Streaming Nested Data Parallelism

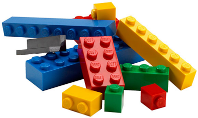
Frederik M. Madsen
fmma@diku.dk

HIPERFIT

December 2014

Array languages: Background

- Functional data-parallel array languages express concise and compositional programs.
- The building blocks are simple but powerful combinators, such as *map*, *reduce*, *scan* and *filter*.
- The combinators run efficiently on many different platforms.
  - From single-core to highly parallel machines.



Array languages: Background

- High-level array languages allows domain experts to focus on the pure algorithm, and still expect high performance.
  - Even when running on specialized hardware.
- Benefits: Low development costs, code re-use.
- In computational finance, application areas include statistical analyses, option pricing and high-frequency trading.

Array languages: Background

- The language implementer must provide an efficient implementation of the language for all supported target architectures.

- Separation of knowledge: The domain expert trusts the compiler to do the right thing.
  - The compiler maps any algorithm to the target architecture with "reasonable" performance.
  - Particularly difficult to achieve for parallel targets.

Array languages: Static analysis

- The static analysis approach: Optimizing compiler.
- The criteria for a given optimization to be applicable are not always obvious to the domain expert.
- Traditional sequential-code optimization:
  - Constant-factor speedup.
  - If the optimization does not fire, it's not the end of the world.
- Parallelizing optimization:
  - #processors-factor speedup.
  - If the optimization does not fire,
    the program will be significantly slower.

- In order to write efficient re-usable programs, the domain expert must have some knowledge of the compiler and the target architecture.

Array languages: Cost models

- A formal cost model, provides a contract between the domain expert and the language implementer.
  - Implementation must respect the cost model.
  - The cost model provides guarantees to the domain expert.
- To have any value, the cost model must be.
  - Optimistic: Provide reasonable performance guarantees.
  - Simple: Understandable by the domain expert.

Array languages: Cost models

- Assuming strict evaluation, what is the cost of (*map f xs*)?
  - The cost of computing $f\ x$ for each x in *xs*.
    - Cost of $f\ x$ may depend on the value of x.
    - Not a static analysis.
  - Time cost:
    - The *total work* is the sum of the work for each $f\ x$.
      $\approx$ time on a single-processor machine.
    - The number of parallel *steps* is the maximum of the parallel steps taken in each $f\ x$.
      $\approx$ time on a ∞-processor machine.
    - Actual time on a P-processor machine is (morally) $work/P + steps$.
  - Space cost:
    - The size of the result plus the sum of the size used in each $f\ x$.
    - Not good enough, exemplified later.

Array languages: Nested data parallelism

- Implementing (*map f*) for a GPU target:
  - Naive approach: In a single kernel call, make each thread computes f for each element.
- This works if f is a simple scalar expression, such as

$$f\ x = sqrt\ (x * x + x * x).$$

- This will *not* work if f contains *nested data parallelism* (NDP):

$$f\ x = sum\ [1..x].$$

- GPUs cannot handle nested kernel-calls (in general).

Array languages: Nested data parallelism

- Therefore, the language implementer must make a choice:
  - Reject NDP expressions.
    - Hand over the problem to the programmer.
    - Perfectly fine (although less expressive) solution.
  - Implement troublesome combinators as sequential loops.
    - Does not obey cost model.
    - Bad work-balancing, non-full utilization.
  - Eliminate NDP by flattening transformation[1].

[1] Blelloch, Guy E. **Vector models for data-parallel computing**, vol. 75. MIT press Cambridge, MA, 1990.

Blelloch's flattening transformation

- Blelloch's flattening transformation eliminates NDP in maps.
- NDP disallowed in other combinators.
- Scans and reductions are restricted to a predefined set of scalar combination functions.
  - $\oplus$-*scan*, $\oplus$-*reduce* $\quad \oplus ::= + \mid \times \mid max \mid \cdots$
- Arrays are (potentially nested) vectors; Vector type is $[A]$.
- The transformation requires *segment-descriptor* representation:

$$[[10], [20, 30], [40, 50, 60]]$$

is represented by

$$([1, 2, 3], [10, 20, 30, 40, 50, 60])$$

Blelloch's flattening transformation

- First, the language implementer must implement basic segmented combinators: $(map\ sum)$, $(zipWith\ (+))$, etc.
- Secondly, all maps containing NDP are fissioned:

$$map\ (\lambda x.sum\ [1..x]) \rightsquigarrow mapSum \circ map\ (\lambda x.[1..x])$$

$$\rightsquigarrow mapSum \circ mapRange \circ map\ (\lambda x.(1, x))$$

- Finally, mapped maps are eliminated by "looking under" the top-most segment descriptor. E.g.:

$$map\ mapSum \rightsquigarrow \lambda(segdsc, xs).(segdsc, mapSum\ xs)$$

The space problem

- Blelloch's language has simple and optimistic time costing.
  - Because of the flattening transformation, all potential parallelism is exposed.
- For the same reason, space is in order of the exposed parallelism.
  - Example: Evaluation of $(map\ (\lambda x.sum\ [1..x])\ [3,5])$:

$$\begin{aligned} &\rightsquigarrow mapSum \circ mapRange \circ map\ (\lambda x.(1,x))\ \$\ [3,5] \\ &\rightarrow mapSum \circ mapRange\ \$\ [(1,3),(1,5)] \\ &\rightarrow mapSum\ ([3,5],[1,2,3,1,2,3,4,5]) \\ &\rightarrow [6,15] \end{aligned}$$

  - Note: Dependent parallel degree prevents GPU-kernel fusion of $(mapSum \circ mapRange)$.

The space problem

- More interesting example:

$$map\ (\lambda x.sum\ [1..x])\ [3E10, 5E10]$$

- Huge data-parallel computation, substantial performance gain from acceleration.
- Unfortunately, space cost $\approx$ 320 gigabytes.
  - GeForce GTX Titan Black has 6 gigabytes.
- *Have to* evaluate in chunks.
  - Domain expert must explicitly encode chunking in program.
    - Loose compositionality.
    - Platform-dependent magic numbers.

- Research question:
  Is it possible to design an expressive functional array language based on implicit chunking that supports nested data parallelism and has a simple optimistic time-space cost model?[2]

[2] Frederik M. Madsen and Andrzej Filinski. **Towards a Streaming Model for Nested Data Parallelism**. In *2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*. Boston, Massachusetts. September 2013.

Space costing

- Eager space costing is insufficient.
- Cost model must account for actual parallelism, not just potential.
  - Similar to work and steps for time costing.
- Proposal, good space cost model:
  - *Sequential space*: Space on a single-processor machine.
  - *Parallel space*: Space on a ∞-processor machine.
  - Actual space on a P-processor machine is (morally) min($P \times$ *sequential space*, *parallel space*).
- $map\ (\lambda x.sum\ [1..x])\ [3E10, 5E10]$:
  - *Sequential space*: 1.
  - *Parallel space*: $8E10$.
  - Actual space: $\min(P \times 1, 8E10) = P$.

Dataflow execution

- Chunked evaluation strategy: Strict, non-eager.
  - Arrays compute as chunk streams.
  - Expressions compile to dataflow networks.
- What is required to realize cost model?
  - Time costing:
    - The chunk size utilizes all processors.
    - Streams are only traversed once.
  - Space costing:
    - Chunks are not persistent; For a given stream, only one chunk is live at any given time.
    - The chunk size is bounded.
- The compiler picks a chunk size bound that is suitable for the target machine.

Sequences

- These requirements precludes:
  - Constant-time random-access.
  - Constant-time length.
  - Array sharing in bulk operations.
    - E.g. assume *xs* and *ys* are bound to arrays. Then *map* (*f* *xs*) *ys* cannot be streamed (*xs* would have to be traversed multiple times).
  - Dependence on future values. E.g. $let\ x = sum\ xs\ in\ map\ (+x)\ xs$
- Language design: Allow explicit use of eager vectors.
  - More general use allowed. Bad space costing.
  - Referred to simply as *vectors*.
  - All other arrays are referred to as *sequences*.

The resulting language: SNESL

$$
\begin{aligned}
\text{Scalars} \ni \pi &::= \mathit{Bool} \mid \mathit{Int} \mid \mathit{Float} \mid \cdots \\
\text{Eager} \ni \tau &::= \pi \mid (\tau_1, \ldots, \tau_k) \mid [\tau] \\
\text{Non-Eager} \ni \sigma &::= \tau \mid (\sigma_1, \ldots, \sigma_k) \mid \{\sigma\}
\end{aligned}
$$

$$
\begin{aligned}
e \quad ::= \quad & \mathit{constant} \\
\mid \quad & x \\
\mid \quad & \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \\
\mid \quad & (e_1, \ldots, e_k) \mid e.k \\
\mid \quad & \mathit{op}\ e \\
\mid \quad & \{e_1 : x\ \mathit{in}\ e_2\}
\end{aligned}
$$

$$
\begin{aligned}
+ \quad &: \quad (\mathit{Int}, \mathit{Int}) \to \mathit{Int} \\
&\vdots \\
\mathit{length}_\tau \quad &: \quad [\tau] \to \mathit{Int} \\
!_\tau \quad &: \quad ([\tau], \mathit{Int}) \to \tau \\
\mathit{mkseq}^k_\sigma \quad &: \quad \overbrace{(\sigma, .., \sigma)}^{k} \to \{\sigma\} \\
\mathit{concat}_\sigma \quad &: \quad \{\{\sigma\}\} \to \{\sigma\} \\
\mathit{part}_\sigma \quad &: \quad (\{\sigma\}, \{\mathit{Bool}\}) \to \{\{\sigma\}\} \\
\oplus\text{-}\mathit{scan}_\pi \quad &: \quad \{\pi\} \to \{\pi\} \\
\oplus\text{-}\mathit{reduce}_\pi \quad &: \quad \{\pi\} \to \pi \\
&\vdots \\
\mathit{tab} \quad &: \quad \{\tau\} \to [\tau] \\
\mathit{seq} \quad &: \quad [\tau] \to \{\tau\}
\end{aligned}
$$

- Type system (almost) ensures schedulability of sequences:
  - Random access and length disallowed for sequences.
  - Array sharing in bulk operations $\{e_1 : x\ \mathit{in}\ e_2\}$: Typing of e_1 is restricted to a non-sequence typing context.
  - Sadly, future dependencies are not handled yet.

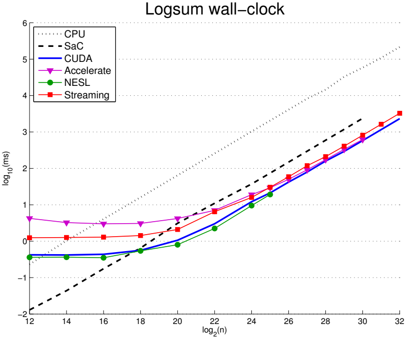
Experiments

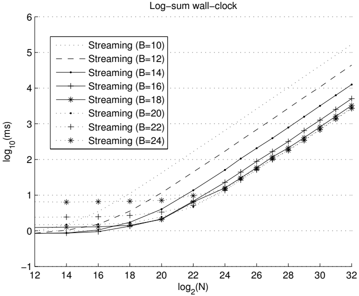
- We do not generate CUDA code yet.
- To test the validity of the execution model, we wrote dataflow CUDA programs, simulating what the compiler would conceivably output.
- Compared to: CPU reference, traditional CUDA code, Accelerate and GPU-NESL[3].
- NVIDIA GeForce GTX 690 (2 GB memory, 1536 cores, 915 MHz). Dual AMD Opteron 6274 (2×16 cores, 2200 MHz).

[3] References can be found in paper (2).

Experiments

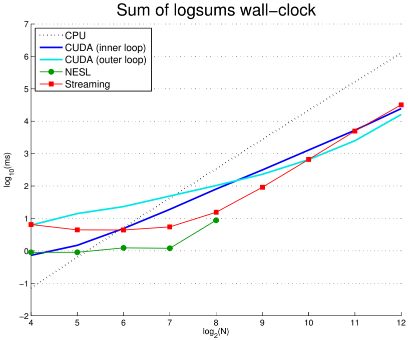
- Computing a simple log-series: $\sum_{i=1}^{n} \log i$

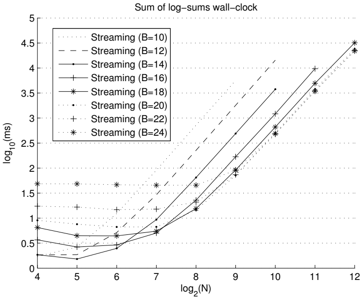




Experiments

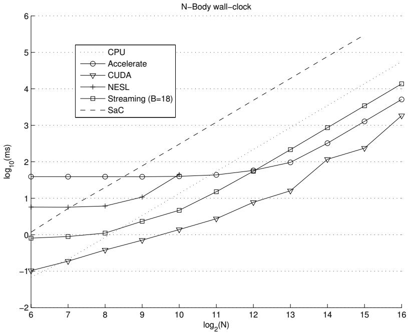
- Sum of multiple log-series: $\sum_{n=1}^{N} \sum_{i=1}^{n} \log i$
- Nested data parallelism.

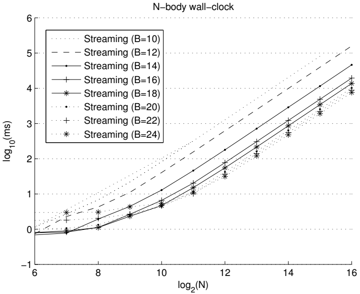




Experiments

- Naive *N*-body simulation.





Experiments

- Positive results:
  - Reasonable execution times.
  - Reliably scales to big problem sizes.
  - Execution time converges as chunk size increases.
- This suggests an optimal chunk size of 2^{18} elements for all three experiments.
  - Knowing the target machine, the compiler can fix the chunk size.

Concluding remarks

- Has the research question been answered? No.
- Standing issues:
  - Space costing of sequences of vectors $\{[\tau]\}$.
    - Sequential space: Maximum of size of elements.
    - Actual space ($P \times$ sequential space) is pessimistic.
  - Future dependencies:
    - Conservative solution: Linear type system.
    - Schedulability analysis.
  - Full compiler stack.
  - More benchmarks.
  - Recursion: Dynamically evolving dataflow network.
- Flat version with shape-polymorphic regular arrays currently being implemented in Accelerate.