Unified memory

GPGPU 2015: High Performance Computing with CUDA
University of Cape Town (South Africa), April, 20th-24th, 2015

Manuel Ujaldón
Associate Professor @ Univ. of Malaga (Spain)
Conjoint Senior Lecturer @ Univ. of Newcastle (Australia)
CUDA Fellow @ Nvidia
Talk outline [28 slides]

1. State of art of technology [12]
3. Examples [8]
I. State of art of technology
A 2015 graphics card: Kepler/Maxwell GPU with GDDR5 memory
A 2017 graphics card: Pascal GPU with 3D memory (stacked DRAM)
The Pascal GPU prototype: SXM2.0 Form Factor

SMX2.0*: 3x Performance Density

(* Marketing Code Name. Name is not final).
Details on silicon integration

DRAM cells are organized in **vaults**, which take borrowed the interleaved memory arrays from already existing DRAM chips.

A logic controller is placed at the base of the DRAM **layers**, with data matrices on top.

The assembly is connected with through-silicon vias, **TSVs**, which traverse vertically the stack using pitches between 4 and 50 um. with a vertical latency of 12 picosecs. for a Stacked DRAM endowed with 20 layers.
Time to fill a typical cache line (128 bytes)

The most popular memory in 2015 is DDR3-1600, with RCD=11 and CL=11. These two latencies represent 27.5 ns. out of 30 ns., 91.6% of the total time.

We have been waiting more than 15 years for this chance, and now with TSVs in 3D it is real.
3D integration, side by side with the processor

Step 1: Partition into 16 cell matrices (future vaults)

Step 2: Gather the common logic underneath.

Step 3: Pile-up DRAM layers.

Step 4: Build vaults with TSVs

Step 5: Buses connecting 3D memory chips and the processor are incorporated.

Links to processor(s), which can be another 3D chip, but more heterogeneous:
- Base: CPU and GPU
- Layers: Cache (SRAM)

A typical multi-core die uses >50% for SRAM. And those transistors switch slower on lower voltage, so the cache will rely on interleaving over piled-up matrices, just the way DRAM does.

Typical DRAM chips use 74% of the silicon area for the cell matrices.
Using 3D chips to build a Haswell-like CPU

We have CPU, GPU and SRAM in different proportions within silicon die, depending on 8 available models:

And, in addition, we want to include some DRAM layers.
Intel already authored a research showing the best choices (*)

Axiom: DRAM is 8 times more dense than a SRAM.

Hypothesis: A core uses similar die area than 2 MB L3 (Ivy Bridge @ 22nm. fulfills this today if we left L2 aside).

Evaluation: 2 layers, with the following alternatives (all reached similar temperatures):

<table>
<thead>
<tr>
<th>Layer #1</th>
<th>Layer #2</th>
<th>Area</th>
<th>Latency</th>
<th>Bandwidth</th>
<th>Power cons.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 cores + 4 MB L3</td>
<td>Empty</td>
<td>1+0 = 1</td>
<td>High</td>
<td>High</td>
<td>92 W.</td>
</tr>
<tr>
<td>2 cores + 4 MB L3</td>
<td>8 MB L3</td>
<td>1+1 = 2</td>
<td>Medium</td>
<td>Medium</td>
<td>106 W.</td>
</tr>
<tr>
<td>2 cores</td>
<td>32 MB. DRAM</td>
<td>1/2+1/2=1</td>
<td>Low</td>
<td>Low</td>
<td>88 W.</td>
</tr>
<tr>
<td>2 cores + 4 MB L3</td>
<td>64 MB. DRAM</td>
<td>1+1 = 2</td>
<td>Very low</td>
<td>Very low</td>
<td>98 W.</td>
</tr>
</tbody>
</table>

Given the higher role played by latency, the last row is the winner: DRAM is the greatest beneficiary of 3D integration.

Today

GPU

GDDR5 Memory

GDDR5 250-350 GB/s

PCle 16 GB/s

CPU

DDR4 Memory

DDR4 50-75 GB/s
In two years

GPU

2.5D memory

CPU

DDR4

Memory stacked in 4 layers: 1 TB/s

NVLINK 80 GB/s

DDR4 100 GB/s
In four years:
All communications internal to the 3D chip
The idea: Accustom the programmer to see the memory that way

The old hardware and software model: Different memories, performances and address spaces.

The new API: Same memory, a single global address space.
Performance sensitive to data proximity.
II. Programming with unified memory
Unified memory contributions

Simpler programming and memory model:
- Single pointer to data, accessible anywhere.
- Eliminate need for cudaMemcpy().
- Greatly simplifies code porting.

Performance through data locality:
- Migrate data to accessing processor.
- Guarantee global coherency.
- Still allows cudaMemcpyAsync() hand tuning.
## CUDA memory types

<table>
<thead>
<tr>
<th></th>
<th>Zero-Copy (pinned memory)</th>
<th>Unified Virtual Addressing</th>
<th>Unified Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA call</td>
<td>cudaMallocHost(&amp;A, 4);</td>
<td>cudaMalloc(&amp;A, 4);</td>
<td>cudaMallocManaged(&amp;A, 4);</td>
</tr>
<tr>
<td>Allocation fixed in</td>
<td>Main memory (DDR3)</td>
<td>Video memory (GDDR5)</td>
<td>Both</td>
</tr>
<tr>
<td>Local access for</td>
<td>CPU</td>
<td>Home GPU</td>
<td>CPU and home GPU</td>
</tr>
<tr>
<td>PCI-e access for</td>
<td>All GPUs</td>
<td>Other GPUs</td>
<td>Other GPUs</td>
</tr>
<tr>
<td>Other features</td>
<td>Avoid swapping to disk</td>
<td>No CPU access</td>
<td>On access CPU/GPU migration</td>
</tr>
<tr>
<td>Coherency</td>
<td>At all times</td>
<td>Between GPUs</td>
<td>Only at launch &amp; sync.</td>
</tr>
<tr>
<td>Full support in</td>
<td>CUDA 2.2</td>
<td>CUDA 1.0</td>
<td>CUDA 6.0</td>
</tr>
</tbody>
</table>
Additions to the CUDA API

**New call:** `cudaMallocManaged(pointer, size, flag)`

- Drop-in replacement for `cudaMalloc(pointer, size)`.
- The flag indicates who shares the pointer with the device:
  - `cudaMemAttachHost`: Only the CPU.
  - `cudaMemAttachGlobal`: Any other GPU too.
- All operations valid on device mem. are also ok on managed mem.

**New keyword:** `__managed__`

- Global variable annotation combines with `__device__`.
- Declares global-scope migratable device variable.
- Symbol accessible from both GPU and CPU code.

**New call:** `cudaStreamAttachMemAsync()`

- Manages concurrently in multi-threaded CPU applications.
Unified memory: Technical details

The maximum amount of unified memory that can be allocated is the **smallest** of the memories available on GPUs. Memory pages from unified allocations touched by CPU are required to **migrate back** to GPU before any kernel launch.

The CPU cannot access any unified memory as long as GPU is executing, that is, a `cudaDeviceSynchronize()` call is required for the CPU to be allowed to access unified memory.

The GPU has **exclusive** access to unified memory when any kernel is executed on the GPU, and this holds even if the kernel does not touch the unified memory (see an example on next slide).
III. Examples
First example:
Access constraints

```c
__device__ __managed__ int x, y = 2;           // Unified memory

__global__ void mykernel()                     // GPU territory
{
    x = 10;
}

int main()                                     // CPU territory
{
    mykernel <<<1,1>>> ();

    y = 20; // ERROR: CPU access concurrent with GPU
    return 0;
}
```
First example:
Access constraints

```c
__device__ __managed__ int x, y = 2;           // Unified memory

__global__ void mykernel()                     // GPU territory
{
    x = 10;
}

int main()                                     // CPU territory
{
    mykernel <<<1,1>>> ();
    cudaDeviceSynchronize();                // Problem fixed!
    // Now the GPU is idle, so access to "y" is OK
    y = 20;
    return 0;
}
```
## Second example: Sorting elements from a file

### CPU code in C

```c
void sortfile (FILE *fp, int N) {
    char *data;
    data = (char *) malloc(N);
    fread(data, 1, N, fp);
    qsort(data, N, 1, compare);
    use_data(data);
    free(data);
}
```

### GPU code from CUDA 6.0 on

```c
void sortfile (FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);
    fread(data, 1, N, fp);
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();
    use_data(data);
    cudaFree(data);
}
```
A “deep copy” is required:

- We must copy the structure and everything that it points to. This is why C++ invented the copy constructor.
- CPU and GPU cannot share a copy of the data (coherency). This prevents `memcpy` style comparisons, checksumming and other validations.

```c
struct dataElem {
    int prop1;
    int prop2;
    char *text;
}
```
void launch(dataElem *elem) {
    dataElem *g_elem;
    char *g_text;

    int textlen = strlen(elem->text);

    // Allocate storage for struct and text
    cudaMalloc(&g_elem, sizeof(dataElem));
    cudaMalloc(&g_text, textlen);

    // Copy up each piece separately, including
    // new "text" pointer value
    cudaMemcpy(g_elem, elem, sizeof(dataElem));
    cudaMemcpy(g_text, elem->text, textlen);
    cudaMemcpy(&(g_elem->text), &g_text, sizeof(g_text));

    // Finally we can launch our kernel, but
    // CPU and GPU use different copies of "elem"
    kernel<<< ... >>>(g_elem);
}
Cloning dynamic data structures WITH unified memory

What remains the same:
- Data movement.
- GPU accesses a local copy of text.

What has changed:
- Programmer sees a single pointer.
- CPU and GPU both reference the same object.
- There is coherence.

To pass-by-reference vs. pass-by-value you need to use C++.

```c
void launch(dataElem *elem) {
    kernel<<< ... >>>(elem);
}
```
Fourth example: Linked lists

Almost impossible to manage in the original CUDA API.

The best you can do is use pinned memory:

- Pointers are global: Just as unified memory pointers.
- Performance is low: GPU suffers from PCI-e bandwidth.
- GPU latency is very high, which is critical for linked lists because of the intrinsic pointer chasing.
Linked lists with unified memory

Can pass list elements between CPU & GPU.
No need to move data back and forth between CPU and GPU.
Can insert and delete elements from CPU or GPU.
But program must still ensure no race conditions (data is coherent between CPU & GPU at kernel launch only).
IV. Final remarks
Unified memory: Summary

- Drop-in replacement for `cudaMalloc()` using `cudaMallocManaged()`.
- `cudaMemcpy()` now optional.
- Greatly simplifies code porting.
- Less Host-side memory management.
- Enables shared data structures between CPU & GPU
  - Single pointer to data = no change to data structures.
- Powerful for high-level languages like C++.
Unified memory: The roadmap. Contributions on every abstraction level

<table>
<thead>
<tr>
<th>Abstraction level</th>
<th>Past: Consolidated in 2014</th>
<th>Present: On the way during 2015</th>
<th>Future: Available in coming years</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Single pointer to data. No cudaMemcpy() is required</td>
<td>Prefetching mechanisms to anticipate data arrival in copies</td>
<td>System allocator unified</td>
</tr>
<tr>
<td>Medium</td>
<td>Coherence @ launch &amp; synchronize</td>
<td>Migration hints</td>
<td>Stack memory unified</td>
</tr>
<tr>
<td>Low</td>
<td>Shared C/C++ data structures</td>
<td>Additional OS support</td>
<td>Hardware-accelerated coherence</td>
</tr>
</tbody>
</table>
NV-Link: High-speed GPU interconnect

2014/15: Kepler
2016/17: Pascal
Final summary

Kepler is aimed to irregular computing, enabling the GPU to enter new application domains. Win: **Functionality**.

Maxwell simplifies the GPU model to reduce energy and programming effort. Win: **Low-power, memory-friendly**.

Pascal introduces 3D-DRAM and NV-Link. Win: **Transfers, heterogeneity**.

- **3D memory** changes memory hierarchy and boosts performance.
- **NV-Link** helps to communicate GPUs/CPUs in a transition phase towards SoC (System-on-Chip), where all major components integrate on a single chip: CPU, GPU, SRAM, DRAM and controllers.