Integrating DMA capabilities into BLIS for on-chip data movement

Devangi Parikh
Ilya Polkovnichenko
Francisco Igual Peña
Murtaza Ali
5 Generations of TI Multicore Processors

- Keystone architecture
  - Lowers development effort
  - Speeds time to market
  - Leverages TI’s investment
  - Optimal software reuse

• 2003: Janus 130nm
• 2006: Faraday 65nm
• 2011: KeyStone I 40nm
• 2013/14: KeyStone II 28nm
• Future: KeyStone III

- ARM A15
- Multicore cache coherency
- 10G Networking

- ARM A8
- C66x fixed and floating point, FPi, VSPi
- Network and Security AccelerationPacs

- C64x+
- Wireless Accelerators
- 6 core DSP
TI 66AK2H12 SoC

- Keystone II architecture
- Cores
  - 4 ARM A15s at 1.0 GHz
    - 4 MB shared L2 cache
    - 32 G flops/s single precision and 8 G flops/s double precision
  - 8 C66x DSPs at 1.0 GHz
    - 64 kB L1 scratch / cache each
    - 1 MB L2 scratch / cache each
    - 128 G flops/s single precision and 32 G flops/s double precision
- Memory
  - 8 GB DDR3 DRAM (external)
  - 6 MB SRAM shared
- Interfaces
  - 2x Gigabit Ethernet ~ 100 MB/s
  - 4x SRIO ~ 400 MB/s
  - 2x Hyperlink ~ 1 GB/s
Development Philosophy

• User view
  – Embedded Linux running on the ARM
  – Standard GCC tool chain
  – Simply link to a TI provided library with an ARM callable API to accelerate applications using multiple ARM cores, DSP cores and processors as appropriate
  – Use TI provided tools and examples to write new applications and libraries which use multiple ARM cores, DSP cores and processors to accelerate performance

• Using multiple cores on a single processor
  – OpenMP for shared memory parallelization across ARM cores
  – OpenCL or OpenMP Accelerator for heterogeneous acceleration with multiple DSP cores

• Using multiple processors
  – Open MPI over Ethernet, SRIO or Hyperlink
ARM + OpenCL DSP Acceleration

**Data parallel**
- A kernel is enqueued
- OpenCL divides into N workgroups
- Each workgroup is assigned a core
- After all workgroups finish a new kernel can be dispatched

**Task parallel**
- A task is enqueued
- OpenCL dispatches tasks to cores
- OpenCL can accept and dispatch more tasks asynchronously

**OpenCL + OpenMP regions**
- A task is enqueued
- OpenCL dispatches the task to DSP 0
- Tasks can use additional DSP cores by entering OpenMP regions
- A task completes before another task is dispatched
- Note: This is a TI extension

**Example use**
- Want to call existing OpenMP based DSP code from the ARM
// OpenMP Accelerator vector add
// OpenMP for loop parallelization
void ompVectorAdd(int N, float *a, float *b, float *c)
{
    #pragma omp target
    map(to: N, a[0:N], b[0:N])
    map(from: c[0:N])
    {
        int i;
        #pragma omp parallel for
        for (i = 0; i < N; i++)
            c[i] = a[i] + b[i];
    }
}

**Data movement**
- to copies variables from the ARM memory to the DSP memory
- from copies variables from the DSP memory to the ARM memory
- TI provides special alloc and free functions to allocate DSP memory such that copies are not needed

**Calling existing DSP code from the ARM**
- Wrapping existing DSP functions with OpenMP Accelerator code is straightforward
Memory

• Shared memory visible by both the ARM and DSP
  – A portion of the 8GB DDR3 DRAM (external)
  – The 6MB SRAM shared memory

• Performance keys
  – Allocate data in the shared memory for ARM setup and DSP acceleration
  – Use clmalloc() to allocate contiguous blocks that can be efficient transferred using DMA

• Options
  – Let the tools take care of the data movement using assign workgroup and strided copy functions
  – Manually manage the data movement using DMA (e.g., define buffers available for the DSP in OpenCL and manage the actual data movement on the DSP)
Dense Linear Algebra Philosophy

TI CBLAS Library

BLIS Cortex-A15

OpenMP

ARM 0

ARM 1

ARM 2

ARM 3

ARM subsystem

OpenCL

DSP 0

DSP 1

DSP 2

DSP 3

DSP 4

DSP 5

DSP 6

DSP 7

BLIS C66x

DSP subsystem

OpenMP

Texas Instruments
Peak performance: 9.6 GFLOPS
DGEMM performance is ~ 8.4 GFLOPS (83% peak)
How can we improve this performance?

- The BLIS implementation on the DSP does not utilize the different levels of memory efficiently.
- Utilize the DMA (Direct Memory Access) capabilities of the DMA to move data in parallel to the computations.
Cache Exploitation and DMA

Registers

L1

L2

MSMC

(384 Kb allocated per core)

DDR

A  B  C

---

DMA data move
CPU data move with packing
Linear CPU load
Cache Exploitation and DMA Details

![Diagram showing cache exploitation and DMA details]
DMA Integration Goals

• Flexible
  User or library developer must be able to select when and where to transfer data for an operation

• Transparent
  User must not be aware of the usage of the DMA, but if desired can manage the DMA

• Integrated into the control tree mechanism
Algorithmic Variants for GEMM

- Blocked Variant 2
  Matrix $B$ is partitioned into $n_c$ columns

- Blocked Variant 3
  Matrix $A$ is partitioned into $k_c$ columns
  Matrix $B$ is partitioned into $k_c$ rows

- Blocked Variant 1
  Matrix $A$ is partitioned into $m_c$ rows
  Blocks of $A$ and $B$ are packed

- Unblocked Variant 2
  Inner Kernel
GEMM Control Tree Definitions

```c
struct gemm_s
{
    impl_t impl_type;
    varnum_t var_num;
    blksz_t* b;
    func_t* gemm_ukrs;
    struct scalm_s* sub_scalm;
    struct packm_s* sub_packm_a;
    struct packm_s* sub_packm_b;
    struct packm_s* sub_packm_c;
    struct gemm_s* sub_gemm;
    struct unpackm_s* sub_unpackm_c;
};

typedef struct gemm_s gemm_t;
```

Control tree definition for `gemm`

```c
struct packm_s
{
    impl_t impl_type;
    varnum_t var_num;
    blksz_t* mr;
    blksz_t* nr;
    bool_t does_densify;
    bool_t does_invert_diag;
    bool_t rev_iter_if_upper;
    bool_t rev_iter_if_lower;
    pack_t pack_schema;
    packbuf_t pack_buf_type;
};

typedef struct packm_s packm_t;
```

Control tree definition for packing routines
Algorithmic Variants for GEMM with DMA Integration

- **Blocked Variant 2**
  Matrix $B$ is partitioned into $n_c$ columns

- **Blocked Variant 3**
  Matrix $A$ is partitioned into $k_c$ columns
  Matrix $B$ is partitioned into $k_c$ rows, copied into $L3$ using the DMA and packed into $L3$

- **Blocked Variant 1**
  Matrix $A$ is partitioned into $m_c$ rows
  Matrix $A$ is partitioned into $m_c$ rows, copied into $L3$ using the DMA and packed into $L2$

- **Unblocked Variant 2**
  Inner Kernel
GEMM Control Tree Definitions with DMA Integration

```
1 struct gemm_s
2 {
3   impl_t impl_type;
4   varnum_t var_num;
5   blksz_t* b;
6   func_t* gemm_ukrs;
7   struct scalm_s* sub_scalm;
8   struct packm_s* sub_packm_a;
9   struct packm_s* sub_packm_b;
10  struct packm_s* sub_packm_c;
11  struct dmam_s* sub_dmam_a;
12  struct dmam_s* sub_dmam_b;
13  struct dmam_s* sub_dmam_b;
14  struct gemm_s* sub_gemm;
15  struct unpackm_s* sub_unpackm_c;
16 }
17 typedef struct gemm_s gemm_t;
```

Control tree definition for gemm with DMA control leaf

```
1 struct dmam_s
2 {
3   impl_t impl_type;
4   varnum_t var_num;
5   blksz_t* mc;
6   blksz_t* nc;
7   buf_t dma_buf_type;
8 }
9 typedef struct dmam_s dmam_t;
```

Control tree definition for DMA routines
Memory Buffers

Listing 1: Allocating intermediate buffers for DMA Integration
Current Status of DMA Integration in GEMM

• Implemented multithreaded prototype of DMA Control Tree with decoding in Block Variant 1 using memcpy instead of DMA

• Pending
  – Decoding of DMA Control Tree in other variants
  – Invoking DMA routines
Thank you!

A special thanks to
Tyler M. Smith
Field G. Van Zee
Robert van de Geijn