Subject: Re: type conversion in GPULIB
Posted by lecacheux.alain on Sat, 10 Mar 2012 11:22:15 GMT
View Forum Message <> Reply to Message

On 9 mar, 19:09, Michael Galloy <mgal...@gmail.com> wrote:
> On 3/9/12 3:39 AM, alx wrote:
>
>> On some computer equipped with a CPU/GPU combination (8 cores intel
>> and Tesla C1050), I have a fast ADC which repeatedly delivers large
>> int16 arrays to CPU memory. I would like to process them "in the fly"
>> by using the GPU and GPULIB software. Functions to convert float and
>> complex of various sizes are available in GPULIB.
>> Is it a way to convert int16 arrays to float arrays in GPU memory by
>> using GPULIB (I would like to get faster conversion and smaller bus
>> data transfer as well) ?
>> Thanks for any insight.
>> alx.
>
> Unfortunately, GPULib currently only supports floating point types.
> Right now, you would have to convert the ints to floats on the CPU
> before transferring to the GPU.
>
> There is no inherent need for this type restriction from CUDA, I would
> like to extend the types to include the various IDL integer types at
> some point.
>
> Mike
> --
> Michael Galloywww.michaelgalloy.com
> Modern IDL, A Guide to Learning IDL:http://modernidl.idldev.com
> Research Mathematician
> Tech-X Corporation
>
>

@Michael: thanks for your answer. But, I do not need for a full
implementation of integer calculous in GPU. I just would like to be
able to convert INT to FLOAT from inside the GPU. Indeed conversion in
CPU is slow and transfering floating numbers from CPU to GPU is
wasting bandwidth. I hope that a next release of GPULIB will provide
this functionality !

@all: regarding INT to FLOAT conversion efficiency in IDL, and trying
to optimize it, I found some unexpected results:
The testing computer was a 3 GHz bi-quad running IDL 8.1 and WIN64. I
checked that IDL thread pool was activated.
After some first initialisations:
IDL> N = 8*1024*1024 & x0 = indgen(N) & x = fltarr(N) & px =

Ptr_New(x)
each of the next five instructions was successively executed within a
FOR loop of 100 iterations. Therefore the test code was simply, in
case 1):

```
IDL> t = systime(/SECONDS)
IDL> for i=0,99 do x = fix(x0, TYPE=4)
IDL> t = (systime(/SECOND) - t)/100
```

Execution times (for one iteration) were measured by using IDL
profiler (2nd column) as well as by computing elapsed times and
averaging (3rd column).

```
                          profiler    elaps.
1) x = fix(x0, TYPE=4)   ->  24.7 ms.    26.3 ms.
2) x = float(x0)         ->  25.0 ms.    26.3 ms.
3) x[0] = float(x0)      ->  37.4 ms.    46.4 ms.
4) x[*] = float(x0)      ->  40.5 ms.    80.3 ms.
5) *px = float(x0)       ->  25.0 ms.    26.4 ms.
```

From where one can derive the following remarks:
Instructions 1) and 2) are equivalent, as expected;
Instruction 5) should have executed faster than 1) and 2) because
allocation of a new 'x' array at each step of the FOR loop is avoided
in 5), but timing remains the same.
While using instructions 3) or 4) (the "usual ways" in IDL to do not
re-allocate 'x' variable), everything get much slower;
some large overhead also appears during the FOR loop, not accounted
for by the profiler. Why ?
Finally, the best timing (~25 ms) lead to a rate of about 3.10^8
conversions by second, which is not well in accordance, I guess,
to effective multicore processing.
Are there some neglected effects due to memory caching ? Some other
thoughts for any optimization ?
alx.