Scaling in the Third Dimension: Communication Architectures and Memory Hierarchies in an Integrated 3D World

Luca Benini
DEIS Università di Bologna
Luca.benini@unibo.it

Thanks to: M. Facchini, P. Marchal (IMEC)
C. Seiculescu, G. De Micheli (EPFL)
S. Murali, A. Pullini (INoCs)
S. Mitra (Stanford)
I. Loi, A. Marongiu (UNIBO)
Applications trends

- Increasingly complex functions
- Highly distributed (multi-standard wireless)
- Widely ranging environments
- Plenty of configurations/customizations
- Tightening power budget
- Key enabler greener home appliances!
- Soft-RT constrained
- Focus on User Quality-of-Experience

Scalable
Adaptive
Green
Predictable
The Application Challenge

1 TOPS/W embedded platforms by 2015!
How are we going to do it?

...today: 10s of cores
The Era of “Power Limited Scaling”

Power trend

![Power Consumption Graph]

- Dynamic Power
- Gate-Oxide Leakage
- Sub-Threshold Leakage
- Technology innovations (e.g. high-k dielectrics)

Power density trend

![Power Density Graph]

- Leakage Power
- Dynamic Power

Technology innovations (e.g. high-k dielectrics)

[STM ASIC]

[Intel, Microsoft and Stanford]
Multi-core and Power

Cache

Large Core

Small Core

Power

Performance

Power = 1/4
Performance = 1/2

More Energy-efficient

Better power & thermal management
Multicore SoC “platforms”

- Microprocessor (→SMP cluster!) is the “master processor”
  - Not energy efficient for “number crunching”

- Domain-specific specific functions
  - Dedicated IPs for legacy functions and ultra-high energy efficiency
  - Domain specific processors (DSPs, VLIW, ASIPs) for complex data-intensive processing (e.g. graphics, multimedia standards)

- “Kitchen-sink” of standard IOs for maximum interface flexibility (chip reuse + platform derivatives)

- Lots of on-chip memories (caches, scratchpads, buffers)
Nomadik ST-Ericsson (n8820) Application Processor

**Memory System**
- Host port interface
- Multichannel DMA controller
- SD/SDIO/MMC Memory stick pro
- NAND flash/NOR flash controller
- DDR SDRAM controller
- Power management
- Timers
- Watchdog
- RTC
- System controller
- Security toolbox
- Interrupt controller
- PLLS

**HW Accelerators**
- Color LCD controller
- Display interfaces
- SDTV/HDTV output
- Camera interfaces
- Smart video accelerator
- Smart graphics accelerator
- Smart audio accelerator
- Smart imaging accelerator
- L2 cache
- Dcache
- Icache
- JTAG/trace

**Main Core**

**I/Os**
- USB2.0 OTG
- PC
- HSI
- MSP
- UART
- SSP
- IrDA
- Rotary encoder I/T keypad interfaces
- GPIO
SoC platform: Quo Vadis?

ITRS 2007 – SoC Consumer Portable
Critical analysis

- Complex custom accelerators have high NRE → only massive reuse justifies risk of a new design!

- Kitchen-sink reuse leads to “spaghetti” inteconnect

- Programming model is messy → limited flexibility or lots of platform knowledge needed

**Ultimately, not a scalable solution!**
Polycore (NoC) Platforms
... the path to 100s cores
Omogeneous processor arrays

Tilera’s TILE64
- 2TB/sec bisection BW
- P (tile) 170-300mW @ 600-1000MHz
- 40 Gbps IO BW
- 200 Gbps DRAM BW

Regular processor fabric for predictability & efficiency
Tilera’s “Gentle Slope” Programming Model

Gentle slope programming philosophy

- Facilitates immediate results using off-the-shelf code
- Incremental steps to reach performance goals

Three incremental steps

1. Compile and run standard C applications on a single tile

2. Run the program in parallel using standard SMP Linux models – pthreads or processes

3. Use stream programming using iLib – a light-weight sockets-like API

Communication-exposed programming!
A look into new GPUs... GeForce 8800

- **Programmable processing:** many cores with specialized functions
- 16 highly threaded SM’s, >128 FPU’s, 367 GFLOPS (265 Sustained), 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU, 90W(!)
Looking closer…

Multiprocessor

Per-Block Shared Memory (PBSM)

Registers

Processing Element 1

Processing Element 2

Processing Element M

Instruction Unit

Storage exposed programming!

Global Device Memory
Bottlenecks
... on the path to 1000 cores
Using Cache size to accommodate increasing thread traffic is VERY expensive – using BW can be cheaper!!

\[ \frac{T}{C^{1/\delta}} \approx B \quad \delta > 1 \ (2-3) \]

2x increased traffic drives 8x cache size (constant memory bandwidth)

4x increased traffic drives 64x cache size (constant memory bandwidth)
What about Embedded MPSoCs?

Frame rate constraint is getting too tight!
3D Integration roadmap
Coming to the rescue of communication starved 2D ICs

Through-silicon vias are at the technology bleeding edge today
Industry interest is growing:  http://www.emc3d.org/
Understanding TSV technology

- For a whole via of 50μm, delay is 16/18.5ps (SOI/bulk)
- For a 1.5mm horizontal link, delay is around 200ps
So far so good, but…

The area news are not so good!

- But TSV pitch is not so small
  - Limited by wafer alignment issues
  - Need large “landing pads” for TSVs
    ~10um pitches seem to be realistic

Manufacturing yield is insufficient

- Redundancy and configuration strategies are needed
- TSV budget for clock, power, thermal
TSV-tuned IOs enable DRAM power savings up to 60%.

- Significantly lower inter-chip communication costs enables the tight integration of logic with DRAM.

- Global benefits are application dependent.
  - E.g., QSDPCM benefits more than MPEG4 ‘cos it consumes less internal DRAM power.
3D boosts “internal” DRAM energy efficiency

- Multi-rank organization
- Ranks are vertically integrated
- Wide and short unidirectional TSV busses for inter-rank connectivity
- Master rank also features off chip IO interface
Architectural Interface: Memory controller

- 2D Data-parallel architecture

Main bottlenecks:
1. Front-end congestion
2. Scheduler scalability
3. Expensive physical channels
3D-multi channel interface

- Multiple logical channels onto multiple vertical physical channels

Slave port/s

- S1
- S2
- ... Sk

SoC front-end + Queue + Scheduler

Memory backend

PHY

CH1

CH2

CH3

Logic DlE

DRAM 1
3D-multi channel interface

- Multiple logical channels onto multiple vertical physical channels

Slave port/s

SoC front-end + Queue + Scheduler

Memory backend

Memory backend

Memory backend

PHY

CH1

CH2

CH3

DRAM 1
3D-multi channel interface

- Multiple logical channels onto multiple vertical physical channels

Slave port/s

SoC front-end + Queue + Scheduler

Memory backend

Memory backend

Memory backend

PHY

CH1

CH2

CH3

DRAM 2
3D-multi channel interface

- Multiple logical channels onto multiple vertical physical channels

**Advantage:** does not require functional changes to DRAM interface
Scalability bottleneck

- Single controller becomes a bottleneck even with many slave ports
  - All cores need to reach it! Even using NoC interconnect, the latency price is high
  - Internal management of multiple slave interface and many transaction queues creates complexity bottlenecks
- This approach does not exploit the possibility of fine-grain distribution of 3D vias
- Creates a single point of failure
Multiple 3D DRAM interfaces

- Relieves single-controller bottleneck
  - Cores have a “friendly neighbor” controller
- Memory is fully accessible to everybody
- Notion of **vicinity** in memory space
- Not without issues
  - Area cost is increased (some hw sharing of memctrl is lost)
  - Many points of entry in the memory dies. Need a regular pattern of memory access ports for “commodity” 3D RAM
Not only DRAM!

- Configurable SRAM fabric: enables process tuning and cost optimization
- Lightweight in-memory configuration support (AND-based switches)
- Multiple vertical ports for coupling with logic chips
Scalable & predictable 3D-platform

3D-Network on-chip

- Packet-based communication with QoS support (TDMA/priorities/regulated traffic)
- Architecturally scalable: more nodes, more bandwidth
- Physically scalable: segmented P2P links

Vertically Integrated main memory

- TSV main-memory communication from 10pJ/bit to 10fJ/bit
- $10^5$ interconnect density increase
- Latency “bottleneck” in horizontal communication $\rightarrow$ Priority/Bandwidth reservation for low-latency memory neighbor
Platform 2012: Architecture Template

- Synchronous Computing Domain. Redundant Grain
- SMP Cluster
- Voltage & Frequency Island. Isolatable

- Decoupled Domains
- Packet Based (NoC) regular Communication Infrastructure

[STM, CEA]
Programming
vertically integrated platforms
A Software viewpoint

• NUMA Architecture
• Fast access to vertical DRAM stacks
• Access to remote DRAM stacks is subject to high latency (increasing with distance)
• non-coherent L1 caches and scratchpads (SPM) at first level of memory hierarchy
3D-memory exposed programming model

A common pattern: **Bulk Synchronous Processing**

1. Large-scale applications as unions of mostly-data-parallel units of work (tasks)
2. Tasks exchange little or no data until a global synchronization barrier is reached
3. No defined task ordering between two barriers
4. Any modified data by a task is made globally visible after the next barrier
5. Any modified data that is shared between barriers must be explicitly annotated by the programmer

- BSP model reflected in successful parallel languages:
  CUDA (NVIDIA), OpenMP, Threading Building Blocks (Intel), STAPL
- With support for distributed Memory and GAS Machines
  Split-C, Titanium (Java dialect), Unified Parallel C
Standardized BSP programming: OpenMP

- Programming annotations to convey parallel regions to the compiler
- Barrier-based parallel programming
- Relaxed Consistency memory model
  - Threads are allowed to have their own temporary view of memory
  - Modified shared objects are published (made globally visible) after a global barrier
  - Critical sections to protect data shared between two barriers
- Consistency issues have to be dealt with at synchronization points
int A[4][6];
int b[8];

#pragma omp parallel sections
{
#pragma omp section
for (i = 0; i < n; i++)
    A[i][rand()] = foo();
#pragma omp section
for (j = 0; j < n; j++)
    B[j] = goo();
}

OpenMP provides means to specify parallel execution by mapping parallel tasks onto different processors.
int A[4][6];
int b[8];

#pragma omp parallel sections
{
  #pragma omp section
  for (i = 0; i < n; i++)
    A[i][rand()] = foo();
  #pragma omp section
  for (j = 0; j < n; j++)
    B[j] = goo();
}

"Green thread": high latency in accessing remote-allocated shared data
This breaks workload balancing and introduces bottlenecks!
Idea: Neighborhood Programming

- Language features to specify data placement onto local neighborhoods for every parallel thread

- In Global Address Space (GAS) machines this can be achieved through a memory allocation process that returns a unique address known by all threads
  1. Programmer annotates arrays for allocation on a given memory neighborhood
     - ..through the extension of the OpenMP with custom directives
  2. If most efficient placement is not known at compile time..
     - ..defer it to runtime (based on Application Profiling and inspection of access count information)
Different arrays can be distributed across several neighborhoods through the custom directive

```c
#pragma omp distributed(var [,mem_id])

int A[m];
#pragma omp distributed (A, <MEM_ID>)
float B[n];
#pragma omp distributed (B)
...
```

Placement onto a specific neighborhood can be either specified within the directive based on a (memory) ID..

..or decided at runtime based on the availability of profile information
The programmer hints the compiler that arrays A and b should be mapped onto different memory neighborhoods. The runtime environment decides actual mapping.
Data Partitioning

• Data parallel applications typically operate on **separate, non overlapping regions** of shared arrays
  
  – This is in accord with workload partitioning in OpenMP (**#pragma omp parallel for**)

• In OpenMP, typically shared arrays reside on the stack of the master thread → Slave threads are pointed to **master’s memory neighborhood**

• Accessing remote data through (non-coherent) L1 cache is subject to:
  
  – Misses
  – Flushes (to maintain a consistent view of shared data)
  – Congestion on the single-ported memory slave containing shared data

• To address these problems, arrays can be **partitioned in smaller chunks**

• Allocation of chunks to neighborhoods should be such that a maximum number of accesses go to **local neighborhood**
#pragma omp parallel for
for (i = 0; i < 4; i++)
    for (j = 0; j < 6; j++)
        A[i][j] = 1.0;

The iteration space is partitioned between 4 processors.
A simple example

```c
#pragma omp parallel for
for (i = 0; i < 4; i++)
    for (j = 0; j < 6; j++)
        A[i][j] = 1.0;
```

Array is accessed with the loop induction variables

Data space overlaps with iteration space. Each processor accesses a different tile
#pragma omp parallel for
for (i = 0; i < 4; i++)
    for (j = 0; j < 6; j++)
        A[i][j] = 1.0;

A simple example
The *split* clause

- We can inform the compiler that a distributed array is accessed in a similar fashion within a parallel region with the custom *split* clause
- It can be coupled with the *parallel* directive
  - it replaces the *shared* qualifier for the array

```c
{
    double a[8];
    #pragma omp distributed (a)
    ...
    #pragma omp parallel for split (a)
    for (i = 0; i < 8; i++)
        a[i] = foo();
}
```
The split clause

The base address is retrieved once for each thread, at the beginning of the parallel region.

The split clause

```c
double a[8];
```

```c
P0
main.omp_fn.0 (...) {
  double * base;
  base = GOMP_get_split_tile_base_addr
  (distvar_id, <tile>);
  for (i=0; i<2; i++) {
    (*base)[i] = foo();
  }
}
```

```c
Library code
Parallel program code
```

GOMP_get_split_tile_base_addr
(int dvar_id, int tile)
{
  double * base;
  base = tiles[dvar_id][tile];
  return (double *)base;
}
```

Then each array reference in the thread is resolved through this pointer.
Data Partitioning

• This access pattern allows an efficient partitioning
  – Low overhead for partition addressing
  – Benefits from the reduced number of accesses to the bus

But..

• ...it requires that every thread accesses a SINGLE partition

• Often real applications feature irregular/strided array access pattern → a thread may access multiple array partitions
  – Addressing difficulties, because at each array reference we need to check which partition we are accessing, and at which offset
#pragma omp parallel for
for (i = 0; i < 4; i++)
    for (j = 0; j < 6; j++)
        hist[A[i][j]]++;
Now processors need to access far-away neighborhoods (higher latencies), since they work on multiple tiles!!!
The **tiled** clause

- We can inform the compiler that a distributed array is accessed in a similar fashion within a parallel region with the custom **tiled** clause

- It can be coupled with the **parallel** directive (it replaces the **shared** qualifier for the array)

```c
{
    double a[8];
    #pragma omp distributed (a)
    ...
    #pragma omp parallel for tiled (a)
    for (i = 0; i < 8; i++)
        a[i] = foo();
}
```
The **tiled** clause

```c
double a[8];

main.omp_fn.0 (...) {
    double * base;
    for (i=0; i<4; i++) {
        base = GOMP_access_tiled_array (<offset>, <distvar_id>);
        (*base)[i] = foo();
    }
}
```

```c
GOMP_access_tiled_array (int offset, int dvar_id) {
    int tile;
    double * base;
    tile = offset/tilesize[dvar_id];
    base = tiles[dvar_id][tile];
    return (double *)(base + offset)
}
```

The address is checked at every array reference.
Efficient Placement of Array Tiles

• As shown, accessing far-away neighborhoods incurs increasingly severe latency penalties.
  – This calls for a placement strategy that minimizes remote references

• We coupled our compiler with tools that gather information on array access count during a profile rune, and automatically devise an efficient allocation scheme

• The outcome of the memory allocator is a header file containing the populated metadata referenced by the instrumentation compiler pass

• Simplest partitioning scheme is that in which we divide the array in as many tiles as processors, but..

• ...this choice may lead to non-optimal locality
Fine Partitioning

- Basic partitioning technique may fail to accommodate the maximum amount of thread-local references in local neighborhood due to the coarse size of tiles.

```c
#pragma omp parallel for
for (i = 20; i < N; i++)
    arr[i] = foo();
```
Fine Partitioning

- Basic partitioning technique may fail to accommodate the maximum amount of thread-local references in local neighborhood due to the coarse size of tiles.

```
#pragma omp parallel for
for (i = 20; i < N; i++)
    arr[i] = foo();
```

- By refining the granularity of partitioning, we obtain a closer matching of memory references and tile-to-memory mapping.
- Reduced number of remote references.
- Increased size of metadata → tradeoff to be found.

1. By properly tuning the granularity of partitioning, remote accesses can be completely removed, if the same location is not shared by threads.
Some Results

- Code is replicated onto each memory neighborhood (SPMD)

- Private data to each parallel thread is created/replicated onto local neighborhoods

- L1 cache only manages code and private data
  - *This eliminates data consistency issues*

- Shared data is managed through array distribution/partitioning
Setup

- The layout of processing tiles as on the right
- Processor hosting the master thread is the one with the highest ID
- Shared data resides on master core’s memory neighborhood
- Master core is in a central position in the floorplan
  - To reduce the maximum latency for slave threads

- Zero-load latencies to access remote neighborhoods are shown on the left
- Fast access to local neighborhoods is modeled with zero additional penalty (w.r.t. interconnect arbitration)
- Additional latency increases with the number of hops traversed to reach far-away neighborhoods
Allocation Policies

- **Baseline**: Shared arrays are entirely placed on the master core’s neighborhood.

- **Basic Tiling**: The program is annotated with our directives for memory neighborhood-aware partitioning. Arrays are partitioned in as many tiles as processors.

- **Fine-Grained Tiling**: Arrays are partitioned in finer-grained tiles to reduce the number of accesses to remote neighborhoods.
JPEG Decoding

- **Baseline**: Parallel thread access shared arrays in the master core’s neighborhood

- **Dynamic loop scheduling**: Iterations are assigned to threads in a FCFS fashion
- All threads access master core’s (ID 15) neighborhood → it is delayed and thus assigned less iterations
- Latencies encountered by threads to access shared data are non-uniform due to varying distance
- Huge overhead due to serialization of accesses to the single-ported memory slave (master core’s neighborhood)
JPEG Decoding

- **Basic Tiling**: Shared arrays are partitioned in as many tiles as cores. Tiles are allocated onto the neighborhood local to the core that mostly references it.
- **Baseline**: Parallel thread access shared arrays in the master core’s neighborhood.

  - A significant number of references is satisfied from local neighborhood → Congestion on single memory port is noticeably reduced.
  - Latencies encountered by threads to access shared data are much more balanced (still non-negligible, since remote references are numerous).
  - Master core is no longer delayed by multiple concurrent accesses to its local neighborhood → balanced workload.
JPEG Decoding

- **Basic Tiling**: Shared arrays are partitioned in as many tiles as cores. Tiles are allocated onto the neighborhood local to the core that mostly references it.

- **Fine-Grained Tiling**: Shared arrays are partitioned in eight times more tiles than cores. Tiles are allocated onto the neighborhood local to the core that mostly references it.

![Graph showing normalized execution time vs. processor ID]
Conclusions

- Applications are pushing toward the 1TOPS/W mark
- From 10 to 1000 processors in 5 to 6 years
- Modular, scalable architectures are the only way forward
- 3D integration is key for addressing the main memory scalability bottleneck
- We need to develop effective programming abstractions and tools to vertically stacked memory hierarchies
- Ultimately storage exposed (e.g. memory neighborhood) and communication exposed (e.g. streaming) programming will coexist
- We’ll be busy...
Walls

- Commodity DRAM organization is **NOT OK** for TOPS/W

  1TOPS, 1% to DRAM (4B) $\rightarrow$ 40GBPS $\rightarrow$ **24W only DRAM (min 12W with 3D)**

  - Over-fetch 8Kb from array for 128b (optimistic) transfer
  - Boosted WL for speed
Revisiting DRAM organization?

[Image of DRAM organization diagrams and bar chart showing speedup with different configurations: 2D, 3D, +wide bus, +true 3D.]

[Loh08]
Array Placement Directive

- Candidate arrays for distributed allocation are initialized through a compiler-inserted call to the custom library function

\[ \text{distributed\_malloc}(\text{int mem\_id, int size}) \]

- If the programmer provides a \textit{MEM\_ID} through the \texttt{distributed} directive, this is directly passed to the \texttt{malloc} function

- If an ID is not specified, the runtime environment checks for the existence of array access count retrieved during a profile run

  - The ID of the processor that accesses the array most frequently is passed to the \texttt{distributed\_malloc}
  
  - When no profile information is found, the ID of the calling processor is used (i.e. data is
Code Instrumentation

• When the compiler encounters a reference to an array annotated with the `tiled` clause, the memory access is instrumented to:
  
  – Compute the offset of current reference (w.r.t. the base address)
  
  – Identify the tile to which this element belongs to (`offset >> tilesize`)
  
  – Load the tile base address from metadata (`base = tiles[offset >> tilesize]`)

• Instrumentation → overhead
  
  – Shift operation
  
  – 1 additional level of indirection (lookup in metadata)