406us	483123us(13.6×)	487898us(13.8×)
Tesla C2050	6700us	10600us(1.58×)	12800us(1.9×)
GeForce GTX 780	4650us	5860us(1.26×)	8420us(1.81×)

Figure 2.8: Run times of an 8192×8192 matrix transposition algorithm are depicted on NVIDIA GPU architectures. In the Full-coalesced case I use local memory to achieve coalescing for memory reads and memory writes at the same time. The other two cases are naive implementations, where either the reads or the writes are coalesced only. In the case of Tesla C2050 there is a noticeable improvement which is thanks to the sophisticated and relatively large caches on the NVIDIA Fermi architecture. The NVIDIA GTX780 contains further improved caching which reflects of the benchmark times.

depicted. It can be seen that both the run times and the coalesced/non-coalesced ratios are improving due to the improvements of the caches.

After we have processed the optimization problem into polyhedra and the access patterns the local memory rearranging can be seen as a local memory sized re-indexing of the data, and computing this becomes feasible. Thanks to the low dimensionality of the problem, and the simple locality constraint on modern GPUs, this optimization can be done by brute-force trying out re-indexing schemes. We will not be able to cover the whole search space on re-indexing this way, however this should produce near optimal solutions in most cases due to the simplicity of the constraints.

This technique is an especially good target for automatic optimization, because of its relative simplicity for computers, and significant complexity for human programmers.

2.6 Algorithmic classes

The algorithmic classes, or more precisely the control-flow data-flow structures which this theory can handle, can be defined by relaxing the constraints of Static Control Parts. The most basic classes of algorithms which can be represented here are the ones with static loop structures and branches, which are only linearly

DOI:10.15774/PPKE.ITK.2014.009

dependent on the loop variables. In this case the loop bounds are allowed to be parametric, but these bound should be known right before the execution of the loop. The dependencies should also be linear, which implies that the arrays are only indexed with a linear expression of the loop variables.

The first relaxation is that I allow the conditional execution of the core of loop to depend on contents of arrays, where the arrays are also linearly indexed, and they are known before the start of the loop execution.

The second relaxations of the constraints is that I allow the dependencies to be conditional in a similar fashion to the conditional execution of loop cores.

2.7 GPU implementation of H.264 video encoder

The H.264 ITU video standard is currently one of the most widely used video coding. This algorithms consist of two bigger parts, the lossy encoding and the lossless encoding. The first part, the lossy encoding consists of two main branches, the inter and intra coding. The second part is a form of entropy coding, which includes the Context-adaptive variable-length coding (CAVLC). The Constrained Baseline Profile feature set of the H.264, which defines the features of both parts was implemented.

This video encoding achieves very good compression ratios at acceptable qualities by employing sophisticated prediction schemes in the lossy part of the encoding. The inter coding represents inter frame encoding where the current frame is predicted from the previous frames, and only the difference (residual) of the actual and predicted frame is stored. The prediction employs block matching to determine the local motion vectors of the image block-by-block. In the version which I have implemented, I use macroblock granularity (16×16 pixes), but the H.264 allows finer block sizes too. By default, everything is processed in macroblocks: motion vectors, inter, intra coding and CAVLC. This has an important implication to the memory representation, because if each thread processes a macroblock, then they should be able to do coalesced access. In my implementation the images are stored in both macroblock coalesced pattern, and in images with 2D caching. There is one exception when the access does not have the macroblock granularity, and this is when we process the previous image. During the

DOI:10.15774/PPKE.ITK.2014.009

motion vector search, we associate a motion vector with every macroblock, but these vectors point to a non-aligned 16×16 block of pixels in the previous frame, therefore we need to store the previous frame in image format also. The motion estimation and the inter coding are parallel for every macroblock.

The intra encoding unlike the inter encoding does not use the previous frames, but instead predicts the macroblock from the top and left neighbors as depicted on Figure 2.9. This dependency results from the particular way that the lossy compression works: every predictive lossy encoding must be followed by a decoding step in order to generate the reference image. This is important, because we cannot use the original images as references in the prediction, they are not accessible in the decoder. When we want to decompress/decode the video, we only have the decoded frame, therefore in order to preserve consistency we need to generate the decoded frames in the encoder too.

The top left neighbor dependencies are designed in a way that there are no circular dependencies. This method fits very well to the serial row-major processing of the intra macroblocks. However this dependency severely limits the number of active parallel threads, because we can only process an intra block if we already have the top and left neighbor computed.

Because the intra coded macroblocks are not dependent on previous frames, they are used as key frames (I-frame), which allows seeking inside the video. However in H.264 the intra blocks are much flexible: inside P-frames the encoder can chose to mix inter and intra blocks arbitrarily in order to achieve better compression or better quality. These choices are governed by heuristics which are not in the scope of this thesis.

The CAVLC encoding is parallel like inter encoding, where every macroblock can be encoded in parallel. However the actual situation is much more complex because parts of the frame are intra encoded, other parts are inter encoded, and the static non moving parts are not encoded at all. Consequently we do not have to run CAVLC for some macroblocks at all.

The block diagram of the implemented H.264 GPU encoder is depicted on Figure 2.10, and the lossy parts of the encoder are depicted on Figure 2.11 which contains the intra and inter coding.

DOI:10.15774/PPKE.ITK.2014.009

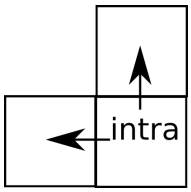


Figure 2.9: Top and left macroblock dependence of the intra coded blocks due to the in-frame prediction

2.7.1 Static polyhedra

The static polyhedral structure of this problem which can be seen on Equation 2.13 is two dimensional and its bounds are frame sizes ($N \times M$) measured in the number of macroblocks. For the intra blocks there are two dependencies, the top dependency, and the left dependency. Because of the static nature of this approach and because of we can have arbitrary mix of inter and intra macroblocks in a frame, the number of parallel threads becomes restricted to the worst case when every macroblock is intra. This practically means that if the frame has $N \times M$ macroblocks, than the $min(N, M)$ will be maximal number of active parallel threads. This is considerably worse than the best case $N * M$. This can be easily improved by handling the two branches separately, but that still fails to significantly improve the intra processing speed.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & N-1 \\ 0 & -1 & M-1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \geq \bar{0} \tag{2.13}$$

The memory access pattern is very regular too, the thread which is processing the (x, y) point of the polyhedron accesses only the (x, y) points of the data structures. The exception is the intra prediction which also accesses $(x-1, y)$ and $(x, y-1)$.

DOI:10.15774/PPKE.ITK.2014.009

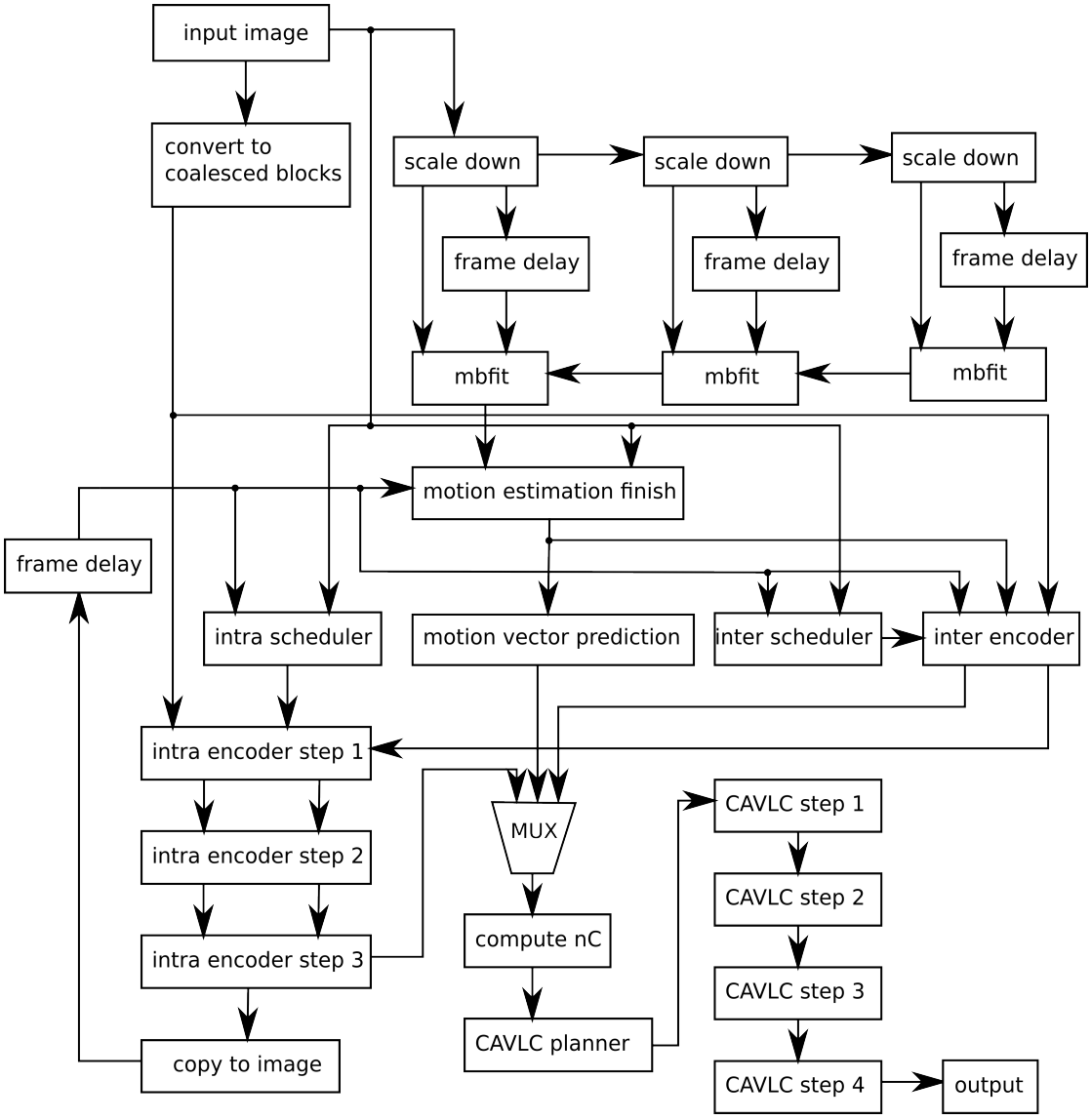


Figure 2.10: Data-flow diagram of the GPU implementation of the H.264 video encoder

DOI:10.15774/PPKE.ITK.2014.009

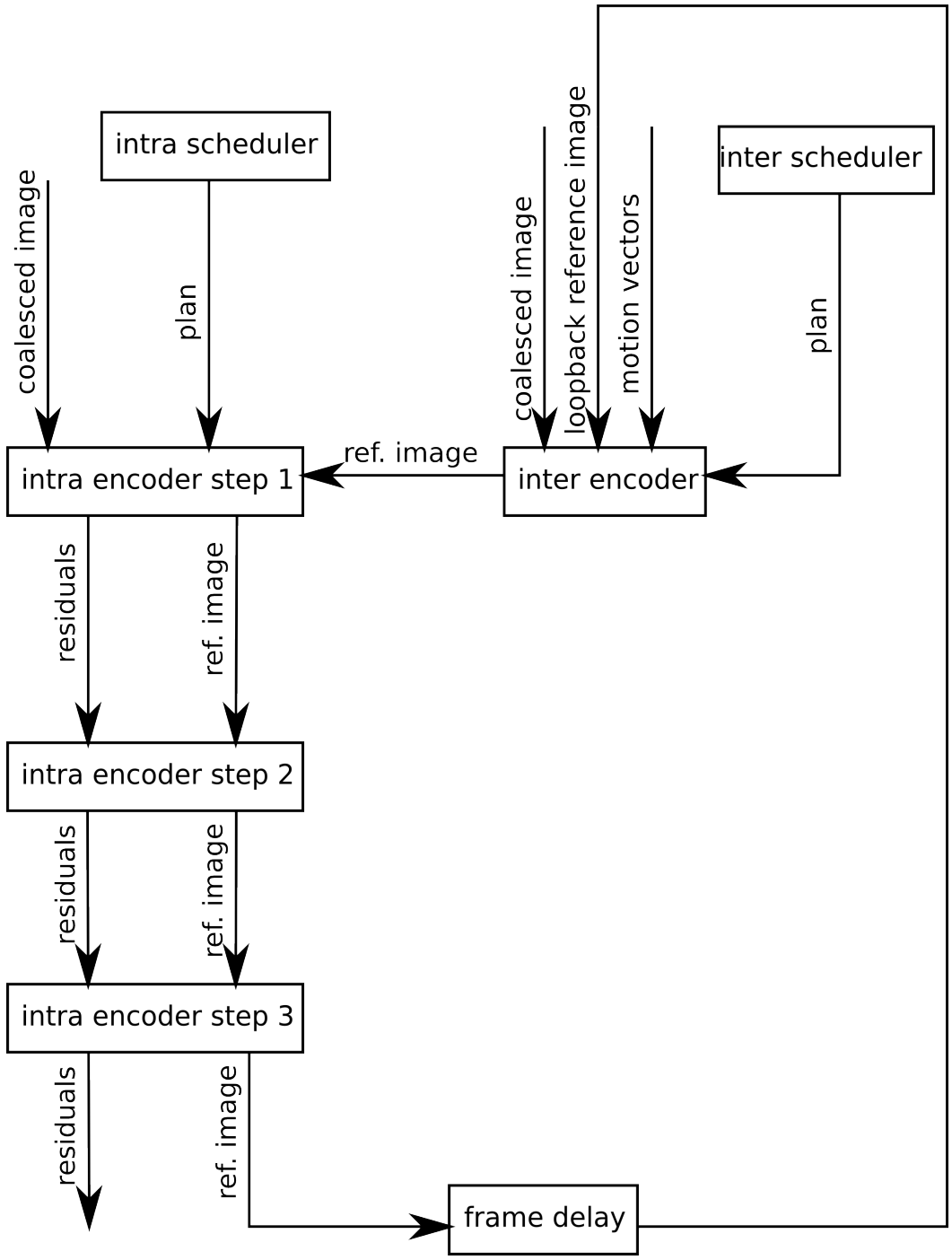


Figure 2.11: Data-flow diagram of the lossy encoding part of the GPU implementation

DOI:10.15774/PPKE.ITK.2014.009

$$f_1(x,y) := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.14}$$

$$f_2(x,y) := \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.15}$$

The top and left dependencies are described by Equation 2.14 and Equation 2.15 respectively. If we apply static parallelization to this problem we get the following transformations:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} := \begin{bmatrix} a \\ b \\ 1 \end{bmatrix} \tag{2.16}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ 1 \end{bmatrix} := \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.17}$$

We can substitute the transformation in Equations 2.16, 2.17 into Equations 2.13, 2.14, 2.15, so we get:

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 0 & N-1 \\ 1 & -1 & M-1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ 1 \end{bmatrix} \geq \bar{0} \tag{2.18}$$

$$f_1(a,b) := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ 1 \end{bmatrix} \tag{2.19}$$

$$f_2(a,b) := \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ 1 \end{bmatrix} \tag{2.20}$$

We can choose to map the a axis to the number of threads and the b axis to the time. This way (a_1, b_1) and (a_2, b_2) polyhedral points are allowed to be executed parallel if $b_1 = b_2$. This is possible because in this transformed space both f_1 and f_2 dependency functions map to an earlier point on the time axis b, $f_1(a,b) := (a, b-1)$, $f_1(a,b) := (a-1, b-1)$.

DOI:10.15774/PPKE.ITK.2014.009

2.7.2 Dynamic polyhedra

This case can be optimized further by my dynamic polyhedral model. The filter functions to the inter and intra coding are a simple flag stored in memory for every macroblock (point of the polyhedra in out case). This flag tells us if we compute the macroblock in inter, in intra or skip it completely.

Because of the nature of inter macroblocks, we do not have dependencies betweem them. This means that when an intra macroblock depends only on inter macroblocks we can compute that intra independently from all other intra. Consequently the mix of inter and intra macroblocks can be computed significantly more efficiently than a frame full of intra macroblocks.

According to my dynamic method, we have to define a scheduler algorithm for inter and intra computation. The dependency between the two types of macroblocks can be resolved by running the inter calculation first. The scheduler for the inter calculation is a very general algorithm, which simply evaluates the flag which indicates the type of the macroblock, and stores all the inter macroblock coordinates inside an array. For high efficiency the parallel computation of each index of the index of the macroblock, can be done by running a parallel prefix sum on the F_{filter} function, where the $true, false$ evaluates to $1, 0$ respectively. This way, in every thread where the F_{filter} function evaluates to $true$, we will have the linear index $(+1)$ where we can store the point of polyhedra (macroblock index), which we want to process.

This kind of scheduling for inter macroblocks improves efficiency because in SIMD GPU architectures group of threads are running in lock-step. This implicates that the when F_{filter} function evaluates to $false$, the thread must wait for other threads in the same group to finish processing, before it can continue. Consequently the F_{filter} function cannot completely realize its speed enhancing function. In order to minimize the time when threads wait, we run the scheduler, which only does a lightweight computation (prefix sum, or atomic sum) to compute the indexes which can ensure that the actual computation can run at full throughput, efficiently utilizing the hardware.

In the case of the intra processing we will have F_{filter} functions for the dependencies too. This means that at first glance the dependencies have to be

DOI:10.15774/PPKE.ITK.2014.009

scanned exhaustively during run-time. Fortunately we can take the polyhedral transformation which was originally meant for the intra processing, and use it for the intra scheduler. We can trivially group the intra macroblocks which pass the F_{filter} function into parallel groups by using the transformed coordinates in Equation 2.18, however this is itself would be a small improvement over the static optimization. The more advanced algorithm can inspect only nearby groups, and merge them. This way we can get the speedup when the intra blocks are sparse inside the P-frame, and we do not need a full dependency search. This is a trade-off between how much we scan dependencies (speed of the scheduler) and the how much parallelism we can achieve in the intra processing. The polyhedral transformation introduces the case where we do not need to scan at all compared to the full scan, where we check all possible dependencies repeatedly.

In case of the I-frames where all macroblocks are intra, the static polyhedral approach cannot be improved further by using dynamic polyhedrons. However in this case I reordered the intra computation in order to minimize the run-time of the parts which are affected by the dependencies.

The non GPU adapted version of the intra encoder is depicted on Figure 2.12. The reference feedback, which causes the dependencies, encompasses all the blocks, so the they cannot be factored out of the dependency. This computation is less efficiently parallelizable due to the dependencies, so moving out blocks from this computation can improve the overall speed.

The dependencies in the intra computation are irreducible in the sense that we cannot easily reduce them to trivial or associative parts, like I have mentioned in Section 2.3.2. I have solved the problem by restructuring and changing the computation, this improved version can be seen on Fgiure 2.13.

Originally the feedback loop, which generates the dependency, exists because we need to use the same reference image which will be generated at the decoder, otherwise the error would accumulate catastrophically. I have moved the DC prediction and the lossy compression (frequency domain transformation, and quantization) outside the feedback loop, so I use the wrong reference image. In order to correct it, I created a new feedback loop, which computes the corrections and creates the actual reference image, and the final results. This is possible because I use DC prediction which mathematically permits the complete decoupling

DOI:10.15774/PPKE.ITK.2014.009

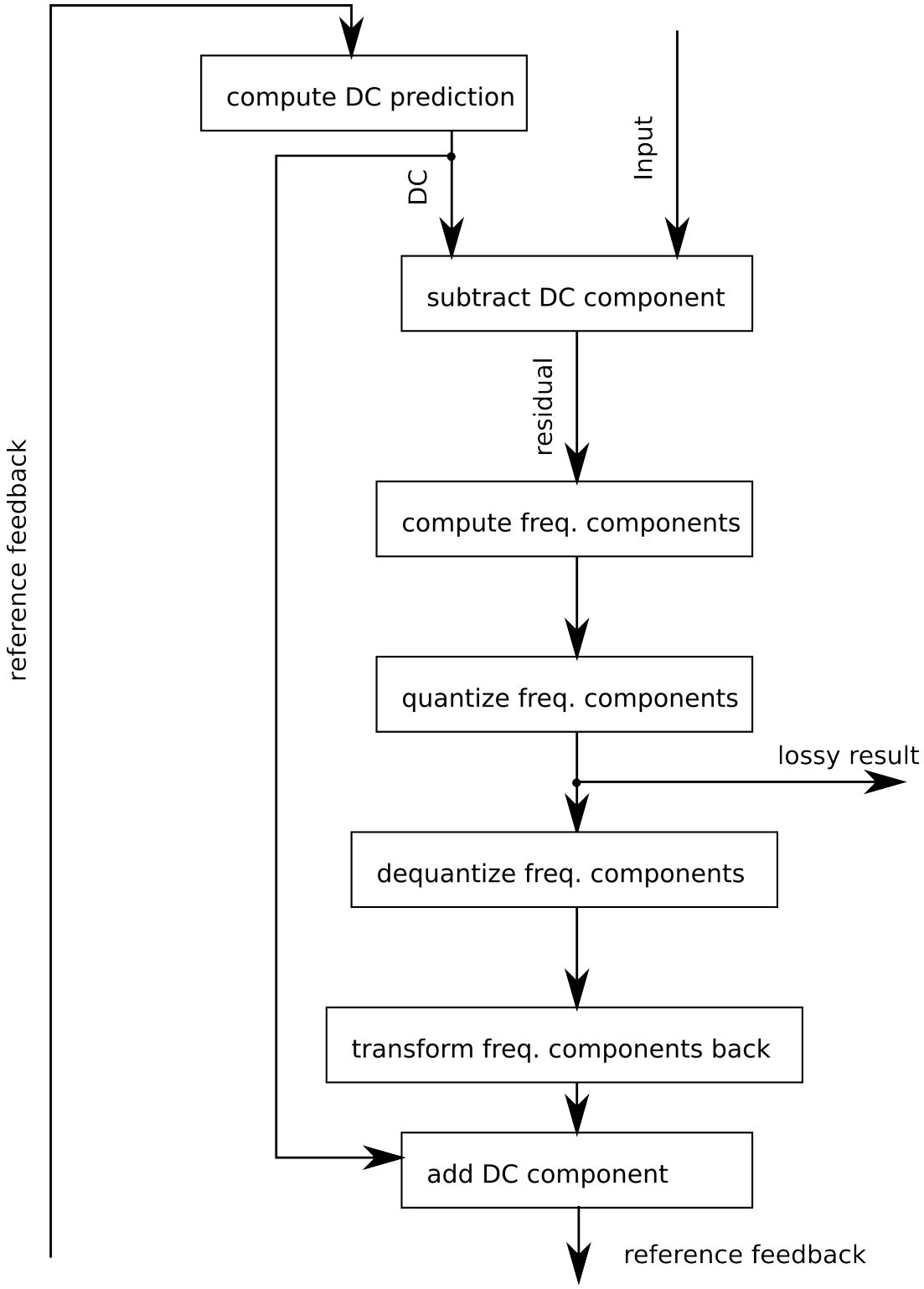


Figure 2.12: Data-flow diagram of the non GPU adapted version of the intra encoder. The reference feedback, which causes the dependencies, loops all the blocks, so the they cannot be factored out of the dependency.

DOI:10.15774/PPKE.ITK.2014.009

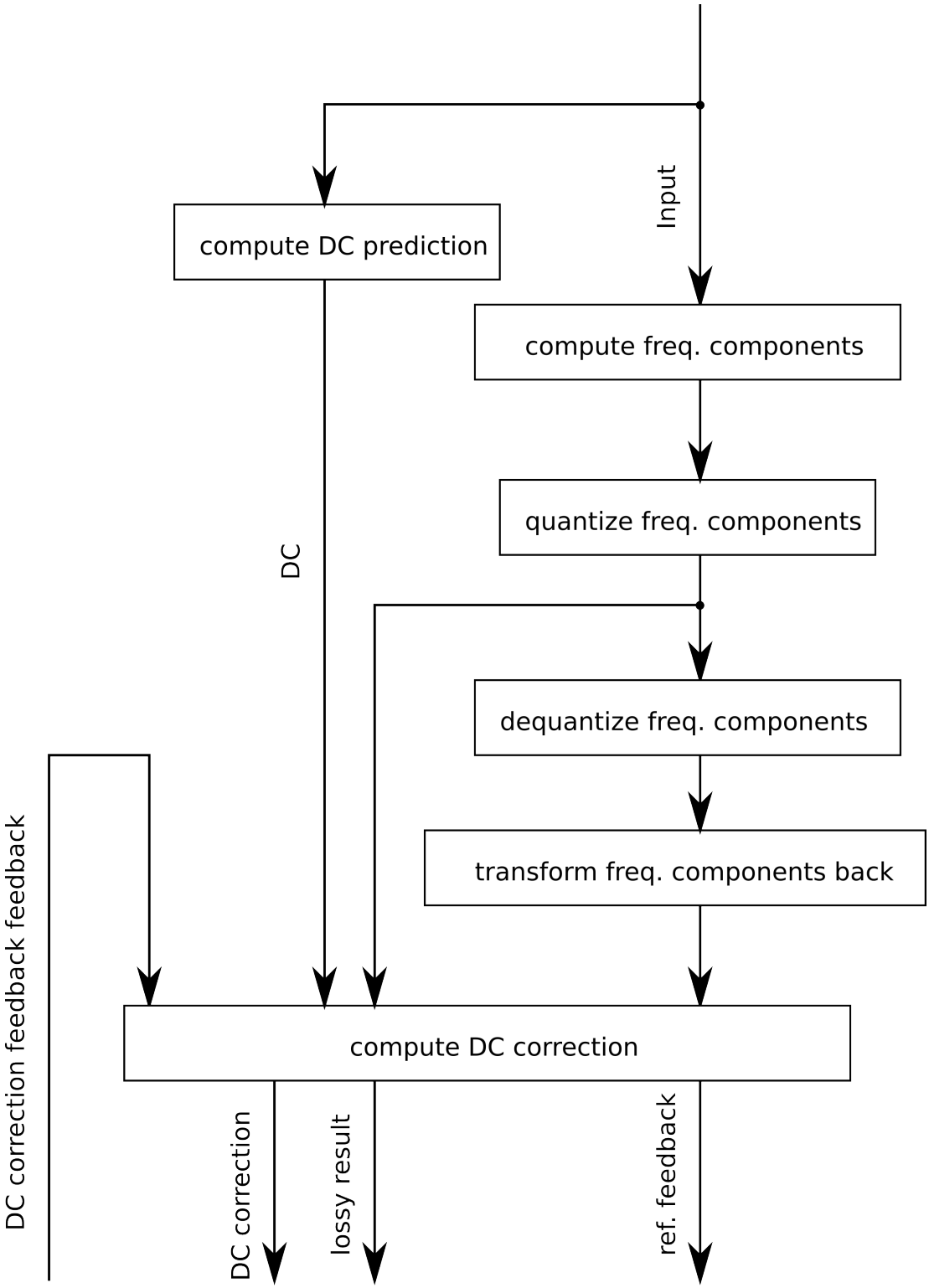


Figure 2.13: Data-flow diagram of the GPU adapted version of the intra encoder. The new feedback loop uses DC correction instead of the reference image inside the intra computation, this way most of the computation is free of dependencies.

DOI:10.15774/PPKE.ITK.2014.009

of the DC component in the frequency domain transformations and quantization, however the frequency domain transformation used by the H.264 standard is an inaccurate Discrete Cosine Transform, which creates a slight coupling between the DC and AC components. Consequently the correction step is needed which computes the correction based on the approximation of the DC-AC coupling.

2.7.3 Benchmarks

25fps	X264 on Intel i7 960	My implementation on GTX 580
Input streams 800x600	12	22
Output streams 640x480(QP=26) 640x480(QP=28) 320x240(QP=26) 160x120(QP=26)	48	88

Figure 2.14: The results of the benchmark of my GPU implementation built into a live video transcoding system compared to the CPU implementation. The inputs stream were decompressed, rescaled and re-encoded into different resolutions and image qualities.

25fps	X264 on Intel i7 960	My implementation on GTX 580
Input streams 960x720	10	18
Output streams 640x480(QP=26) 640x480(QP=28) 320x240(QP=26) 160x120(QP=26)	40	72

Figure 2.15: The results of the benchmark of my GPU implementation built into a live video transcoding system compared to the CPU implementation. The inputs stream were decompressed, rescaled and re-encoded into different resolutions and image qualities.

DOI:10.15774/PPKE.ITK.2014.009

My GPU H.264 implementation was built into a live video transcoding system, which made practical benchmarks possible. Intel i7 960 and NVIDIA GTX 580 was used as the testing hardwares whose release date is only one year apart, so the benchmarks are comparable. There are measurements depicted in Figure 2.14 and in Figure 2.15. In the tests, higher resolution stream were received, decoded, scaled and re-encoded in multiply resolutions and qualities (QP). The quality of the encoded streams were measured by human observer, and it was found to be corresponding to the configured quality.

2.8 Conclusion

Polyhedron optimizations themselves can not solve the huge problem of parallelization, but based on my experiments it seems to be a promising start. The basic idea is to reduce these optimization problems into topological transformations. I went on this road, and reduced other parts of the problem, too. Along the way I determined possible pitfalls and bottlenecks, which need to be worked around. The most important ones are data-flow dependencies, so investigating more data-flow dependence operator primitives provides more handy tools to be able to perform polyhedral optimizations, therefore expanding the scope of algorithmic problems we can handle inside this framework.

It is an important lesson learned that problems should be treated in their simplest forms, optimized forms are usually difficult or impossible to handle. There are quite many hardware specific heuristics, and sometimes not even the manufacturer knows them completely, so benchmarking everything is necessary. Fortunately benchmarking access patterns and thread scheduling is quite easy in this framework.

We are seeing exponential growth in core number (according to Moore's Law), which implies that very soon only parallelizable algorithms will be important as optimization targets. I strongly suspect that every practical algorithm is parallelizable, but actually implementing these will be even more important in the near future.

The given model is designed for many core programming and the theoretical aspects were derived from practical experience on GPUs and FPGAs. The model

DOI:10.15774/PPKE.ITK.2014.009

can be easily extendable to FPGAs, Cell BE and CPU clusters. However, the importance of different aspects of this formalism strongly vary depending on architectures. For example, local memory is the most important part of Cell BE architecture, but less important for GPUs, and missing for CPUs.

DOI:10.15774/PPKE.ITK.2014.009

Chapter 3

RACER data stream based array processor

In this Chapter, first I give an introduction of the current trends in computational architectures. I summarize the main issues of the most popular approaches by highlighting the disadvantages of these architectures. The introduction includes the main targeted problems to be solved as well. In Section 3.2 I introduce my invention, the RACER architecture, which is a novel massively parallel heterogeneous data stream architecture. I describe the programming principle of the proposed architecture. To maximize computational and data transfer speed per chip area I designed an active memory device, which is presented in detail. Next, the fine structure of its processing array and the applied pipeline processing are described. RACER programming and its emulated functionality are presented through simulations.

3.1 Introduction

The trend of the evolution of the processors is obvious, the number of cores per processor will increase exponentially in the next five-ten years. However, the difference between each of the following architectures is not only the number of processors, but the changes in their architectures. This evolution leads not only to a higher number of processing units, but to the more efficient and optimized operation and also to an increased computational power per area.

DOI:10.15774/PPKE.ITK.2014.009

3.1.1 Compromises of common architectures

If we examine the current parallel architectures, we find that their objective is the maximization of the general-purpose computing power per unit area. This is achieved through technological improvements and trade-offs. These compromises of their memory and computing structures at the most common architectures are the following:

CPU (Central Processing Unit) [29]: CPUs usually have only one type of explicitly addressable memory and general caching is applied. Caching generally means that data are stored by a special method to be available more quickly than in direct memory access. The so-called cache memory contains selected data elements and it is configured to provide the data elements to the CPU as quickly as possible. The computing architecture of CPU is characterized by the "out-of-order execution" method, which is required for the run-time rearrangement of the order of the instructions. A significant part of the chip area of CPU is used by the cache memory, the number of transistors of the arithmetic logic circuits is less than the number of transistors of other logic parts. At the same time, relatively only a few parallel threads can be run on one CPU using its small number of arithmetic units, but these units are very well utilized. In case of a multi-core CPU each core can have separate cache memory.

The difficulties of memory reading and writing of CPUs are hidden by using traditional hierarchical cache. This solution, especially if we have more processor units, increases untenable the ratio between the chip area of the cache memory and the chip area of pure computing. It is a good balance for the less computationally intensive tasks, but quite wasteful in the case of scientific or graphical computations.

DSP (Digital Signal Processor) [30, 31]: These devices are very similar to CPUs, the difference is mainly between their parameters. DSPs are designed for running signal processing algorithms efficiently (FFT, matrix-vector operations) with low power consumption and competitive price. The chip area (ie. the cost of manufacturing) is much smaller than the CPUs', because of optimization reasons, DSPs have less cache memory. Therefore the system memory access patterns of DSPs are more restricted if we want to exploit the available bandwidth.

DOI:10.15774/PPKE.ITK.2014.009

GPU (Graphics Processing Unit) [32, 33, 34]: The widely used GPU has many same processor units which can access the required data through a hierarchical memory system. The processing units of the GPU are organized into groups and units can locally communicate with each other within a group. The GPU is not characterized by locality, its memory system is organized in hierarchic tree structure. For the completely local organization of the same SIMD (single instruction, multiply data) type processor units, more global wires have to be connected to each processing unit. Despite local connections between the processor units, the global wires limit the maximum size. On the transistor level, the dimensions are so small and a chip is proportionately so large that the communication between the two farthest processor takes too much time.

The computing architecture of GPU has many very simple core processing units, that are usually SIMD type vector processors. Most of the transistors of GPU are parts of a processing unit, which usually consists of a combination of an ALU (Arithmetic Logic Unit) and an FPU (Floating Point Unit). GPU has a very simple pipeline management with deep pipelining. Pipelines are chains of data processing stages. Deep pipelining means that the chain of sequential steps of the processing operations is long. Consequently, compared to CPU, GPU contains many processing cores, but they are typically less utilized.

A further difficulty of GPU-based devices is the delivery of data to the processing units. The maximum utilization is measured by the number of computational operations per data elements in order to fully exploit the capacity of the architecture. In case of ideal memory utilization, this number means usually 25-30 operations. For GPUs this number is low because the processing is overlapped with data transfer, but generally this number is still too high to utilize optimally their performance since the algorithms typically do not perform 25-30 operations on the same data element before loading it back into the memory. If the GPU device does not perform the optimally required operations on a data element then the processing units are "starving" by reason of slow memory access and they are inactive for a significant part of time.

The hierarchic memory structure of GPU usually contains addressable local memory for each processing unit or unit group. GPU is connected to global memory too, which applies caching, but less than CPU has. Generally, GPU is

DOI:10.15774/PPKE.ITK.2014.009

also capable of using two-dimensional caching that is treated as a two-dimensional array of memory blocks for image processing.

The vector based SIMD architecture of GPUs has a very strong constraint to the implementation of threads. In a workgroup every thread has to do the same operation on different data and read the data from the adjacent memory. But working with this architecture the programmer has to solve the efficient use of memory, because contrary to the CPU, this system does not hide the details and does not solve the related problems automatically.

Cell BE (Cell Broadband Engine) [35]: this is a hybrid architecture, which includes a classic PowerPC CPU processor connected to synergistic processing units. The synergistic processing units are very simplified processing units, which have relatively large local memory on chip. The programmers are responsible to solve every tiny technical problem, from the appropriate feeding of the pipeline to organize the internal logic of the memory operations. This device has only indirect memory access via the local memory.

FPGA (Field Programmable Gate Array) [36, 37]: On this architecture, arbitrary logic circuit can be implemented within certain broad limits. Usually the implemented circuit is relatively efficient, since the desired circuit is realized physically on the FPGA by connecting on-chip switching circuit components. Consequently the logic circuits of the FPGA can be adapted directly to the given task, therefore this architecture can exploit most efficiently the available processing units. However, the cost of this enormous flexibility is the low density of the processing units on the chip surface, since the switching circuits and universal wiring need large chip area.

FPGA is usually connected to on-board memory and next to its arithmetic units there are local memory modules too. Usually on an FPGA there is no or very simple caching, but if cache memory is implemented, it requires too much chip surface. On an FPGA, usually there are more than a thousand general processing units (e.g. arithmetic units), and these units are complemented by hundreds of thousands of more simple logic processing units (CLB). The reprogramming of the FPGA, which is a hardware implementation of an appropriate computing architecture is slow compared to the computation power of FPGAs. During the redefinition of the computing architecture the auto-routing process has to be

DOI:10.15774/PPKE.ITK.2014.009

carried out, which means essentially the hardware design. In the course of auto-routing procedure the compiler converts the designed FPGA program to physical FPGA logic circuit. This compilation is slow (off-line), because there are many different potentially working circuits but we need to find the optimal (or quasi-optimal) solution. The reprogramming of a standard sized FPGA can take up to half an hour using a currently available PC, because it includes a high dimensional combinatorial optimization problem, which is NP-complete.

FPOA (Field Programmable Object Array) [38]: The memory architecture of FPOA is essentially identical to the FPGA's. Compared to FPGA, FPOA contains higher level processing units such as ALUs or FPUs, and a smaller number of freely programmable universal logic units.

Systolic Array [39]: this classical topological array processor architecture contains effectively only execution (computing) units, adder and multiplier circuits, which usually solve some linear algebra operations in parallel. Its applicability is very limited, because its topology is specific to the executed algorithm. This architecture does not contain either memory architecture, or program control structure. These units should be provided by another system. The flexibility is sacrificed for efficiency, since the computing units are utilized almost fully during operation and the surface of the silicon chip contains effectively only computing units.

The systolic array is a topological array, which receives input, and gives output at the edge of the array. According to the design of the systolic array, only a single algorithm can be executed at the same time, so there is no task level parallelism. The processing elements of the systolic array are ALUs or FPUs, which are connected to each other by one-way connections, thus a loop can not be defined. The stream of the data is predefined on the hardware by the one-way directed connections, therefore it can not be changed. It practically has no control-flow, but it can be programmed by the order of the input data and/or arithmetic instructions sent to the processing elements. This architecture is characterized by good computing performance per area, but only very specific, predefined tasks can be performed efficiently. The systolic array is not Turing-complete.

CNN (Cellular Nonlinear/Neural Networks) [40, 41]: this architecture is efficient at local image processing operations (low resolution image processing algo-

DOI:10.15774/PPKE.ITK.2014.009

rithms on gray-scale images) with extremely high speed and low power consumption. Every pixel is associated with a processing unit, the process is analog and there is only a very little analog memory. Accessing the global memory compared to the internal speed is very slow and also needs the analog to digital conversion of the pixels. It is optimized for 2D topological computations with low memory.

Dataflow architectures [42]: Based on the implementations, dataflow architectures can be classified as static and dynamic architectures. MIT Dataflow Architecture [45], DDM1 [44], LAU [50] and HDFM [51] were designed using the static model. Manchester Dataflow Machine [46], the MIT Tagged-Token [43], DDDP [48] and PIM-D [47] were designed using the dynamic model. To overcome the main disadvantage of the dynamic model, which is the overhead of matching tokens, expensive associate memory implementations are included in the architecture (e.g. Monsoon architecture [49]). Although the dataflow architecture is very promising because of its execution paradigm, but in practice it can not exploit efficient parallelism based on its inherent limitations. Another problem of this approach is that it is difficult to program because of its functional languages.

3.1.2 The main targeted problems

In case of multiprocessor computer architectures, moving data efficiently between memory and processing cores poses a significant programming challenge. We need to transfer the data - that we want to process to the cores - in the order which is efficient to process, however the efficient pattern of memory access usually conflicts with the efficient processing pattern. The more cores we have on a single chip, the more sensitive we are to the memory access patterns, because more cores have to share the same global memory bandwidth and less space is allocated to cache memory, which can partially mitigate this problem.

Further disadvantage of the most of the well known multiprocessor array based computer architectures is that the clock signal is distributed to each processing elements through specially dedicated global wiring. This global wiring is optimized for synchronized operation, because these architectures are completely synchronized by themselves. This wiring is difficult to implement efficiently on

DOI:10.15774/PPKE.ITK.2014.009

semiconductors, because the wavelength of the clock frequency is comparable to the length of the wire and delivering the clock signal everywhere on the chip in the exact same time is a highly complex task. On the other hand, not just the clock signal, but the data has to arrive at the exact right time at the processing units as well. Most often the faster data pipelines need to be slowed down, so the slowest pipeline is the speed limiting factor. This problem can be solved by highly complex execution and pipeline control, or alternatively by much simpler locally controlled pipelines. However, locally controlled pipelines are fundamentally incompatible with the classically programmed architectures.

Considering these problems, my aim was to design a computational architecture (RACER architecture), which is not limited by the disadvantages of the previous parallel architectures, is Turing complete and fully general algorithms can be implemented efficiently on it, moreover its performance per area is maximized as much as possible.

The purpose of this work is to create a multiprocessor array based computer architecture, which is able to accomplish the ordering and re-ordering of data elements of the data streams during processing more efficiently than the well-known architectures.

Another purpose of this work is to design a multiprocessor array based computer architecture, wherein the computer architecture does not require globally wiring the clock signal to each processing unit. The desired architecture should have greater fault tolerance against the synchronization delays of data streams and it should provide synchronization of data streams more effectively than other well-known architectures.

The proposed RACER computer architecture ensures the ordering of the data elements of the data streams with greater efficiency than the well-known systems, therefore the architecture contains an active data memory which, besides storing the data, is able to reorder the data elements of the data stream. This ordering unit is integrated in the active data storage. Moreover, without an external unit, the ordering unit is able to re-order the data streams every time when the stream passes through a memory unit.

The RACER architecture has a simpler structure and smaller parametric search space such as FPGA. In this architecture, the connections of process-

DOI:10.15774/PPKE.ITK.2014.009

ing units are simpler and higher level processing units are applied. The RACER architecture tolerates the sub-optimal paths better, therefore data stream paths can be created much more simply and quickly.

3.2 RACER architecture

The main functional blocks of the RACER computer architecture can be seen in Figure 3.1. The architecture includes a *central processing unit (RCPU)*, *memory units (MU)*, *periphery units (PU)*, *control unit (RCU)* and *instruction stream router unit (ISRU)*. The units of the architecture are connected to each other through the RCPU. The RCPU processes the *program stream* which consists of *instruction stream* and *data stream* divided into *data elements*, see Figure 3.2. The ISRU, which defines the instruction stream, can be integrated in RCPU, but in any case it is still a separated unit within the RACER architecture. The RCPU contains an array of blocks of *processing elements* and each block is surrounded by *data routing elements*. *Lateral processing elements* are connected to the neighboring data transfer elements. Lateral processing elements are those which are physically located on the edge of the block.

The computer program of the RACER architecture contains an instruction stream which defines the *program stream path* and the hardware parameter details of the implementation of the program. The route of the program stream through the parts of the architecture especially through the RCPU and memory devices defines the stream path, which is essentially the topology of the computing hardware structure of the problem. The RACER architecture reconfigures the hardware structure before each program execution, namely it plans the routes of the program streams which may vary per each execution.

Every MU of the RACER architecture contains a *sorting processing unit* (SPU) to be able to order or reorder the data elements of the program stream. The design of such memories, namely an SPU is integrated in the MUs, allows the extremely rapid reordering of the data elements. While the unordered data stream is being stored in the memory, the properly ordered data stream can be read out. This sorting process can be solved very efficiently by integrating the data sorting function into the MUs. For the investigated algorithms the number

DOI:10.15774/PPKE.ITK.2014.009

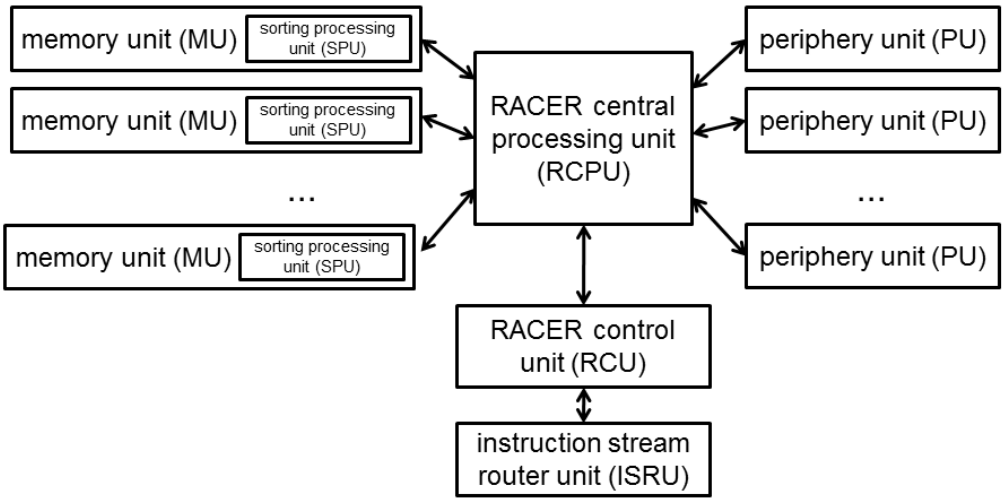


Figure 3.1: The main functional blocks of the RACER computer architecture can be seen. The architecture includes a central processing unit (RCPU), memory units (MU), periphery units (PU), control unit (RCU) and instruction stream router unit (ISRU). The units of the architecture are connected to each other through the RCPU.

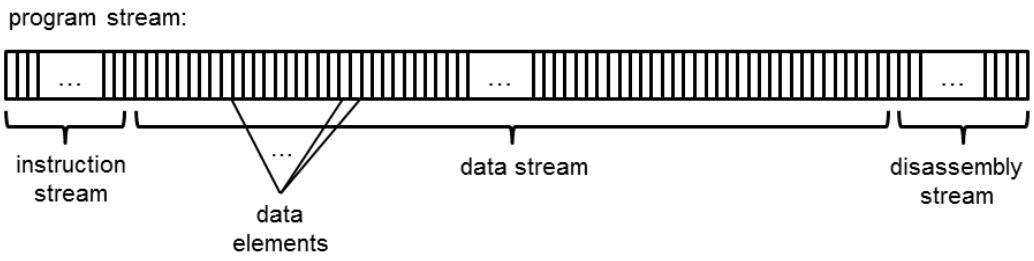


Figure 3.2: The program stream which consists of the instruction stream and the data stream divided into data elements, and the disassembly stream.

DOI:10.15774/PPKE.ITK.2014.009

of MUs connected to RCPU was typically between four and eight, but more MUs can be connected to the RCPU.

The SPU of RACER architecture prepares the data streams and controls the appropriate data feeding of the RCPU. The data stream starts from a PU or MU and arrives at a PU or MU too. It can flow through even several times in different MUs while the calculations are carried out by the processing elements. Because PU and MU provide the same interface, they both can be used as a source or a destination of a program stream. The instruction stream of the program stream contains the tasks of MU too, which is the sorting of the data elements of data streams into different patterns.

In the common stream processing based multiprocessor computer architectures, the program streams, which are leaving the central processing unit, are reordered by the cache and the processor itself. In the literature this reordering is usually called "coalescing". The reordering is necessary to obtain the appropriate order of data elements from the computing point of view, moreover to read and write the memory by a continuous data transfer. In the RACER architecture the task of coalescing is done by the MU. Thus, unlike the well-known architectures, where the contents of the memory are ordered close to the processor, the MU contains a special purpose processor for this task. Thus, the sorting of the data can be performed using a so-called on-chip processor much faster than in the well-known devices. The usage of these memories in the RACER architecture is especially beneficial for performing local data reordering.

The better the advantages of SPU can be exploited, the higher the bandwidth utilization of the memory is. The data bus bandwidth of a computer architecture determines the maximum transferable data per unit time. In order to provide adequate speed of data processing, the data bus of the RACER computer architecture is typically much wider than in conventional architectures.

3.2.1 Programming principle

As mentioned, the MU can do more than just store the data, because it has an integrated sorting processor. Therefore, it is naturally optimized for comparing and moving instead of computing, however it should be possible to run simple

DOI:10.15774/PPKE.ITK.2014.009

algorithms on it like maximum or minimum search, or random shuffling. This opens a new possibility: using the memory as an active part of the algorithm as seen in Figure 3.4. The link between the memory and RCPU should be able to transfer data both ways at the same time for optimal performance, so the memory can be used as a part of the data stream pipeline. This way the bottleneck introduced by the strictly coalesced memory reading can be avoided, without using caching.

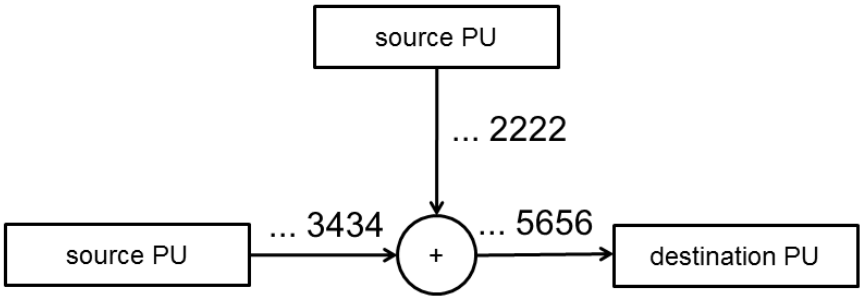


Figure 3.3: A simple data stream processing example is shown, where the addition of two data streams from different sources is realized element-by-element by an adder operator realized in the RCPU. The width of data streams can be 4×32 bits, but this parameter can vary depending on the hardware implementation.

The head of the program stream contains the instruction stream and the central part of the program stream contains the data stream. The data stream contains the ordered data elements, which are processed by the RACER architecture. The head of the program stream contains the instructions for the memory and the arithmetic and control operations for the processing elements. The data elements of the central part of the program stream are arranged according to the computing structure, essentially arranged in a topology as the processing is carried out by the processing elements on the data elements of the stream. The tail part of the program stream includes the *disassembly stream* which ends with a *disassembly message*. The *disassembly stream* effectively terminates the program by resetting all the hardware elements to their respective default state. In every case in the program stream of RACER architecture, a given instruction

DOI:10.15774/PPKE.ITK.2014.009

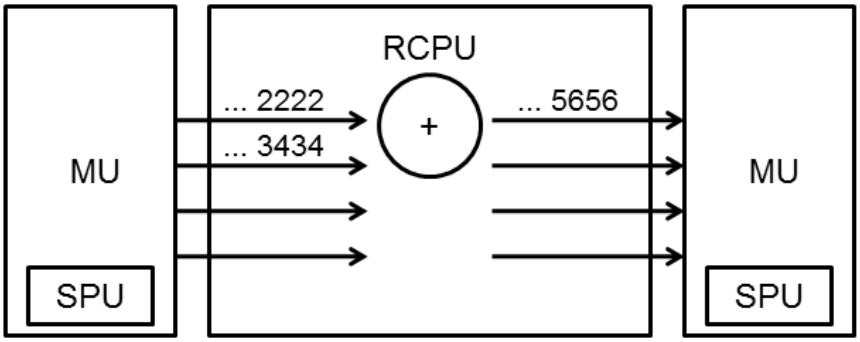


Figure 3.4: The computation of Figure 3.3 realized on RACER architecture. The addition of the first and second data channel of the input data stream is computed, and the result is placed in the first channel of the output data stream. Of course the RCPU can perform more complex operations, too

precedes the data element, which it is executed on. It is possible that the instruction stream and data stream is not separated in the program stream, i.e., the elements of the instruction stream and data stream can alternate.

In Figure 3.3 a simple example is shown for processing data streams, where the addition of two data streams from different sources is realized element-by-element by an adder operator realized in the RCPU. The width of data streams can be 4×32 bits, but this parameter can vary depending on the hardware implementation. In Figure 3.4 the addition of the first and second data channel of the *input data stream* can be seen, and the result is placed in the first channel of the *output data stream*. Arriving from the MU and passing through the RCPU the data stream is at least partially processed, as it can be seen in Figure 3.4. Of course the RCPU can perform more complex operations, too.

3.2.2 Active memory

The role of the RACER MU is twofold: data storage and data sorting. Figure 3.5 shows a simple example of the reordering of the data in the MU. The input program stream is stored in a suitable storage and then during the readout process, the SPU creates the rearranged output data stream. In order to minimize the processing time of the MU, the MU and its SPU is optimized for data move and

DOI:10.15774/PPKE.ITK.2014.009

comparison operations. The RACER memory is capable of performing only simple algorithms, such as searching maximum or minimum elements or generation of index for data elements.

In Figure 3.5 the sorting of an input data stream is depicted. In this sorting process one of the channels of the data stream contains the indexes. These indexes/tags are related to the data elements of the data stream, which the SPU arranges by their indexes and it creates the corresponding output data stream. In a more advanced example, the MU sorts the data elements mixed by a loop, because the loops and other control structures have the tendency to shuffle the order of the data elements. However, we also may need to reorder the data elements if the elements are not explicitly mixed but other different parts of the algorithm require different orderings. All nontrivial programs would contain control structures for example loops. This underlines the importance of the sorting capability of the MU.

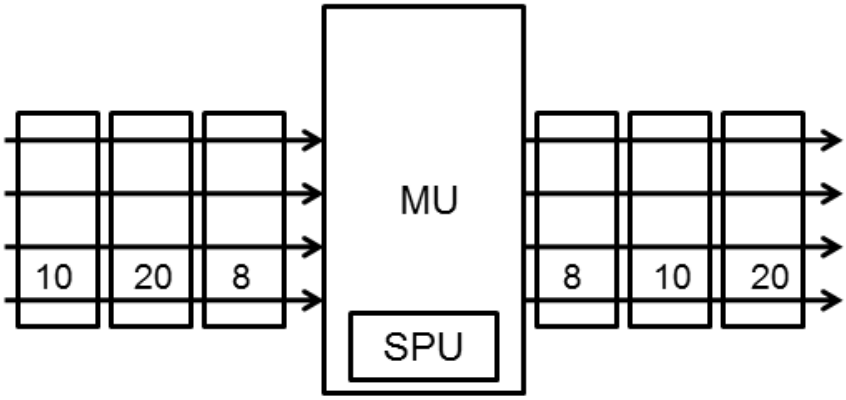


Figure 3.5: Illustration of the RACER memory, it sorts the information by tags. In this sorting process one of the channels of the data stream contains the indexes. These indexes are related to the data elements of the data stream, which the SPU arranges by their indexes and it creates the corresponding output data stream.

In Figure 3.6 the block diagram of the RACER MU can be seen. The RACER MU contains a *memory processing unit* (MPU) for comparing the data elements. This MPU can be an ALU or an FPU, it should have at least minimally adequate computing capability for data sorting, and is controlled by the *memory*

DOI:10.15774/PPKE.ITK.2014.009

control unit (MCU). The MCU receives and executes the related parts of the instruction stream. The MU also includes a *data link controller unit* (DLCU), which is connected through the data bus to a port of RCPU, where the data transmission can be implemented by optical fiber or conventional wiring. The DLCU has a data buffer storage and frames the program streams and send them to the RCPU. It is important to note that the MU is not a homogeneous block of memory, and its structure is optimized for sorting and processing, by intermixing memory (DRAM) and compare blocks. Data sorting is based on comparisons, consequently the highest throughput operation inside the MU is data comparison. This is very important, because in order to achieve optimal efficiency the comparison and sorting should be fast enough so it can run parallel with the MU reading and writing operations. In the ideal case when only local sorting is needed on the data stream, this high throughput allows the full parallel operation of the MU, so the receiving of the stream (write), the sorting, and the sending(read) operations can all run parallel. This drastically lowers the delay, caused by the sorting operation.

3.2.3 Structure of the RACER central processing unit

In Figure 3.7 the details of the internal structure of the RCPU can be seen. This device contains the block of processing elements and the data routing elements. The processing elements are connected to the neighbors and are able to process operations on program streams based on the instruction stream. The role of the data routing elements enables the passage of the stream to the processing elements. Usually data routing elements connected to their neighbors can be placed at all four sides of a block of processing elements. More data routing elements are connected to a block of processing elements, more different paths can be used to transfer and enter the program stream into the *processing block*. The elements of the data stream pass through the data routing elements without processing the stream. In case of the processing of multiple or branching data streams in the same time, streams can cross each other through the routing elements. The data routing elements ensure that a data stream can enter at the right place of the block of processing elements, so the processing elements can be

DOI:10.15774/PPKE.ITK.2014.009

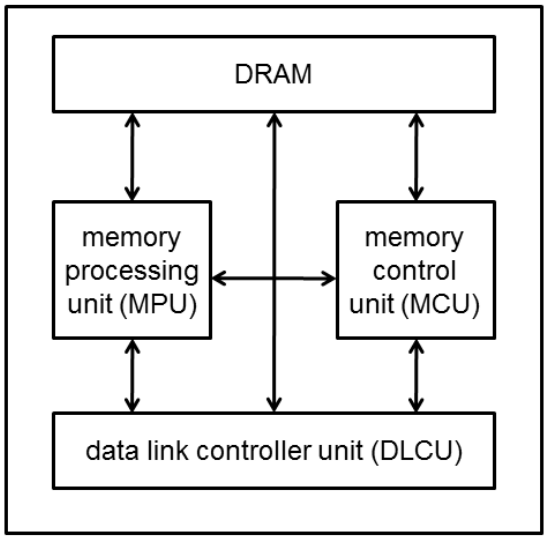


Figure 3.6: Structure of the RACER MU with the communication paths. The RACER MU contains a memory processing unit (MPU) for comparing the data elements. This MPU can be an ALU or an FPU, it should have at least minimally adequate computing capability for data sorting, and is controlled by the memory control unit (MCU). The MCU receives and executes the related parts of the instruction stream. The MU also includes a data link controller unit (DLCU), which is connected through the data bus to a port of RCPU, where the data transmission can be implemented.

DOI:10.15774/PPKE.ITK.2014.009

distributed optimally between the data streams whose processing is overlapped in time.

In Figure 3.7 there are multiple *processing blocks*, where the processing blocks are connected by data routing elements. The data stream can be processed while it passes through multiple processing blocks, before leaving the RCPU through the input/output ports. The data stream can leave to the RACER memory or other peripheries through the input/output *communication ports*, that is the RCPU communicates with the units of the architecture through the ports. The RCPU has a lot of possible communication ports, in case of a square arrangement of processing elements, the number of communication ports can be, as an example, $4 \times M$, where M is the number of processing elements at the edges.

The data routing elements are along and between the processing blocks that the data streams can enter processing blocks at the optimal location. The data streams can even cross each other in the data routing elements. The intersecting data streams do not interact with one another, they cross each other without exchanging data.

It is important to note that since using longer *data stream paths* are relatively non-resource and non-time consuming, delay can be tolerated, the RCPU scales well in space, and the computing area can be extended easily by placing modular processing blocks next to each other and connecting them with data routing elements.

A processing block generally contains $n \times m$ processing elements. In Figure 3.7, in the given example $n = m = 4$, but n and m can be chosen arbitrarily. The processing elements are arranged in square blocks including *general processing elements* and elements with enhanced functionality. These special elements are placed at the bottom and the right side of the block. In Figure 3.7 the processing elements are labeled by P and the *enhanced processing elements* are labeled by G, accordingly. The enhanced functionality processing elements have all functionality of general processing elements, in addition, they may have conditional data stream control, which is implemented by the conditional control of their multiplexers, namely they are particularly suitable for the implementation of certain types of branching. Some of these branches can be implemented by

DOI:10.15774/PPKE.ITK.2014.009

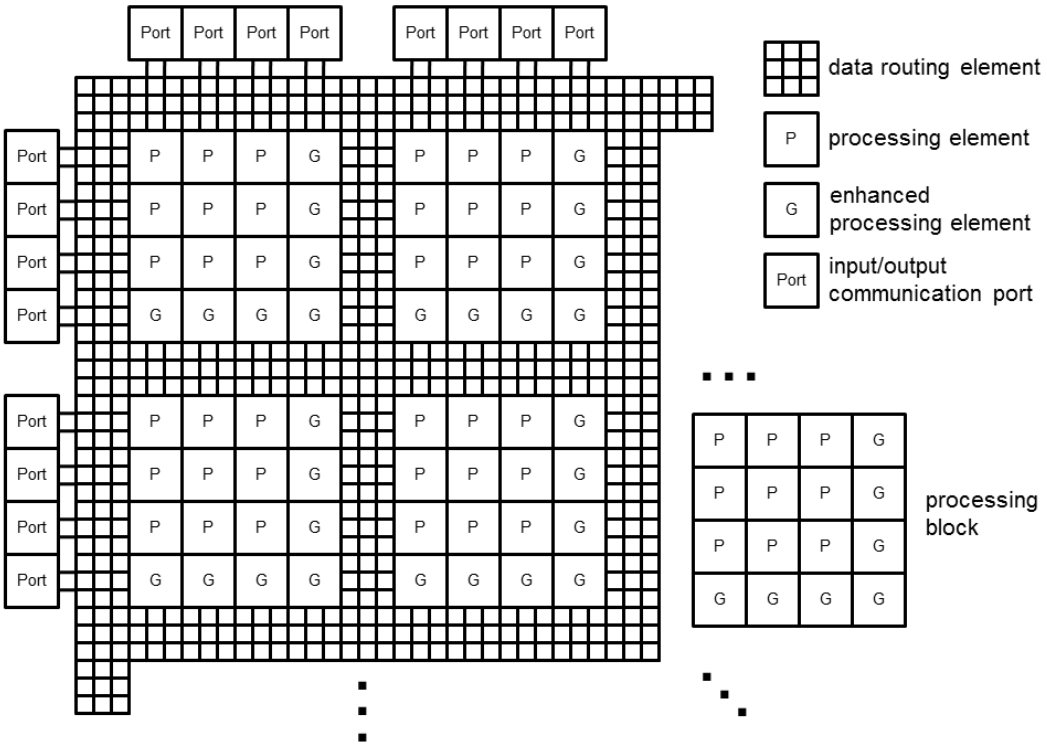


Figure 3.7: Internal structure of the RACER central processing unit. This device contains the block of processing elements and the data routing elements. The processing elements are connected to the neighbors and are able to process operations on program streams based on the instruction stream. The role of the data routing elements enables the passage of the stream to the processing elements. The processed data stream can leave to the RACER memory or other peripheries through the input/output communication ports.

DOI:10.15774/PPKE.ITK.2014.009

general processing elements too. The types of branching are described in detail in Section 3.3.

Each processing element contains *programming logic*, which is configured by the local memory. This programming logic controls the multiplexers, and also implements control flow operations. The simplest approach is to view the programming logic as LUTs (lookup tables). These LUTs tell the multiplexers how they should behave in each situation. Different situations arise depending on the state of the incoming and outgoing pipelines, and the control-flow operation which the processing element implements.

On the side of the processing blocks, each processing element has three data routing neighbors on one side and can be connected to only one data routing element.

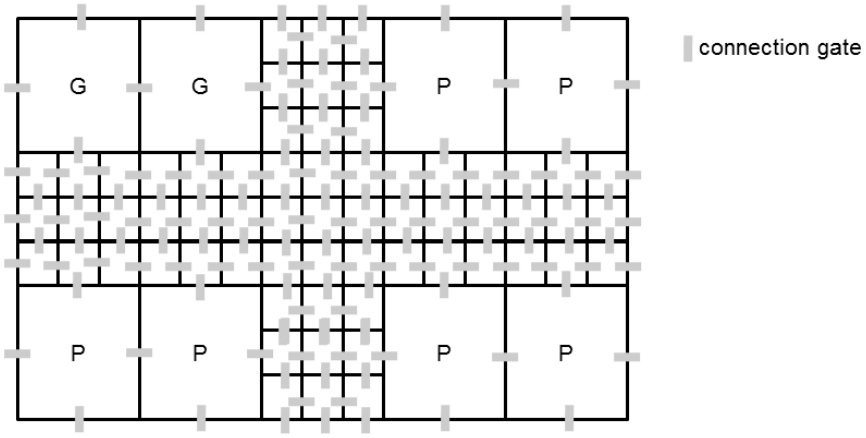


Figure 3.8: Internal connections of the RACER CPU, the structure is black, the connections are gray. These connection gates provide the passage of the data stream between processing elements and data routing elements, between data routing elements and between processing elements.

Figure 3.8 shows a part of Figure 3.7 where *connection gates* are also included. These connection gates provide the passage of the data stream between processing elements and data routing elements, between data routing elements and between processing elements.

DOI:10.15774/PPKE.ITK.2014.009

In Figure 3.7 three dots note that the structure of RCPU can be extended in the specified directions. In the indicated directions more blocks of processing elements can be arranged in the same structure. The data routing elements are connected to the ports on all sides of the RCPU, as it is shown in Figure 3.7 and also at the right side and the bottom.

3.2.4 Program stream

In Figure 3.9, as a simple example of RACER algorithms, the path of its program stream is depicted. According to the Figure, the data streams pass through the data routing elements and processing elements. The processing elements perform calculation operations on the data streams. The data streams merge into one data stream in the junction, which leaves the RCPU through the output port. In Figure 3.9, the data streams are displayed by their traversed routes.

In case of using program streams, the implemented program included in the instruction stream travels before the data stream. The program stream assigns the tasks to each processing element, the processing elements receive their instructions as the program stream passes through them, therefore global control of processing elements is not needed. Using such program streams, processing elements are locally controlled only. The *instruction stream router unit* (ISRU) defines the route of the data stream, which will configure the required architectural layout for a given algorithmic calculation. The ISRU is optimized for especially this task, so it can run to solve this optimization problem efficiently in real-time. This is very important, because this allows the RCPU to be a truly multitasking architecture.

In the RACER architecture, there are several options to concatenate the instruction stream with the data stream. The program stream may be received from an external periphery, in this case it does not contain an instruction stream yet. When the program stream reaches a place where the routes are not defined and not included in the stream, it will stop. So the ISRU provides the instruction stream, which includes the program, and then the data and instruction streams are concatenated into a program stream. The instruction stream is transferred

DOI:10.15774/PPKE.ITK.2014.009

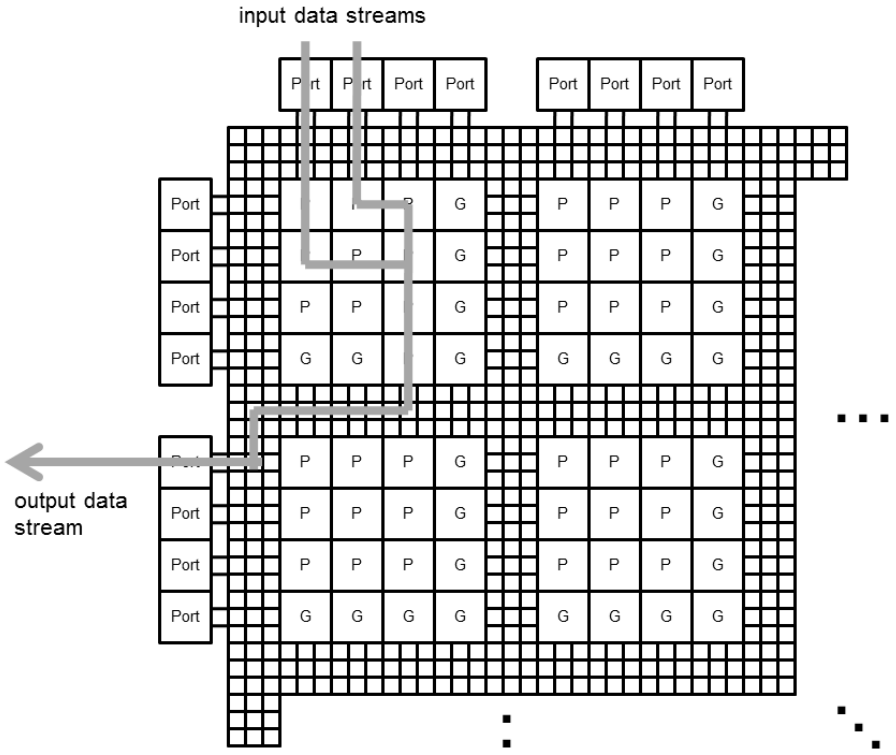


Figure 3.9: A simple algorithm routed on the RACER CPU. Grey depicts the used elements. The data streams pass through the data routing elements and processing elements. The processing elements perform calculation operations on the data streams. The data streams merge into one data stream in the junction, which leaves the RCPU through the output port. The data streams are displayed by their traversed routes.

DOI:10.15774/PPKE.ITK.2014.009

through the data routing elements to the concatenation junction. Because the instruction stream is capable of creating and breaking down its route dynamically, it uses the data routing elements only temporarily to reach the concatenation junction. This junction should be at a PU, or inside an MU. In the case of the MU, the actual concatenation of the streams is done by the MCU inside the MU.

The processing elements operate on the data of the program stream according to the program implemented in the instruction stream. The data elements of the data stream are provided in the requested order of the program. Thus, the processing elements do not have to wait for the data and contrary to known architecture global connections are not needed to implement the appropriate layout of this architectural design. The number of operations processed by the processing elements can be changed dynamically depending on the task to be performed. This means that based on the load of the processing elements, the route of the data elements of the data stream can be dynamically changed within a given limit. It is possible to construct alternative paths, which can receive the data elements in case the other paths were overloaded. This approach is analogous to the traditional SIMD multi-threaded execution.

The processed data streams may have further merges and branches in RCPU as it can be seen in Figure 3.9. Two data streams can be merged if previously they logically belonged to the same stream. In Figure 3.9, the route of the merge is designed so that the conditions of the convergent data streams are appropriate to the requirement of the program. It is possible that the program requires the implementation of a loop, then the route of the data stream is a closed path. The data stream enters at a given element of the closed route, and also the processed data stream leaves at a given element of the closed route. An important feature of a loop is that it mixes the data which demonstrates that the data sorting ability of the MU has high importance.

3.2.5 Detailed structure of the processing element

In Figure 3.10 the internal structure of the processing element can be seen. The processing element includes an *input multiplexer* (input MUX) capable of processing the input of the ports, a local memory connected to the input MUX, a

DOI:10.15774/PPKE.ITK.2014.009

local processing unit (LPU) connected to the output of the input MUX, and an *output multiplexer* (output MUX) connected to the output of the LPU, providing the output of the processing element.

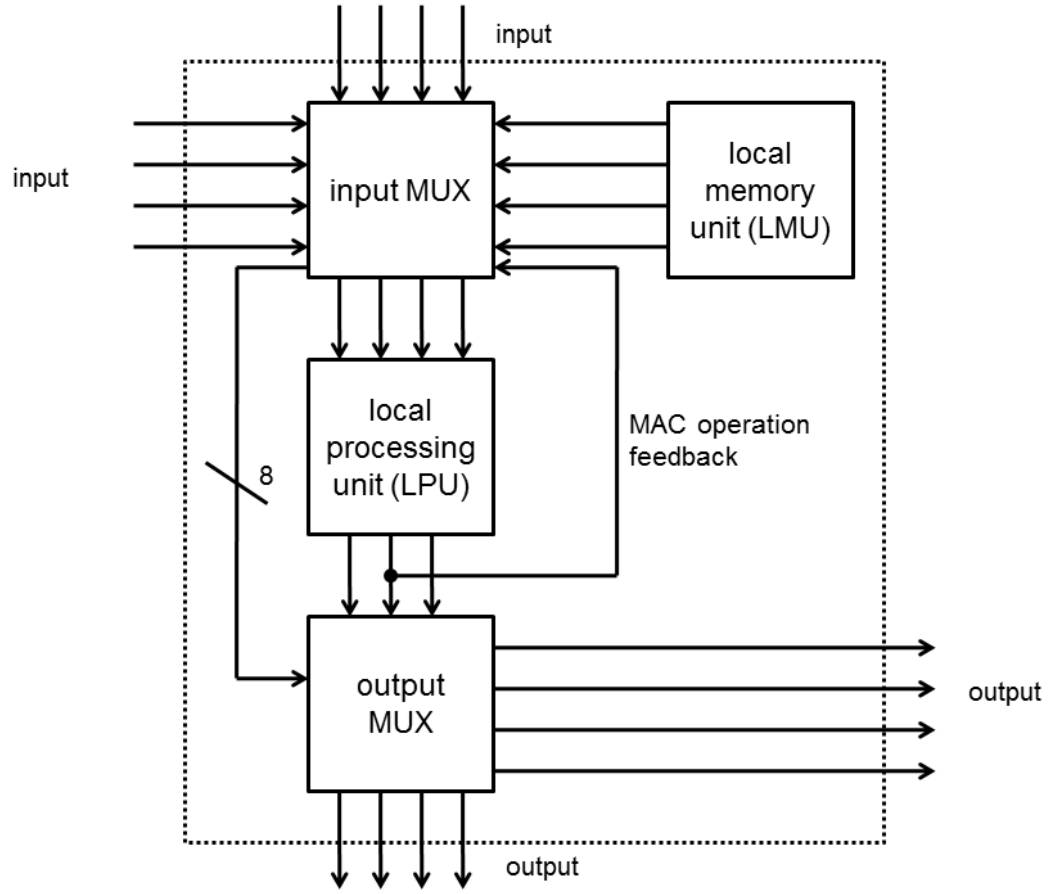


Figure 3.10: The internal structure of the processing element of the RACER architecture is depicted. The processing element includes an input multiplexer (input MUX) capable of processing the input of the ports, a local memory connected to the input MUX, a local processing unit (LPU) connected to the output of the input MUX, and an output multiplexer (output MUX) connected to the output of the LPU, providing the output of the processing element.

The LPU can be an ALU and/or FPU. The currently used ALUs and/or FPUs (like in GPU architecture) can be applied in RACER architecture, but small changes may be required. Applying of an FPU as an LPU can be suitable

DOI:10.15774/PPKE.ITK.2014.009

for 3D visualization, scientific calculations and simulations. An ALU is much smaller than an FPU, therefore it can be integrated easier into the processing elements.

The processing element also contains the input and output MUXes, which controls the travel of the data elements between the input, LPU and output. As previously mentioned, the MUXes are controlled by LUTs stored in the LMU. These LUTs can implement all kinds of RACER control-flow including conditional branches, by receiving all the relevant pipeline status information along with the condition. This status information is the occupancy of each ingoing and outgoing pipeline stage of MUX. The clock signal can be also included, which used to implement fork and PHI branches for evenly splitting and merging data streams.

The LMU is a read-only memory with 256-512 bits size. It usually contains a few arithmetic constants and multiplexer control LUTs which are initialized when the head of the program stream reaches the given processing element.

In Figure 3.10, the inputs are the upper and left side of the processing element, the outputs are the lower and right side of the processing element. Accordingly, the inputs of the processing elements are connected to their upper and left neighbors (to processing or data routing elements), respectively, the outputs of the processing elements are connected to their lower and right neighbors (to processing or data routing elements). It is also possible that in other implementations of the architecture, at each side of the processing elements there are inputs as well as outputs too. In these implementations, two-way data exchange is also allowed between the neighboring processing elements. For the implementation of a closed route, utilization of enhanced processing elements is necessary.

In Figure 3.10 the input and output of the processing element is interconnected by bypass and feedback connections. If it is required, the LPU can be bypassed or its output can be feedback (MAC operation feedback for signal processing) to the input MUX.

3.2.6 Applied pipeline processing

In Figure 3.11, the internal structure of an FPU, which is an example for an LPU, can be seen. The FPU contains a *multiplier unit*, two *compare units* (compara-

DOI:10.15774/PPKE.ITK.2014.009

tors) and an *adder unit*, which are connected to each other as it is illustrated in Figure 3.11. The RACER computer architecture operates based on the pipelined parallel principle, accordingly pipeline stages are designed. A data element is defined as an amount of data, which can be processed during the processing time of a single pipeline stage. *Pipeline registers*, that are suitable to store a data element, are assigned to the *pipeline stages* of the processing and data routing elements of the RACER architecture. While processing the data stream, a data element is stored in the corresponding pipeline registers of its pipeline stage and transferred to the pipeline register of the next pipeline stage at least one processing time unit later.

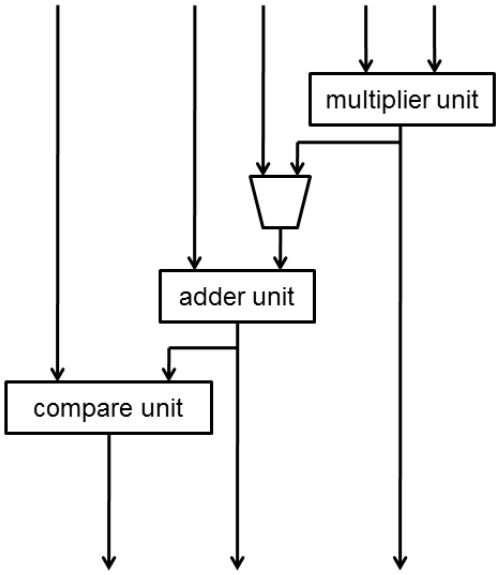


Figure 3.11: The internal structure of the FPU without pipeline stages. The FPU contains a multiplier unit, two comparative units (comparators) and an adder unit, which are connected to each other.

In Figure 3.12, the internal structure of FPU can be seen depicted with pipeline stages. The number of pipeline stages depends on the applied technology. The given pipeline stage allocation is typically around 1GHz clock frequency. In Figure 3.12 the multiplier, the adder and also the comparison unit comprises more than one pipeline stage. One of the important features of the RACER

DOI:10.15774/PPKE.ITK.2014.009

computer architecture is that it does not include global wiring, the clock signal propagates its waves locally. Because of this, single cycle delay may occur during the processing of data streams, depending on the relative angle of propagation of the data stream and the clock single wave. The length of the delay in the FPU can also depend dynamically on the processed data too, because all these extra and less predictable delays are implicitly handled by the locally controlled pipeline nature of the architecture.

A processing element is staying typically identical for a long time during processing, doing the same operation on different data. This behavior is expected from stream architectures (data-flow driven architectures), furthermore the RACER architecture can be defined as a super-set of classic data-flow machines.

The clock wires are smaller, shorter, and the clock signal amplifiers (clock buffers) are fewer, because RACER architecture uses a locally wired asynchronous clock, which has the same frequency everywhere on the chip, but the phase is spatially different. As a consequence, the frequency is not limited by the capacity of global wiring, besides using higher processing frequency, lower power consumption can be achieved than in the case of global wiring solutions.

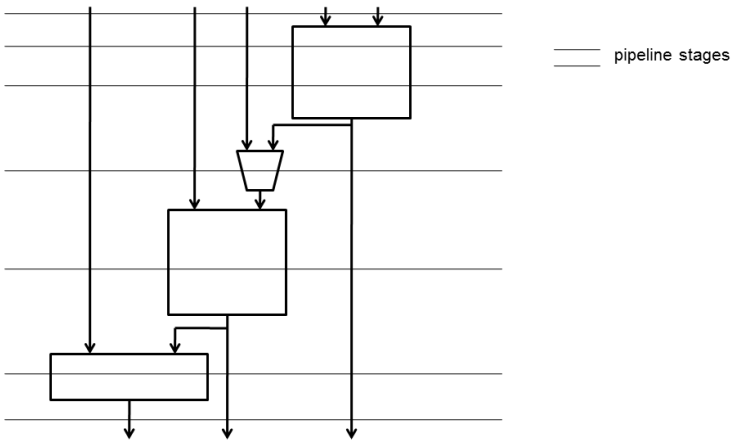


Figure 3.12: The internal structure of the FPU with pipeline stages. The multiplier, the adder and also the comparison unit comprises more than one pipeline stages. Pipeline registers, that are suitable to store data elements, are assigned to the pipeline stages of the processing.

DOI:10.15774/PPKE.ITK.2014.009

In case of pipeline processing, the data streams are routed through processing elements, data routing elements of RCPU and the MUs. Typically more stages are in the processing elements and at least one stage is in the data routing elements. In order to store the data elements, a pipeline register (e.g. D-flip-flop) belongs to each pipeline stages. The route of the data stream can be described by the sequentially visited pipeline stages. Where these visited pipeline stages are neighboring pipeline stages, they are responsible for the processing or the transfer of the data element.

The data stream is constructed in a way that during the normal processing of the data stream, the data elements of the data stream are stored in every other pipeline register of the pipeline stages. The state of the data stream is called half-speed processing, when every other pipeline stage is loaded with a data element, ie. one data element and one empty stage follow each other repeatedly.

The half-speed processing is preferred, because in case of an obstruction caused by a loop, a congestion may occur and the data stream can stop. In the case of a congestion the processing of the data stream is not in normal operation state, the empty pipeline stages propagate backwards. This happens because the indication of the empty stage propagates backwards at the same speed as the data elements propagate forward.

If there is any fork branch in the route of the data stream, in case of an obstruction the data elements at the junction can choose the bypass direction. So obstructions can be effectively overcome by branches and bypass routes, which means that the architecture can dynamically divide the work between alternative paths. The compiler can take into account the potentially dangerous congestion locations and determine the route of the processing providing such detours. Determining the congestion points can happen heuristically, or by benchmarking the program. From the algorithmic viewpoint, congestion happens because the bandwidth is higher than the processing speed. This is often the case with more complex algorithms. Consequently the congestion is a useful feature because it can gracefully decrease the bandwidth to exactly match the processing speed of the implemented algorithm. Solving the congestion problem involves adding bypass routes, which just means the increasing of processing power by adding more parallelism. these bypass routes are effectively duplications of the bottleneck part

DOI:10.15774/PPKE.ITK.2014.009

of the program. Consequently bypass routes execute the same computations, so their results can be simply merged back to the original route.

The length of the detour does not necessarily have to match with the length of the normal route. The mixture of data elements caused by the difference can be fixed efficiently with the sorting memory process of the RACER architecture.

The use of half-speed processing is also advantageous because the architecture will not be sensitive to processing delays, and clock-cycle phase delay up to 90? is allowed without any effect to the processing. The unusually high tolerance of out-of-phase clocks can be explained by the pipeline behavior where a moving data element is always surrounded by empty pipeline stages, which avoids collisions. Full-speed processing may also be used, in this case data elements fill each pipeline stages, and we have double processing speed compared to the half-speed, however we lose the sophisticated pipeline control, and all looped control-flow capabilities.

Compared to the full-speed processing we lose speed at half-speed processing, but because of the above-described properties (local wiring only, difficulties of full-speed processing, etc.), the half-speed processing of the RACER architecture is more effective than full-speed processing.

3.2.7 Half-speed mode pipeline timing

In half-speed mode the data only propagates between pipeline stages in a way that it leaves every other stage empty. This allows the flexible and local control of the data flow. This feature could be implemented by semi-synchronized logic, where both self timing state changes and clock driven state changes happen. As a side effect of the self timing mechanism, each pipeline stage only listens to every other clock cycle. This makes the pipeline much more robust against clock phase variations between neighbor pipeline stages, since the stages are effectively running at half clock speed. In certain designs where we enforce explicit data stream synchronization in the program level, which makes program implementation much simpler, we can effectively tolerate random clock phase variations in the half-speed mode pipeline, because of the self timing nature of the architecture. Two simple pipeline stages connected serially is depicted in Figure 3.13, the extra part compared to a simple full-speed pipeline is the feedback mechanism.

DOI:10.15774/PPKE.ITK.2014.009

The feedback propagates backwards compared to the direction of the data flow, pipeline control is implemented by this local feedback connection between every pipeline stage.

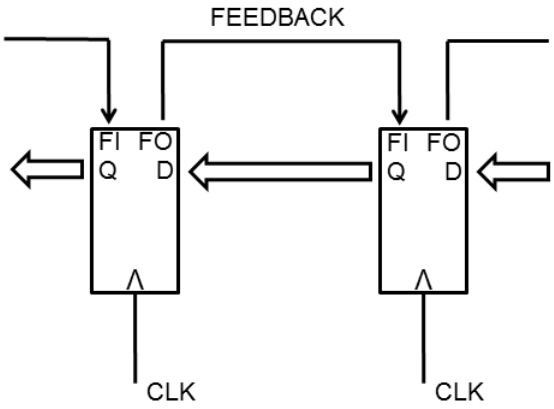


Figure 3.13: Connection of two consecutive pipeline stages for half-speed mode. The feedback plays an important role in controlling when the data propagates, and it effectively enforces the half-speed operation.

Timing diagram of a single pipeline stage during half-speed operation, where every other stage is filled, is depicted in Figure 3.14.

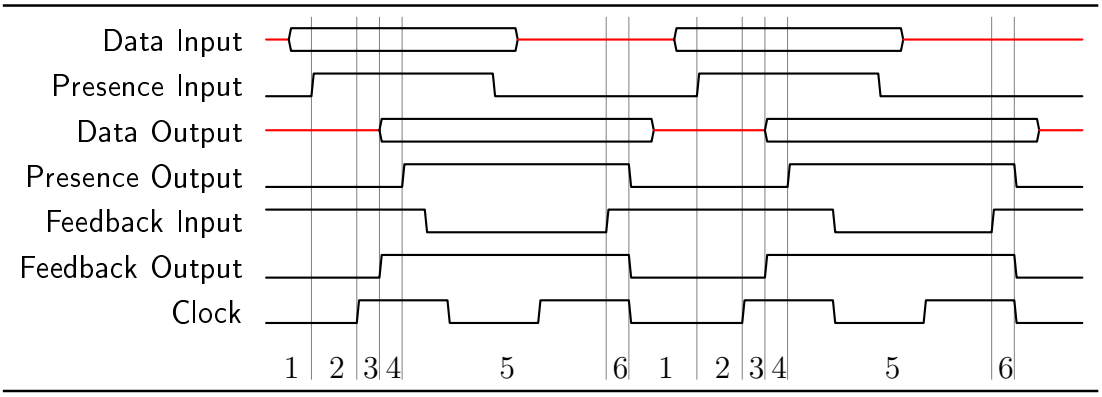


Figure 3.14: Timing diagram of a single pipeline stage during normal operation. It can be seen that data transfer speed is half of the clock speed because of the half-speed operation.

DOI:10.15774/PPKE.ITK.2014.009

State number	Pipeline Register Storage	
1	Empty	waiting for data
2	Empty	waiting for clock
3	Filled	signaling feedback, data propagates to the output
4	Filled	signaling data presence, waiting for feedback event
5	Clearing	feedback event received
6	Clearing	presence output and feedback output goes low

Figure 3.15: The pipeline control cycles through various states during the operation of the pipeline

Current state	Current inputs				Next state	Next outputs		
	CLK	FI	DPI	DI		FO	DPO	DO
6-1	X	X	1	Data	2	0	0	X
2	rising edge	X	1	Data	3-4-5	1	1	Data
3-4-5	x	rising edge	X	X	6-1	0	0	X

FI	Feedback Input
FO	Feedback Output
DI	Data Input
DO	Data Output
DPI	Data Presence Input
DPO	Data Presence Output

Figure 3.16: The state table of a single pipeline stage for half-speed mode. The state change only happens when the inputs trigger it, otherwise it remains the same. The Data Input is copied to the Data Output, and is latched till it is cleared by the rising edge on the Feedback Input.

The pipeline stage cycles through the states, which are depicted in Figure 3.15. The states, which are separated by delay, are grouped together in one state, depicted in Figure 3.16. The separation of these sub-states is necessary to eliminate the hazards concerning the Presence Input and Presence Output signals. The feedback signal coming from the next stage which allows it to return to the empty state. The pipeline stage cycles between filled and empty states, where the clearing of the pipeline stage only happens when the next stage received the data, and signals backwards using the Feedback. This ensures that during half-speed operation each moving data is followed by an empty pipeline stage. If the propagation of the data stops for any reason, this mechanism uses the empty spaces to ensure that the whole pipeline stops in an orderly fashion.

DOI:10.15774/PPKE.ITK.2014.009

3.3 RACER programming

Unconditional program stream operators like forking and PHI functions can be easily implemented by the processing elements (see Figure 3.10), input and output MUXs, the half-speed mode further simplifies the circuit design. A program stream fork can be seen as a forking road, when the priority way is clogging the data travels to the second possible way. The PHI function is the inverse situation where data comes from more than one direction and data streams become one stream. In conditional processing the forking and PHI function like operations are governed by a condition calculated by the processing element's comparator (or any other way, Boolean values are represented as floating values). This can be implemented by conditionally switching the input or output MUXs.

A computer program is processed on the RACER architecture as follows. A message of a control unit or computational device includes the computer program. In this message the code for MU (control instructions) and processing tasks for RCPU (arithmetic operations) are included. This message is encoded in a graph representation, which will be processed in accordance with the hardware implementation of the architecture. The graph representation is a low level general formulation of the program, which has not been mapped topologically to the computing array yet. The ISRU is the module, which adds the appropriate path structure to the program. This structure defines how the stream gets to the RCPU, the route of the stream inside the RCPU and which memory modules are used. This routing also depends on the structure of the computer program and the routing of the other already running programs also.

While the program stream passes through the RCPU, the computing topology is configured based on the instruction stream. In the RCPU, during the programming the corresponding operations and the computational topology encoding parts are removed from the instruction stream. Therefore the instruction stream slowly disappears and at the end of the instruction stream an activation command indicates where the data stream begins. Like its name implies, the activation signal activates each and every programmable part of the architecture which it passes through. After activation, the entering data stream is processed by the computing elements. The program stream may contain branches, in this

DOI:10.15774/PPKE.ITK.2014.009

case the programming of the streaming system is more complicated, because it also had to branch, in order to reach each part of the streams. Different type of branches or their combinations are allowed. The following branches can be used:

- *Copy branch*: branch in the graph representation of the computer program.
- *Calculation branch*: Two different cases are possible, depending on the direction of the data-flow. The first where an operation produces one output from multiply inputs, or the second where an operation produces two outputs from a single input. Alternatively, these two cases can happen at the same time, so the operation would produce many outputs from many outputs. In these cases, the operation can only be completed if every input is available and every output is free at the same time. As a results of this constraint, every operation can implicitly used for synchronizing data streams.
- *Conditional branch*: there are two cases as well. In the first case, data elements are accumulated from both two directions and the next transmitted element is chosen according to a specific condition. In the other case, data elements may be transferred into two directions and the path of each data element is chosen according to an arithmetic condition.
- *PHI / Fork Branch*: in the case of PHI, data elements arrive from both directions, they are merged by priority, or in an alternating way. In the case of Fork, data elements can leave to two directions. Direction can be chosen by priority (if both directions are free, the output direction can be chosen in an alternating way).

3.3.1 Program example of RACER architecture

In Figure 3.17, the algorithm of Mandelbrot-set calculation can be seen implemented on RACER computer architecture. The Mandelbrot-set consists of those complex numbers in the complex plane, which, if substituted in the given complex-valued recursive sequence, the result does not converge to infinity. The

DOI:10.15774/PPKE.ITK.2014.009

Mandelbrot-set depicted in the complex plane is the well-known Mandelbrot fractal. To draw the fractal in the x-y coordinate system, a complex arithmetic is iterated in each pixel until in each pixel the absolute value of the complex number exceeds a given threshold. The iteration number of the complex arithmetic in a pixel defines the color of the given pixel. In order to have a finite number of iterations, a maximum iteration limit is used. The conventional C source code of the computation of the Mandelbrot-set is as follows:

```
int mandelbrot ( double x , double y )
{
  double z1 = 0 , z2 = 0;
  int iter = 0;
  double len;
  do{
    double a = z1 * z1 - z2 * z2 + x;
    double b = 2 * z1 * z2 + y;
    z1 = a;
    z2 = b;
    len = z1 * z1 + z2 * z2 ;
    iter++;
  }while ( iter < 200 and len < 16 );
  return iter ;
}
```

In Figure 3.17, the branches or logic operations are depicted by rectangles and are implemented by processing elements. Depending on the design of the processing elements, more than one logic operation can be implemented by one processing element. In Figure 3.17, the branches implement the functionality of PHI-type of branches, i.e. priority based program stream merge is implemented by them. The feedback streams depicted by filled triangles have priority in the junctions. Without giving priority to particular feedback data streams, deadlock occurs and the data streams would be waiting for each other. In data merging junctions, data streams arriving from a triangle depicted branch has priority to be processed while a stream arriving from the other branch has to be waiting.

The constants are locally stored in the LMU of processing elements. Conditional branches are marked by trapezoid symbols. In these conditional branches a logic operation, the so-called loop termination criterion decides which way the

DOI:10.15774/PPKE.ITK.2014.009

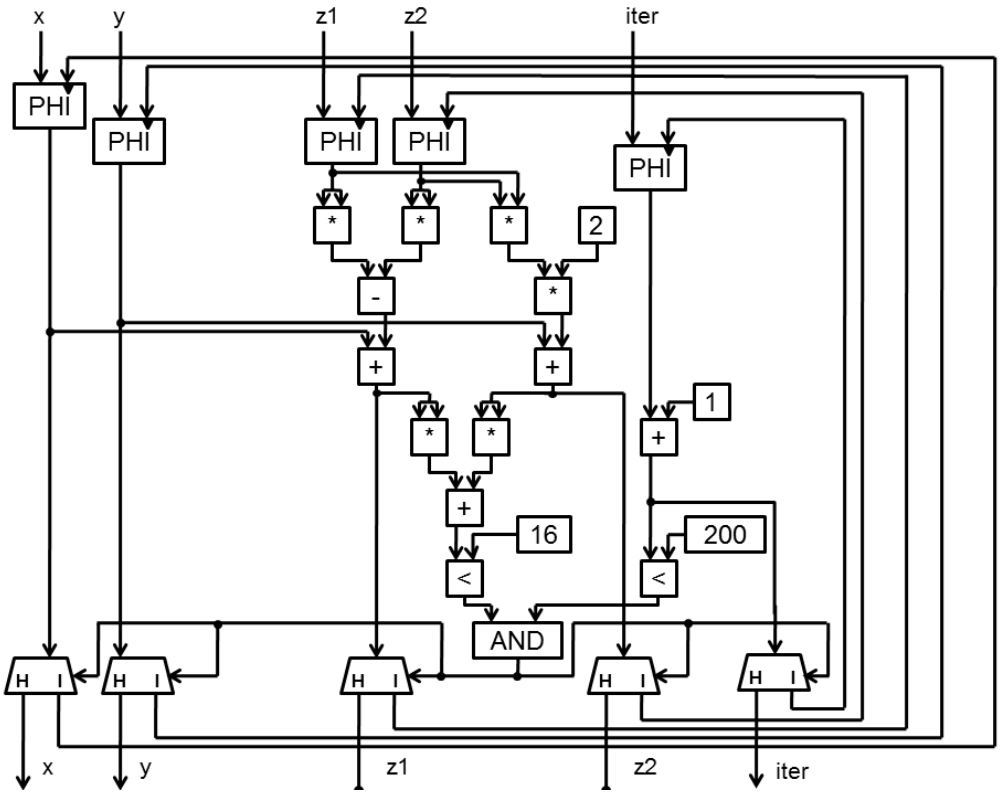


Figure 3.17: The algorithm of the Mandelbrot-set calculation implemented on RACER computer architecture. The branches or logic operations are depicted by rectangles or squares and are implemented by processing elements. Depending on the design of the processing elements, more than one logic operations can be implemented by one processing element. The feedback streams depicted by filled triangles have priority in the junctions. Without giving priority of particular feedback data streams deadlock occurs and the data streams would be waiting for each other. In data merging junctions, a data stream arriving from a triangle depicted branch has priority to be processed while a stream arriving from the other branch has to be waiting.

DOI:10.15774/PPKE.ITK.2014.009

stream and its data elements will be transmitted. The calculated variables are depicted at the inputs and the outputs of the branches. The values of x and y variables of the pixels are initialized with the current coordinates of the pixels, for the other variables the initial values are zero. The outputs provide the finished results to be transmitted to the MU. The MU organizes the data elements by their coordinates, to be able to read out the image of Mandelbrot fractal continuously. Some of the outputs are not required, since only the pixel coordinates and iteration number are used later, therefore other information is discarded. The looping data-flow mechanism can be understood by observing the operations done on each data element. In the Mandelbrot set each pixel is associated to a data element, so each data element (pixel) spends as much as iterations inside the loop, as it needs to determine the convergence. The convergence is determined by the exit condition threshold, or reaching the maximal allowed iteration count. From the viewpoint of a data element, every loop cycle is equivalent to every iteration of the original serial implementation on the pixel. However we have more than one data element in the loop at the same time, so we are parallel processing multiple pixels.

The RACER implementation of the algorithm of Mandelbrot fractal demonstrates that during computation the data elements are mixed up in the loop, because the coordinate pairs stay for a different length of time in the loop depending on the number of their iterations. The coordinates of the pixel enter the loop in order, but those that can be calculated sooner leave the loop sooner, so these overtake the more slowly calculated pixel coordinate pairs. Since the variance of processing the time of data elements is limited, the data elements are locally reordered by the loop so it can be sorted real-time by the MU. The example in Figure 3.17 also emphasizes the pipeline structure of the RACER architecture: a lot of data elements are circulating in the loop overlapped in time and all operations are executed in parallel. During the calculation of the fractal the processing elements are almost always in operation, so the utilization of the architecture is very good.

3.3.2 Simulation

DOI:10.15774/PPKE.ITK.2014.009

The Mandelbrot fractal algorithms has been also implemented by NVIDIA for their GPUs. This implementation will serve as a reference for comparing the results and the estimated speeds. The GPU was chosen as the speed reference because this algorithm is very GPU friendly. Each pixel in the fractal is computed independently, by a loop. The computation is an arithmetic iteration, which does not use any global memory and only uses a few registers (less than 16). The iteration counts are potentially different at each point, but they strongly correlate for the nearby pixels. The exceptions are the regions near the unstable island, but during the benchmark the fractal was scaled in the usual way, which reasonably minimizes this area. This ensures that iterations running in the same SIMD finish roughly at the same time. As a result, this algorithm very well utilizes the available computing resources of the GPU, because most of the time is spent on running arithmetics inside the loop.

A single Mandelbrot loop mapped to RACER consumes the following resources:

Used	Number available in a processing element	Name
2	1	numeric compare
4	1	floating point multiply
3	1	floating point add
1	1	integer add
5	4	data-flow branching
5	4	data-flow merging
11	4	explicit synchronization

The algorithms fits comfortably into a 4×4 processing block, so it consumes only 16 processing element maximum. The simulated program run at the expected speed, which averaged to 42 clock cycles / pixel in half-speed mode. The arrival of the first processed pixel at the output happened after 181 clock cycles, and the time difference between the last input pixel and the last output pixel was 442.

Because of the low routing requirements, this program can be mapped to every processing block in the RCPU, so it can fill up the whole 32×32 array. If we suppose that we are in half-speed mode and the clock frequency is 700MHz, and

DOI:10.15774/PPKE.ITK.2014.009

for every two clock cycles every processing element can execute a double precision Multiply-Add operation, then we can compare the benchmark to a reference Mandelbrot implementation on Tesla C2050 GPU.

The chip area, power consumption and technology size estimations are taken from the Considerations of VLSI implementation section, with the exception of peak performance, because in order to assure fairness in this comparison, I assumed that the processing elements contain the same double precision Mutiply-Add arithmetic cores as the NVIDIA C2050. This ensures that the relative to peak performance algorithmic efficiency is comparable.

Arch. Arch.	Mandelbrot speed	Mandelbrot relative to peak speed	Technology size	Clock freq.	Peak double performance	Chip area	Power consumption
C2050	$668\frac{Mpixel}{s}$	$1.3\frac{Mpixel}{GFLOP}$	40nm	1150MHz	512GFLOPS	529mm^2	215W
RACER	$1066\frac{Mpixel}{s}$	$1.5\frac{Mpixel}{GFLOP}$	65nm	700MHz	717GFLOPS	436mm^2	330W

It is important to note that even the relatively little optimized Mandelbrot implementation on the RACER outperforms the optimized reference CUDA implementation of the Mandelbrot fractal algorithms in peak relative metric. It means that the resource utilization of the RACER architecture is better for Mandelbrot algorithm than for the NVIDIA C2050, which is an important fact, because this algorithm maps particularly well to the GPU architecture, which should make it hard to beat.

3.4 Turing Completeness

There are several approaches to prove that the RACER architecture is Turing-complete. The trivial approach is that the RACER architecture is Turing-complete by definition because the MUs contain a control unit. This control unit is similar to a traditional processor, which if given infinite memory (and address space) can simulate any Turing machine. One other approach is detailed in the subsection below.

3.4.1 Implementing Conway's Game of Life

Conway's Game of Life is a well known Turing-complete cellular automaton, which only uses local rules to describe the time evolution of the states. The

DOI:10.15774/PPKE.ITK.2014.009

states are represented by a 2D discrete binary grid. The next state of any cell on this grid can be described by its current state, and states of cells in the Moore neighborhood (3×3 neighborhood). Let N be the number of **True** cells in the neighborhood, so $N \in [0..8]$, x is the current state of the cell, y is the next state of the cell. Then the rules are:

- if N $<$ 2 then y := False
- if N $=$ 2 then y := x
- if N $=$ 3 then y := True
- if N $>$ 3 then y := False

There are two parts of the RACER program implementing Conway's Game of Life. The first is the neighborhood tiling depicted in Figure 3.19, which is implemented by local delay elements and MU based stream delays and copies. The second part is counting the **True** cells and the rules of the time evolution, implemented by 7 adders, 3 comparators and 3 logic operations, as depicted in Figure 3.18.

The 2D grid can be transferred from the memory in a row major order. In this case the memory should create 3 delayed copies of the stream, each delayed by a whole line, so each represents a consecutive line. The column delays in order to obtain the whole 3×3 neighborhood can be done by local delays. This way the processing cores can tile through the 2D grid efficiently.

The processing cores, where the time evolution rules are executed, produce a single stream as an output which contains the input to the next iteration in row major order. This stream goes back to the memory, so it can be read out in three replicated delayed streams in the next iteration.

In this algorithm we do not even use the ordering function of the memory, we only read streams from multiple locations at once. This proves that the RACER array processor is theoretically Turing-complete even if we do not consider the advanced functionality of the MUs.

DOI:10.15774/PPKE.ITK.2014.009

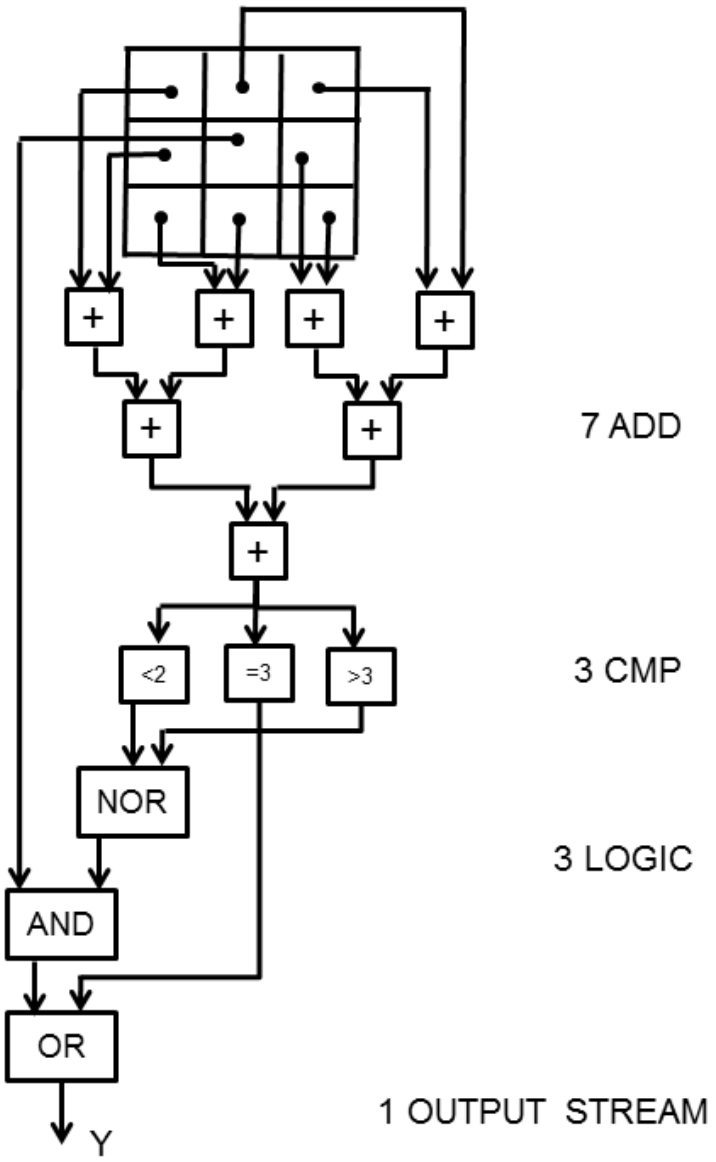


Figure 3.18: The possible implementation of the Game of Life cellular automaton rule processing is depicted in RACER graph representation.

DOI:10.15774/PPKE.ITK.2014.009

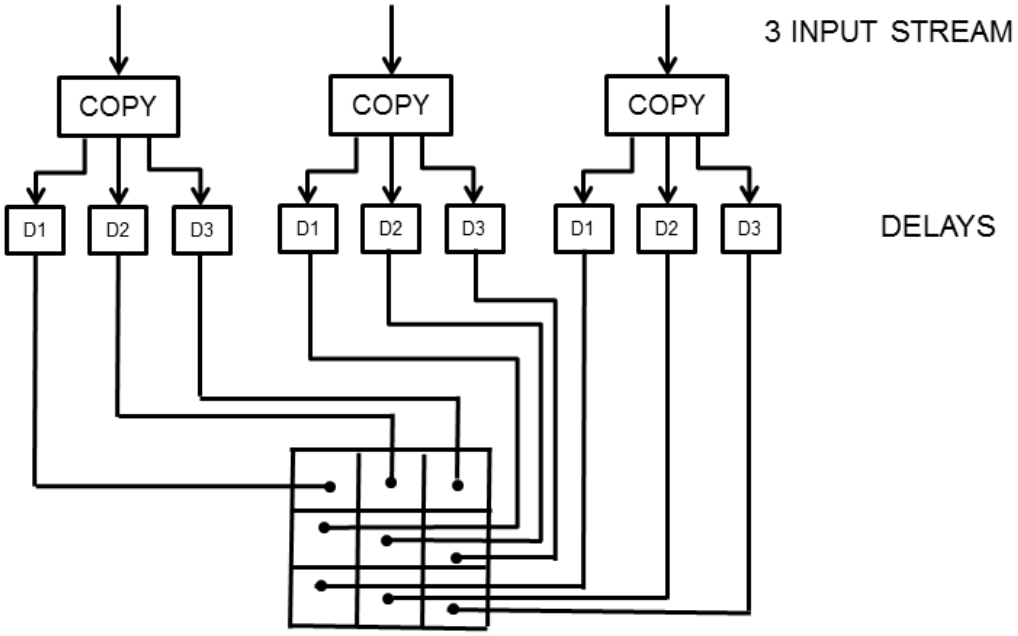


Figure 3.19: The local tiling for 3×3 neighborhood with delay elements on RACER architecture is depicted. The delay chains receive three inputs from the MUs corresponding to the three consecutive lines, and generate the whole 3×3 tile from it.

DOI:10.15774/PPKE.ITK.2014.009

3.5 Considerations of VLSI implementation

According to the available information about the latest AMD GPU micro-architecture, the Graphics Core Next (GCN), the area inside the cores is divided equally between the computing cores, the memory and the control logic (not considering the texture units). My estimation of gate count is based on on-chip register memory sizes, where the on-chip memory in gates is approximately the number of cores $\times$ 100K. Coincidentally the IP core I have chosen (GRFPU-1 from Aeroflex Gaisler) as a processing element equivalent for my estimation is also 100K gates big. From this I can estimate that a single core together with control logic and memory is about 300K gates.

If I assume that in the equivalent gate counting, every gate is made of 4 transistors, and the gates/transistors are spread evenly on the chip die, then I can make a guess of the total transistor count of the chip: 2048 number of cores in AMD Tahiti $\times$ 300K gates $\times$ 4 transistors/gates $\times$ 1.25 (+25 % texture unit) $\times$ 1.33 (+33% other graphics oriented logic circuits) $\approx$ 4000 million transistors. The actual transistor count of the AMD Tahiti is 4310 million, so this estimation is roughly accurate.

If we exclude the strictly graphics oriented hardwares on this GPU (AMD Tahiti) and consider only the computing parts of the cores, the surface ratio of the computing cores is $\approx$ 33%. If we want to compare this to other architectures we need to speculate from other values because these exact surface/gate ratios are very rarely published. We know that the GFLOPS / area and GFLOPS/Watt is one of the biggest for GPU if we only consider general purpose architectures. Since the number of computing cores and peak performance is very closely related, it is reasonable to state that the GPUs have one of the highest computing core surface/gate ratios compared to other architectures.

The chosen processing element equivalent IP core was GRFPU-1 from Aeroflex Gaisler, its size is 100K gates which is roughly equivalent to the size of the computing core in AMD Tahiti. However, IP cores from smaller feature technologies tend to be bigger due to bigger pipelines and higher frequencies, so my estimation, where I have used 100K gates for 65nm and 90nm technologies is an overestimation. However, the AMD Tahiti cores execute double precision only at $\frac{1}{4}$ speed

DOI:10.15774/PPKE.ITK.2014.009

while the processing element does it at full speed, so for the comparison I have limited the Processing Elements to $\frac{1}{4}$ speed as well.

The GRFPU-1 is a fully featured single and double precision Floating Point Unit. It is fully pipelined and it can do an operation for every clock cycles. These operations include: add, subtract, multiply, divide, square-root, convert, compare, move, abs, negate. It can also compute square root and divide parallel to other operations. Unfortunately the add and multiply operations are not pipelined together into a MAD or FMA operation but we can assume this capability because it does not increase the gate count significantly. If we ignore the divide and square root capability and assume it has the MAD/FMA operation then this FPU is similar to the GPU compute cores, except it can operate in double precision at full speed.

Each routing element contains:

- 4 pipeline stages (2 is the minimum, this is an overestimation)
- 32bit program memory for configuring the routing element
- 4way bidirectional multiplexer
- control logic
- extra amplifiers for the buses

My estimation for the number of gates of each routing element: 128 bit bus width $\times$ 4 connections $\times$ 6 gates per DFF + 128 bit bus width $\times$ 16 gates per 4way MUX + 32bit $\times$ 2 gates per program bit + 1024 gates for control logic + 128 bit bus width $\times$ 4 connections $\times$ 2 amplifier gates = 7232. If we use 3×3 topology for routing elements we get about 4600 routing elements for 1024 cores, so we have 33M gates in the whole array inside the routing elements. This covers about 20% of the surface. If we suppose that the routing element $\frac{1}{3}$ is size of the Processing element, then from the topology in Figure 3.7 we get 36% coverage, so Figure 3.7 obviously overestimates the size of the routing elements.

From these values I estimated the final chip sizes and the power consumptions with Cadence InCyte Chip Estimator for 90nm and 65nm technologies. This chip estimator assumed that I use regular clock distribution network where clocks are

DOI:10.15774/PPKE.ITK.2014.009

tightly controlled. This adds approximately 2M gates and 34W-81W (about 20%) extra power to the estimation. For the RACER architecture, a much less tightly controlled clock distribution network is sufficient, so this could be in theory significantly reduced but estimating this reduction would need precise VLSI design of the architecture.

Technology feature size	Clock frequency	Chip surface	Total power consumption	Clock tree power	Double precision speed	PE surface ratio	Routing element surface ratio
90nm	400MHz	561mm^2	224W	42W	819GFLOPS	72%	21%
90nm	600MHz	564mm^2	454W	81W	1229GFLOPS	72%	21%
65nm	500MHz	355mm^2	226W	34W	1024GFLOPS	70%	21%
65nm	600MHz	369mm^2	280W	46W	1229GFLOPS	67%	20%
65nm	700MHz	436mm^2	330W	60W	1434GFLOPS	57%	17%

I have compared GPU peak performances to the estimated RACER peak performances in order to highlight the possible performance gains coming from the higher number of computing cores (Processing Elements).

GPU name	Technology feature size	Nearest estimation feature size	Single prec. speedup	Double prec. speedup	Power ratio
Radeon HD 2900XT	80nm	90nm	1.7×	6.9×	1.05
Radeon HD 4870	55nm	65nm	1.2×	1.5×	2.1
GeForce 8800 GTS	65nm	65nm	2.3×	4.6×	2.5

AMD GPUs have 33% of their surface covered by computing core, excluding graphics specific modules. NVIDIA GPUs usually have lower ratios, or similar. By estimation the RACER architecture coverage is between 57% and 72%, if we consider using the same cores for Processing Elements, this translates to 2× speedup, but at the cost of higher power consumption, because the surface is better utilized.

3.6 Conclusions

Based on my research into parallelization of algorithms and many-core architectures, I have designed a massively parallel scalable architecture called RACER. This architecture aims to support the arbitrary scaling of the number of cores and the closer integration of memory while providing good performance for less parallel algorithms also. In this architecture I redefined the job the memory, making

DOI:10.15774/PPKE.ITK.2014.009

it an integral part of the implemented algorithm, which allowed the processing elements (cores) to be more specialized, and much more efficient.

The RACER architecture includes a RCPU, MUs, PUs, RCU and ISRU. The units of the architecture are connected to each other through the RCPU. The RCPU processes the program stream, which consists of an instruction stream and a data stream divided into data elements. The ISRU, which defines the instruction stream, can be integrated in RCPU, but in any case is still a separated unit within the RACER architecture. The RCPU contains an array of blocks of processing elements and each block is surrounded by data transfer elements. Lateral processing elements are connected to the neighboring data transfer elements. Lateral processing elements are partly those which are physically located on the edge of the array, secondly those processing elements through which the data stream enters and leaves a block.

The computing power per unit area of multiprocessor architectures can be estimated and compared. The RACER computer architecture provides hopefully more computing power per unit area than the current GPU architectures based on the following reasons:

- There is no cache memory in the RACER, which can cover up to 33% of the chip surface of the GPU.
- There is no register-file memory in the RACER, which can cover up to 17% of the chip surface of the GPU.
- The connections between the processing elements are local in the RACER. The connection layer can be overlapped with the other layers, therefore it only needs little extra surface.

According to our calculations, even though using additional data routing elements and ISRU the utilization of the surface of the RACER architecture is significantly more effective than GPU's.

The RACER computer architecture is well applicable for 3D visualization, ethernet routing, cryptography, management of large databases, simulations and scientific calculations. The RACER architecture is Turing-complete, which means

DOI:10.15774/PPKE.ITK.2014.009

that an arbitrary Turing machine can be implemented on the architecture. The components of the RACER architecture can be integrated into a single integrated circuit, and their parameters can be tuned on demand according to the development of technology.

Based on Patent and Trademark Attorneys' prior art search, their official opinion declares that RACER computer architecture is new (the closest systems are [56, 57]) and novel improvements are included.

DOI:10.15774/PPKE.ITK.2014.009

Chapter 4

The BRUSH Algorithm

In this Chapter a new algorithmic approach is presented, developed to evaluate two-electron repulsion integrals based on contracted Gaussian basis functions in a parallel way. This new algorithm scheme provides distinct SIMD (Single Instruction Multiple Data) optimized paths which symbolically transforms integral parameters into target integral algorithms. Contrary to the common solutions, this method uses off-line selection of the optimal path and off-line code generation. This approach is optimized for GPUs, my measurements indicate that the method gives a significant improvement over the CPU-friendly PRISM algorithm. The benchmark tests (evaluation of more than 10^8 integrals using the STO-3G basis set) of our GPU (NVIDIA GTX 780) implementation showed up to 750-fold speedup compared to a single core of Athlon II. X4 635 CPU.

4.1 Introduction

The direction of the development of information technologies shows that in the next decade the trends will be determined by the exponential growth in the number of processors of parallel, many-core architectures [59, 58, 75]. The GPUs are increasingly used in supercomputers and scientific computing [60]. In addition to a large number of computing units of GPUs, its hierarchical memory structure also plays a prominent role in data processing and computing [58]. If the algorithms of computational quantum chemistry could be implemented efficiently on already available parallel systems, the researchers would be able to simulate

DOI:10.15774/PPKE.ITK.2014.009

larger molecules than could have been simulated before. My goal is to efficiently implement the two-electron integration task - which is the most computationally intensive part of quantum chemistry calculations [61] - based on GPU to solve general simulation problems.

The first GPU based implementations of quantum chemistry related to computational tasks had significant limitations in both accuracy and programming difficulties of the technology [63, 62, 64]. Although the latest GPU architectures can be programmed in a much more user-friendly way, moreover industrial standards are provided (CUDA, OpenCL) [65, 66], still, the efficient programming of GPUs still lacks deep knowledge of the detailed architecture and fundamentally different algorithm design. The existing industrial standard programming interfaces (CUDA, OpenCL) were also used for quantum chemistry calculations in recent years [67, 68, 69, 70, 71, 72, 73, 74], but there are still technical and algorithmic problems with these approaches which researchers are unable to solve code implementation over "f" orbital on GPU [81, 80]. I hope that these fundamental problems would be solved effectively with my approach.

In this Chapter, I propose a new meta-algorithm called BRUSH, and test it on several different molecules comparing my results between different GPU and CPU implementations.

In molecular integral evaluation over Gaussian basis functions, the recursive algorithms [76, 77] play a central role, which trace back the problem to a large number of elementary integral terms. Due to the unrolling of this recursive algorithm, we can do a significant optimization by algebraic simplifications on contracted and un-contracted integrals. Moreover we can place the contraction step not just between, but inside the integral transformation step, by using algebraic transformations. While the atomic centers in the basis functions are different in every molecule, the Gaussian exponents only depend on the type of atom and the basis set library [100]. It means that specific integral solvers can be compiled for specific atom types and basis sets, which enables us to compute the Gaussian exponent part of the solution off-line, using constant substitution and propagation on the generated code. The only drawback is that we generate far too many integral solvers, but we can mitigate this problem by only doing constant substitution on contracted basis functions. Contraction is usually used on lower orbitals

DOI:10.15774/PPKE.ITK.2014.009

(s, p) [101], where the integral computation is much simpler, which means that doing this off-line optimization is computationally cheap. However we can choose to optimize only the very often used configurations, so we can keep the amount of compilation work under control.

It is a well known fact, that applying the contraction in the right place while solving the integral can significantly boost the computation speed of the heavily contracted integrals [78]. The PRISM meta algorithm [78] uses this approach to heuristically choose the best algorithm most suited to the given quartet to solve. Unfortunately the PRISM algorithm is neither SIMD optimized and nor entirely compatible with my GPU based approach.

BRUSH, based on Head-Gordon-Pople (HGP-) and McMurchie-Davidson (MD-) PRISM [77, 82] algorithms, specially tailored for SIMD architectures and off-line unrolling of control structures. In my meta algorithm we apply similar paths to HGP and MD integral solvers, sometimes mixes of the two. But in case of heavy contraction, we can afford to analytically split the generated code parts to contraction variant and invariant parts, and place the contraction between them. This way the code can be further optimized.

Another significant difference is that because we are generating the unrolled code off-line, we can do all decisions about the optimal solution path off-line. My algorithm consists of main-paths, and sub-paths. Main-paths are split to sub-paths where main-paths of the solution can be decided by simply looking at the contracted and non-contracted parts of the integral quartet. It is generally hard to choose the optimal sub-path that is why we compile all of them, and decide after looking at the code complexity and memory usage of the code.

According to the measurements, my GPU algorithm in single precision run on Nvidia GTX780 was over 700× faster than NWChem 6.3 [79] run on a single core of Athlon II X4 635, and over 100× faster the NWChem 6.3 running on all four cores of Intel i7-3820 (Sandy Bridge) processor.

This chapter is organized as it follows: Section 1 gives the introduction and outlines the background of the problem. In Section 2, the basic notations and definitions are described. Section 3 provides the detailed description of the BRUSH algorithm for two-electron integrals on GPU. In Section 4, the discussed bench-

DOI:10.15774/PPKE.ITK.2014.009

marks and my timing measurements are presented for different molecules. Section 5 gives a brief summary of the conclusions.

4.2 Notations and definitions

In this Section, to overview the theoretical background and introducing our representation, some of notations and definitions - used in references [83, 78, 82] - were followed.

Let us define an unnormalized primitive Cartesian Gaussian function in the following manner

$$\varphi_{ak}(\mathbf{r}) = (r_x - A_x)^{a_x}(r_y - A_y)^{a_y}(r_z - A_z)^{a_z} e^{-\alpha_k|\mathbf{r}-\mathbf{A}|^2} \tag{4.1}$$

where $\mathbf{a} = (a_x, a_y, a_z)$ is the angular momentum vector, $\mathbf{A}$ is the position vector, and α_k is its exponent. The angular momentum can be calculated as $a = (a_x + a_y + a_z)$. Primitive functions are defined by primitive shells, and each shell is constituted by a given center and a given exponent.

Contracted Cartesian Gaussian function is defined by the linear combination of primitive functions

$$\phi_{\mathbf{a}}(\mathbf{r}) = \sum_{k=1}^{K_A} D_{\mathbf{a}k}\varphi_{\mathbf{a}k}(\mathbf{r}), \tag{4.2}$$

where the contraction coefficients are $D_{\mathbf{a}k}$ and K_A is the degree of contraction of $\Phi_{\mathbf{a}}$.

The primitive four-center Gaussian electron repulsion integral is defined by the following formula

$$[\mathbf{a}_k\mathbf{b}_i|\mathbf{c}_m\mathbf{d}_n] = \int_{\mathbf{r}_1}\int_{\mathbf{r}_2} \varphi_{\mathbf{a}k}(\mathbf{r}_1)\varphi_{\mathbf{b}l}(\mathbf{r}_1)\frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}\varphi_{\mathbf{c}m}(\mathbf{r}_2)\varphi_{\mathbf{d}n}(\mathbf{r}_2)d\mathbf{r}_1 d\mathbf{r}_2 \tag{4.3}$$

Commonly this integral is denoted by $[\mathbf{ab}|\mathbf{cd}]$ and the subscripts of the left-side are expunged because of its particular interest. The contracted four-center Gaussian electron repulsion integral is defined by the combination of Equation (4.2) and Equation (4.3)

DOI:10.15774/PPKE.ITK.2014.009

$$(\mathbf{ab}|\mathbf{cd}) = \sum_{k_A}^{K_A}\sum_{k_B}^{K_B}\sum_{k_C}^{K_C}\sum_{k_D}^{K_D} D_{\mathbf{a}k_A} D_{\mathbf{b}k_B} D_{\mathbf{c}k_C} D_{\mathbf{d}k_D} [\mathbf{a}_{k_A}\mathbf{b}_{k_B}|\mathbf{c}_{k_C}\mathbf{d}_{k_D}] \tag{4.4}$$

$L_{tot} = (a+b+c+d)$ is defined as the total angular momentum of the electron repulsion integral. $K_{bra} = K_A K_B$ is the bra degree of contraction, $K_{ket} = K_C K_D$ is the ket degree of contraction. $K_{tot} = K_{bra} K_{ket}$ is the total degree of contraction.

A new center $\mathbf{P}$ can be assigned to $\varphi_{\mathbf{a}}$ and $\varphi_{\mathbf{b}}$ primitive Gaussian functions, and another center $\mathbf{Q}$ is analogously assigned to the primitive functions $\varphi_{\mathbf{c}}$ and $\varphi_{\mathbf{d}}$. Center $\mathbf{P}$ and center $\mathbf{Q}$ and their parameters ζ, G_{AB}, U_P and η, G_{CD}, U_Q are defined in the following manner

$$\zeta = \alpha + \beta \qquad \eta = \gamma + \delta \tag{4.5}$$

$$G_{AB} = e^{-\frac{\alpha\beta}{\zeta}|\mathbf{A}-\mathbf{B}|^2} \qquad G_{CD} = e^{-\frac{\gamma\delta}{\eta}|\mathbf{C}-\mathbf{D}|^2} \tag{4.6}$$

$$U_P = D_A D_B G_{AB} \left(\frac{\pi}{\zeta}\right)^{\frac{3}{2}} \left(\frac{1}{2\zeta}\right)^{a+b} \qquad U_Q = D_C D_D G_{CD} \left(\frac{\pi}{\eta}\right)^{\frac{3}{2}} \left(\frac{1}{2\eta}\right)^{c+d} \tag{4.7}$$

$$\mathbf{P} = \frac{\alpha\mathbf{A} + \beta\mathbf{B}}{\zeta} \qquad \mathbf{Q} = \frac{\gamma\mathbf{C} + \delta\mathbf{D}}{\eta} \tag{4.8}$$

where $\varphi_{\mathbf{a}}$, $\varphi_{\mathbf{b}}$, $\varphi_{\mathbf{c}}$ and $\varphi_{\mathbf{d}}$ are centered at $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$, with angular momenta a, b, c and d, with exponents α, β, γ and δ, and with contraction coefficients D_A, D_B, D_C and D_D.

Based on the previous interpretation of theory [83] the following equations are defined for the computation of the class defined by Equation (4.4)

$$\mathbf{R} = \mathbf{Q} - \mathbf{P} \tag{4.9}$$

$$R^2 = R_x^2 + R_y^2 + R_z^2 \tag{4.10}$$

$$\vartheta^2 = \frac{\zeta\eta}{\zeta+\eta} \tag{4.11}$$

$$T = \vartheta^2 R^2 \tag{4.12}$$

$$U = U_P U_Q \tag{4.13}$$

$$[\mathbf{0}]^{(m)} = U\vartheta^{2m+1}\sqrt{\frac{2}{\pi}}\int_0^1 t^{2m} e^{-Tt^2} dt \tag{4.14}$$

DOI:10.15774/PPKE.ITK.2014.009

The representation of contraction unlike on Equation (4.4) can be separated by *Bra* and *Ket* contractions like on Equations (4.15) and (4.16)

$$(\mathbf{bra}|\mathbf{ket}] = \sum^{K_{bra}} [\mathbf{bra}|\mathbf{ket}] \tag{4.15}$$

$$(\mathbf{bra}|\mathbf{ket}) = \sum^{K_{ket}} [\mathbf{bra}|\mathbf{ket}] \tag{4.16}$$

However, as it was shown earlier [83], generalizing this approach by exponent ratios can not be useful while simultaneous scaling of the uncontracted quantities is included. Equations (4.17) and (4.18) explain this type of contraction as follows

$$_{a'b'p'}(\mathbf{r}] = \sum^{K_{bra}} \frac{(2\alpha)^{a'}(2\beta)^{b'}}{(2\zeta)^{p'}} [\mathbf{r}] \tag{4.17}$$

$$[\mathbf{0}]^{m}_{c'd'q'} = \sum^{K_{ket}} \frac{(2\gamma)^{c'}(2\delta)^{d'}}{(2\eta)^{q'}} [\mathbf{0}]^{(m)} \tag{4.18}$$

According to these generalizations on Equations (4.15), (4.16), (4.17) and (4.18) we can formally handle the integral transformation steps and contraction steps, and perhaps choose an optimal order. Our notation is closely based on this generalized braket representation.

In the following I briefly review the MD and HGP algorithms, which serve the base of the new methods, outlined in this article. Here I present only the most important features, the reader is referred to references [77, 82] for further details.

4.2.1 McMurchie-Davidson (MD) algorithm

$[\mathbf{0}]^{(m)}(0 \leq m \leq (a+b+c+d))$ is given as a set, it was presented in [82, 83] and also by MD, that a recurrence relation (noted as RR) can be used, forming $[\mathbf{r}](0 \leq r \leq L_{tot})$ electron integral repulsion set. With the same sign, the set of $[\mathbf{r}]$ is equal to the electron integral repulsion set which is defined by $[\mathbf{p}|\mathbf{q}]$ representing the electrostatic integration between two, **P** and **Q** primitive Hermite functions. The **P** centered function is called p-bra, and symbolized by $[\mathbf{p}|$. Respectively, the **Q** centered function is called q-ket, and symbolized by $|\mathbf{q}]$. By MD and [82, 83] it was presented, that using recurrence relations $[\mathbf{p}|\mathbf{q}]$ can be bra-transformed to $[\mathbf{ab}|\mathbf{q}]$ and $[\mathbf{ab}|\mathbf{q}]$ can be ket-transformed to an appropriate $[\mathbf{ab}|\mathbf{cd}]$.

DOI:10.15774/PPKE.ITK.2014.009

This algorithm allows more efficient contraction placement and recurrence relations than the classical Obara-Saika because the recurrence relations in Equations (4.19) and (4.20) are respectively left and right contraction invariant. And the order of Equations (4.19) and (4.20) is a degree of freedom which can used to further optimize the computation. The recurrence relation in Equation (4.25) is a trivial transformation and it does not increase the complexity of the computation at all. The last recurrence relation in Equation (4.26) can only be done on primitives because the virtual **R** center is contraction variant, that is, the complexity of this recurrence relation is quadratic only at first glance, in fact, the number of resulting [**0**] brakets is proportional to the angular moment **r**.

$$[(\mathbf{a}+\mathbf{1}_i)\,\mathbf{bp}| \equiv p_i\,[\mathbf{ab}\,(\mathbf{p}-\mathbf{1}_i)| + (P_i - A_i)\,[\mathbf{abp}| + \frac{1}{2\zeta}\,[\mathbf{ab}\,(\mathbf{p}+\mathbf{1}_i)| \tag{4.19}$$

$$|(\mathbf{c}+\mathbf{1}_i)\,\mathbf{dq}] \equiv q_i\,|\mathbf{cd}\,(\mathbf{q}-\mathbf{1}_i)] + (Q_i - C_i)\,|\mathbf{cdq}] + \frac{1}{2\eta}\,|\mathbf{cd}\,(\mathbf{q}+\mathbf{1}_i)] \tag{4.20}$$

It should be noted that we can obtain the Equation (4.20) from (**cbad**) mirroring the Equation (4.19), so here we are only talking about three symmetrically irreducible recurrence relations in this algorithm. Similar recurrence relations could also be generated for $[\mathbf{a}\,(\mathbf{b}+\mathbf{1}_i)\,\mathbf{p}|$ on Equation (4.23) and $|\mathbf{c}\,(\mathbf{d}+\mathbf{1}_i)\,\mathbf{q}]$ Equation (4.24) by (**badc**) mirroring. We can remove the explicit dependence of P and Q center from the Equation (4.20) and Equation (4.19) by using the Equation (4.8). This way we would obtain the following recurrence relations:

$$[(\mathbf{a}+\mathbf{1}_i)\,\mathbf{bp}| \equiv p_i\,[\mathbf{ab}\,(\mathbf{p}-\mathbf{1}_i)| + (B_i - A_i)\,\frac{2\beta}{2\zeta}[\mathbf{abp}| + \frac{1}{2\zeta}\,[\mathbf{ab}\,(\mathbf{p}+\mathbf{1}_i)| \tag{4.21}$$

$$|(\mathbf{c}+\mathbf{1}_i)\,\mathbf{dq}] \equiv q_i\,|\mathbf{cd}\,(\mathbf{q}-\mathbf{1}_i)] + (D_i - C_i)\,\frac{2\delta}{2\eta}|\mathbf{cdq}] + \frac{1}{2\eta}\,|\mathbf{cd}\,(\mathbf{q}+\mathbf{1}_i)] \tag{4.22}$$

$$[\mathbf{a}\,(\mathbf{b}+\mathbf{1}_i)\,\mathbf{p}| \equiv p_i\,[\mathbf{ab}\,(\mathbf{p}-\mathbf{1}_i)| - (B_i - A_i)\,\frac{2\alpha}{2\zeta}[\mathbf{abp}| + \frac{1}{2\zeta}\,[\mathbf{ab}\,(\mathbf{p}+\mathbf{1}_i)| \tag{4.23}$$

$$|\mathbf{c}\,(\mathbf{d}+\mathbf{1}_i)\,\mathbf{q}] \equiv q_i\,|\mathbf{cd}\,(\mathbf{q}-\mathbf{1}_i)] - (D_i - C_i)\,\frac{2\gamma}{2\eta}|\mathbf{cdq}] + \frac{1}{2\eta}\,|\mathbf{cd}\,(\mathbf{q}+\mathbf{1}_i)] \tag{4.24}$$

$$[\mathbf{p}|\mathbf{q}] \equiv (-1)^q\,[\mathbf{p}+\mathbf{q}] \tag{4.25}$$

$$[\mathbf{r}+\mathbf{1}_i]^{(m)} \equiv R_i\,[\mathbf{r}]^{(m+1)} - (r_i)\,[\mathbf{r}-\mathbf{1}_i]^{(m+1)} \tag{4.26}$$

DOI:10.15774/PPKE.ITK.2014.009

This algorithm can be converted into the MD-PRISM by converting the recurrence relations and contraction steps into a PRISM graph, as depicted on Figure 4.1.

4.2.2 Head-Gordon-Pople (HGP) algorithm

This algorithm uses two vertical recurrence relations in Equations (4.27) and (4.28), these two recurrence relations are special because they do not change the sum of the angular momenta and they also do not use any gaussian exponents. As a result of this, Equations (4.27) and (4.28) are contraction invariant, and we are also free to choose their order. This is very efficient because the reduced form $[\mathbf{m0}|\mathbf{n0}]$ implies a much more simple recurrence relation in Equation (4.29) when converting it to $[\mathbf{00}|\mathbf{00}]$, for in this way, the significant part of the computation is being done outside the contraction.

$$|\mathbf{c}\,(\mathbf{d}+\mathbf{1}_i)) \equiv |(\mathbf{c}+\mathbf{1}_i)\,\mathbf{d}) + (C_i - D_i)\,|\mathbf{cd}) \tag{4.27}$$

$$(\mathbf{a}\,(\mathbf{b}+\mathbf{1}_i)| \equiv ((\mathbf{a}+\mathbf{1}_i)\,\mathbf{b}| + (A_i - B_i)\,(\mathbf{ab}| \tag{4.28}$$

$$\begin{aligned}[\mathbf{m0}|\,(\mathbf{n}+\mathbf{1}_i)\,\mathbf{0}] \equiv \frac{m_i}{2\eta}\,[(\mathbf{m}-\mathbf{1}_i)\,\mathbf{0}|\mathbf{n0}] + \frac{n_i}{2\eta}\,[\mathbf{m0}|\,(\mathbf{n}-\mathbf{1}_i)\,\mathbf{0}] \\ -\frac{2\zeta}{2\eta}\,[(\mathbf{m}+\mathbf{1}_i)\,\mathbf{0}|\mathbf{n0}] - \left[\frac{2\beta}{2\eta}\,(A_i - B_i) + \frac{2\delta}{2\eta}\,(C_i - D_i)\right][\mathbf{m0}|\mathbf{n0}]\end{aligned} \tag{4.29}$$

This algorithm can be converted into the HGP-PRISM by converting the recurrence relations and contraction steps into a PRISM graph, as depicted on Figure 4.2.

4.2.3 Generalized braket representation

Our generalized braket representation is based on the one proposed by Gill et al. in [78].

$$[\mathbf{0}]^{(m)} := [\mathbf{00}|\mathbf{00}]^{(m)} := \left[\begin{array}{ccc|cccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 0 & 0 & 0 & 0 & 0 & 0 & \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right]^{(m)} \tag{4.30}$$

DOI:10.15774/PPKE.ITK.2014.009

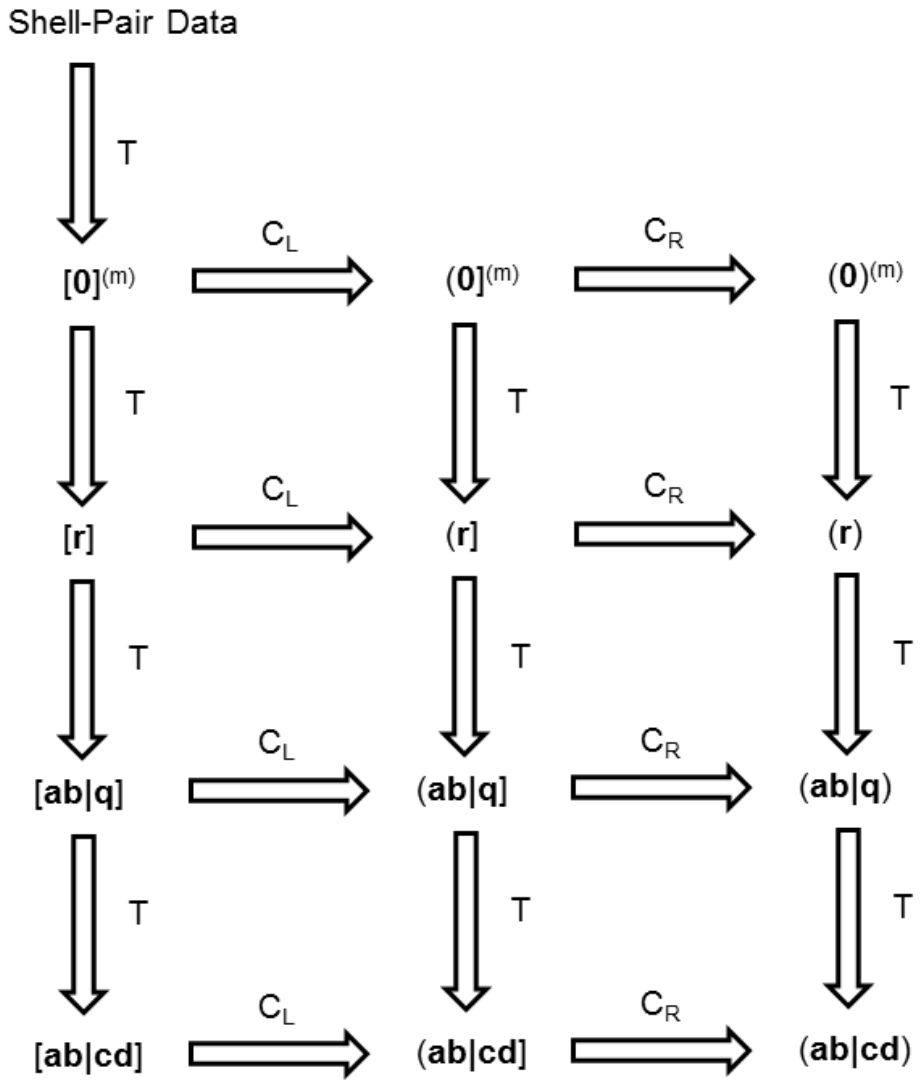


Figure 4.1: MD PRISM algorithm consists of a set of interrelated pathways from shell-pair data to the desired brakets. It consist of the McMurchie-Davidson recurrence relations and contraction steps. Every possible path from the shell-pair data to the $(ab|cd)$ braket format represents a possible solution of the integral problem. Depending on the degree of contractions and the angular moments walking different paths can result in different run-times. The PRISM meta algorithm tries to the find the ideal path.

DOI:10.15774/PPKE.ITK.2014.009

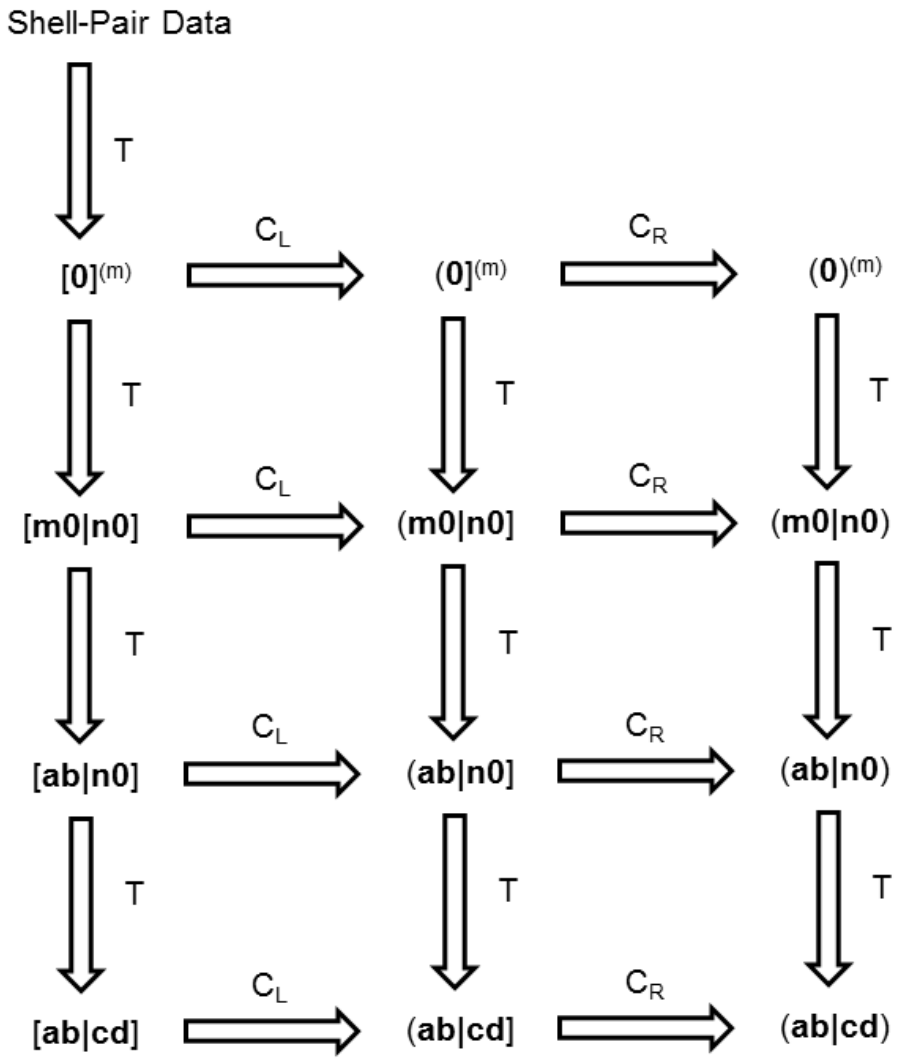


Figure 4.2: HGP PRISM algorithm consists of a set of interrelated pathways from shell-pair data to the desired brakets. It consist of the Head-Gordon-Pople reuccrence relations and contraction steps. Every possible path from the shell-pair data to the $(ab|cd)$ braket format represents a possible solution of the integral problem. Depending on the degree of contractions and the angular moments walking different paths can result in different run-times. The PRISM meta algorithm tries to the find the ideal path.

DOI:10.15774/PPKE.ITK.2014.009

$$\left[\begin{array}{ccc|cccc} \mathbf{a} & \mathbf{b} & \mathbf{p} & \mathbf{c} & \mathbf{d} & \mathbf{q} & \mathbf{r} \\ a' & b' & p' & c' & d' & q' & \\ \mathfrak{a}^* & \mathfrak{b}^* & \mathfrak{c}^* & \mathfrak{e}^* & \mathfrak{f}^* & \mathfrak{g}^* & \mathfrak{r}^* \end{array}\right]^{(m)} := \left[\begin{array}{ccc|cccc} \mathbf{a} & \mathbf{b} & \mathbf{p} & \mathbf{c} & \mathbf{d} & \mathbf{q} & \mathbf{r} \\ 0 & 0 & 0 & 0 & 0 & 0 & \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right]^{(m)} \cdot$$
$$(\mathbf{B}-\mathbf{A})^{\mathfrak{a}^*} \cdot (\mathbf{C}-\mathbf{D})^{\mathfrak{b}^*} \cdot (\mathbf{D}-\mathbf{B})^{\mathfrak{c}^*} \cdot (\mathbf{C}-\mathbf{A})^{\mathfrak{e}^*} \cdot (\mathbf{D}-\mathbf{A})^{\mathfrak{f}^*} \cdot (\mathbf{C}-\mathbf{B})^{\mathfrak{g}^*} \cdot (\mathbf{R})^{\mathfrak{r}^*} \cdot$$
$$\frac{(2\alpha)^{a'}(2\beta)^{b'}}{(2\zeta)^{p'}} \cdot \frac{(2\gamma)^{c'}(2\delta)^{d'}}{(2\eta)^{q'}} \cdot \tag{4.31}$$

The symmetrically irreducible recurrence relations of the McMurchie-Davidson algorithm written in our general braket representation are depicted on Equations (4.32), (4.33) and (4.34). Where Equation (4.32) corresponds to Equation (4.21), Equation (4.33) corresponds to Equation (4.25), and respectively, Equation (4.34) corresponds to Equation (4.26):

$$\left|\begin{array}{ccc} \mathbf{a}+\mathbf{1_i} & \mathbf{b} & \mathbf{p} \\ a' & b' & p' \\ \mathfrak{a}^* & \mathfrak{b}^* & \mathfrak{c}^* \end{array}\right| \equiv \mathbf{p}_i \left|\begin{array}{ccc} \mathbf{a} & \mathbf{b} & \mathbf{p}-\mathbf{1_i} \\ a' & b' & p' \\ \mathfrak{a}^* & \mathfrak{b}^* & \mathfrak{c}^* \end{array}\right| + \left|\begin{array}{ccc} \mathbf{a} & \mathbf{b} & \mathbf{p} \\ a' & b'+1 & p'+1 \\ \mathfrak{a}^*+\mathbf{1_i} & \mathfrak{b}^* & \mathfrak{c}^* \end{array}\right| + \left|\begin{array}{ccc} \mathbf{a} & \mathbf{b} & \mathbf{p}+\mathbf{1_i} \\ a' & b' & p'+1 \\ \mathfrak{a}^* & \mathfrak{b}^* & \mathfrak{c}^* \end{array}\right| \tag{4.32}$$

$$\left[\begin{array}{ccc|cccc} \mathbf{0} & \mathbf{0} & \mathbf{p} & \mathbf{0} & \mathbf{0} & \mathbf{q} & \mathbf{0} \\ a' & b' & p' & c' & d' & q' & \\ \mathfrak{a}^* & \mathfrak{b}^* & \mathfrak{c}^* & \mathfrak{e}^* & \mathfrak{f}^* & \mathfrak{g}^* & \mathfrak{r}^* \end{array}\right]^{(m)} \equiv \left[\begin{array}{ccc|cccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{p}+\mathbf{q} \\ a' & b' & p' & c' & d' & q' & \\ \mathfrak{a}^* & \mathfrak{b}^* & \mathfrak{c}^* & \mathfrak{e}^* & \mathfrak{f}^* & \mathfrak{g}^* & \mathfrak{r}^* \end{array}\right]^{(m)} \tag{4.33}$$

$$\left[\begin{array}{ccc|cccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{r}+\mathbf{1_i} \\ a' & b' & p' & c' & d' & q' & \\ \mathfrak{a}^* & \mathfrak{b}^* & \mathfrak{c}^* & \mathfrak{e}^* & \mathfrak{f}^* & \mathfrak{g}^* & \mathfrak{r}^* \end{array}\right]^{(m)} \equiv$$
$$\left[\begin{array}{ccc|cccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{r} \\ a' & b' & p' & c' & d' & q' & \\ \mathfrak{a}^* & \mathfrak{b}^* & \mathfrak{c}^* & \mathfrak{e}^* & \mathfrak{f}^* & \mathfrak{g}^* & \mathfrak{r}^*+\mathbf{1_i} \end{array}\right]^{(m+1)} - \mathbf{r}_i \left[\begin{array}{ccc|cccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{r}-\mathbf{1_i} \\ a' & b' & p' & c' & d' & q' & \\ \mathfrak{a}^* & \mathfrak{b}^* & \mathfrak{c}^* & \mathfrak{e}^* & \mathfrak{f}^* & \mathfrak{g}^* & \mathfrak{r}^* \end{array}\right]^{(m+1)} \tag{4.34}$$

4.3 BRUSH algorithm for two-electron integrals on GPU

In this Section the branches of the meta algorithm are summarized. I distinguish eight different cases which are pictured in Figure 4.3 in merged form. Some of the

DOI:10.15774/PPKE.ITK.2014.009

paths follow the part of the PRISM algorithm, but they are expanded in compile time. The cases are separated by the number and the place of the contractions in the quartets. For sample, (**cx**|**xx**) symbolizes a quartet where the first function is contracted and the other three functions are not contracted. All the cases are grouped in eight sets based on the symmetry of the integral.

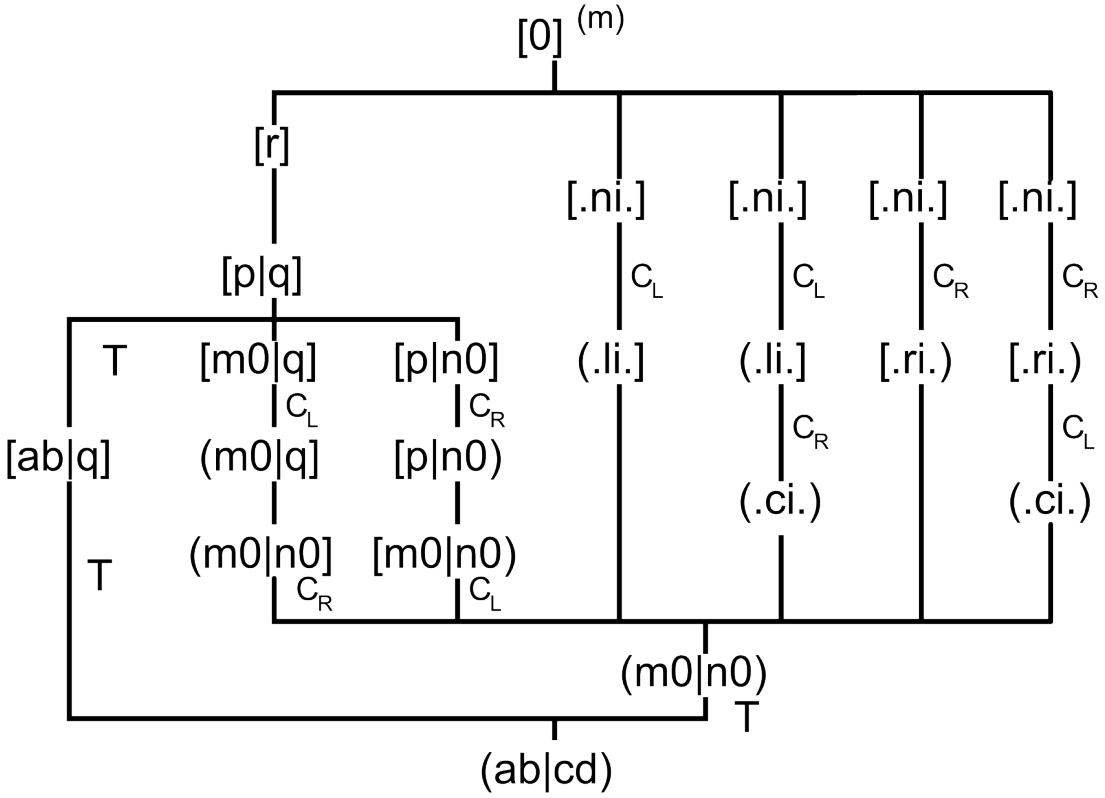


Figure 4.3: The structure of the BRUSH meta algorithm. All possible paths are merged together into a graph. The algorithm includes the MD-PRISM and HGP-PRISM transformation steps along with my symbolic transformation steps. The left C_L and right C_R contraction steps are depicted in the Figure. In the branches for partially uncontracted brakets the contraction step is missing, since it is a trivial identity transformation.

Case 1: set of (**xx**|**xx**) type integrals, where there are no contractions, the algorithm has the following integral rules:

$$(\mathbf{ab}|\mathbf{cd}) = [\mathbf{ab}|\mathbf{cd}] \rightarrow [\mathbf{ab}|\mathbf{q}] \rightarrow [\mathbf{p}|\mathbf{q}] \rightarrow [\mathbf{r}] \rightarrow [\mathbf{0}]^{(m)} \tag{4.35}$$

DOI:10.15774/PPKE.ITK.2014.009

this is similar to the solution of MD-PRISM algorithm in the case of non-contracted integrals [78] where Equations (4.21) (4.22) (4.23) (4.24) (4.25) are used.

Case 2: set of (**cc**|**cc**) type integrals, where all the functions are contracted, the Br_{1a} path of the algorithm has the following integral rules:

$$(\mathbf{ab}|\mathbf{cd}) \rightarrow (\mathbf{m0}|\mathbf{n0}) \rightarrow (\mathbf{.ci.}) \rightarrow (\mathbf{.li.}] \rightarrow [\mathbf{.ni.}] \rightarrow [\mathbf{0}]^{(m)} \tag{4.36}$$

and Br_{1b} has the integral rules:

$$(\mathbf{ab}|\mathbf{cd}) \rightarrow (\mathbf{m0}|\mathbf{n0}) \rightarrow (\mathbf{.ci.}) \rightarrow [\mathbf{.ri.}) \rightarrow [\mathbf{.ni.}] \rightarrow [\mathbf{0}]^{(m)} \tag{4.37}$$

where (**m0**|**n0**) is solved by following the steps of the HGP-PRISM algorithm (Obara-Saika rule) assuming that there are no contractions according to Equation (4.29). After we have obtained this result we multiply out the (left or right) contraction invariant parts and insert a contraction (left or right) and next we multiply out the (right or left) contraction invariant parts from the remaining part and insert a contraction (right or left). The order of the sides is based on the size of the contraction. This is symbolized in Figure 4.3. with two branches, where [**.ni.**] means non invariant part, (**.li.**] means the left invariant part, [**.ri.**) means the right invariant part, and (**.ci.**) is the invariant part. Where brackets change from "(" to "[" the contraction steps are inserted (C_L and C_R). The special distinction from the normal bra-ket notation is important for these transformation steps (**.ci.**)(**.li.**][**.ri.**)[**.ni.**] because they have no physical interpretation, they are purely symbolic transformations on the generalized braket representation.

Case 3: set of (**cc**|**xx**) type integrals, where left functions are contracted, the algorithm computes two different branches (Br_{3a} and Br_{3b}) and chooses the better one. Br_{3a} has the following integral rules:

$$(\mathbf{ab}|\mathbf{cd}) = (\mathbf{ab}|\mathbf{cd}] \rightarrow (\mathbf{m0}|\mathbf{n0}] \rightarrow (\mathbf{m0}|\mathbf{q}] \rightarrow [\mathbf{m0}|\mathbf{q}] \rightarrow [\mathbf{p}|\mathbf{q}] \rightarrow [\mathbf{r}] \rightarrow [\mathbf{0}]^{(m)} \tag{4.38}$$

Br_{3b} has the following integral rules:

$$(\mathbf{ab}|\mathbf{cd}) = (\mathbf{ab}|\mathbf{cd}] \rightarrow (\mathbf{m0}|\mathbf{n0}] \rightarrow (\mathbf{.ci.}] = (\mathbf{.li.}] \rightarrow [\mathbf{.ni.}] \rightarrow [\mathbf{0}]^{(m)} \tag{4.39}$$

DOI:10.15774/PPKE.ITK.2014.009

where (**m0**|**n0**) is solved by following the steps of the HGP-PRISM algorithm (Obara-Saika rule) assuming that there are no contractions. After we have obtained this result we multiply out the left contraction invariant parts and insert a left contraction in a similar way as it is described in Case 2.

Case 4: set of (**xx**|**cc**) type integrals, where right functions are contracted, the algorithm computes two different branches (Br_{4a} and Br_{4b}) and chooses the better one. The Br_{4a} and Br_{4b} paths are (**cd**|**ab**) mirrored versions of the Br_{3a} and Br_{3b} respectively. They also have the same heuristics.

Case 5: set of (**cx**|**cx**) or (**xc**|**xc**) or (**xc**|**cx**) or (**cx**|**xc**) type integrals, where one of the functions from both sides is contracted, the algorithm computes two different branches (Br_{5a} and Br_{5b}) and chooses the better one. Br_{5a} has the following integral rules:

$$(\mathbf{ab}|\mathbf{cd}) \rightarrow (\mathbf{m0}|\mathbf{n0}) \rightarrow (\mathbf{m0}|\mathbf{n0}] \rightarrow (\mathbf{m0}|\mathbf{q}] \rightarrow [\mathbf{m0}|\mathbf{q}] \rightarrow [\mathbf{p}|\mathbf{q}] \rightarrow [\mathbf{r}] \rightarrow [\mathbf{0}]^{(m)} \tag{4.40}$$

Br_{5b} has the following integral rules:

$$(\mathbf{ab}|\mathbf{cd}) \rightarrow (\mathbf{m0}|\mathbf{n0}) \rightarrow [\mathbf{m0}|\mathbf{n0}) \rightarrow [\mathbf{p}|\mathbf{n0}) \rightarrow [\mathbf{p}|\mathbf{n0}] \rightarrow [\mathbf{p}|\mathbf{q}] \rightarrow [\mathbf{r}] \rightarrow [\mathbf{0}]^{(m)} \tag{4.41}$$

Case 6: set of (**cx**|**xx**) or (**xc**|**xx**) type integrals, where one of the functions from left side is contracted, the algorithm computes two different branches (Br_{6a} and Br_{6b}) and chooses the better one. Br_{6a} has the following integral rules:

$$(\mathbf{ab}|\mathbf{cd}) \rightarrow (\mathbf{m0}|\mathbf{n0}) \rightarrow [\mathbf{m0}|\mathbf{n0}) \rightarrow [\mathbf{p}|\mathbf{n0}) \rightarrow [\mathbf{p}|\mathbf{n0}] \rightarrow [\mathbf{p}|\mathbf{q}] \rightarrow [\mathbf{r}] \rightarrow [\mathbf{0}]^{(m)} \tag{4.42}$$

Br_{6b} has the following integral rules:

$$(\mathbf{ab}|\mathbf{cd}) \rightarrow (\mathbf{m0}|\mathbf{n0}) \rightarrow (\mathbf{.ci.}) \rightarrow [\mathbf{.ri.}) \rightarrow [\mathbf{.ni.}] \rightarrow [\mathbf{0}]^{(m)} \tag{4.43}$$

where (**m0**|**n0**) is solved by following the steps of the HGP-PRISM algorithm (Obara-Saika rule) assuming that there are no contractions. After we have obtained this result we multiply out the left contraction invariant parts and insert a left contraction in a similar way as it is described in Case 2.

Case 7: set of (**xx**|**cx**) or (**xx**|**xc**) type integrals, where one of the functions from right side is contracted, the algorithm computes two different branches (Br_{7a}

DOI:10.15774/PPKE.ITK.2014.009

and Br_{7b}) and chooses the better one. The Br_{7a} and Br_{7b} paths are (**cd**|**ab**) mirrored versions of the Br_{6a} and Br_{6b} respectively. They also have the same heuristics.

Case 8: set of (**cc**|**xc**) or (**cc**|**cx**) or (**xc**|**cc**) or (**cx**|**cc**) type of integrals, based on the high number of contraction the algorithm is practically the same as at Case 2.

We are more efficient than the unrolled PRISM, because we exploit the opportunities arisen from unrolling. These extra solution paths are closely related to the naming, because my recursion rules look more chaotic, so instead they look more like a brush than a prism, hence the naming.

4.4 Measurements

I have applied two integrator implementations as our CPU speed references, NWchem and Libint, and MRCCsoftwares were tested for validating the precision. I have used a relatively new CPU, the Intel i7-3820 3.6GHz and a few years older AMD Athlon II X4, as our speed references, where I have completed my measurements on a single core. We can assume that these problems scale linearly with the number of cores in CPUs, but are not affected by HyperThreading, because the floating point calculations benefit very little from HyperThreading in this particular case, and these calculations are almost exclusively done on the floating point units.

There were four compiler back-ends developed for my GPU integrator: OpenCL, CUDA-C, CUDA-PTX, NVIDIA-assembly called BRUSH-OCL, BRUSH-C, BRUSH-PTX, BRUSH-ASM respectively. For higher than ”p” angular moments or big contracted basis functions neither AMD nor NVIDIA software could compile OpenCL or CUDA-C. However it was only compiling for the STO-3G basis set. The CUDA-PTX generated codes fared a little better, but caused problems for compiling the whole cc-pVdz basis set with ”d” angular moments. Only the BRUSH-ASM back-end where I output NVIDIA assembly code and assemble it into a CUDA binary was able to compile the whole cc-pVdz basis set. I have chosen to introduce only the STO-3G basis-set measurements because the stable code is yet to be finished for ”d” angular moments and above.

DOI:10.15774/PPKE.ITK.2014.009

It is important to note that our OpenCL back-end, the BRUSH-OCL, was largely unoptimized, because the vendor supplied compiler was only able to compile it by cutting the optimization time. Consequently the BRUSH-OCL measurements were significantly suboptimal, but it still performed well due to the high computation power of the hardware. On the AMD HD7970 GPU hardware we were forced to use only the BRUSH-OCL back-end because we have yet to complete our AMD GPU machine code compiler.

Above a certain complexity (angular moment and contraction size, $(dd|dd)$) the run-time and the memory usage of the NVIDIA and AMD compilers are unpractical. The cause of this behaviour is that, algorithms up to the complexity $\mathcal{O}(n^3)$ are favoured in compiler because of their efficiency, where n is the number of instructions in the function body. This is true for middle-end optimizations, and the register allocator too. The code sizes for BRUSH on GPUs can easily grow more than millions of instructions, so standard compilers are unable to process them. The code transformations and optimizations used for compiling the output of BRUSH in the BRUSH-ASM, are limited in time and space complexity to $\mathcal{O}(n \cdot log(n))$.

I have done most of the GPU measurements in single precision, and one in double precision. It was throughly investigated [75, 80, 96, 61] that using a single precision integrator is appropriate even for large molecules to converge the SCF procedure to near optimum, and later into the SCF iterations the double/mixed precision could be enabled, only using it for a few iterations to achieve chemical precision.

It should be noted that the speed difference between single and double precision is only $\times 5.5$, while the the theoretical peak difference of the hardware is $\times 4$. This is a good result, because these computations tend to suffer from register pressure, which doubled as our register sizes doubled. However, neither of these two potential bottlenecks multiplied to $\times 8$, bur rather, only multiplied to $\times 5.5$.

The performance of Libint integrator, using the vertical recurrence relations on Equations (4.27), (4.28) and (4.29), was found to be similar to NWchem in non-vectorized mode, so it was not included it in the table of measurements.

I have done my measurements on several different GPGPU graphics cards, so the trend of improvement can also be seen. I have depicted the used software or

DOI:10.15774/PPKE.ITK.2014.009

software module in case of my BRUSH algorithm, the hardware, the integration time, and the relative speedup to the AMD Athlon II X4 processor. I have also measured the precision of the ground state energy computed from these values by an SCF algorithm. The relative precision is practically the ratio of the ground state energy and the energy error. The absolute precision is the magnitude of the measured energy error compared to the reference sources: MRCC, Gaussian, Molpro, NWChem.

I have summarized my measurements in Table 1, where each column contains a measurement on a software and hardware environment across different molecules, and each row contains the results of a single molecule measurements across different environments. In the first row I depicted the most important information about the environment, hardware details like the processor type and frequency and if we have used only a single core for the measurement, or software details like the name of software module, of the precision of the number representation. In the first column we have the name of the molecule used in the computations, or the chemical formula for long alkanes. We also have labels for the absolute precision and relative precision. Where absolute precision means difference of the computed and reference ground state energy values, the relative precision is similar is the scaled version of the absolute precision with the energy relative precision $:= \frac{\text{Computed}-\text{Reference}}{\text{Reference}}$. The first sub-row of a measurement row depicts the integration time, where all integrals passing the Swartz screening were computed. The second sub-row depicts the speedup relative of the single core of Athlon II X4 635 desktop CPU.

DOI:10.15774/PPKE.ITK.2014.009

Hardware	Athlon II X4 635	i7-3820	C2050	C2050	HD7970	GTX580	GTX780
Software	NWchem 6.3	NWchem 6.3	BRUSH-PTX	BRUSH-ASM	BRUSH-OCL	BRUSH-ASM	BRUSH-PTX
Precision	double	double	double	single	single	single	single
Frequency	1 core 2.9GHz	1 core 3.6GHz	1.15GHz	1.15GHz	1.05GHz	1.54GHz	1.72GHz
benzene	<500ms	<200ms	49ms	8.9ms	42ms	6.7ms	5.1ms
	×1	×2.5	×10	×56	×11.9	×72.4	×98
abs. precision	10^{-6} Ha	10^{-6} Ha	10^{-6} Ha	10^{-5} Ha	10^{-5} Ha	10^{-5} Ha	10^{-5} Ha
rel. precision	10^{-9} Ha	10^{-9} Ha	10^{-10} Ha	10^{-8} Ha	10^{-8} Ha	10^{-8} Ha	10^{-8} Ha
superbenzene	22s	11s	746ms	167ms	240ms	111ms	77ms
coronene	×1	×2	×29	×132	×92	×198	×286
abs. precision	10^{-6} Ha	10^{-6} Ha	10^{-6} Ha	10^{-3} Ha	10^{-4} Ha	10^{-3} Ha	10^{-3} Ha
rel. precision	10^{-9} Ha	10^{-9} Ha	10^{-9} Ha	10^{-6} Ha	10^{-7} Ha	10^{-6} Ha	10^{-6} Ha
sucrose	30s	16s	643ms	115ms	470ms	80ms	67ms
	×1	×1.9	×46	×261	×64	×375	×447
abs. precision	10^{-5} Ha	10^{-5} Ha	10^{-6} Ha	10^{-4} Ha	10^{-3} Ha	10^{-4} Ha	10^{-4} Ha
rel. precision	10^{-9} Ha	10^{-9} Ha	10^{-10} Ha	10^{-7} Ha	10^{-6} Ha	10^{-7} Ha	10^{-7} Ha
$C_{30}H_{62}$	37s	24s	572ms	105ms	175ms	70ms	52ms
	×1	×1.5	×64	×352	×211	×529	×711
abs. precision	10^{-5} Ha	10^{-5} Ha	10^{-6} Ha	10^{-4} Ha	10^{-3} Ha	10^{-4} Ha	10^{-4} Ha
rel. precision	10^{-9} Ha	10^{-9} Ha	10^{-10} Ha	10^{-7} Ha	10^{-6} Ha	10^{-7} Ha	10^{-7} Ha
$C_{60}H_{122}$	173s	95s	2506ms	467ms	505ms	308ms	232ms
	×1	×1.8	×69	×370	×343	×562	×746
abs. precision	10^{-5} Ha	10^{-5} Ha	10^{-6} Ha	10^{-3} Ha	10^{-3} Ha	10^{-3} Ha	10^{-3} Ha
rel. precision	10^{-9} Ha	10^{-9} Ha	10^{-10} Ha	10^{-7} Ha	10^{-6} Ha	10^{-7} Ha	10^{-7} Ha
$C_{100}H_{202}$	493s	264s	6979ms	1332ms	1100ms	870ms	655ms
	×1	×1.9	×70	×370	×448	×567	×752
abs. precision	10^{-5} Ha	10^{-5} Ha	10^{-5} Ha	10^{-3} Ha	10^{-2} Ha	10^{-3} Ha	10^{-3} Ha
rel. precision	10^{-9} Ha	10^{-9} Ha	10^{-9} Ha	10^{-7} Ha	10^{-6} Ha	10^{-7} Ha	10^{-7} Ha

The NWchem software was chosen because of simplicity and because we can easily measure the speed of its integrator, and it also provides information about how many integrals passed the Swartz screening threshold. While the integral screening is out of the scope of this dissertation, I have to mention that our screening (Cauchy-Schwarz screening) was more conservative than the NWchem, as a consequence, we have computed 20% more integrals on average, so the actual speedup of our integrator is theoretically bigger than depicted.

Most modern computational chemistry programs utilize also advanced algorithms to speed up the evaluation of long-range interactions. The most well known of these is the Continuous Fast Multipole Method [84], however many other techniques also exist, which makes it difficult to compare directly the efficiency of the BRUSH algorithm with these. We are also working on the GPU implementation of these methods.

DOI:10.15774/PPKE.ITK.2014.009

4.5 Conclusions

I have presented the PRISM-like meta algorithm BRUSH as a better choice for GPU based two-electron integrators, given the set of constraints, because the BRUSH algorithm is a superset of the HGP/MD-PRISM algorithms, but I have also added GPU friendly optimizations, and more general integrals paths.

I have measured that the speedup of this calculation on GPUs, and for single precision I have obtained 750× speedup compared to a single core of Athlon II X4 635 desktop processor which closely agrees with the best-case GPU versus CPU speedup, which is in the range of ×50 - ×100, for all four cores of the mentioned Athlon and Intel i7 processors.

It is easy to see why my approach to unroll the recursion and compute the Gaussian exponents at compile time is advantageous on GPU architectures since it allows us to stay mostly in register memory which is by far the fastest on the GPU, on the other hand the huge amount of executable code is not much a hindering factor because of the SIMD nature of this architecture, unlike for CPUs. However it is very difficult to determine why the BRUSH meta algorithm is faster because I have obtained my algorithm by through benchmarking of various combinations of recurrence relations. In case of the compiler optimizations, as well as using unrolling and Gaussian exponent (contraction) propagation, the computation time no longer simply depends on theoretical Flops and Mops counts.

DOI:10.15774/PPKE.ITK.2014.009

Chapter 5

Conclusions

The polyhedron based algorithm optimization formalism and method was presented in Chapter 2, which gave a theoretical basis to my work using GPUs, and other many core architectures. This work centered on the formalism and dealing with the data-flow dependencies, which are critical in loop optimizations including parallelization. Additionally, I have also advanced this topic by including dynamic control-flow structures into the previously static theory. This way we can handle much more practical programming situations.

Based on my theoretical results I proposed an architecture named RACER in Chapter 3, which eliminates many often occurring bottlenecks of parallelization. Following the theory, described previously in Chapter 2, I came to conclusion that memory operations, like sorting and searches, should be outsourced to the memory. This approach allows the significant increase of performance of the memory operations while simultaneously making the processing elements of the RACER processor to be much more simple and efficient. As a result the RACER architecture can implement more parallelism, allowing the efficient implementation of a wider range of algorithms than in other GPU or CPU architectures. An algorithm example is described to explain the process of the program running on the architecture.

Chapter 4 presents a new algorithmic approach developed to evaluate two-electron repulsion integrals based on contracted Gaussian basis functions in a parallel way. This approach utilizes my earlier theoretical results in a practical way, and experience from developing low level compiler systems. I show in bench-

DOI:10.15774/PPKE.ITK.2014.009

marks that significant speed improvements can be achieved by my optimization approach.

DOI:10.15774/PPKE.ITK.2014.009

Summary

5.1 Methods used in the experiments

In the course of my work, instruments of numerous disciplines were applied. One of the most important of these is the theory of automatic parallelization. In case of automatic parallelization loop structures, which can be reformulated in a parallel way, are identified in the implemented algorithm. The loop structures are usually described by polyhedral representation, where the static loop structure is represented in a multi-dimensional discrete space and converted to the desired shape by affine geometric transformations. I supplemented this theory to work the dynamic control structures too. For the representation I used data-flow and control-flow description of the program, which enable the efficient automatic management and optimization of the code. I have learned the internal operation of the two currently most popular open source compilers (GCC, LLVM) and the optimization algorithms they use.

It was necessary to study in detail the following most widely used general-purpose programmable GPU architectures:

- NVIDIA GeForce8
- NVIDIA Fermi
- NVIDIA Kepler
- AMD(ATI) R800(Evergreen)
- AMD(ATI) R900(NI Cayman)

DOI:10.15774/PPKE.ITK.2014.009

- AMD(ATI) R1000(Southern Islands GCN)

For the AMD architectures, the manufacturer provided to me the documentation of the machine code, the detailed structure of the architectures and also the general-purpose hardware-level programming. In the case of NVIDIA, the architecture dependent information was collected by disassembly and careful measurements.

While designing the RACER architecture, I used general engineering design methods of digital processors, such as pipeline design, digital synthesis, H-fractal clock routing method, resonant network and asynchronous network. During the construction, I tried to use as much existing IP-core modules as possible. I combined my experience of implementing complex algorithms on GPU (e.g. video compression, sparse-matrix algebra) with the local data processing methods of array processors and systolic arrays. I got acquainted with the operation of the processor memory communication and the limitation of memory circuits. The elimination of these limitations plays an important role in the RACER architecture. I have learned efficient simulation methods of digital circuits and their high-level design in VHDL and Verilog languages.

My results are in use in a GPU optimized quantum chemistry software computing the two electron integrals. I have implemented a specialized compiler software for achieve the massive parallelism and GPU optimized program code. This compiler is able to expand the integrals symbolically, in this reduction the following algorithms were taken as a basis:

- Boys
- Pople-Hehre
- Obara-Saika-Schlegel
- Head-Gordon-Pople
- McMurchie-Davidson

DOI:10.15774/PPKE.ITK.2014.009

- MD-PRISM, HGP-PRISM

While understanding these algorithms, I became acquainted with the details of mathematical methods of numerical quantum chemistry, in particular with the computation of two electron integrals of Gaussian basis and the mathematical method of the general bra-ket. I have learned the methods which use the integrals for calculating the electrostatic potentials:

- Self Consistent Field : SCF-HF
- Density Function Theory : DFT-KS
- Moller-Plesset Perturbation Theory : MPPT
- Configuration Integration : CI
- Coupled Cluster computation : CC

5.2 New scientific results

1. Thesis: *I showed that in case of programming many-core architectures, besides classical static polyhedral representation, dynamic polyhedral loops and dynamic control structures can be represented by polyhedrons. In the case of dynamical polyhedrons, I showed and gave a formalism how to manage memory access patterns. I defined those algorithm classes, which can be managed efficiently with my proposed methods. [7, 8, 9]*

I proposed a new mathematical formalism for the high-level manipulation of the dynamical control structures of the programs. I reduced the dynamical control structures to infinite static structures with specific dynamical dependences. The infinite limits are necessary, because the

DOI:10.15774/PPKE.ITK.2014.009

dynamic limits are parametric in compilation time therefore they can be overestimated by infinity. Dependencies can be dynamic, which makes necessary to execute the parallelization in runtime. Thus, in this theory, the scheduling is the part of program execution, and this process determines that which computations where and when are executed. The theory is demonstrated by the parts of my H.264 video encoder implementation.

2. Thesis: *I designed a new data stream based parallel computing architecture (RACER), in which the tasks are distributed between the memory and processing units more efficiently than in previous architectures. For this achievement, the parallelism is extended to the memory as well. [14]*

I designed the modules of RACER data stream driven computational architecture. Both the program and data streams pass through the array processor. The program stream forms the appropriate structure which processes the following data stream. The control of the data stream processing can be dynamic too including branches, loops, merges and forks. The connected memory system is also a very important part of the architecture, which contrary to the conventional memory, contains computing elements too, in particular the comparison arithmetic units. Thus, appropriate algorithm dependent ordering of the data can be achieved, which provides continuous data feed of the array processor.

2.1. I showed that due to the simplicity and locality of the control electronics and wiring, in case of the realization of VLSI, the 57-72 percent of the chip area is used by arithmetic processing units. Implementing RACER architecture, one of

DOI:10.15774/PPKE.ITK.2014.009

the highest arithmetic density could be reached compared to available GPU architectures.

I have chosen GRFPU-1 IP core from Aeroflex Gaisler for processing elements. In my estimations I have used 100K gates in one core for 65nm and 90nm technologies. I have estimated the number of gates of the routing elements also, which covers about 20 percent of the surface. From these values I estimated the final chip sizes and the power consumptions with Cadence InCyte Chip Estimator. I have compared GPU peak performances to the estimated RACER peak performances in order to highlight the possible performance gains coming from the higher number of computing cores. By estimation the RACER architecture's surface covered by computing cores is between 57 and 72 percent of the chip area.

2.2. I showed that Mandelbrot and Conway's Game of Life algorithms can be implemented on the RACER architecture, while RACER remains a general architecture. With the implemented applications I demonstrated the functionality of the architecture and proved that the architecture is Turing complete.

I designed an possible hardware implementation and I proved the viability of the RACER architecture in simulations. I applied low level simulations where the input is the graph representation of the control and data flows. Thus, the proper functioning of the algorithms can be verified on the proposed RACER architecture.

3. Thesis: *I utilized my techniques for accelerating the computation of two-electron integrals in quantum chemistry algorithms in particular to the compatibility of the single-instruction-multiply-data (SIMD)*

DOI:10.15774/PPKE.ITK.2014.009

architecture. I designed a meta algorithm (BRUSH) for GPUs, which assigns the optimal computational path for each Gaussian two electron integral. I showed that in the case of special contractions, the constant substitution and propagation is efficient on these architectures. [12, 1]

I designed and implemented a specialized compiler for computing two-electron integrals of quantum chemistry methods. This compiler allows the efficient exploitation of parallel SIMD architectures. In quantum chemistry, the most important numerical problem is the calculation of two-electron integrals. The input of my compiler is the actual integral problem, which is unfolded in compiling time contrary to the previous methods. All the dynamic control operations are executed during compilation. The optimal computational paths are calculated and chosen beforehand. The hardware specific machine code is generated from the received computational graph which contains a huge number of arithmetic operations. While I designed this transformation I paid special attention to the exploitation of the properties of the architecture. For example, the usage of multi-level memory structure to store temporary values, or the optimization of parallel processing of the SIMD cores.

5.3 Examples for application

My work and its theoretical results were motivated by practical utilization. The presented algorithms provide solutions for problems in real application domains.

The results of the first thesis group assist compilation of algorithms on many-core architectures (GPU, FPGA).

My second thesis group presents an architecture which has excellent computational performance in many different fields. These applications including but not limited to: 3D graphics rendering, raytrac-

DOI:10.15774/PPKE.ITK.2014.009

ing, computation on unstructured grid, computer games, dealing with large databases and all problems which can be solved efficiently on GPU.

In the third thesis group, an algorithm was presented which can be used for general purpose applications. The GPU acceleration of quantum chemical calculations can assist the synthetic molecule design by significantly reducing the running time.

DOI:10.15774/PPKE.ITK.2014.009

References

The author's journal publications

[1] **A. Rák** and G. Cserey, "The BRUSH algorithm for two-electron integrals on GPU," *MATCH Communications in Mathematical and in Computer Chemistry*, 2014. submitted. 3

[2] **A. Rák** and G. Cserey, "Macromodeling of the memristor in SPICE," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 4, pp. 632–636, 2010.

[3] **A. Rák**, G. Gandhi, and G. Cserey, "Chua's circuit topology evolution using genetic algorithm," *International Journal of Bifurcation and Chaos*, vol. 20, no. 3, pp. 687–696, 2010.

[4] G. B. Soós, **A. Rák**, J. Veres, and G. Cserey, "GPU boosted CNN simulator library for graphical flow based programmability," *EURASIP Journal on Advances in Signal Processing*, 2009. Article ID 930619, 11 pages doi:10.1155/2009/930619.

[5] **A. Rák**, G. B. Soós, and G. Cserey, "Stochastic bitstream based CNN and its implementation on FPGA," *International Journal of Circuit Theory and Applications*, vol. 37, no. 4, pp. 587–612, 2009.

DOI:10.15774/PPKE.ITK.2014.009

The author's international conference publications

[6] G. Cserey, **A. Rák**, B. Jákli, and T. Prodromakis, "Cellular neural networks with memristive cell devices," in *Proceedings of 17th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2010*, (Athens, Greece), pp. 938–941, Dec. 2010.

[7] **A. Rák**, G. Feldhoffer, G. B. Soós, and G. Cserey, "Standard C++ Compiling to GPU with Lambda Functions," in *Proceedings of 2010 International Symposium on Nonlinear Theory and its Applications (NOLTA 2010)*, (Krakow, Poland), 2010. 1

[8] **A. Rák**, G. Feldhoffer, G. B. Soós, and G. Cserey, "Standard c++ compiling to GPU," in *3rd HUNGARIAN-SINGAPOREAN WORKSHOP on SYSTEMS BIOLOGY and COMMUNICATION SYSTEMS*, (Budapest, Hungary), 2010. 1

[9] **A. Rák**, G. Feldhoffer, G. B. Soós, and G. Cserey, "CPU-GPU hybrid compiling for general purpose: Case studies," in *Proceedings of 12th International Workshop on Cellular Neural Networks and their Applications, CNNA 2010*, (Berkeley, USA), Feb. 2010. 2.1, 1

[10] G. J. Tornai, G. Cserey, and **A. Rák**, "Spatial-temporal level set algorithms on CNN-UM," in *Proceedings of 2008 International Symposium on Nonlinear Theory and its Applications, NOLTA 2008*, (Budapest, Hungary), pp. 696–699, 2008.

[11] G. B. Soós, **A. Rák**, J. Veres, and G. Cserey, "GPU powered CNN simulator (SIMCNN) with graphical flow based programmability," in *Proceedings of 11th International Workshop on Cellular Neural Networks and their Applications, CNNA 2008*, (Santiago de Compostela, Spain), pp. 163–168, 2008.

DOI:10.15774/PPKE.ITK.2014.009

The author's other publications

[12] **A. Rák**, and Feldhoffer, G., and Soós, G.B. and Höltzl, T., and Oroszi, B. and Cserey, György, "Eljárás és rendszer integrál kiszámításának párhuzamos architektúra szálára való leképezésére." Hungarian and PCT patent, 2012. 2013. 3

[13] G. Cserey and **A. Rák**, "High accuracy time-to-digital converter on FPGA." Hungarian patent, 2009.

[14] **A. Rák** and G. Cserey, "Számítógépes architektúra és feldolgozási eljárás." Hungarian patent (beadott PCT), 2012. 2

[15] **A. Rák**, G. Cserey, and B. Jákli, "Eszköz és eljárás mért jel időbeliségének meghatározására." PCT patent, 2013.

DOI:10.15774/PPKE.ITK.2014.009

Publications connected to the dissertation

[16] L. Dagum, R. Menon, and S. Inc, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998. 2.1

[17] J. Reinders, "Intel threading building blocks," 2007. 2.1

[18] M. Wolfe, "Implementing the PGI Accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 43–50, ACM, 2010. 2.1

[19] P. Becker, "Working draft, standard for programming language C++," *ISO/IEC, Tech. Rep*, vol. 2798, 2009.

[20] AccelerEyes, "Jacket: a GPU engine for MATLAB," 2009. 2.1

[21] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Grösslinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011. 7

[22] D. Novillo, "Gcc-an architectural overview, current status, and future directions," in *Linux Symposium*, vol. 2, pp. 185–200, Citeseer, 2006. 2.1.2

[23] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 75–86, IEEE, 2004. 2.1.2

[24] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, (Juan-les-Pins, France), pp. 7–16, September 2004. 2.2.3

[25] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. A. Silber, and N. Vasilache, "GRAPHITE: Loop optimizations based on the polyhedral model for GCC," in *Proc. of the 4th GCC Developper's Summit*, pp. 179–198, June 2006. 7

DOI:10.15774/PPKE.ITK.2014.009

[26] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Ieee Micro*, vol. 28, no. 2, pp. 39–55, 2008. 2.5

[27] A. Corp., "White paper - amd graphics cores next (GCN) architecture," 2012. 2.5, 2.5.4

[28] R. Hochberg, "Matrix multiplication with cuda - a basic introduction to the cuda programming model," 2012. 2.5.7

[29] E. Corporation, "1108 user's guide (manual)," *Envos Corporation*, p. 1, 1988. 3.1.1

[30] S. A. Dyer and B. K. Harms, "Digital signal processing," vol. 37 of *Advances in Computers*, pp. 59 – 117, Elsevier, 1993. 3.1.1

[31] B. G. Lipták, *Instrument Engineers' Handbook, Volume Two: Process Control and Optimization*, vol. 2. CRC press, 2005. 3.1.1

[32] J. Owens, "Gpu architecture overview," in *ACM SIGGRAPH*, vol. 1, pp. 5–9, 2007. 3.1.1

[33] A. Al Maashri, G. Sun, X. Dong, V. Narayanan, and Y. Xie, "3d gpu architecture using cache stacking: Performance, cost, power and thermal analysis," in *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pp. 254–259, IEEE, 2009. 3.1.1

[34] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi gf100 gpu architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011. 2.5, 2.5.4, 2.5.7, 3.1.1

[35] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *Micro, IEEE*, vol. 26, no. 2, pp. 10–24, 2006. 3.1.1

[36] I. Kuon, R. Tessier, and J. Rose, "Fpga architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2008. 3.1.1

DOI:10.15774/PPKE.ITK.2014.009

[37] R. Wiśniewski, *Synthesis of compositional microprogram control units for programmable devices.* University of Zielona Góra, 2009. 3.1.1

[38] K. Atkinson, R. Bell, F. Ng, L. Nguyen, D. Phil, and D. Trawick, "Field programmable semiconductor object array integrated circuit," Dec. 5 2006. US Patent App. 11/567,146. 3.1.1

[39] H. Kung, *Systolic array.* John Wiley and Sons Ltd., 2003. 3.1.1

[40] L. O. Chua and T. Roska, "The cnn paradigm," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 40, no. 3, pp. 147–156, 1993. 3.1.1

[41] T. Roska and L. O. Chua, "The cnn universal machine: an analogic array computer," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 40, no. 3, pp. 163–173, 1993. 3.1.1

[42] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, "Hybrid dataflow/von-neumann architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1489–1509, 2013. 3.1.1

[43] D. E. Culler, "Dataflow architectures," *Annual review of computer science*, vol. 1, no. 1, pp. 225–253, 1986. 3.1.1

[44] A. L. Davis, "The architecture and system method of ddm1: A recursively structured data driven machine," in *Proceedings of the 5th annual symposium on Computer architecture*, pp. 210–215, ACM, 1978. 3.1.1

[45] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic dataflow processor," in *ACM SIGARCH Computer Architecture News*, vol. 3, pp. 126–132, ACM, 1975. 3.1.1

[46] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985. 3.1.1

DOI:10.15774/PPKE.ITK.2014.009

[47] N. Ito, M. Sato, E. Kuno, and K. Rokusawa, *The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D*, vol. 14. IEEE Computer Society Press, 1986. 3.1.1

[48] M. Kishi, H. Yasuhara, and Y. Kawamura, "Dddp-a distributed data driven processor," in *ACM SIGARCH Computer Architecture News*, vol. 11, pp. 236–242, ACM, 1983. 3.1.1

[49] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," in *ACM SIGARCH Computer Architecture News*, vol. 18, pp. 82–91, ACM, 1990. 3.1.1

[50] A. Plas, D. Comte, O. Gelly, and J. Syre, "Lau system architecture: A parallel data driven processor based on single assignment," in *Proceedings of the International Conference on Parallel Processing*, pp. 293–302, 1976. 3.1.1

[51] R. Vedder and D. Finn, "The hughes data flow multiprocessor: Architecture for efficient signal and data processing," in *ACM SIGARCH Computer Architecture News*, vol. 13, pp. 324–332, IEEE Computer Society Press, 1985. 3.1.1

[52] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. M. Caamano, "Dynamic and speculative polyhedral parallelization using compiler-generated skeletons," *International Journal of Parallel Programming*, vol. 42, no. 4, pp. 529–545, 2014. 2.1

[53] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 101–113, 2008. 2.1

[54] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "Posh: a tls compiler that exploits program structure," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 158–167, ACM, 2006. 2.1

DOI:10.15774/PPKE.ITK.2014.009

[55] C. Li, F. Gava, and G. Hains, "Implementation of data-parallel skeletons: a case study using a coarse-grained hierarchical model," in *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, pp. 26–33, IEEE, 2012. 2.1

[56] P. Athanas and R. A. Bittner Jr, "Worm-hole run-time reconfigurable processor field programmable gate array (fpga)," Oct. 27 1998. US Patent 5,828,858. 3.6

[57] A. Agarwal and D. Wentzlaff, "Managing data provided to switches in a parallel processing environment," Feb. 28 2012. US Patent 8,127,111. 3.6

[58] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013. 4.1

[59] J. Nickolls and W. J. Dally, "The GPU computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, 2010. 4.1

[60] C.-W. Hsieh, C.-Y. Chou, T.-C. Tsai, Y.-F. Cheng, and S.-H. Kuo, "NCHC's formosa V GPU cluster enters the TOP500 ranking," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pp. 622–624, IEEE, 2012. 4.1

[61] I. S. Ufimtsev and T. J. Martinez, "Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation," *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222–231, 2008. 4.1, 4.4

[62] A. G. Anderson, W. A. Goddard III, and P. Schröder, "Quantum Monte Carlo on graphical processing units," *Computer Physics Communications*, vol. 177, no. 3, pp. 298–306, 2007. 4.1

[63] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of computational chemistry*, vol. 28, no. 16, pp. 2618–2640, 2007. 4.1

DOI:10.15774/PPKE.ITK.2014.009

[64] K. Yasuda, "Two-electron integral evaluation on the graphics processor unit," *Journal of Computational Chemistry*, vol. 29, no. 3, pp. 334–342, 2008. 4.1

[65] C. Nvidia, "Compute unified device architecture programming guide," 2007. 4.1

[66] A. Munshi *et al.*, "The opencl specification," *Khronos OpenCL Working Group*, vol. 1, pp. l1–15, 2009. 2.1, 4.1

[67] X. Andrade and A. Aspuru-Guzik, "Real-space density functional theory on graphical processing units: computational approach and comparison to Gaussian basis set methods," *arXiv preprint arXiv:1306.2953*, 2013. 4.1

[68] M. Cawkwell, E. Sanville, S. Mniszewski, and A. M. Niklasson, "Computing the density matrix in electronic structure theory on graphics processing units," *Journal of Chemical Theory and Computation*, vol. 8, no. 11, pp. 4094–4101, 2012. 4.1

[69] W. A. De Jong, E. Bylaska, N. Govind, C. L. Janssen, K. Kowalski, T. Müller, I. M. Nielsen, H. J. van Dam, V. Veryazov, and R. Lindh, "Utilizing high performance computing for chemistry: parallel computational chemistry," *Physical Chemistry Chemical Physics*, vol. 12, no. 26, pp. 6896–6920, 2010. 4.1

[70] A. E. DePrince III and J. R. Hammond, "Coupled cluster theory on graphics processing units i. the coupled cluster doubles method," *Journal of Chemical Theory and Computation*, vol. 7, no. 5, pp. 1287–1295, 2011. 4.1

[71] N. Luehr, I. S. Ufimtsev, and T. J. Martínez, "Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs)," *Journal of Chemical Theory and Computation*, vol. 7, no. 4, pp. 949–954, 2011. 4.1

[72] R. Olivares-Amaya, M. A. Watson, R. G. Edgar, L. Vogt, Y. Shao, and A. Aspuru-Guzik, "Accelerating correlated quantum chemistry calculations using graphical processing units and a mixed precision matrix multiplication library," *Journal of Chemical Theory and Computation*, vol. 6, no. 1, pp. 135–144, 2009. 4.1

DOI:10.15774/PPKE.ITK.2014.009

[73] G. Shi, V. Kindratenko, I. Ufimtsev, and T. Martinez, "Direct self-consistent field computations on GPU clusters," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–8, IEEE, 2010. 4.1

[74] M. Watson, R. Olivares-Amaya, R. G. Edgar, and A. Aspuru-Guzik, "Accelerating correlated quantum chemistry calculations using graphical processing units," *Computing in Science & Engineering*, vol. 12, no. 4, pp. 40–51, 2010. 4.1

[75] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten, "GPU-accelerated molecular modeling coming of age," *Journal of Molecular Graphics and Modelling*, vol. 29, no. 2, pp. 116–125, 2010. 4.1, 4.4

[76] A. Szabo and N. S. Ostlund, *Modern quantum chemistry: introduction to advanced electronic structure theory*. Courier Dover Publications, 1989. 4.1

[77] P. M. Gill, "Molecular integrals over gaussian basis functions," *Advances in quantum chemistry*, vol. 25, pp. 141–205, 1994. 4.1, 4.2

[78] P. M. Gill and J. A. Pople, "The prism algorithm for two-electron integrals," *International journal of quantum chemistry*, vol. 40, no. 6, pp. 753–772, 1991. 4.1, 4.2, 4.2.3, 4.3

[79] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, *et al.*, "High performance computational chemistry: An overview of NWChem a distributed parallel application," *Computer Physics Communications*, vol. 128, no. 1, pp. 260–283, 2000. 4.1

[80] A. V. Titov, V. V. Kindratenko, I. S. Ufimtsev, and T. Martinez, "Generation of kernels for calculating electron repulsion integrals of high angular momentum functions on GPUs–preliminary results," *Proceedings of SAAHPC 2010*, pp. 1–3, 2010. 4.1, 4.4

[81] A. V. Titov, I. S. Ufimtsev, N. Luehr, and T. J. Martinez, "Generating efficient quantum chemistry codes for novel architectures," *Journal of Chemical Theory and Computation*, vol. 9, no. 1, pp. 213–221, 2012. 4.1

DOI:10.15774/PPKE.ITK.2014.009

[82] L. E. McMurchie and E. R. Davidson, "One- and two-electron integrals over Cartesian Gaussian functions," *Journal of Computational Physics*, vol. 26, no. 2, pp. 218–231, 1978. 4.1, 4.2, 4.2, 4.2.1

[83] P. M. Gill, M. Head-Gordon, and J. A. Pople, "Efficient computation of two-electron-repulsion integrals and their nth-order derivatives using contracted gaussian basis sets," *Journal of Physical Chemistry*, vol. 94, no. 14, pp. 5564–5572, 1990. 4.2, 4.2, 4.2, 4.2.1

[84] C. A. White, B. G. Johnson, P. M. Gill, and M. Head-Gordon, "The continuous fast multipole method," *Chemical physics letters*, vol. 230, no. 1, pp. 8–16, 1994. 4.4

[85] H. J. Kulik, N. Luehr, I. S. Ufimtsev, and T. J. Martinez, "Ab initio quantum chemistry for protein structures," *The Journal of Physical Chemistry B*, vol. 116, no. 41, pp. 12501–12509, 2012.

[86] Y. Furukawa, R. Koga, and K. Yasuda, "Acceleration of computational quantum chemistry by heterogeneous computer architectures,"

[87] V. P. Vysotskiy and L. S. Cederbaum, "Accurate quantum chemistry in single precision arithmetic: Correlation energy," *Journal of Chemical Theory and Computation*, vol. 7, no. 2, pp. 320–326, 2010.

[88] M. M. Mehine, S. A. Losilla, and D. Sundholm, "An efficient algorithm to calculate three-electron integrals for Gaussian-type orbitals using numerical integration," *Molecular Physics*, no. just-accepted, 2013.

[89] A. Harju, T. Siro, F. F. Canova, S. Hakala, and T. Rantalaiho, "Computational physics on graphics processing units," in *Applied Parallel and Scientific Computing*, pp. 3–26, Springer, 2013.

[90] L. Genovese, M. Ospici, T. Deutsch, J.-F. Méhaut, A. Neelov, and S. Goedecker, "Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures," *The Journal of chemical physics*, vol. 131, p. 034103, 2009.

DOI:10.15774/PPKE.ITK.2014.009

[91] G. Knizia, W. Li, S. Simon, and H.-J. Werner, "Determining the numerical stability of quantum chemistry algorithms," *Journal of Chemical Theory and Computation*, vol. 7, no. 8, pp. 2387–2398, 2011.

[92] B. M. Gosswami, "Implementing density functional theory (DFT) methods on many-core GPGPU accelerators," 2011.

[93] K. Yasuda, "Accelerating density functional calculations with graphics processing unit," *Journal of Chemical Theory and Computation*, vol. 4, no. 8, pp. 1230–1236, 2008.

[94] W. Ma, S. Krishnamoorthy, O. Villa, and K. Kowalski, "GPU-based implementations of the noniterative regularized-CCSD (T) corrections: applications to strongly correlated systems," *Journal of Chemical Theory and Computation*, vol. 7, no. 5, pp. 1316–1327, 2011.

[95] K. Bhaskaran-Nair, W. Ma, S. Krishnamoorthy, O. Villa, H. J. van Dam, E. Apra', and K. Kowalski, "Noniterative multireference coupled cluster methods on heterogeneous CPU–GPU systems," *Journal of Chemical Theory and Computation*, vol. 9, no. 4, pp. 1949–1957, 2013.

[96] D. Ye, A. Titov, V. Kindratenko, I. Ufimtsev, and T. Martinez, "Porting optimized GPU kernels to a multi-core CPU: Computational quantum chemistry application example," in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pp. 72–75, IEEE, 2011. 4.4

[97] M. P. Haag and M. Reiher, "Real-time quantum chemistry," *International Journal of Quantum Chemistry*, vol. 113, no. 1, pp. 8–20, 2013.

[98] X. Wu, A. Koslowski, and W. Thiel, "Semiempirical quantum chemical calculations accelerated on a hybrid multicore CPU–GPU computing platform," *Journal of Chemical Theory and Computation*, vol. 8, no. 7, pp. 2272–2281, 2012.

[99] C. M. Isborn, B. D. Mar, B. F. Curchod, I. Tavernelli, and T. J. Martínez, "The charge transfer problem in density functional theory calculations

DOI:10.15774/PPKE.ITK.2014.009

of aqueously solvated molecules," *The Journal of Physical Chemistry B*, vol. 117, no. 40, pp. 12189–12201, 2013.

[100] J. A. Pople and W. J. Hehre, "Computation of electron repulsion integrals involving contracted gaussian basis functions," *Journal of Computational Physics*, vol. 27, no. 2, pp. 161–168, 1978. 4.1

[101] W. J. Hehre, R. F. Stewart, and J. A. Pople, "Self-consistent molecular-orbital methods. i. use of gaussian expansions of slater-type atomic orbitals," *The Journal of Chemical Physics*, vol. 51, p. 2657, 1969. 4.1