ECE 508
Manycore Parallel Algorithms

Lecture 4: Joint Register and Shared Memory Tiling
Background: Shared Memory Tiling Not Enough

Convolutional neural network (CNN) layers
• can be transformed into dense matrix multiplications.
• Linear algebra libraries then use a range of techniques to obtain good performance,
• of which you can get about 60%
• using the shared memory tiling taught in ECE408/CS483.

You may enjoy applying what you learn here in this context, especially if you already have code from 408.
Objective

• To learn more advanced tiling techniques
  – used in ultra-high-performance dense linear algebra libraries,
  – by placing data tiles in registers to enhance available data access bandwidth to the compute units, and
  – by understanding tradeoffs in use of on-chip memory.
Registers are Fast But Per-Thread

GPU chips have both registers and shared memory. In the last lecture, we compared their characteristics.

Registers offer

• low latency and high throughput,
• but are private to each thread, so
• must be loaded serially by each thread, and
• register tiling requires thread coarsening.
Shared Memory is Slower But More Flexible

Shared memory offers

- comparable latency, lower throughput
  (but still much higher than global memory)
- visible to threads in a block, so
- can be loaded cooperatively, and
- does not require thread coarsening, but
- does need to be moved to registers before use.
Tile Differently in Different Dimensions!

The best option? 
Use both.

- Hardware **paths** are **separate**, and **throughputs** can be **combined**.
- To do so,
  - we typically **use both types of tiling**
  - for distinct dimensions of multidimensional data.
Review Dense Matrix Multiplication

Given two $W \times W$ matrices, $M$ and $N$,

- we can multiply $M$ by $N$
- to compute a third $W \times W$ matrix, $P$:

$$P = MN$$

In terms of the elements of $P$, matrix multiplication implies computing…

$$P_{ij} = \sum_{k=1}^{W} M_{ik} N_{kj}$$
Visualize the Computation

\[ P_{ij} = \sum_{k=1}^{W} M_{ik} N_{kj} \]

Graphically, imagine

- taking each element in a row of \( M \),
- multiplying it by the corresponding element in a column of \( N \), and
- summing up the products.

Do that for every row and every column to produce \( P \).
Computing Part of P Requires Part of M and N

Parallelize over outputs (elements of $P$). To speed up computation, we need to reuse values from global memory.

Consider a 4x4 tile (block) of $P$.
- Which part of M do we need?
  - 4 rows
- Which part of N?
  - 4 columns
Use Tiling to Boost Performance

**Rows and columns are large** (W).
Instead, **repeat** the following...
• one load per thread
• to bring **tile of M into shared memory**
• one load per thread
• to bring **tile of N** into shared memory
• **synchronize**,  
• **compute**, and
• **synchronize** again.

© Steven S. Lumetta, ECE508/CS508/CSE508, 2020-2021
Tiling with $T \times T$ Thread Blocks Gives $T \times$ Reuse

In the formula,

- **each value** of $M$ (and $N$)
- is **used** $W$ **times**.
- That’s the **theoretical limit on reuse**.

**Tiling with $T \times T$ thread blocks**

- provides **reuse of $T$**
- **for each value** in $M$ (and $N$).

(That’s a lab in 408.)
Input Values are Reused Across…

What is the source of the reuse?
• Threads (>1 thread using a value),
• computation within one thread (same thread for several computations), or
• both?

Reuse is only across threads!
Threads Use Each Value Exactly Once

Each thread uses
• a piece of a row of $M$, and
• a piece of a column of $N$.

Inner product is computed
• in loop of $T$ iterations
• using each value once.
Full Tile is Not Needed for Reuse of $T \times$

Consider the first iteration of the inner product loop.

Which data are used (by any thread)?

So … the reuse would be the same even if we read only those elements.
Full Tile is Not Needed for Reuse of $T \times$

The **takeaway**:  
- the **number of strips loaded**, $S$,  
- **is an independent** value,  
- a **tuning parameter**!

Keep that in mind.
Thread Blocks Need Not Be Square

Another point:

- The thread block need not be square.
- Here’s $T \times U$.

Another tuning parameter.

Why not use $S=1$?

Only few threads load, and Barrier between each tile.
Cost of Loading Smaller Input Tiles

*(more detail with S=1)*

In each iteration of the outer loop,
- only $T+U$ (of $TU$ total) threads load $M$ and $N$ elements *(divergence or load imbalance)*,
- synchronize,
- calculate $S$—that is, $1$—terms of the inner product,
- and synchronize again.

`__syncthreads()` is efficient, but putting one multiply-and-add between two barriers is going to hurt.
Can Do Much Better Than 408 Code

That’s where we were in early 2008.

Then along came Volkov and Demmel*

- **Treat GPU** as a **vector machine**, and
- **Apply knowledge** of numerical methods.
- **50% performance boost** on dense matrix multiply.
- **World record** at the time.
- **SC19 test of time award**—**still used** in NVIDIA’s dense linear algebra libraries today!

It’s Not Magic, It’s Application of Knowledge

• To many at the time, these results seemed like magic.
• But vector machines had been around for decades.
• The Cray supercomputers, for example.
• Volkov & Demmel reapplied knowledge to a new architecture.

What you learn in this class will also apply to future architectures.
The Secret of Success: Prepare Your Mind

“Dans les champs de l'observation le hasard ne favorise que les esprits préparés.” —Pasteur, 1854
(In the fields of observation, chance favors only prepared minds.)

• Prepare your mind:
  – study known architectures and algorithms,
  – think about important problems, and
  – make progress as you can.
The Secret of Success: Act Quickly

“Most great scientists know many important problems. … And when they see a new idea come up, … They drop all the other things and get after it.” —Hamming, 1986*

- Be at the right place at the right time.

- If your mind’s not ready, being there won’t matter.

Joint Tiling: First Pick a Dimension for Registers

How does joint tiling work?

- Let’s start here.
- We **need to choose** the register dimension.
- Let’s say **elements of M go into registers** (you can pick $N$ if you’d like in Lab 3).
- **Elements of P stay in registers**, too.
Use T Threads per Thread Block

- **Elements of M** are in registers, so they **cannot be shared** by threads.
- Thus **need a thread for each row** of our $T \times U$ computation tile.
- Could coarsen (>1 row per thread); more $N$ reuse but uses more registers.
- Let’s have **T threads**.
Use Thread Coarsening to Reuse M Values

M values are reused U times.

U > 1 requires thread coarsening.

• Each thread
  – computes U values of P, and
  – requires U registers to accumulate the partial sums.
Choose $U=16$ to Manage Register Use

**GPU resources provide**
- 10s of registers per thread
- (reg. file size / max. # of threads).

$U=16$ is a reasonable choice.
Choose T=64 for Reasonable Reuse Overall

Reuse of 16 is a bit low.

We can balance
• by using a larger value of T.
• Each value of N is reused T times.

Let’s choose T=64, which gives reuse similar to 32×32 shared memory tiling (the 408 code).
Use Shared Memory Tiling to Reuse N

What about S?

- We use shared memory tiling for N, so
  - threads load values collaboratively,
  - write them into shared memory, then
  - all threads use all values.
- Total inner product terms computed per tile is UTS.
- Each of the T threads computes and accumulates US inner product terms.
- US should not be too small, since we must synchronize twice for each tile.
Choose $S$ as $T / U = 4$

So what about $S$?

For $S = 1$,
- only 1 in 4 threads ($U/T$)
- loads a value of $N$.

Why not have all threads load?
Thus $S = T / U = 4$.

Requires more registers for $M$ (load in parallel to hide latency).
An Overview of Joint Tiling SGEMM

In each iteration,
• Thread block collaboratively loads an $S \times U$ tile of $N$ into shared memory ($T = US$; every thread loads one $N$ element; no divergence/idle threads).
• Each thread loads $S$ elements from $M$ into registers (together, these form a $T \times S$ tile of $M$).
• Synchronize.
• Each thread calculates $S$ steps for $T \times U$ elements of $P$.
• Synchronize again.

Each thread needs $U$ registers for $P$ and $S$ registers for $M$, so values of $U$ and $S$ are limited by the number of registers available.
Summary and Comments

These numbers from Volkov & Demmel for GTX280.

- $T = 64$: 64 threads, $64 \times$ reuse of $N$
- $U = 16$: 16 registers for $P$, $16 \times$ reuse of $M$
- $S = T/U = 4$
  - 4 loads per tile per thread from $M$
  - 1 collaborative load per tile per thread from $N$

- Our discussion gives you some insights as to why they work.
- For a deeper understanding, read the paper (link on class web page).

© Steven S. Lumetta, ECE508/CS508/CSE508, 2020-2021
A Comparative Analysis

Tiled SGEMM introduced in ECE408
- Each thread block computes $32 \times 32 = 1024$ results.
- Uses 12 kB on-chip memory (registers + shared memory).

Register/Shared-Memory tiled version of SGEMM
- Each thread block computes $64 \times 16 = 1024$ results.
- Uses only $5 \frac{1}{4}$ kB on-chip memory.
- Similar degree of reuse; $\sim 2 \times$ more efficient than tiled MM.

<table>
<thead>
<tr>
<th>Tiling algorithm</th>
<th># of reuse per data in M</th>
<th># of reuse per data in N</th>
<th># of data computed per block in P</th>
<th>(N) Shared memory usage per block</th>
<th>(M+P) Register usage per TB</th>
<th>Performance on GTX280 in GFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jointly tiled SGEMM</td>
<td>16</td>
<td>64</td>
<td>16x64</td>
<td>4x16x4 = 256 B</td>
<td>(64x4 + 64x16) x 4 = 5 kB</td>
<td>$\sim 430$</td>
</tr>
<tr>
<td>Shared-Memory Tiled SGEMM</td>
<td>32</td>
<td>32</td>
<td>32x32</td>
<td>32x32x4 = 8 kB</td>
<td>32x32x4 = 4 kB</td>
<td>$&lt; 300$</td>
</tr>
</tbody>
</table>

© Wen-mei W. Hwu, David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021
Coalescing Loads from Global Memory

One last question…

**What about coalescing memory accesses?**

Loading *tile of N*…
- **thread order good** already, and
- could use corner-turning.*

*Use of a transposed thread order to allow memory loads to coalesce when loading global to shared.

© Steven S. Lumetta, ECE508/CS508/CSE508, 2020-2021
May Need to Transpose One Input

What about loads from $M$?

Loading *tile of $M$*...

- non-contiguous, so *not coalesced*;
- *can not use corner-turning*
- as each row of tile loaded by one thread.

Need to transpose!
Data Layout for FORTRAN

The **C library APIs** reflect the need to **transpose one input matrix**.

FORTRAN, in contrast, uses a column major layout:

- **M** accesses are coalesced.
- **N** needs to be transposed.
- **P** may need to be transposed.
Performance Data: One Register for M or Several?

Lumetta’s implementations

\( T = 128, \ U = 16, \ S = 8, \) as given in lab code

1. Array for tile of \( M \) (\( A \) in code): read tile N, read tile M, synchronize, compute, synchronize

2. One register for tile of M: read tile N, synchronize, loop to read value of M and compute, synchronize

Executed on 6 September 2021 NOT using exclusive queue on a 4096×4096 matrices (Titan V GPU).

Time with array for tile of M:* 40.76 msec (1.686 TF/s)

Time with register for tile of M: 15.76 msec (4.360 TF/s)

*Much more competitive (4.076 TF/s) with \( T=64 \).
ANY QUESTIONS?