GPU Microarchitecture Note Set 5—Interval Analysis

Note: Loosely based on Eyerman 09 ToCS a3, Fields 02 ISCA p.47, Fields 04 TACO p.272.

B		Number of threads per block.
G		Number of blocks in grid (launch).
N		Input size. (Depends on problem.)
L_ι		Latency of one iteration.
d_ι		Amount of data crossing chip boundary per iteration.
L_F	6 cyc	Latency of one floating-point operation.
L_M	400 cyc	Minimum global load latency for loads that hit memory.
L_2		Minimum global load latency for loads that hit L2 cache.
Θ_M	500 GB/s	Off-chip bandwidth.
θ_M		Off-chip throughput during execution of some kernel.
P		Number of threads in a launch.
ϕ	1 GHz	Clock frequency of GPU.
$\Theta_{\rm IF}$		FP instruction throughput.

Θ_{IM}	Load and store instruction throughput.
n_{IF}	Number of FP instructions executed in interval.
n_{IM}	Number of load and store instructions executed in interval.

Interval Analysis

Interval Analysis:

A method of modeling the execution of code on some device using repeating regions of execution (such as loop bodies).

Interval:

A well-defined region of execution, such as a loop body.

Goal of analysis is to find minimum number of threads, P, ...

... needed to achieve full performance.

The idea is to first compute the latency of an interval, L_ι ...

... and then compute the number threads needed to fully utilize ...

... resources such as thread dispatch and off-chip bandwidth.

Intervals

Analysis can use a single interval (considered first)...
... or several intervals (usually separated by barriers).

An interval is well chosen if:

It covers much of a program's execution time.

Conditions during each interval execution are similar ...
... for example, the number of cache misses is the same in each interval.

A set of intervals, say $L_{\iota 0}, L_{\iota 1}, \ldots$ is well chosen if:

They cover much of a program's execution time.

Conditions during each execution of $L_{\iota 0}$ are similar to each other ...
... conditions during each execution of $L_{\iota 1}$ are similar to each other, etc.

Latency of Interval

Interval Latency

Minimum execution time of interval on device of interest.

Symbol, L_ι.

L_ι can be …
… roughly estimated,…
… carefully estimated, …
… or measured.

Latency estimated based on a single thread or warp.

Example

Illustration of *interval* and *interval latency*.

For Code Fragment:

```
  for ( int h=tid; h<N; h += n_threads ) { dout[h] = din[h] + 1; }
```

A good interval is the loop body for one iteration: `dout[h] = din[h] + 1;`

Simplified SASS (CC 3.5) for Interval:

```
.Loop
I0:   LD.E R2, [R2];
I1:   FADD R7, R2, 1;
I2:   ST.E [R4], R7;
I3:   BRA .Loop

# Cycles:  0   1   2   3  ..  400 401 402 403 404 405 406 407 408 409 410
wp0:       [I0         ]      I1                          [I2            ] I3

           ! <---- One Interval -----------------------------------------> !
                   Interval Latency, L_{\iota}, = 410 cyc
```

Execution can't be less than L_ι . . .

. . . how many threads, P, will it take to use up off-chip bandwidth . . .

. . . or some other resource.

Use of Interval Latency for Code with One Interval, no Barriers

Let N denote the number of times the interval needs to be executed.

For example, operating on an N-element array, L_ι computes one element.

Let P denote the number of active threads during the interval.

For example, $P = 640$ in a launch of 10 blocks of 64 threads.

Then minimum execution time is: $t(N,P) = \dfrac{N}{P} L_\iota$.

Goal of analysis is to find largest P for which minimum is possible.

Number of Active Threads

Analysis based on the *number of active threads (P)* . . .
. . . which is the product of the number of active blocks and the block size.

In a launch of G blocks of B threads $P \leq BG$.

The number of active blocks is determined by a number of factors . . .
. . . such as SM capabilities and shared memory use. (See the CUDA documentation for details.)

In these notes we will assume that B and G are chosen such that all G blocks (BG threads) are active.

Illustration of Interval Latency Concept

Kernel below is to operate on N elements.

Interval is one iteration of loop.

Suppose a kernel of G blocks of B threads is launched . . .
. . . on a GPU with M SMs. . .
. . . and we know that G/M blocks per SM are active. . .
. . . *then* $P = BG$.

CUDA Code:

```
  for ( int h=tid; h<N; h += n_threads ) dout[h] = din[h] + 1;
```

Simplified SASS (CC 3.5) for Interval:

```
   LD.E R2, [R2];
   FADD R7, R2, 1;
   ST.E [R4], R7;
```

If we somehow conclude $L_\iota = 450\,\text{ns}$ . . .
. . . then total execution time $\geq \frac{N}{BG} 450\,\text{ns}$.

Work and Resources

Work [done in an interval]:
Data to be moved (d_ι), number of instructions to be executed $(n_{\iota I})$, or some other effect of instruction execution that uses a finite resource.

The amount of work will be computed per thread, such as d_ι, ...
... so the total work during the interval for P threads is Pd_ι.

Resource [needed for work in an interval]:
The thing provided to do a particular kind of work. A resource has bandwidth limitations, such as Θ_M for off-chip data bandwidth.

If P threads do Px_ι work (for some generic work x)...
... the corresponding resource is occupied for $\dfrac{Px_\iota}{\Theta_x}$ time.

Resources

Two broad types of resources considered here …
… data movement and instruction execution (issue and dispatch).

For Data Movement

Need to consider boundary being crossed:

SM to L2 cache: Use Θ_2 for bandwidth (and L_2 for latency).

L2 cache to DRAM: Use Θ_M for bandwidth (and L_M for latency).

Note: The value of Θ_M is available from NVidia APIs …
… whereas Θ_2, and the latencies must be measured.

Instruction Execution

Instruction execution is more tedious to account for because:

Must consider instruction type (such as load/store, 32-bit FP, etc.)

Must guess or count number of instructions of each type.

Instructions (work) vary by generation (though not by much).

Resources vary a great deal by device.

Major Types of Instructions (for these notes anyway):

32-bit FP (non-special): Work: n_{IF}, Bandwidth: Θ_{IF}.

Load/store: Work: n_{IM}, Bandwidth: Θ_{IM}.

Instruction Bandwidth Scope

In analysis use chip bandwidth, not SM bandwidth.

Typically memory bandwidths given for whole chip ...
... and instruction bandwidths given for one SM.

For resource consumption, use bandwidth for whole GPU chip.

For example, consider Θ_{IM} (load/store instructions):

In a CC 6.0 GP100 device there are 8 LS units per scheduler, ...
... $2 \times 8 = 16$ LS units per SM, ...
... and $\Theta_{\text{IM}} = 56 \times 16 = 896\,\text{insn/cyc}$ per chip.

Example

Find work and resource usage on a CC 6.1 GP104 device for the following code:

```
LD.E R2, [R2];
FADD R7, R2, 1;
ST.E [R4], R7;
```

Data movement:

For a GP104: $\Theta_{\mathrm{M}} = 320\,\mathrm{GB/s}$, $\Theta_2 = 900\,\mathrm{GB/s}$.

One load and one store, each of 4 bytes: $d_\iota = 2 \times 4\,\mathrm{B}$.

Assuming sufficient L2 cache, use of off-chip data resource: $\dfrac{2 \times 4P\,\mathrm{B}}{\Theta_{\mathrm{M}}}$.

Based on request size, use of SM/L2 data transfer: $\dfrac{2 \times 32P\,\mathrm{B}}{\Theta_2}$.

Example, continued.

```
LD.E R2, [R2];
FADD R7, R2, 1;
ST.E [R4], R7;
```

GP104 Characteristics:

CC 6.1, Number of SMs: 20. $\Theta_{\rm IM} = 20 \times 32 = 640\,\text{insn/cyc}$, $\Theta_{\rm IF} = 20 \times 128 = 2560\,\text{insn/cyc}$.

Instruction Execution:

Two load/instructions: $n_{\rm IM} = 2\,\text{insn}$.

One FP instruction: $n_{\rm IF} = 1\,\text{insn}$.

Usage of dispatch: $P\left(\frac{n_{\rm IM}}{\Theta_{\rm IM}} + \frac{n_{\rm IF}}{\Theta_{\rm IF}}\right) = P\left(\frac{2\,\text{insn}}{\Theta_{\rm IM}} + \frac{1\,\text{insn}}{\Theta_{\rm IF}}\right) = P\left(\frac{2}{640} + \frac{1}{2560}\right)$ cyc.

Analysis of Intervals Without Barriers

Consider an interval with time L_ι and data use d_ι ...
... on a system with $\Theta_{\rm M}$ ...
... in a launch with P active threads.

Amount of time resource is in use per interval: $P\frac{d_\iota}{\Theta_{\rm M}}$.

Then: $t(P) \geq \begin{cases} \frac{N}{P}L_\iota & \text{if } L_\iota > P\frac{d_\iota}{\Theta_{\rm M}} \\ N\frac{d_\iota}{\Theta_{\rm M}} & \text{otherwise.} \end{cases}$

Based on this time is minimized at $P = L_\iota\frac{\Theta_{\rm M}}{d_\iota}$ ...
... increasing P further has no benefit.

Bounds can be obtained using other resources ...
... performance is determined by the smallest P.

Interval Analysis Assumptions

Assumption:

> A resource can be fully utilized during an interval . . .
> . . . without increasing the thread latency above L_ι.

In practice thread latency does increase, . . .
. . . and so P is underestimated.

The assumption is closer to reality . . .
. . . when there are more warps per scheduler.

The assumption does not hold when barriers are used . . .
. . . (see or wait for) material to be added further below.

Analysis Steps

Compute Latency (This is the hardest part.)

Use code generated by CUDA toolchain ...
... or based on assumptions about such code ...
... or just measure latency.

Find the minimum number of threads for each kind of work.

The execution time is then $\frac{N}{P} L_\iota$.

If Too Slow

Re-code to reduce L_ι.

Exploit re-use to reduce data.

Re-code to reduce number of instructions.

Example: Compute L_ι for a simple loop.

Compute for a CC 6.1 device with:

8 LS per scheduler, 32 FP32 per scheduler.

$L_M = 400\,\text{cyc}$ and $L_F = 6\,\text{cyc}$.

```
.Loop
I0:   LD.E R2, [R2];     // t_is = 0.           t_re = L_m  =  400.
I1:   FADD R7, R2, 1;    // t_is = 400. (R2). t_re =  400  +  L_f  =  406.
I2:   ST.E [R4], R7;     // t_is = 406. (R7).
I3:   BRA .Loop          // t_is = 410.
```

```
# Cycles:  0   1   2   3  ..  400 401 402 403 404 405 406 407 408 409 410
wp0:       [I0         ]      I1                      [I2           ] I3
```

Total time: $L_\iota = 410\,\text{cyc}$.

Example: Resource usage for the simple loop.

Suppose

The loop operates on a total of N elements.

The loop is to run on a M2200 CC 5.2 device.

$\Theta_M = 88.1\,\text{GB/s}$, $\phi = 1.04\,\text{GHz}$

$M = 8$.

Work Performed by Loop:

Data: $d_\iota = 2 \times 4\,\text{B} = 8\,\text{B}$. (Four bytes for load and store.)

Limit $P = 408\,\text{cyc}\frac{88.1\,\text{GB/s}}{8\,\text{B}}\frac{1}{1.04\,\text{GHz}} = 4320$.

Load/store instructions (one of each): $n_{\text{IM}} = 2$.

Regular FP instruction (the `FADD`): $n_{\text{IF}} = 1$.

Control flow: $n_{\text{IC}} = 1$.

Example, Find L_ι for Actual SASS code.

Complete machine language code (SASS) for region:

```
.L_2:  // Note: .L_2 is a line label.
I0 :       MOV R2, R6;                    // $t_{\rm is} = 0$.        $t_{\rm re} = 6$
I1 :       MOV R4, R8;                    // $t_{\rm is} = 1$.        $t_{\rm re} = 7$
I2 :       LD.E R2, [R2];                 // $t_{\rm is} = 6$ (R2).   $t_{\rm re} = 406$.
I3 :       IADD32I R8.CC, R8, 0x4;        // $t_{\rm is} = 10$ (R8).  $t_{\rm re} = 16$
I4 :       MOV R5, R9;                    // $t_{\rm is} = 11$        $t_{\rm re} = 17$
I5 :       IADD32I R0, R0, 0x1;           // $t_{\rm is} = 12$        $t_{\rm re} = 18$
I6 :       ISETP.GE.AND P0, PT, R0, R11, PT;// $t_{\rm is} = 18$ (R0)   $t_{\rm re} = 24$
I7 :       IADD.X R9, RZ, R9;             // $t_{\rm is} = 19$        $t_{\rm re} = 25$
I8 :       IADD32I R6.CC, R6, 0x4;        // $t_{\rm is} = 20$        $t_{\rm re} = 26$
I9 :       IADD.X R3, RZ, R3;             // $t_{\rm is} = 26$  (CC)  $t_{\rm re} = 32$
I10:       FADD R7, R2, 1;                // $t_{\rm is} = 406$ (R2)  $t_{\rm re} = 412$
I11:       ST.E [R4], R7;                 // $t_{\rm is} = 412$ (R7)
I12:  @!P0 BRA `(.L_2);                   // $t_{\rm is} = 416$       Next iter: 417
```

```
# Cycle  0   1   2 ..5   6   7   8   9   10  11  12  .. 18  19  20  .. 26 .. 406 .. 412 413 414 415 416
wp0:     I0  I1          [I2          ]  I3  I4  I5      I6  I7  I8      I9     I10    [I11          ] I13
```

Based on analysis above: $L_\iota = 417$ cyc.

Hand-Analyze a Matrix/Vector Kernel (MXV-LCS)

Note: *MXV-LCS* → Matrix × Vector, Load, Compute, Store.

```
for ( int h=start; h<stop; h += inc )
  {
    Elt_Type vout[M], vin[N];
    for ( int r=0; r<M; r++ ) vout[r] = 0;
    for ( int c=0; c<N; c++ ) vin[c] = d_app.d_in[ h * N + c ];
    for ( int c=0; c<N; c++ ) for ( int r=0; r<M; r++ ) vout[r] += d_app.matrix[r][c] * vin[c];
    for ( int r=0; r<M; r++ ) d_app.d_out[ h * M + r ] = vout[ r ];
  }
```

Goals:

Perform analysis for a CC 5.1 device.

Perform analysis using assumed instructions for code above.

Analyze minimum number of threads for given N and M.

Compare predictions to measurements.

Find square matrix size (N) that requires fewest threads.

CC 5.1 Characteristics

For the CC 5.1:

Assumed SASS Code

We expect that the compiler will generate code like this:

```
L0:    LDG.E R16, [R14];
L1:    LDG.E R12, [R14+0x4];
L2:    LDG.E R11, [R14+0x8];
...
F0:    FFMA R13, R16, c[0x3][0x4], RZ;
F1:    FFMA R17, R16, c[0x3][0x24], RZ;
F2:    FFMA R18, R16, c[0x3][0x44], RZ;
F3:    FFMA R19, R16, c[0x3][0x64], RZ;
F4:    FFMA R21, R16, c[0x3][0x84], RZ;
F5:    FFMA R22, R16, c[0x3][0xa4], RZ;
F6:    FFMA R14, R16, c[0x3][0xc4], RZ;
F7:    FFMA R15, R16, c[0x3][0xe4], RZ;
F8:    FFMA R13, R12, c[0x3][0x8], R13;
...
S0:    STG.E [R4], R2;
S1:    STG.E [R4+0x4], R14;
S2:    STG.E [R4+0x8], R13;
```

Estimation of Interval Latency

Possible execution on a 5.1 device.

```
Time  0   1   2   3   4 .. 7   8 .. 11   ... 400 401 402 403 404 405 406 407 408
wp0: [L0            ][L1     ][L2     ]      F0  F1  F2  F3  F4  F5  F6  F7  F8
```

Accounting only for the load, multiply/add, and store instructions we get

$$L_\iota(K_0) = L_\mathrm{M} + MN + M$$

if $N \geq L_\mathrm{F}$ and $L_\mathrm{M} \geq N/2$, where K_0 is a shorthand name for the modified kernel and its assumed code. For values $M = N = 16$ and the latencies above we get

$$L_\iota(K_0) = 400 + 256 + 16 = 672\,\text{cyc.}$$

Off-Chip Data Bandwidth Limit

Once the latency for an interval is computed, it is possible to bound the number of threads that can be executed in parallel by using resource constraints. For example, let d_ι denote the amount of off-chip data read or written by a thread during one iteration and let Θ_M indicate the off-chip bandwidth. Let P indicate the number of threads in a launch. The amount of data read during the interval is Pd_ι, the data throughput would be $\frac{Pd_\iota}{L_\iota}$, solving $\frac{Pd_\iota}{L_\iota} = \Theta_M$ for P yields $P = \frac{L_\iota}{d_\iota/\Theta_M}$, the number of threads needed to saturate off-chip bandwidth.

For our matrix/vector kernels, $d_\iota = 4(M+N)\,\mathrm{B}$. Then

$$P = \frac{(L_\mathrm{M} + MN + M)\,\mathrm{cyc}}{\frac{1}{\Theta_M}4(M+N)\,\mathrm{B}}.$$

For $N = M$ (square matrices) we have

$$P = \frac{(L_\mathrm{M} + N^2 + N)\,\mathrm{cyc}}{\frac{1}{\Theta_M}8N\,\mathrm{B}} = \frac{\Theta_M}{8}\left(\frac{L_\mathrm{M}}{N} + N + 1\right)\frac{\mathrm{cyc}}{\mathrm{B}}$$

for $N \geq L_\mathrm{F}$.

Matrix Size Minimizing Number of Threads

$$P = \frac{\Theta_M}{8}\left(\frac{L_{\mathrm{M}}}{N} + N + 1\right)\frac{\mathrm{cyc}}{\mathrm{B}}$$

for $N \geq L_{\mathrm{F}}$.

Notice that this expression has a minimum at $N = \sqrt{L_{\mathrm{M}}}$, or 20 for our default global memory latency. As N shrinks below 20 more and more threads are needed because each thread is loading less data but the iteration time, L_ι, can't fall below L_{M}. When N grows above 20 the computation time, N^2, grows quadratically while data grows linearly, so more threads are needed. Note that Θ_M is often given in GB/s while the latency is given in cycles. Cycles can be converted to seconds by dividing by the clock frequency, often denoted ϕ. In that case $L_\iota(K_0) = (L_{\mathrm{M}} + MN + M)\,\mathrm{cyc} = \frac{1}{\phi}(L_{\mathrm{M}} + MN + M)\,\mathrm{s}$.

Instruction Bandwidth Limit

Another limiter is instruction throughput. For that one must compute the time needed during an interval to issue the instructions. Let Θ_{IF} indicate the bandwidth of FP instructions and Θ_{IM} indicate the bandwidth of load and store instructions on a multiprocessor. Let n_{IF} and n_{IM} indicate the number of instructions of the respective type executed by a thread in the interval. Then the time for Q threads to issue for one interval is $Q(\frac{n_{\mathrm{IF}}}{\Theta_{\mathrm{IF}}} + \frac{n_{\mathrm{IM}}}{\Theta_{\mathrm{IM}}})$ and based on this instruction issue limit $Q = L_\iota / \left(\frac{n_{\mathrm{IF}}}{\Theta_{\mathrm{IF}}} + \frac{n_{\mathrm{IM}}}{\Theta_{\mathrm{IM}}}\right)$. Note that Q is the number of active threads on a multiprocessor, which is some multiple of the block size.

For K_0 we have $n_{\mathrm{IF}} = MN$ and $n_{\mathrm{IM}} = M + N$. Substituting these values and L_ι gives

$$\begin{aligned}
Q(K_0) =& L_\iota / \left(\frac{n_{\mathrm{IF}}}{\Theta_{\mathrm{IF}}} + \frac{n_{\mathrm{IM}}}{\Theta_{\mathrm{IM}}}\right)\\
&= (L_{\mathrm{M}} + MN + M) / \left(\frac{MN}{\Theta_{\mathrm{IF}}} + \frac{M+N}{\Theta_{\mathrm{IM}}}\right)\\
&= \left(L_{\mathrm{M}} + N^2 + N\right) / \left(\frac{N^2}{\Theta_{\mathrm{IF}}} + \frac{2N}{\Theta_{\mathrm{IM}}}\right)\\
&= (L_{\mathrm{M}}/N + N + 1) / \left(\frac{N}{\Theta_{\mathrm{IF}}} + \frac{2}{\Theta_{\mathrm{IM}}}\right)\\
&= \left(L_{\mathrm{M}}/N^2 + 1 + 1/N\right) / \left(\frac{1}{\Theta_{\mathrm{IF}}} + \frac{2}{N\Theta_{\mathrm{IM}}}\right)
\end{aligned}$$

For large N the interval time is dominated by the N^2 instructions so the number of threads is based on FP instruction bandwidth Θ_{IF}. When N is smaller load latency determines the thread count.

Comparison with a CC 6.1 Device

Does the interval analysis above explain the behavior of the `mxv_o_lbuf` kernel obtained on a CC 6.1 device?

The calculations below show that for a GTX 1080 the iteration latency would be 388.4 ns so to saturate the 320.3 GB/s bandwidth would require 971.9 threads or just 48.6 threads per SM. To saturate instruction issue assuming only N loads, N stores, and N^2 multiply/adds would require 268.8 threads per SM.

The performance counters show 1.2 instructions per FMA, or a total of $256 \times 0.2 = 51.2$ non-FMA instructions. We've already accounted for 32 loads and stores, so there are just 19 extra instructions. Assume that these add 19 cycles to latency (which means that they are before the first loads or after the first FMA). Including those extra instructions only slightly changes P and Q.

The execution on the GTX 1080 reaches a peak at 4 wp / SM or 128 thds per SM, which is consistent with this.

Comparison with a CC 3.5 Device

A similar analysis for the K20, a CC 3.5 device, finds 118 threads per SM are needed to saturate off-chip bandwidth. That is inconsistent with observed behavior where a configuration with 4 wps (128 threads) per SM yielded just 1/2 the performance of larger blocks. Inspection of the SASS code shows that dependent FMA instructions are closer together than L_F and so the latency of these instructions is exposed.

```
% Solution Calculations

% For a GTX 1080.  SM: 20.  Data 320.3 GB/s
%  Clock: 1.73 GHz
%  Theta_IF = 128
%  Theta_IM =  64

%   n_IF_1 = 16 * 16 = 256
%   n_IM_1 = 16 + 16 = 32
%   Extra: 256 * .2 = 51.  Assume same thpt as IF

%
%  N = 16
% Latency/iter: 400 + N^2 + N = 672 cyc = 388.4 ns
% Latency/iter (correct store II): 400 + N^2 + 2N = 688 cyc = 397.7 ns
% Data/iter: 4(16+16) = 128 B
% Theta_M = 320.3 GB/s
% P = 320.3 GB/s 388.4 ns / 128 B = 971.9
%   => 971.9 / 20 = 48.6 thds / SM
% Issue Time Ideal:
%  256/128 + 32 / 64 = 2.5 cyc
%  Q = 672 / 2.5 = 268.8  thds per SM

% Accounting for 1.2 I/op.
%  Latency/iter: 400 + N^2 + N + 19 = 691 cyc = 399.4 ns
%  P = 320.3 GB/s 399.4 ns / 128 = 999 = 50 thds / SM
% Issue Time based on 1.2 I/op (which is 51 insn total or 51-32= 19 non l/s)
%  (256+19)/128 + 32/64 = 2.648 cyc
%  Q = 691 / 2.648 = 261 thds per SM

% For a GTX K20c.  SM: 13.  Data 208.0 GB/s
%  Clock: .71 GHz
%  Theta_IF = 128
%  Theta_IM =  64

%   n_IF_1 = 16 * 16 = 256
%   n_IM_1 = 16 + 16 = 32
%   Extra: 256 * .2 = 51.  Assume same thpt as IF

%
%  N = 16
% Latency/iter: 400 + N^2 + N = 672 cyc = 946.5 ns
% Data/iter: 4(16+16) = 128 B
% Theta_M = 208.0 GB/s
% P = 208.0 GB/s 946.5 ns / 128 B = 1538
%   => 1538 / 13 = 118
% Issue Time Ideal:
```

```
%  256/128 + 32 / 64 = 2.5 cyc
%  Q = 672 / 2.5 = 268.8  thds per SM
% Issue Time based on 1.2 I/op
%  (256+51)/128 + 32/64 = 2.898 cyc
%  Q = 672 / 2.898 = 231.8 thds per SM
```

Hand-Analyze a Less Efficient Matrix/Vector Kernel (MXV-(LC)*S)

Note: *MXV-(LC)*S* → Matrix × Vector: Load, Compute, Load, Compute, ···, Load, Compute, Store.

```
__global__ void mxv_o_lbuf() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x,  num_threads = blockDim.x * gridDim.x;
  for ( int h=tid; h<d_app.num_vecs; h += num_threads ) {
      Elt_Type vout[M];
      for ( int r=0; r<M; r++ ) vout[r] = 0;
      for ( int c=0; c<N; c++ ) {
          const Elt_Type vin = d_app.d_in[ h * N + c ];
          for ( int r=0; r<M; r++ ) vout[r] += d_app.matrix[r][c] * vin;
        }
      for ( int r=0; r<M; r++ ) d_app.d_out[ h * M + r ] = vout[ r ];
    }}
```

Assume (unrealistically) that compiler **does not** schedule (rearrange) instructions.

Difference with MXV-LCS (from previous example):

An input vector component is loaded and used immediately ...
... rather than loading all in local space and then using them.

This *exposes* load latency (a bad thing) ...
... but reduces the number of registers needed per thread (a good thing?).

Goals

Find the minimum number of threads based on data and instruction bandwidth.

We expect that more threads will be needed.

Compare analysis with measured results from a GTX 1080 (CC 6.1).

Determine if savings in number of registers makes up for need for more threads.

Device Characteristics

Use $L_{\mathrm{M}} = 400\,\mathrm{cyc}$ for the latency of a load that misses all caches.

Use $L_2 = 200\,\mathrm{cyc}$ for the latency of a load that hits the level 2 cache.

Use $L_{\mathrm{F}} = 6\,\mathrm{cyc}$ for instruction latency of non-memory instructions.

Interaction with Level 2 Cache

The CUDA C code loads an element and uses it in M FMADD instructions, then loads the next element, etc. Using an element size of 4 bytes and a request or line size of 32 bytes, we would expect that $\frac{1}{8}$ of the loads would miss the L2 cache and $\frac{7}{8}$ would hit.

```
W0: [L  ]          F  F  ....  F  [L  ]          F  F  ....  F
    !--- L_m ----> ! -- M ----->  !--- L_2 ----> ! -- M ----->
    ! <--  One occurrence ----->  ! <-- 1st of 7 occur.-- -->
    ! <------------ First of N occurrences -------------------------->
```

Interval Time

The time for an iteration accounting for the loads and FMAs would be

$$\begin{aligned} L_\iota(K_1) &= [L_{\mathrm{M}} + M + (L_2 + M) + \cdots + (L_2 + M)] + [L_{\mathrm{M}} + \cdots \\ &= \frac{1}{8} N L_{\mathrm{M}} + \frac{7}{8} N L_2 + NM. \end{aligned}$$

Assuming that the stores issue at one per cycle the total iteration time is

$$L_\iota(K_1) = \frac{1}{8} N L_{\mathrm{M}} + \frac{7}{8} N L_2 + NM + M.$$

Data Use Limits

The expressions for the amount of data and issue time are the same as those used in the MXV-LCS example.

Considering square matrices $N = M$:

$$\begin{aligned}
P &= \Theta_M \frac{\left(\frac{1}{8}NL_{\mathrm{M}} + \frac{7}{8}NL_2 + N^2 + N\right)\ \mathrm{cyc}}{8N\,\mathrm{B}} \\
&= \Theta_M \frac{\left(\frac{1}{8}L_{\mathrm{M}} + \frac{7}{8}L_2 + N + 1\right)\ \mathrm{cyc}}{8\,\mathrm{B}} \\
&= 320.8\,\mathrm{GB/s} \frac{\left(\frac{1}{8}400 + \frac{7}{8}200 + 16 + 1\right)\ \mathrm{cyc}}{8\,\mathrm{B}} \\
&= 5609.4
\end{aligned}$$

This works out to 280 threads or 8 warps per SM.

That's substantially more but still easily attainable. That means the compiler can conserve per-thread registers. That doesn't save per-block registers because more threads per block are needed.

Example: `mxv_o_per_thd`

Difference with MXV-(LC)*S:

Each matrix/vector multiplication performed by M threads.

```
__global__ void mxv_o_per_thd() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x, num_threads = blockDim.x * gridDim.x;
  const int start = tid / M;  // First vector number computed by this thread.
  const int r = tid % M;      // Vector element computed by this thread.

  for ( int h=start; h<d_app.num_vecs; h += num_threads / M ) {
      Elt_Type vout = 0;
      for ( int c=0; c<N; c++ ) vout += d_app.matrix[r][c] * d_app.d_in[ h * N + c ];
      d_app.d_out[ h * M + r ] = vout;
}}
```

Goals:

Show the number of threads needed as expressions.

Compare with measured values for a GTX 1080 CC 6.1 GPU.

Interval Latency

Consider an iteration of the h loop:

```
Elt_Type vout = 0;
for ( int c=0; c<N; c++ ) vout += d_app.matrix[r][c] * d_app.d_in[ h * N + c ];
d_app.d_out[ h * M + r ] = vout;
```

Assuming that the `c` loop is unrolled, there will be N loads followed by N FMA instructions. Notice that the N FMA instructions are dependent on each other (since they are adding to `vout`), so the iteration latency includes the load latency plus N FMA latencies, and one issue time for the store:

$$L_\iota = L_{\mathrm{M}} + NL_{\mathrm{F}} + 1.$$

Data Limit

Each thread loads $4N$ B of data, but because M threads are accessing the same data item the contribution of one thread to the amount of data accessed is

$$d_\iota = 4(\frac{N}{M} + 1)\,\mathrm{B},$$

or 8 B when $N = M$and assuming that the L2 cache is effective.

The number of threads to saturate latency is:

$$\begin{aligned} P &= \Theta_M \frac{L_\mathrm{M} + NL_\mathrm{F} + 1\,\mathrm{cyc}}{8\,\mathrm{B}} \\ &= 320.8\,\mathrm{GB/s}\frac{(400 + 16 \times 6 + 1)\ \mathrm{cyc}}{8\,\mathrm{B}} \\ &= 11502 \end{aligned}$$

This works out to 575 threads per SM or 18 warps per SM (remembering that a 1080 has 20 SMs).

The number of threads is consistent with the data from MXV-LCS in which performance improvement for `mxv\_o\_per\_thd` starts leveling off between 16 and 20 warps.

```
Iter Latency:
 L_M + N L_f + 1
 400 + 16 6 + 1 = 497 cyc = 287.3 ns
Data: 8 B
 P = 320.3 GB/s 287.3 ns / 8 B = 11502 = 575 / SM = 18 wp / SM
```