Parallellization of C code

Jos van Eijndhoven
jos@vectorfabrics.com

GPU symposium at TUE
Sept. 1, 2010

Vector Fabrics

Introducing myself

- Completed my PhD on this TUE in 1984. Worked in the EE dept. until 1996. Did a sabbatical in IBM T.J. Watson Research Center, pioneering high-level synthesis.
- Moved to Philips Research to work on programmable media processing architectures, covering processor architectures, compilation techniques, video-domain applications. Joined the corporate patent portfolio review team. Cooperated with Philips' IC design team in San Jose, CA.
- Co-founder of 'Vector Fabrics' in 2007. Vector Fabrics creates tools for embedded system design, covering the path from C-code input to system HW architecture and embedded software output.
- Published about 100 scientific publications, holds 14 worldwide patents.

Presentation summary

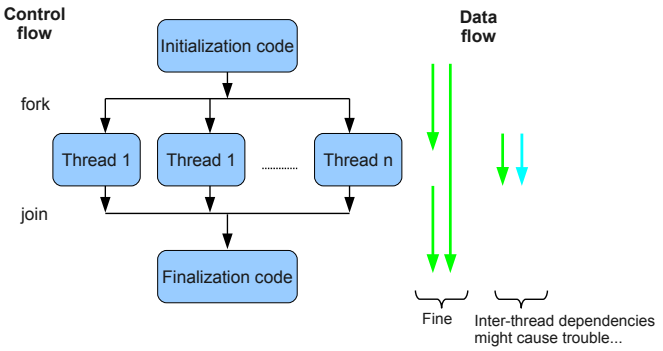
- C language: memory, dataflow, control flow
- Loop-based parallellizations
- Data dependencies that hinder parallelization
- Handling / resolving data dependencies
- Tooling support for parallellization
- Conclusion

The C language: sequential by nature

- Procedural (imperative) programming language:
  - State in variables / memory locations
  - Data flow (value assignment & use through expressions)
  - Control flow (loops, conditionals, function calls)

  Strictly sequential semantics by nature of 'State'.
- Alleviation of the sequential nature requires knowledge of data-flow between memory locations.

Inter-thread data dependencies

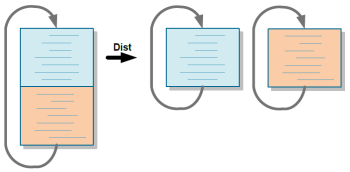


Analysis of data-dependencies, compile-time static or run-time dynamic, is an active research area...

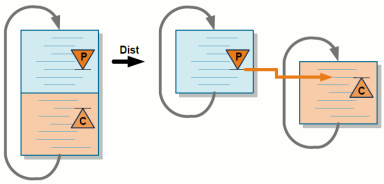
Parallellization: threads from loops

- Partition the compute load, such that parts can be distributed over concurrent processors.
- Partitioning almost directly leads to investigation of loops:
  - Loops contain most of the workload
  - Loops provide nice opportunity for distributing pieces of work
- Typically, a loop *induction variable* needs to be captured together with its *induction expression*. This allows explicit derivation of loop indexes. The induction variable itself is exempt from the loop-carried data dependencies.
- For parallellization, literature distinguishes between:
  - Loop distribution: Partition body in pieces, keep index space
  - Loop splitting: Keep body, partition loop index space.

Loop distribution



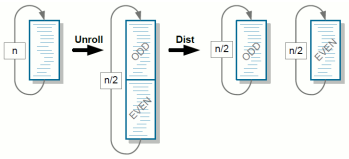
Depicts ideal distribution:
- good load balance
- no data dependencies



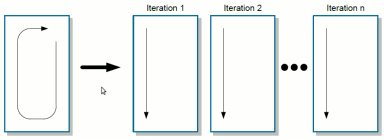
Might need to synchronize data from production to consumption...

Loop splitting

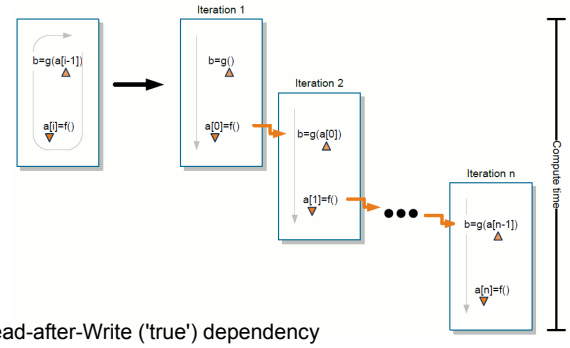
- Implemented as *loop unrolling* followed by *loop distribution*:



- Or implemented directly:



Loop carried data dependencies (RaW)



- Read-after-Write ('true') dependency
- Requires data *communication* and *synchronization*
- Reduces available parallellism

Other dependency types

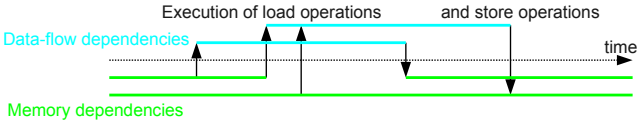
- Write-after-Read (anti-)dependency:
  Data must be consumed before it can be over-written.
- Write-after-Write (output-)dependency:
  Data must be over-written in proper order

In general, these types of dependencies also:
- Require data *synchronization*
- Reduce available parallellism

Data-flow versus memory dependencies

- Data-flow dependencies relate to consumption and production of scalar values in expressions. These values are mapped by the C-compiler in registers. Mapping to registers involves a classic (static) life-time analysis. Accessing these values does **not** involve load/store operations.
- Memory dependencies relate to accessing values on a particular address in memory through **load/store operations**.

Unfortunately, there is no standard/direct relation between C code syntax and mapping to registers versus memory.



Capturing data-dependencies is hard

In **real-world C programs**, capturing data dependencies is hard:
- Dependencies occur between stores and loads beyond function- and file-boundaries, beyond the scope of the C compiler.
- Beyond file boundaries, the linker decides on mapping of variable-names and function-names. Linker semantics is tricky.
- Due to data-dependent control and/or pointer arguments, multiple invocations of the same function result in different dependency patterns.
- With data-dependent control, the discovered dependencies depend upon the actual application input test data.
- Dependency analysis should cover basic C libraries, supporting e.g. malloc(), memcpy(), read(), write(), ...

Resolving data dependencies (1)

Typically, many data dependencies **can be removed**. Those are just a side-effect of an unfortunate implementation, NOT essential for the algorithm.

E.g.: replace a linked-list datastructure ('`p = p→next`') by an array with object pointers ('`p = elem[i];`'), in which '`i`' is (derived from) the loop induction variable.

Clearly, this can be a significant task...

Obviously, removing *all* inter-thread dependencies allows the creation of an optimal parallel system....

Resolving data dependencies (2)

Some remaining data-dependencies **are irrelevant:** their ordering does not affect application semantics.

E.g.1: '`a[i] = malloc(sizeof(..));`'
The implementation of malloc has internal (global) variables that create dependencies between successive calls.

E.g.2: a thread stores its final result by attaching it to some global datastructure.

Typically, such dependencies are resolved by protecting critical code sections against multi-entrant execution:
Different threads can then execute such code without global ordering constraints.
The penalty on overall completion time might be low.

Resolving data dependencies (3)

Some data-dependencies **are essential** for the algorithm.

- True data dependencies **must** be honoured by correct scheduling (static or run-time dynamic schedules).
- Anti-dependencies might be (partially) resolved by duplicating storage locations.

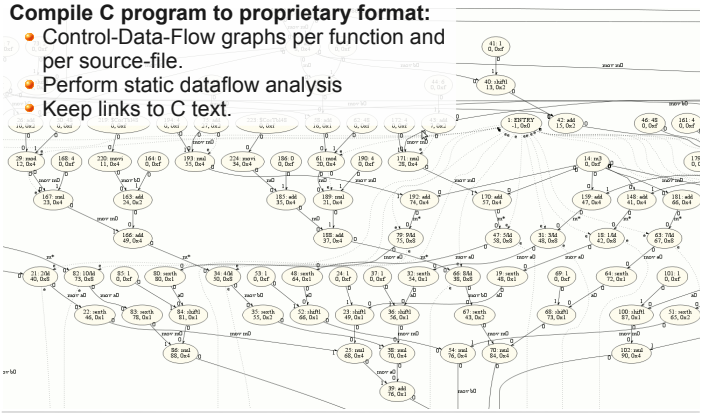
A proven method to simultaneously resolve anti-dependencies and run-time scheduling is the introduction of explicit (FIFO-buffered) communication channels, leading to **process networks**:
A 'producer' can write several copies of a variable into the channel before the 'consumer' reads them.

Otherwise, memory-mapped semaphores are used to control inter-thread communication.

Vector Fabrics' tooling (1: Compilation)

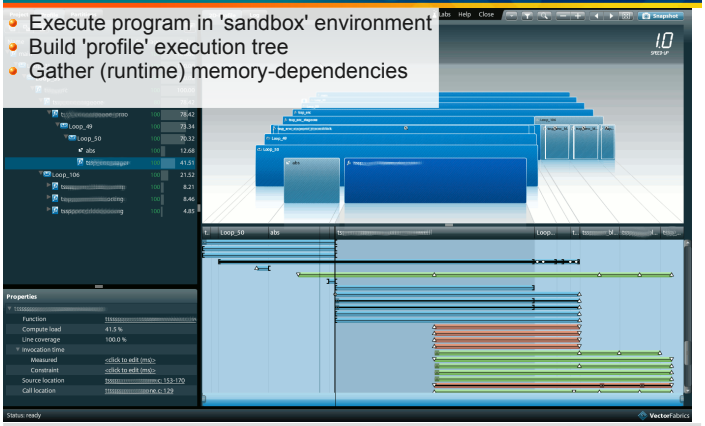
Compile C program to proprietary format:

- Control-Data-Flow graphs per function and per source-file.
- Perform static dataflow analysis
- Keep links to C text.



Vector Fabrics' tooling (2: Analysis)

- Execute program in 'sandbox' environment
- Build 'profile' execution tree
- Gather (runtime) memory-dependencies



Vector Fabrics' tooling (3: Xform, Output)

Code transformations to enable parallellism (target dependent):

- Insert Fork/Join of threads
- Insert Channel read/writes, Semaphore acquire/release
- Modify allocation of variables

Create output text:

- Generic C source code, for mapping to CPU's
- Verilog code for mapping of a thread to (FPGA-) hardware
- OpenCL for threads mapped to GFX hardware??

Conclusions

- C is a relatively simple programming language with mature and advanced compilation technology.
- Data-flow analysis is still a hard problem, in particular for applications with irregular behavior.
  (this is an application problem, not a language problem)
- Tooling for creating parallellism, by automatic C-to-C transformations, is still in its infancy.

C-based tooling for parallellisation allows that:

- The application programmer creates sequential C code, which is easier and less error-prone.
- Tooling creates a target-dependent parallel output, analysed for safe behavior.

Questions?

Vector Fabrics