Nested Data Parallelism for Image Processing Algorithms

An exposé for a bachelor thesis

Chandrakant Swaneet Kumar Sahoo

17.04.2015

Summary This exposé describes my motivation, the contents and the goal for my bachelor thesis on the application of the Nested Data Parallel approach on Image Processing Algorithms. The contents of this document are guided by the checklist [Köhler-Bußmeier, 2012].

Contact swaneet06@hotmail.com

Location Hochschule für Angewandte Wissenschaften Hamburg

Department Dept. Informatik

Examiner Prof. Dr. Michael Köhler-Bußmeier

Second examiner Prof. Dr. Andreas Meisel

Contents

1 Introduction

1.1 Context

In the past few decades the raw processing power of CPUs has not increased by any significant portion. Instead, we are presented with hardware with an exponentially increasing number of processors - each of them as fast as the previous generation. A natural objective is adapting our programs to make the most of the availible processors. A few questions arise:

- How can we effectively exploit the higher number of processors to keep increasing performance?
- How can we write programs which can run on varying number of processors?

Research and Development in this field is generally referred to as Parallel Computing. For my thesis, I have further reduced the context to the programming language Haskell and Image Processing. An explanation is given next.

1.2 Haskell

I am fascinated by the language as it enables high-level abstract programming without sacrifising efficiency. Throughout the years I have learned a lot in Haskell. The abstractions (Monads, GADTs, Type Classes, Lawful-Thinking,...) exceed the level I have encountered in other languages whereas other elements (Stream Fusion, Lazyness, ...) enable optimiziations which are difficult or entirely missing in conventional languages. Haskell has become popular language for research in functional programming. Choosing Haskell for my thesis additionally enables me to demonstrate my experience and knowledge on it.

1.3 Image Processing

Image Processing is a topic of high personal interest. I particularly liked the elective course "Robot Vision" (Prof. Dr. Meisel). Image Processing Algorithms have a natural tendency to be applicable in Parallel Computing (as many algorithms end up on high-performance GPUs). Therefore, I think such a focus is adequate. I am also focusing on Image Processing Algorithms because I want to express my interest on it.

1.4 General approaches in Parallel computing

Generally recognized approaches are:

- Concurrent programming (threads & locks)
- Flat data Parallelism (`map/reduce`)
- GPU Parallelism (CUDA,OpenCL, mainly matrix/vector operations)
- and in the Haskell world: [Marlow, 2012]
  - Algorithm + Strategy = Parallelism, a task-based approach [Trinder et al., 1998]
  - The Par Monad, a dataflow approch, [Marlow et al., 2011]
  - Repa, a regular-array data-parallel approach, [Keller et al., 2010]
  - Accelerate, a GPU approach, [McDonell et al., 2013]
  - Nested Data Parallel, a parametric nested-data-parallel approach, [Chakravarty et al., 2007]

1.5 Leading question

Given an image processing algorithm (e.g. Split&Merge-Segmentation), the leading question for this thesis is:

How can we implement this high-level irregularly-parallel algorithm such that it compiles to efficient mashine level code?

Subquestions arising are:

- How much faster is a parallel variant against the sequential one?
- How much faster is a compiled variant to human-written low-level parallel code?

1.6 Specific Approaches in Haskell and a comparision

The following is a brief comparision of the candidate approaches for my thesis.

- **Repa: Regular, shape-polymorphic Parallel Arrays**
  - \+ Retains high level of abstraction
  - \+ Highly efficient mashine-code (e.g. fusioning)
  - \+ Feels like conventional functional programming
  - \+ Can compile to OpenCL/CUDE for high performance GPU execution
  - \+ Multi-dimensional regular arrays (e.g. matrices)
  - \- No support for irregular data structures (e.g. sparse-matrices, trees, graphs, recursion)
  - Recommended for matrix-/pixel-algorithms
- **Algorithm + Strategy = Parallelism**
  - \+ Algorithm and Parallel evaluation strategy are entirely independent
  - \+ (Therefore) Highly irregular algorithms can be paralellized
  - \+ Feels like conventional functional programming
  - \+ Very easy to use
  - \- The programmer has to ensure subtle pre-/post-conditions for successful parallelization
  - \- Cannot compile to OpenCL/CUDE for high performance GPU execution
  - Recommended as simple-parallization approach
- **Nested Data Parallel**
  - \+ Retains high level of abstraction
  - \+ Feels like conventional functional programming
  - \+ Supports irregular data and program flows (e.g. sparse-matrices, trees, graphs, recursion)
  - \+ Highly efficient mashine code (fusioning & flattening)
  - \- Work in progress / experimental (setting up Data Parallel Haskell was tiresome ...)
  - +/- GPU support in future
  - Recommended for irregular algorithms

1.7 Deciding on an approach

I have decided for Nested Data Parallelism. My choice will be explained now:

Nested Data Parallel Nested Data Parallelism is an extension to conventional Flat Data Paralellism. Flat data parallelism is characterized by parallel implementations of `map/reduce` functions and is widely used in industry and already heavily developed. However is it only limitedly applicable - as the programmer has to transform the algorithm to make use of these constructs. Out algorithms however usually operate on nested data (e.g. nested lists/arrays, trees, recursion,...). This is a discrepancy - a flat data parallel program is efficient to run, but hard to write - and a nested data algorithm (like they appear in books or papers) are easy to write, but run inefficiently. The Nested Data Parallel approach [Chakravarty et al., 2007] combines the best of both. It enables us to write our algorithms as we like them and applies a non-trivial transformation to produce efficient flat data parallel code. I choose this topic, because its an excellent example of high level programming compiling to highly performant low level code - which is why I discarded all other apporaches expect for Repa. I chose NDP rather than Repa, because it spreads a wiedly undeveloped/unknown approach on parallelism - that is the idea of allowing nested data for parallel programs.

1.8 What is novel in this approach?

It implements a non-trivial flattening transformation (also called vectorization) and takes a large amount of workload off the programmer. By this, it follows the general idiom 'let the compiler do the hard work for you'.

1.9 When is the thesis regarded as completed?

The thesis is considered finished, once various strategies of implementing a chosen algorithm (e.g. Split&Merge-Segmentation) have been implemented and compared in effectiveness/efficiency/human-workload.

1.10 Which tasks have to be completed for this thesis?

My work is defined by creating the programs P_s, $P_m$and $P_{np}$and comparing them. The focus is on showing the various transformations applied in NDP, which the programmer would have had done manually.

A_K := A conceptional Problem (e.g. Sorting)
A_S := A sequential Algorithm to A_K (e.g. Mergesort)
P_P := A parallel Algorithm to A_K (e.g Parallel Mergesort)
P_s:= An ordinary Haskell implementation of A_S (e.g. using Lists)
P_m:= A manually-parallelized implementation of A_P (e.g using explicit threads)
P_{np}:= A Nested-Data-Parallel implementation of A_P (using parallel arrays)

Additional tasks

- Read further papers on how NDP is implemented (especially. vectorization and fusioning)
- Read futher literature on how to make quantified comparisions of the programs (e.g. parallel complexity, benchmarks, etc...)

- Decide whether I will work with the true Haskell-Core code generated in P_{np}or apply the flattening and fusioning transformations manually. The first variant is the actual source code that will later be executed - but it is hard to read. [1] The implementation of DPH is less developed than its theoretical background. The second variant is easier to work with and simpler to present, however it will almost certainly be less optimized than the actual code.
- Decide which algorithm(s) is/are to be used as examples. They should be easily visualizable and should have irregular behaviour. Connected-Components-Labeling on images is such an candidate. Combining multiple algorithms into a pipeline is also attractive, since cross-algorithm fusionsing and optimization is where Haskell can shine truely.
- Learn how make benchmarks/create runtime statistics

1.11 Which tasks are realistic to be accomplished within the scope of this thesis?

I am optimistic that these task can be accomplished in a timeframe of 2 months. However, I might need to keep a tight focus or choose a simple (and small) algorithm to keep the task accomplishable.

1.12 Time plan

My time plan is visible at table 1.

Table 1: Time table

CW	monday	thesis work
17	20.04	reading remaining papers, reading parallel complexity theory
18	27.04	deciding on an algorithm, learning benchmarking
19	4.05	implementing $P_s$and P_m
20	11.05	implementing P_{np}
21	18.05	vectorizing and optimizing P_{np}/ understanding generated core
22	25.05	analysis and benchmarking
23	1.06	*puffer*
24	8.06	*puffer*
25	15.06	Begin to write down, prepare for exams
26	22.06	Writing..., prepare for exams
27	29.06	Writing..., prepare for exams
28	6.07	Prepare Colloquium, Writing..., exams week 1
29	13.07	Prepare Colloquium, Finalize writing, exams week 1
30	20.07	Colloquium and Release
31	27.07	Last week for Colloquium and Release
32	3.08	Fin `:D`

2 Draft - Contents

The following presents a draft structure of my thesis. Since the algorithm I am going to implement is not decided yet, I am using "Split&Merge-Segmentation" as its placeholder.

[1] See `DotP.hs` and `DotP.vect.hs` at my github repo.

1. Introduction
    1.1. Context
    1.2. Goal
    1.3. Structure
2. Basics
    2.1. Parallel Computing and Complexity
    2.2. Haskell
    2.3. Nested Data Parallelism
        i. Parallel Arrays
        ii. Sparse Matrices - An Example
        iii. Execution model
    2.4. A_K: Split&Merge-Segmentation
3. P_s: Sequential Split&Merge-Segmentation
    3.1. Implementation
    3.2. Runtime analysis (e.g. sequential/parallel complexity)
    3.3. Benchmark
4. P_m: Manually-parallel Split&Merge-Segmentation
    4.1. Implementation
    4.2. Runtime analysis (e.g. sequential/parallel complexity)
    4.3. Benchmark
5. P_{np}: Nested-Data-Parallel Split&Merge-Segmentation
    5.1. High-level Implementation
    5.2. Transformations
        i. Non-Parametric representation
        ii. Lifting
        iii. Vectorization
        iv. Fusioning
    5.3. Final low-level form
    5.4. Runtime analysis (e.g. sequential/parallel complexity)
    5.5. Benchmark
6. Evaluation
    6.1. Sequential vs. Parallel
    6.2. Manually-Parallel vs. Nested-Data-Parallel
7. Conclusion
    7.1. Effectivenesss of Nested Data Parallel
    7.2. Related work (on parallel computing in Haskell)

7.3. Future work

i. Alternate Algorithms
ii. Best of Repa and NDP
iii. Distributed NDP
iv. NDP on GPUs

3 References

[Chakravarty et al., 2007] Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., Keller, G., and Marlow, S. (2007). Data parallel haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA. ACM.

[Keller et al., 2010] Keller, G., Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., and Lippmeier, B. (2010). Regular, shape-polymorphic, parallel arrays in haskell. *SIGPLAN Not.*, 45(9):261–272.

[Köhler-Bußmeier, 2012] Köhler-Bußmeier, M. (2012). Checkliste für ein expose. `http://www.informatik.uni-hamburg.de/TGI/lehre/abschlussarbeiten/expose-checklist.pdf`.

[Marlow, 2012] Marlow, S. (2012). Parallel and concurrent programming in haskell. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, pages 339–401, Berlin, Heidelberg. Springer-Verlag.

[Marlow et al., 2011] Marlow, S., Newton, R., and Jones, S. P. (2011). A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA. ACM.

[McDonell et al., 2013] McDonell, T. L., Chakravarty, M. M. T., Keller, G., and Lippmeier, B. (2013). Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 49–60, New York, NY, USA. ACM.

[Trinder et al., 1998] Trinder, P. W., Hammond, K., Loidl, H. W., and Jones, S. L. P. (1998). Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60.

4 Personal notes

- Zeitplan für 2.5 Monate
- English als Ausarbeitungssprache
- Wahl eines Bildverarbeitungsalgorithmus welcher nicht offensichtlich parallelisierbar ist
- A collection of image processing algorithms with varying degree of parallization: (from a discussion with Prof. Meisel)
  1. 99%: Median, Faltungsmasken
  2. 80%: Parallele Kantenverfolgung, μ-Momente
  3. 70%: Rekursive Algorithmen (Connected Components Labeling, SplitMerge Segmentierungsverfahren,...)
  4. 60%: Shortest Paths (e.g. Seam Carving)
  5. 10%: Sequentielle Kantenverfolgung
  6. 0%: Zufallsgenerator