Aalto-yliopisto
Perustieteiden
korkeakoulu

On Static Timing Analysis of GPU Kernels

Vesa Hirvisalo
Department of Computer Science and Engineering
Aalto University

14th International Workshop on
Worst-Case Execution Time Analysis
Madrid, Spain, 8th July 2014

Talk outline

Introduction to SIMT executed kernels

- Co-operating Thread Arrays (CTA)
- warp scheduling
- thread divergence

Static WCET estimation

- divergence analysis
- abstract warp creation
- abstract CTA simulation

An example

- based on a simple machine model

Introduction

Data parallel programming and accelerators

- we try to maximize occupancy of the hardware

GPGPU computing as an example

- heterogeneous computing
- we concentrate on the accelerator (GPU) side timing
- hardware scheduling essential

Launches

- Co-operating Thread Arrays (CTA)
- the computation is prepared on the host (CPU) side
- input data and a number of threads
- these are launched to the accelerator (GPU)

Example (1/2): a kernel

Consider the following code in a language resembling OpenCL (note the use of the thread identifier `Tid`):

```
__kernel TriangleSum(float* m, float* v, int c) {
  int d = 0;              /* each thread has its own variables */
  float s = 0;            /* s is the sum to be collected */
  int L = (Tid + 1) * c;
  for (int i = Tid; i < L; i += c) {
    if ((d % (Tid + 1) == 0)
      s += 1;
    if (d % 2)
      s += m[i];
    __syncthreads();      /* assuming compiler support */
    d += 1;
  }
  v[d-1] = s;
}
```

SIMT execution

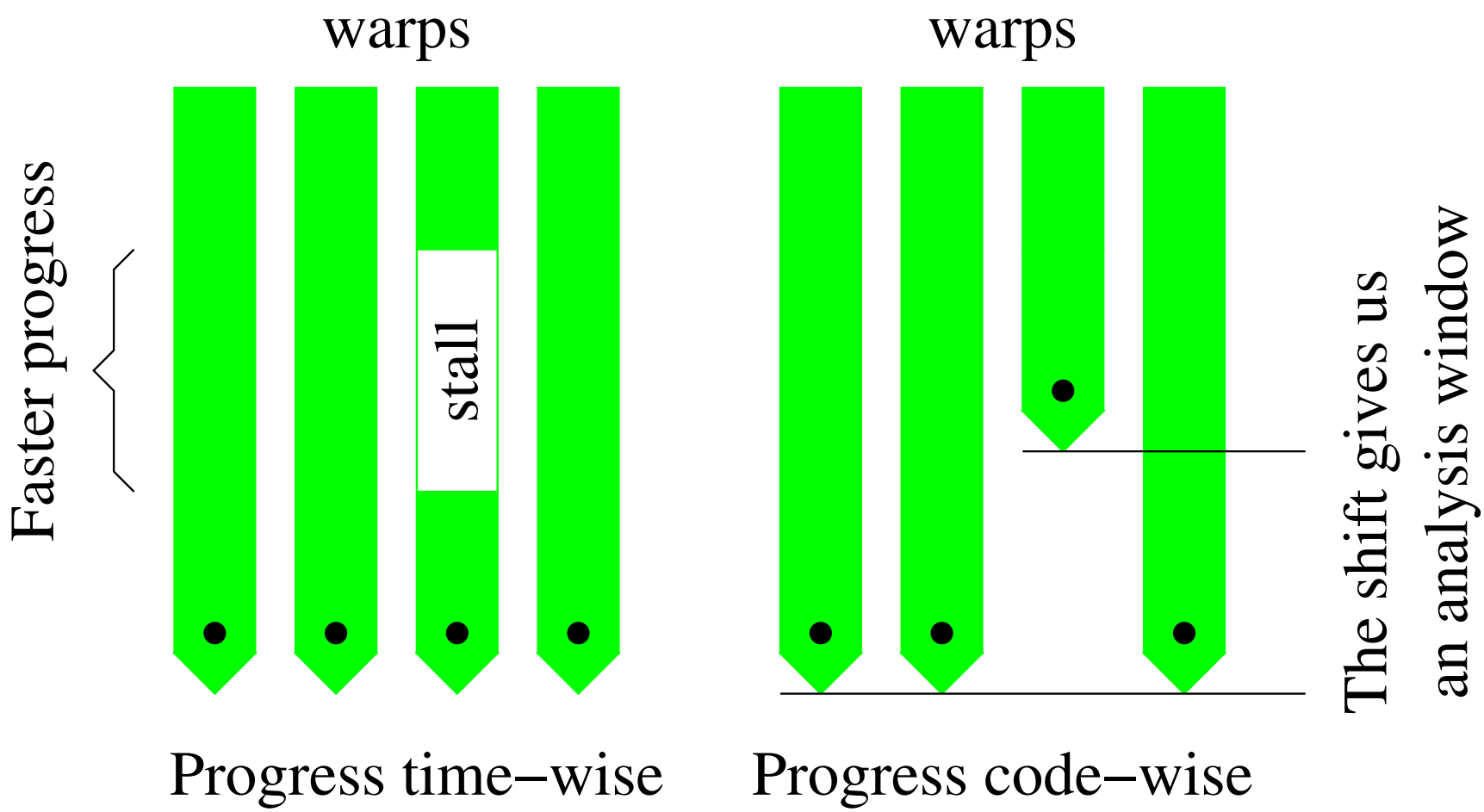
Threads are processed by computing units (CU)

- the following we assume: a single CU
  - able to handle a single *work group* (set of threads)

The threads are executed in warps

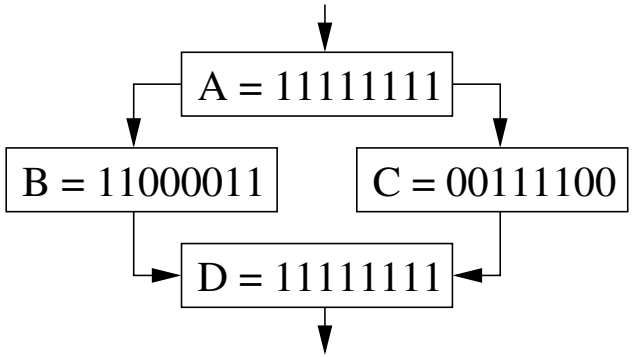
- warp width equals to the number of cores
  - The warp has a PC, which applies to all its unmasked threads
- SIMT = Single Instruction Multiple Threads
- there are typically several warps
- the *warp scheduler* makes the choice
  - round-robin is typical
  - the warp must be ready
  - if none – the execution stalls

Small analysis windows mean few paths

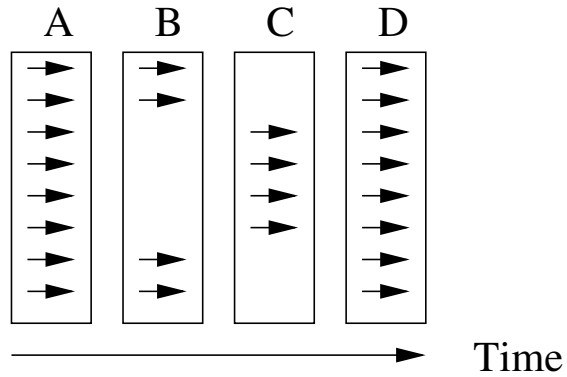


Divergence in execution

Program flow



Active threads



Initial stack contents

R–pc	Next–pc	Mask
–	A	11111111

stack top

Stack after divergence

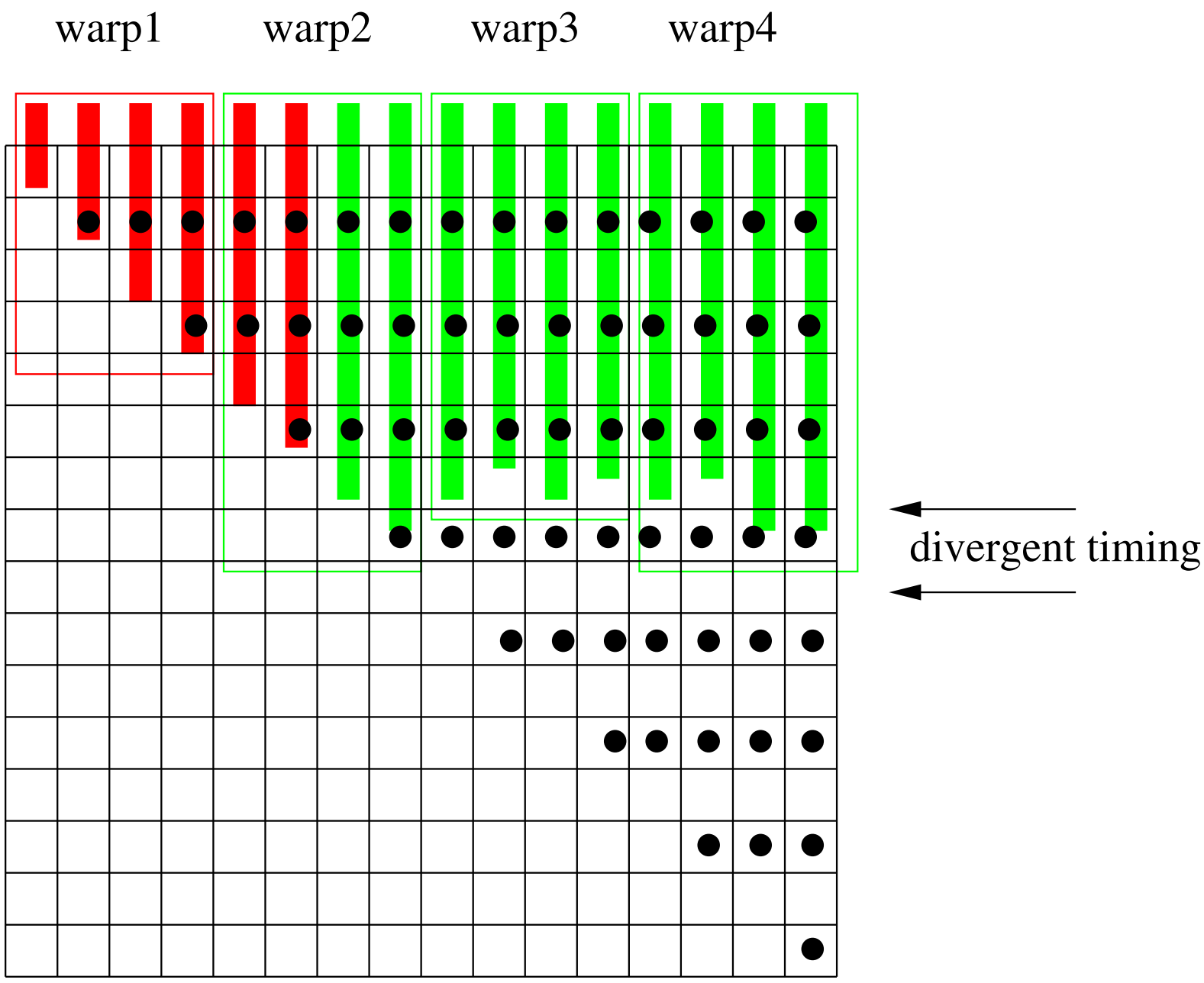
R–pc	Next–pc	Mask
–	D	11111111
D	C	00111100
D	B	11000011

After branch completion

R–pc	Next–pc	Mask
–	D	11111111
D	C	00111100

After reconvergence

R–pc	Next–pc	Mask
–	D	11111111



threads proceeding on matrix

WCET estimation

We define the total time spent in execution as

$$T_{exec} = T_{instr} + T_{stall}$$

Considering (structured) branching we have

$$T_{if\_else} = \begin{cases} T_{true\_branch} & \text{if all threads converge to true} \\ T_{false\_branch} & \text{if all threads converge to false} \\ T_{false\_branch} + T_{true\_branch} & \text{if threads diverge} \end{cases}$$

The warp scheduling hides the memory latencies.
On the worst case we have

$$T_{stall} = max(0, T_{memory} - N_{warps})$$

For loops, we use the time of the longest thread in the warp.

Static divergence analysis

We base our static divergence analysis on GSA. It uses three special functions: μ, γ, and η instead of the ϕ-function of SSA that it resembles:

- γ function is a join for branches. $\gamma(p, v_1, v_2)$ is v_1 if the p is true (or else v_2).
- μ function is a join for loop headers. $\mu(v_1, v_2)$ is v_1 for the 1^{st} iteration and v_2 otherwise.
- η is the loop exit function $\eta(p, v)$. It binds a loop dependent value v to loop predicate p.

We say that a definition of a variable is *divergent* if the value is dependent on the thread.

- if there are no divergent definitions for a branch predicate, we know the branch to be non-divergent.

Abstract warp construction

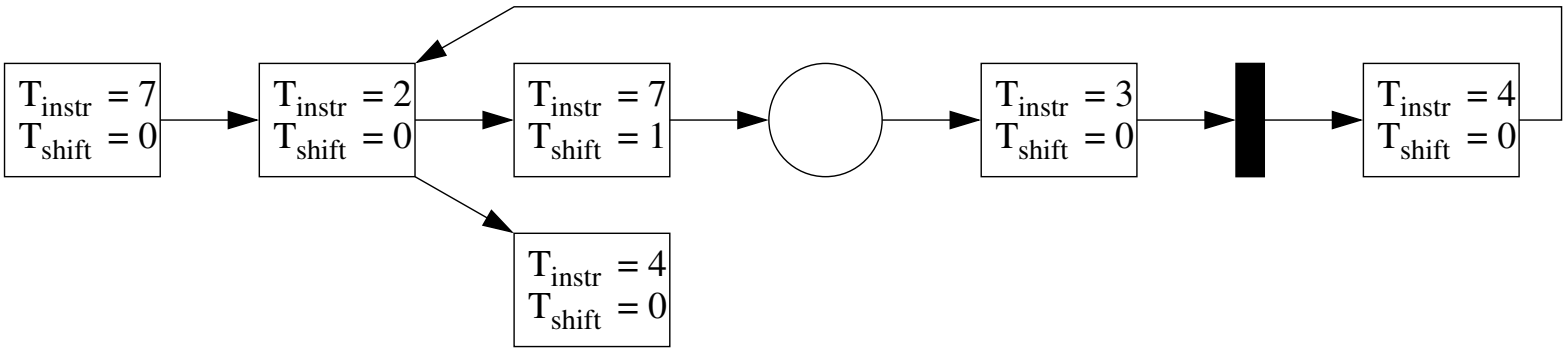
An *abstract warp* $A = (V, E)$ is directed graph. The nodes V have three node types:

- *time nodes* describe code regions with two values. T_{instr} is the upper bound of the instruction execution time consumed. T_{shift} is the upper bound of the variation of the instruction execution time caused by thread divergence.
- *memory access nodes* that mark places where memory access stalls may happen.
- *barrier nodes* that mark places where barrier synchronization must happen.

An abstract warp is constructed from the code in a recursive bottom-up way

Example (2/2): CTA simulation

Assuming a simple machine model (1 instr/cycle), we get the following abstract warp



The abstract CTA simulation

- begins from the leftmost node
- assuming warp width = 4, we have 4 warps

A final estimate $T_{WCET} = 804$

- a cycle accurate simulator gives 688 cycles

Conclusions

Static WCET estimation

- divergence analysis
- abstract warp creation
- abstract CTA simulation

We allow *some* divergence

- understanding divergence is essential
- uniform (non-divergent) execution is simpler

We demonstrated an approach

- we used a simple machine model
  - modeling real hardware is complex
- however, GPUs are rather predictable
  - they are designed for real-time (i.e., graphics)