Unified Execution Mode in a GPU-style Softcore

Peera Thontirawong, Prabhas Chongstitvatana
Department of Computer Engineering
Chulalongkorn University
Bangkok 10300, Thailand
prabhas@chula.ac.th

***Abstract*— A GPU-style processor has large amount of processing power on a given die compared to a general purpose processor. However, a Graphic Processing Unit must be executed in lock-step where a group of cores execute the same instruction. This constraint puts a real limitation on programming of a GPU. This work proposed a design of a processor that unifies the execution of Graphic Processing Units and a general purpose processor. The discussion of programming model of vectorised instructions and the extension to allow multi-cores to run independently is presented. The proposed design required only 3% additional resource compared to the original design. This design is suitable for embedded applications.**

Keywords—Graphic Processing Units; Softcore; General Purpose GPU; Programming GPU

I. INTRODUCTION

GPU has become "standard" in high performance computing. Early days of computing saw the availability of GPU allowed real-time applications such as video decoding [1]. As time progresses, GPU design has been more mature, there are attempts to make it more general purpose [2]. GPU has advantage of energy efficiency in terms of computing power per watt. It has been an important factor for media applications in mobile devices and its energy efficiency has been studied [3]. However, programming a GPU required special skill [4]. It is also difficult to do general purpose computing on a GPU. Therefore, GPU becomes a second processor to a general purpose processor. Having both CPU and GPU in one machine serves the purpose of running mixed work environment and media centric applications. This arrangement has become a *de facto* standard of PCs, notebooks and mobile phones.

We would like to unify two processors. The advantage would be that it eliminates interfacing between the two. Rather than two processors communicating by sharing a common memory, it becomes one processor (with many cores) with uniform memory. Programming would be more flexible and less idiosyncratic. Performance would be higher too (by the advantage of being on the same die).

Previously, we have designed a GPU-style softcore [5] intended for embedded applications (hence it is simple). It has the instruction set that is similar to a GPU. The execution is in a Single Instruction Multiple Data (SIMD) mode. All cores execute same instruction but perform on different data. To eliminate the memory access conflict, our design has all cores go through a special unit, called Local Data Store, which *serialises* multiple accesses to the memory. This is quite effective.

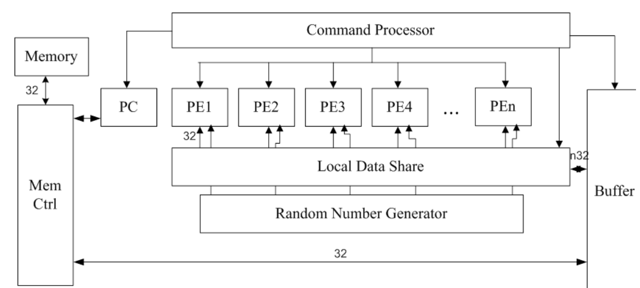
In order the make it a general purpose processor, this softcore should behave as a multi-core processor. The design has been extended by additional instructions. The execution cycle is changed to a Multiple Instruction Multiple Data (MIMD) mode [6]. How to reconcile the two instruction sets (one of a vectorised operation, another of control-oriented operation) is the challenge of this work.

II. GPU SOFTCORE

A. Procesor Organization

This is a simple GPU with four 32-bit cores. It contains four Processing Elements (PE or core). Each PE has 32 registers, one ALU and Local Data Store units (LDS). It also includes a 32-bit random number generator organised as 4 by 8-bit. It has 1Kx32 bits of memory. The memory is interfaced with processor through a buffer unit (BUF) connected to LDS. LDS communicates to all PEs in parallel. The processor operates in Single Instruction Multiple Data (SIMD) mode. That is, every PE runs the same program in synchrony. It has only one control unit, one Program Counter (PC) and one Instruction Register (IR). Its instruction has fixed size of 32 bits.

Fig. 1. The diagram of GPU organization.



B. *Instruction Set*

The Processing Elements perform Arithmetic and Logic with three-address format instructions, such as:

```
add r3 r1 r2
```

For branching, the command processor (control unit) performs these instructions:

```
jmp @ads
jz r @ads
jnz r @ads
```

The conditional jump instructions read the result stored in a register. In SIMD mode, the condition in that register of all PEs must be satisfied for a branch to be taken.

The Local Data Store unit transfers data between PEs and the main memory. It joins a narrow 32-bit bus, with a wide 32x4 -bit bus to PEs. It also performs *broadcast* from one of its register to all PEs.

```
ld/st ls @ads      load/store local-memory
ldr/str r          load/store LDS-PE
ldw @ads           load memory to all LDS
bc r ls            broadcast LDS-PE
```

III. Operations in SIMD Mode

This section will begin with describing how the vectorised operations work and then how to incorporate control-oriented operations on the same design.

To perform a control-flow, as all cores must be synchronised, they must be doing exactly the same work independently. The condition to transfer the control must be that all cores meet the condition. For example, an n-time loop.

```
  ldw @n     ; load Mem[n] to all LDS
  ldr 2      ; use reg 2
:loop
  ...        ; body of loop
  dec 2
  jnz 2 @loop
```

Register 2 of all cores are doing the same work and `jnz` performs test for zero on ALL register 2. Because of the restriction on Single Instruction execution, some conditional must be carefully written so that only the data is different (between cores) but the instruction that being executed must be the same. For example, move-if-true is such a conditional instruction.

```
mv_t r3 r1 r2
```

if R[3] is true then move R[2] to R[1]. The result will depend on the value of R[3] of each core but they all execute this instruction. In a program where each core computes a different point and it might terminate the loop at different time. To allow this different termination, we use move-if-true to update the value until the termination time (different between cores) but all cores continue to run until completion. The core that is already finished will not further update the value (to prevent the overflow). So, when the loop is complete, each core has x,y that terminates at the different time. The following code snippet illustrates this situation. This is the pseudo code and the assembly code for this operation.

```
while x*x + y*y < bound
  compute next x,y
```

```
:while
  ; (x*x + y*y < bound) stored to R[8]
  ...
  jz 8 @exit  ; jump all cores complete
  ...
  ; compute next x',y'
  mv_t 8 x x'    ; update x
  mv_t 8 y y'    ; update y
  jmp @while
:exit
```

So, for vectorised operations, SIMD mode is very good. It is also good for synchronised loop. But for general conditional (such as if..then) the program must be carefully written and when it is not synchronised, it is difficult and it wastes a lot of cycles (hence waste energy) to run until all cores come to completion.

IV. Operations in Multi-core mode

How to extend this GPU to run in MIMD mode? First and foremost, each core must have its own trace of execution. So, each core must contain its own program counter (PC) and Instruction Register (IR). The instruction that alters control-flow must be specific to individual core rather than having a synchronised execution over all cores. All vectorised arithmetic and logic instructions do not require any change when they are operated independently. Lastly, the access to memory must include independent load and store to registers of each core. This can go through Local Data Store. The additional instructions that allow the processor to run in MIMD mode will be described next.

Let us start with the memory access. The load/store instructions to LDS are:

```
ld k @ads     load Mem[ads] to LDS of core k
```

For MIMD mode, this instruction will affect only the core k. Other cores will take this instruction as no-operation.

```
ldr r         load LDS[k] to R[r]
str r         store R[r] to LDS[k]
```

Each core can execute these instructions independently.

```
st @ads k     store LDS of core k to Mem[ads]
```

Now, this instruction can cause memory access conflict when it runs in MIMD mode. Local Data Store unit must resolve this event. When LDS requests a write to memory, it must serialise the access. If there are more than one core request a write, then only one core is granted the request, all other cores must be stalled. And this will take care of LDS memory access in MIMD mode.

For control-flow instructions, a new mode must be created (beside synchronised execution). `x_jz` and `x_jnz` behave similar to a normal processor, they check only the condition of their own registers.

`x_jz r @ads` if R[r] == 0 then PC = ads

`x_jnz r @ads` if R[r] != 0 then PC = ads

One more instruction is `sync`. This is to *synchronise* all cores. It is important to be able to synchronise all cores when running a multi-core program.

`sync` wait for all cores to reach this point

To illustrate the extended processor running in MIMD mode, the Mandelbrot program will be used. It is slightly changed in order to run each core independently. This loop computes one pixel.

```
:while
  ; (x*x + y*y < bound) stored to R[8]
  ...
  x_jz 8 @exit     ; independent jump
  ...
  ; compute next x',y'
  mov x x'         ; update x
  mov y y'    ; update y
  jmp @while
:exit
```

The next example shows a complex if-then-else sequence that makes it difficult to write a vectorised code running on a GPU. This is a part of Compact Genetic Algorithm [7] that update the probability vector (p[i]) according to the pattern of two genes (a[i], b[i]) and their fitness (fa, fb). There are four cases to update p[i].

```
for i = 1 to k do
   if fa >= fb then
      if a[i] = 1 and b[i] = 0 then
         p[i] = min(1, p[i] + 1/n)
      if a[i] = 0 and b[i] = 1 then
         p[i] = max(0, p(i) - 1/n)
   else
      if a[i] = 1 and b[i] = 0 then
         p[i] = max(0, p[i] - 1/n)
      if a[i] = 0 and b[i] = 1 then
         p[i] = min(1, p(i) + 1/n)
```

To write a vectorised code, we decompose the operations into logical operations which are executed in lock-step, as shown below.

```
; select & update

xor 29 11 12  ; lb[t][i]= a[t][i] ^ b[t][i]
and 30 29 12  ; mb[t][i]= lb[t][i] & b[t][i]
lt 28 21 22
jnz 28 @check
xor 30 30 31  ; mb[t][i] ^= fa < fb ? 1 : 0

:check
; check upper bound
; lb[t][i]= ~ub[t][i]|mb[t][i] ? lb[t][i]:0
and 28 3 4
and 28 28 5
and 28 28 6
xor 28 28 31
or 28 28 30
and 29 29 28

:update
; 4-bit adder
...
and 29 30 4
xor 4 30 4
and 28 27 4
xor 4 27 4
or 27 29 28
...
xor 6 30 6
xor 6 27 6
```

In contrast to the vectorised code, a multi-core version is simply following the pseudo code. Here is a snippet of the code for the first case. Please note that at the end of the loop, we synchronise all cores.

```
:loop
...
ge 11 9 10    ; fa >= fb
jz 11 @else

x_jz 1 @L1
x_jnz 2 @ending
inc 0         ; a[i]==1 && b[i]==0, p[i]++
lt 11 12 0    ; 255 < p[i], check up-bound
mv_t 11 0 12  ; p[i] = 255
jmp @ending
:L1           ; next case ...
...
:ending       ; check terminate
sync
...
jnz 11 @loop
```

From the above code sequence, it becomes clear that this is more like an ordinary (non-vectorised) code. Each core will continue its own path without concerning other cores.

V. EXPERIMENTS

We ran two benchmark programs: matrix multiplication and Mandelbrot. Matrix multiplication is 4x4 and the program has fully unrolled the loop so it becomes essentially a straight line code. Mandelbrot calculation on the grid size 64x64 with fixed point arithmetic with 12 bits of fractional part. Both benchmarks were run in two modes: SIMD and MIMD. Table 1 reports the number of execution cycles required.

TABLE I. THE NUMBER OF CYCLES REQUIRED TO RUN BENCHMARKS

Program	SIMD	MIMD
Matrix Multiplication	1,406	1,406
Mandelbrot	15,350,074	13,476,882

Because matrix multiplication program does not contain any branch, the results from both modes are the same. For Mandelbrot program the MIMD mode is faster. This is due to the fact that in MIMD mode each PE can finish its loop independently while in SIMD mode all PEs have to synchronise by waiting for the longest computation to finish.

Adding the MIMD execution mode into a GPU involves a tradeoff between flexibility and resources. Each PE has its own control unit and the memory controller. The question is how much additional resource is needed and whether it worth the tradeoff? The design in [6] has been synthesized on the Field Programmable Gate Array, Xilinx Spartan3, XC3 S1500L. The amount of resource used is shown in Table 2. To measure the resource, we convert the synthesis result into the number of gates. If we assume that a register is equivalent to 8 gates, LUT is 6 gates, and a multiplier unit is 5,000 gates. The total gate count for SIMD GPU is 230,708 gates. Breaking down the detail of the resource used, the PE cores consume most resource, 96%, while the control unit is only 0.4%. So, duplicating the control unit to each core is not expensive. To understand why the control unit is very small, one has to consider that the core design is really simple, no pipeline, multi-cycle execution. Therefore the control unit is simple.

To add MIMD execution mode, we assume the control unit will be larger, 2 times for each core due to additional instructions and addressing modes. The memory controller also needs to be duplicated for each core as they can access the memory independently. The estimate result for this proposed design is shown in Table 2. The conclusion is that it is larger than the original design by only 3%. This is an excellent trade off.

VI. CONCLUSION

This work proposed a method to improve GPU-style processors in order to make them more flexible in programming. By extending instructions and allow each Processing Element to execute independently, the processor can perform similar to multi-core processors. The effect of the new mode of execution has been demonstrated. Although the proposed design uses more resource, it is shown that this additional resource is small, only 3%.

The resource for this design is available online at http://www.cp.eng.chula.ac.th/faculty/pjw/project/npu.htm. The site contains a simulator and an assembler, including benchmark programs.

REFERENCES

[1] G. Shen, L. Zhu, S. Li, H. Shum, Y. Zhang, “Accelerating video decoding using GPU,” IEEE Int. Conf. Acoustics, Speech, and Signal Processing, 6-10 April 2003, vol 4, pp. 772-775.

[2] J. Wang, A. Sun, Y. Li, H. Liu, “Programmable GPUs: New General Computing Resources Available for Desktop Grids,” Int. Conf. Grid and Cooperative Computing, Oct. 2006, pp. 46-49.

[3] J. Pool, A. Lastra, M. Singh, “An energy model for graphics processing units,” IEEE Int. Conf. on Computer Design (ICCD), 3-6 Oct. 2010, pp. 409-416.

[4] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, 2011, ISBN 978-0-13-138768-3.

[5] N. Thammasan, and P. Chongstitvatana, “Design of a GPU-styled Softcore on Field Programmable Gate Array,” Int. Joint Conf. on Computer Science and Software Engineering (JCSSE), 30 May - 1 June 2012, pp. 142-146.

[6] Chongstitvatana, P., "Putting General Purpose into a GPU-style Softcore," Int. Conf. on Embedded Systems and Intelligent Technology, Jan 13-15, 2013, Thailand.

[7] G. R. Harik, F. G. Lobo, D. E. Goldberg, “The compact genetic algorithm,” IEEE Trans. on Evolutionary Computation, vol.3, no.4, pp.287-297, Nov 1999, doi: 10.1109/4235. 797971.

TABLE II
BREAKDOWN OF THE SYNTHESIS RESULT OF THE ORIGINAL GPU AND THE PROPOSED DESIGN

	GPU	4 PEs	Control U	LDS & PC	Random	Memory Controller
Registers	4576	4232	44	268	32	0
4 input LUTs	12350	11620	92	593	1	44
Slices	8430	8035	56	300	16	23
Multiplier Units	24	24	0	0	0	0
Est. Gate	230708	223576	904	5702	262	264
Gate %	100.00	96.91	0.39	2.47	0.11	0.11
	GPU2					
Est. Gate	237828	223576	7232	5702	262	1056
Gate %	100.00	94.01	3.04	2.40	0.11	0.44