Parallelization of C code compilation to embedded system

Jos van Eijndhoven
jos@vectorfabrics.com

SiliconSouthWest: The Multicore Challenge
Sept. 14, 2010
Vector Fabrics

- Vector Fabrics is a high-tech startup developing tools for the design and implementation of multicore, multi-threaded applications and embedded systems.
- Founded in 2007, Eindhoven, The Netherlands
- Its first product “vfAnalyst” was recently released.
Scope of this presentation

- **C language** input: sequential specification of application functionality.
- Analyze for available parallelism
- Translate to multi-threaded behavior
- Create target-specific implementation
- Final thoughts...
Inter-thread data dependencies

Control flow

```
<table>
<thead>
<tr>
<th>Initialization code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>Thread 2</td>
</tr>
<tr>
<td>Thread n</td>
</tr>
<tr>
<td>Finalization code</td>
</tr>
</tbody>
</table>
```

Data flow

- fork
- join

Fine

Inter-thread dependencies might cause trouble...
Design effort in parallelization

**Data dependencies** limit the parallelization opportunities, and add complexity to the implementation:

- Except when parallelizing just small inner loops, finding dependencies by code inspection is error-prone. (pointer arithmetic, global variables, system/library calls, multiple calls to functions with different arguments, ...)
- Removing data-dependencies can be hard: might require complex algorithm/data-structure rewrites.
- Implementing the parallel solution takes target-specific effort. Distinguish two different design patterns:
  - *Loop distribution*: Partition loop body in pieces, keep index space
  - *Loop splitting*: Keep loop body, partition loop index space.
Loop distribution: “functional partitioning”

E.g. Call two functions in parallel
- Need good load balance - or -
- Map to different target
  (heterogeneous architecture)

Might need to synchronize data from production to consumption.
  (See 'channel' in a later slide)
Loop splitting: “data partitioning”

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

- Or implemented directly:

- Good for homogeneous targets with many threads (GPU's)
Vector Fabrics' tooling (vfAnalyst)

- Show workload and dependencies
- Quick 'what-if' parallelization scenarios
- Support for implementation (in progress)
Rewrite data dependency as channel (1)

```
int A[N][M];

while (..)
{ produce();
  consume();
}

produce()
{ for (i ...)
  for (j ...)
    A[i][j] = ...
}

consume()
{ for (i ...)
  for (j ...)
    ... = A[i][j];
}
```

Threaded:

```
Thread1: while (..)
  produce();

Thread2: while (..)
  consume();
```

Must obey dependencies:
- 'True' dependency: consumer must wait for valid data
- 'Anti' dependency: producer must wait with over-writing until after consumption
Rewrite data dependency as channel (2)

```c
int A[N][M];

while (..)
{ produce();
  consume();
}

produce()
{ for (i ...)
  for (j ...)
    A[i][j] = ...
}

consume()
{ for (i ...)
  for (j ...)
    ... = A[i][j];
}

Stream s;

Thread1: while (..)
    produce();
    consume();

Thread2: while (..)
    consume();
    produce()
    { for (i ...)
      for (j ...)
        write_int( s, ...)
    }

consume()
{ for (i ...)
  for (j ...)
    ... = read_int(s);
}
```
Create implementation (1): multi-core

**Generic multi-core software** thread implementation:
- Thread create/join, semaphores for synchronization, are provided by OS.
- Shared memory semantics:
  Rely on CPU HW cache coherency.
- Opportunity with vector-instructions (SIMD, such as SSE)

Relatively easy platform to program functionality.
Hard-to-predict timing behavior: rely on over-dimensioning
Map application into embedded system:
- heterogeneous hardware ('accelerators') and distributed memory
  - Map threads to different compute units
  - Map data to embedded memories
  - Employ channel communication model (?): efficient data transport, resolve cache-coherency issues

Complexities in data mapping, memory throughput. Manual creation of channels costs design effort.
Embedded system compiler flow

1. Create thread-level parallelism with synchronization
   - thread1
   - thread2
   - thread3
   - thread4
   - thread5

2. Re-allocate data to improve locality
   - thread1
   - thread2
   - thread3
   - thread4
   - thread5
   - Memory segment 1
   - Memory segment 2

3. Create target architecture description and mapping
   - CPU1
   - CPU2
   - IP-block
   - IP-block
   - peripheral
   - DDR
   - Local memory

Maintain executable software semantics
Microcontroller, accelerators and peripherals embedded in FPGA: flexible system-on-chip

- Distributed memories and point-to-point channel interconnects to achieve performance.
- Basic OS (no MMU)
- No manual tweaking of code, VHDL, linker-scripts, OS additions, ...
Synthesized embedded system: example2

- More powerful CPU and PC-style peripherals
- OS with virtual/physical memory protection
- OS device driver to map FPGA memory and channels into user application space.
Final thoughts

- C is a relatively simple programming language with mature and advanced compilation technology. It is well-suited for analysis and automated manipulation.
- Data-flow analysis is a hard problem, in particular for applications with irregular behavior.
- Tooling for creating multi-threaded parallelism, by automatic C-to-C transformations, is still in its infancy.

C-based tooling for parallelization allows that:
- The application programmer creates sequential C code, which is easier and less error-prone.
- Tooling helps to create a target-dependent parallel output, analyzed for safe behavior.
Questions?

Check [www.vectorfabrics.com](http://www.vectorfabrics.com) for a free demo
Create implementation (3): GPU

Generic software, map kernels to GPU (Graphics card):
- GPU's like to handle tens-of-thousands of threads.
- Limited facilities to implement data-dependencies
- Mapping of data to (card/cluster) local memories

Fine to handle small 'kernels' (inner loops),
Yields enormous compute power.
Vector Fabrics' tooling (in development)

Automate code transformations to enable parallelism:
- Insert Fork/Join of threads
- Insert Channel read/writes, Semaphore acquire/release
- Modify allocation of variables

Create output:
- Generic C source code, for mapping to CPU's
- Dedicated C source code, for C-to-RTL translation (mapping of a thread to (FPGA-) hardware)
- System infrastructure code for embedded systems (Interconnect, memory maps, OS configuration, ...)

*19 Sep. 2010*
vfStream: HW&SW IP

- VfStream library with C-language API
- Implements inter-thread streaming data (stream create, read, write, ...)
- Provides thread synchronization (sleep and wake-up)
- Has equivalent HW IP implementation, supporting point-to-point HW channels, or HW-to-SW channels.

HW-to-SW channels:
- Thread wake-up and sleep through interrupts
- Supports paged (virtual) memory systems
- User-mode to kernel-mode memory protection