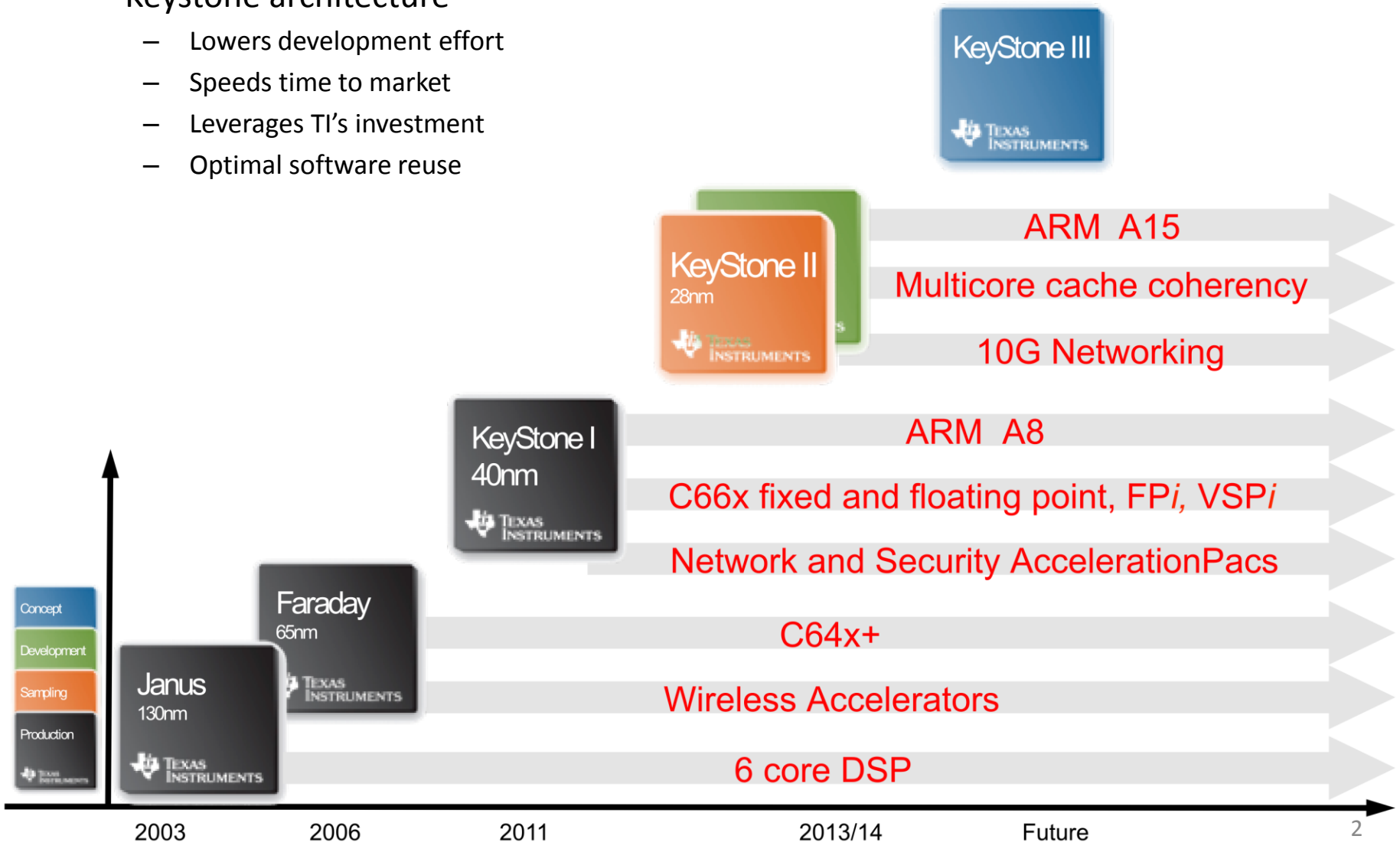
Integrating DMA capabilities into BLIS for on-chip data movement

Devangi Parikh
Ilya Polkovnichenko
Francisco Igual Peña
Murtaza Ali

Texas Instruments

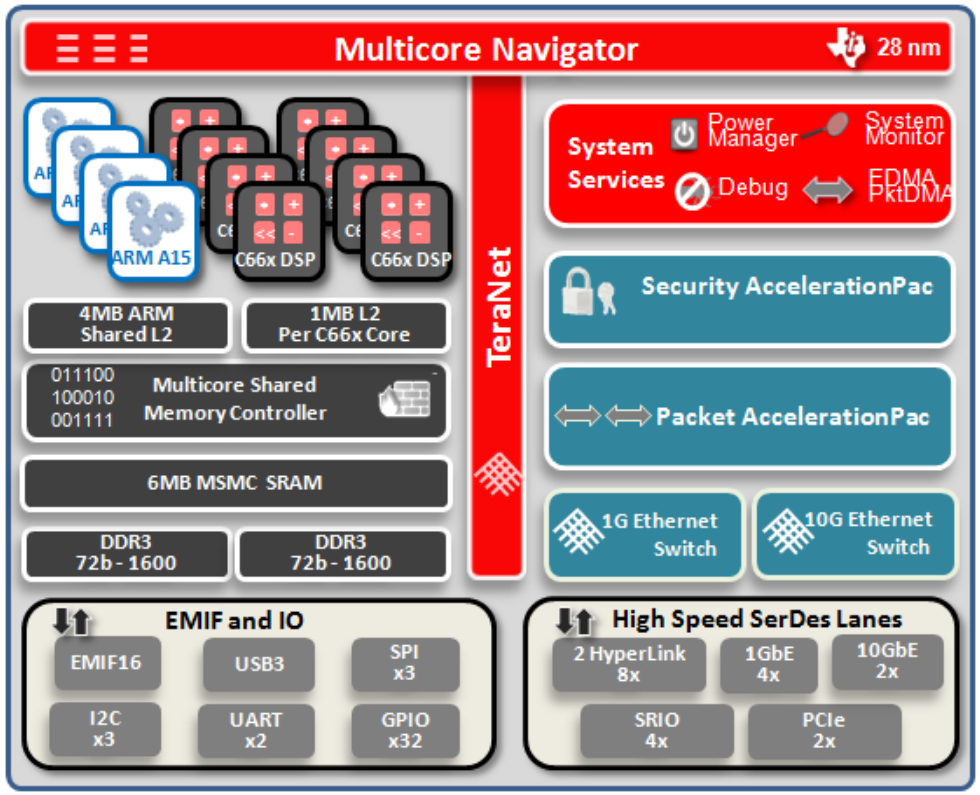
5 Generations of TI Multicore Processors

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



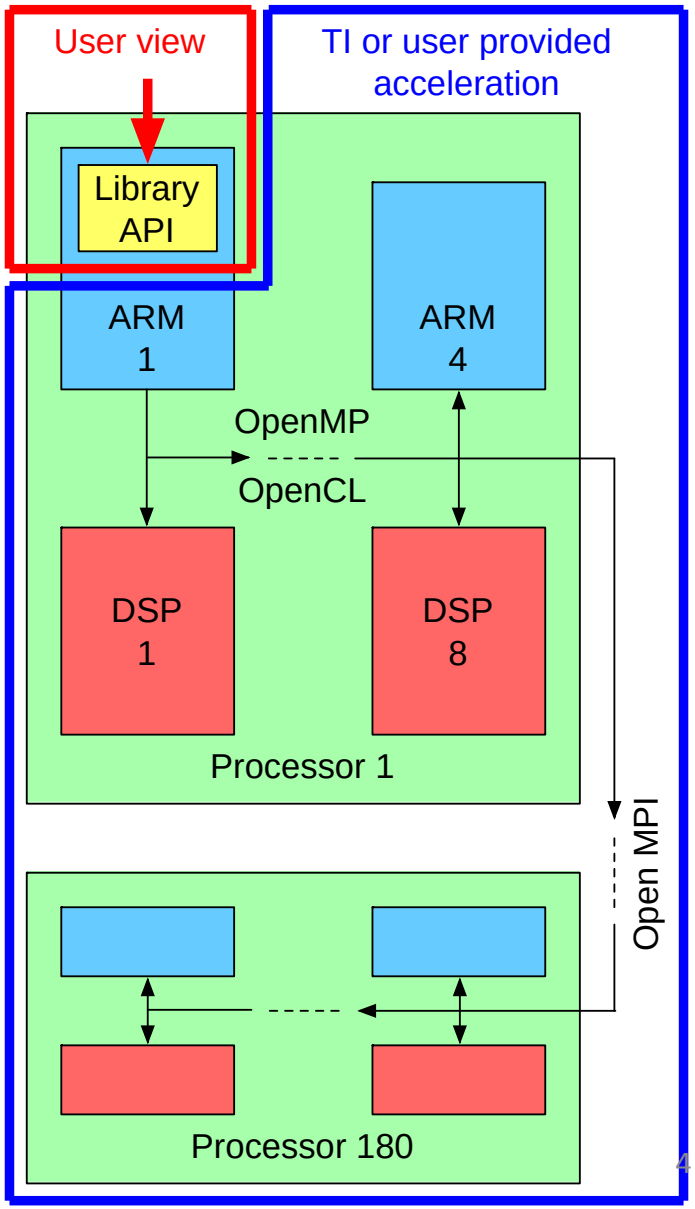
TI 66AK2H12 SoC

- 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

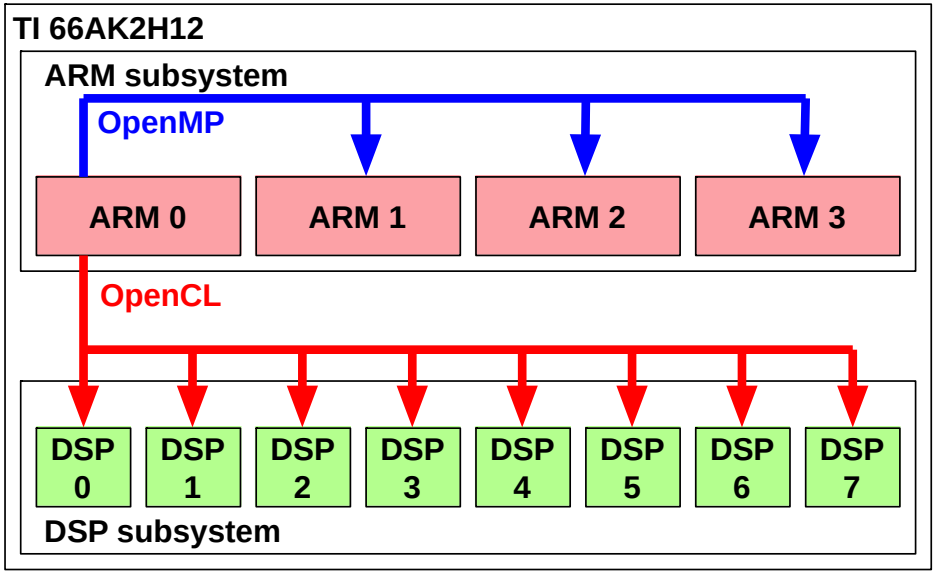


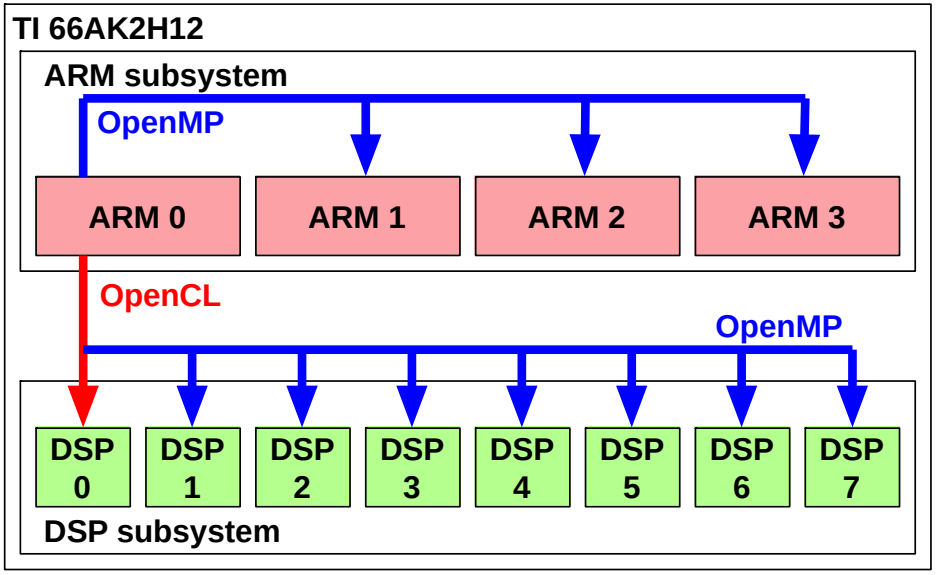
Development Philosophy

- 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

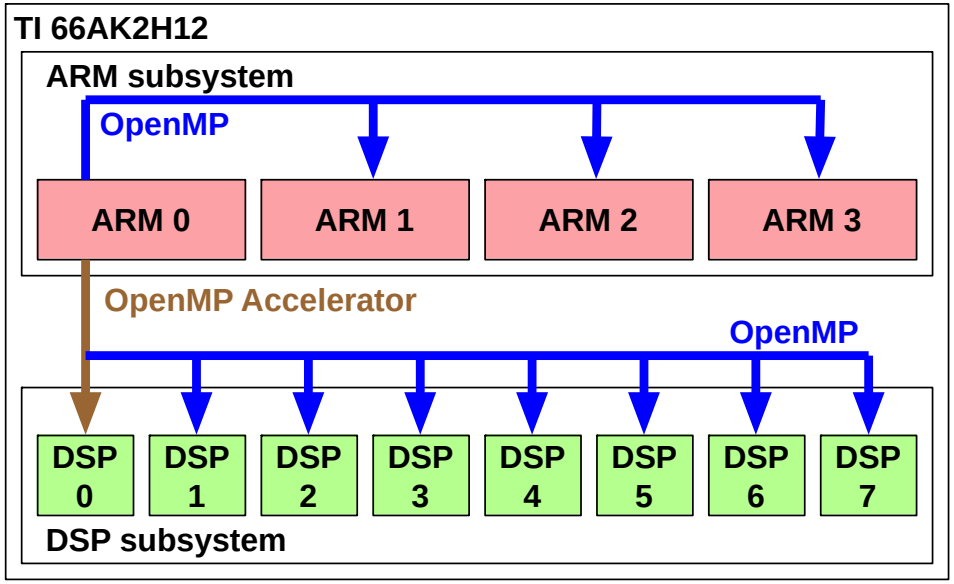
Task parallel
- A task is enqueued
- OpenCL dispatches tasks to cores
- OpenCL can accept and dispatch more tasks asynchronously

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

ARM + OpenMP Accelerator DSP Acceleration

```
// 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

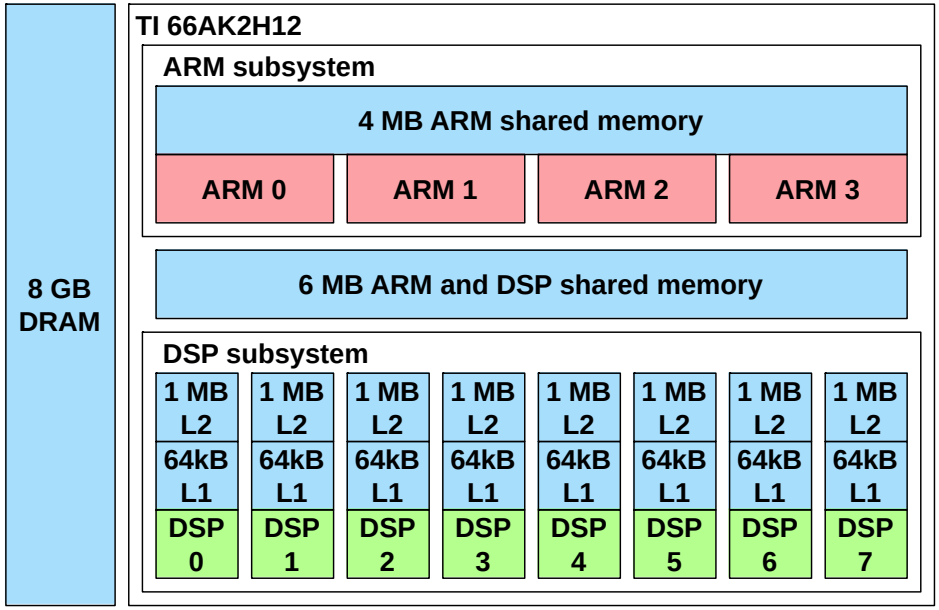
- **`to`** copies variables from the ARM memory to the DSP memory
- **`from`** copies variables from the DSP memory to the ARM memory
- TI provides special **`alloc`** and **`free`** functions to allocate DSP memory such that copies are not needed

Calling existing DSP code from the ARM

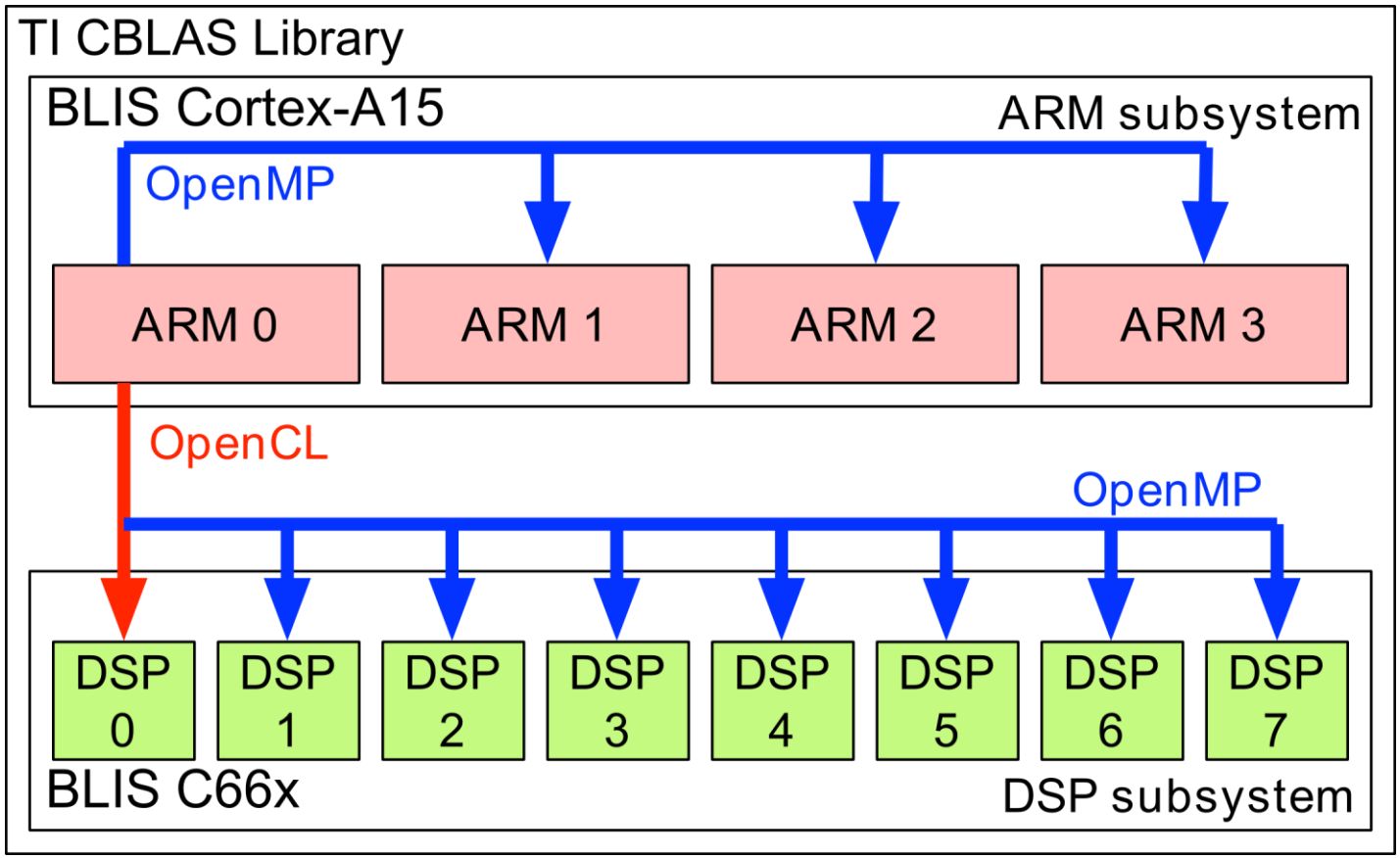
- Wrapping existing DSP functions with OpenMP Accelerator code is straightforward

Memory

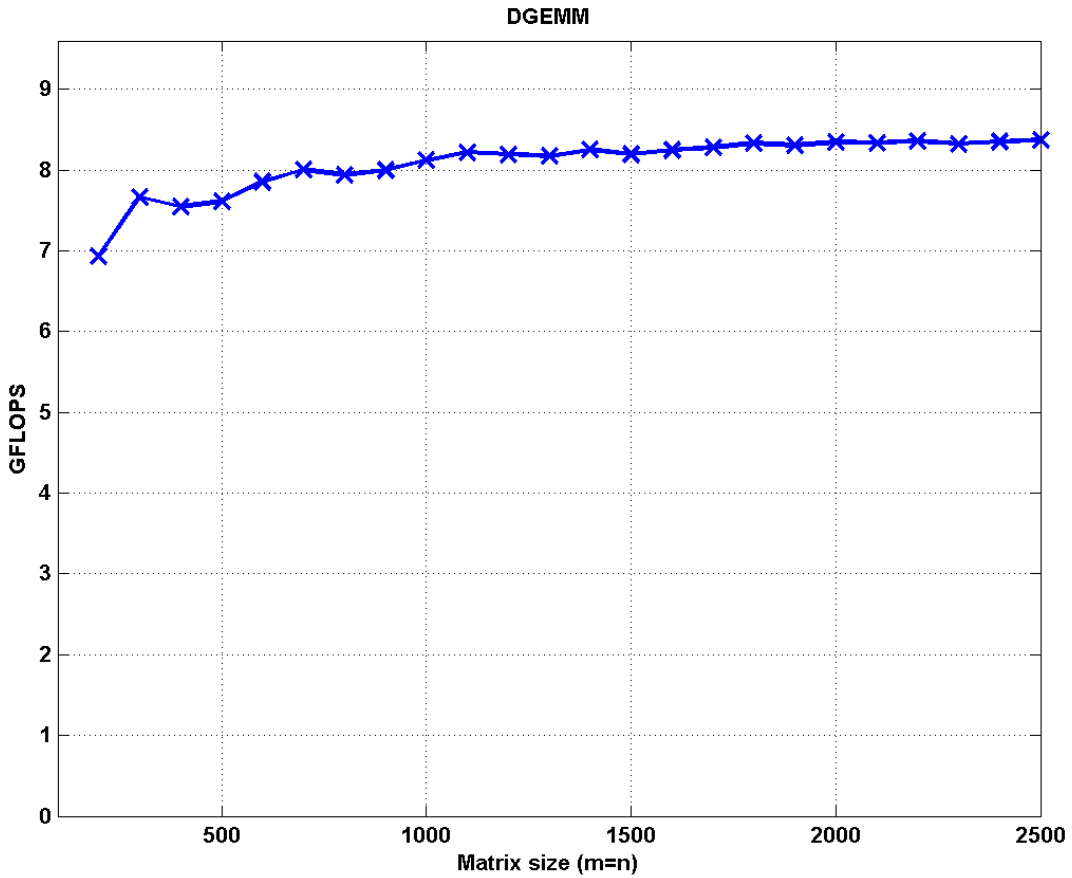
- Shared memory visible by both the ARM and DSP
  - A portion of the 8GB DDR3 DRAM (external)
  - The 6MB SRAM shared memory
- Performance keys
  - Allocate data in the shared memory for ARM setup and DSP acceleration
  - Use clmalloc() to allocate contiguous blocks that can be efficient transferred 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



BLIS Cortex-A15
DGEMM Multicore Performance

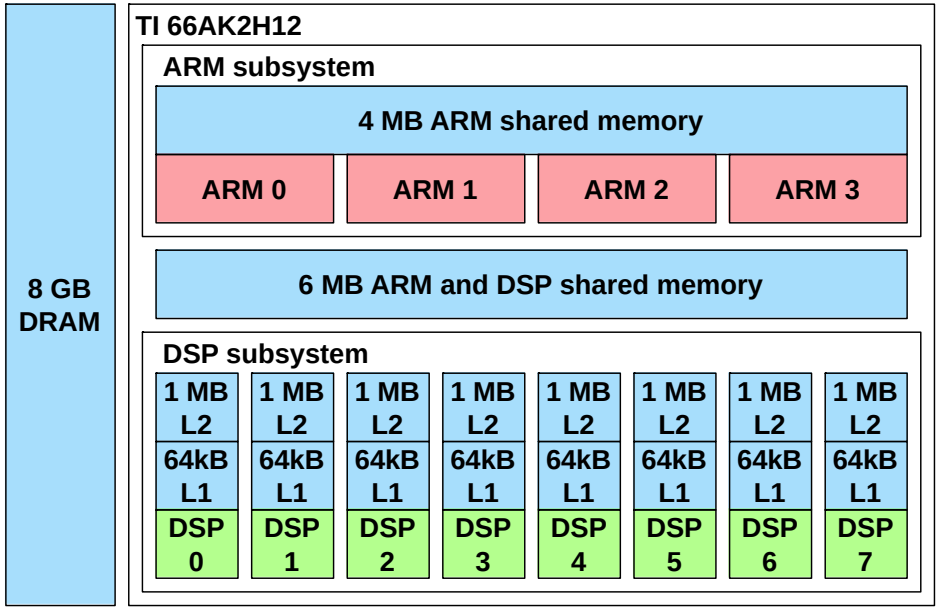


- Peak performance: 9.6 GFLOPS
- DGEMM performance is ~ 8.4 GFLOPS (83% peak))

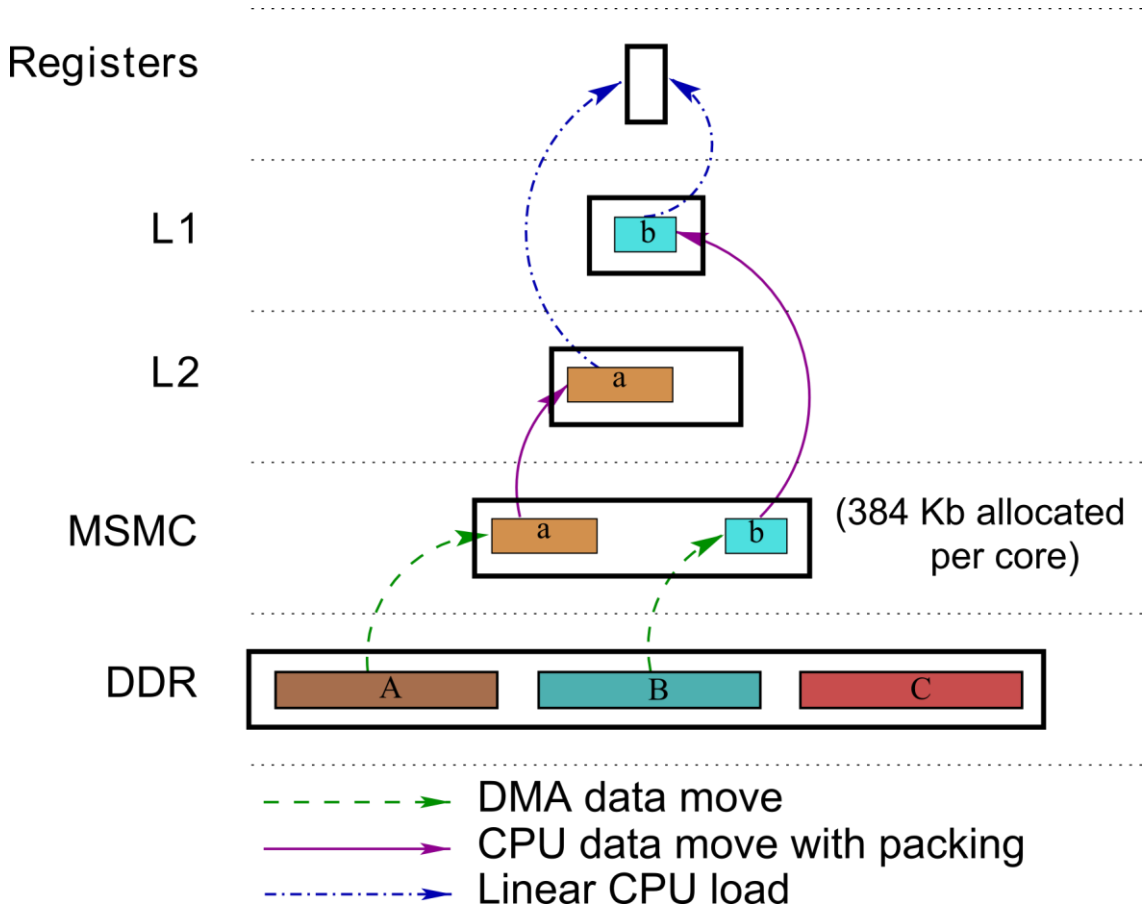
Recall - Memory

How can we improve this performance?

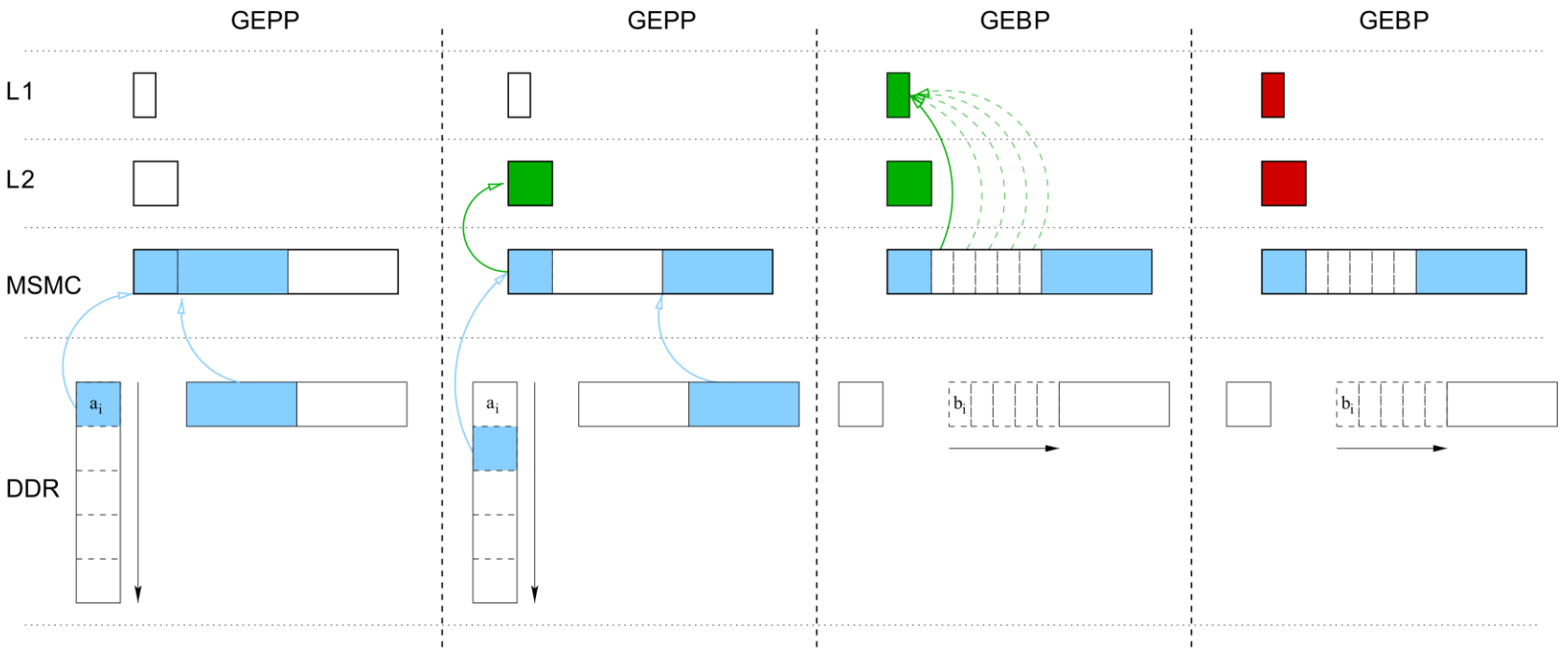
- The BLIS implementation on the DSP does not utilize the different levels of memory efficiently.
- Utilize the DMA (Direct Memory Access) capabilities of the DMA to move data in parallel to the computations



Cache Exploitation and DMA



Cache Exploitation and DMA Details



DMA Integration Goals

- Flexible

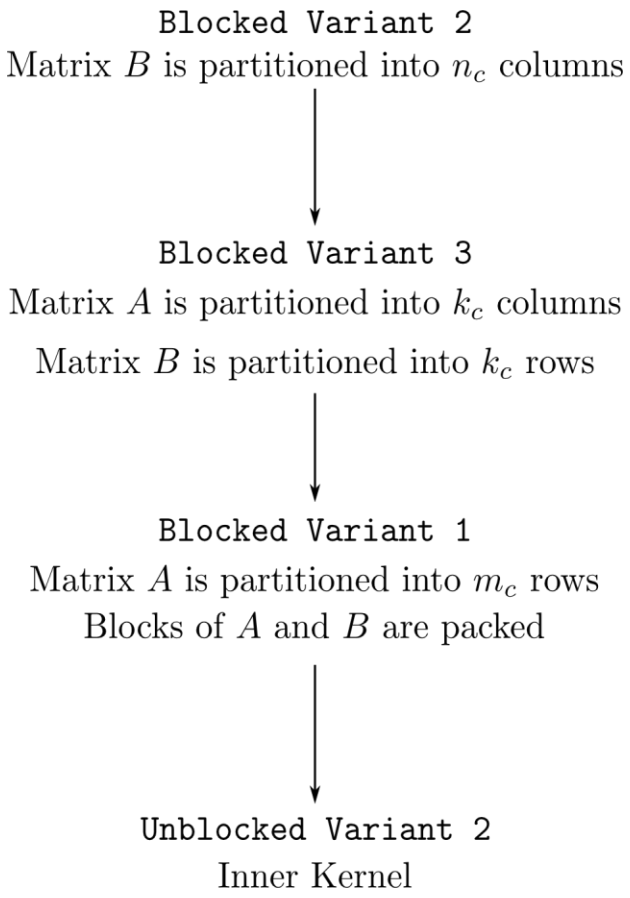
  User or library developer must be able to select when and where to transfer data for an operation

- 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



GEMM Control Tree Definitions

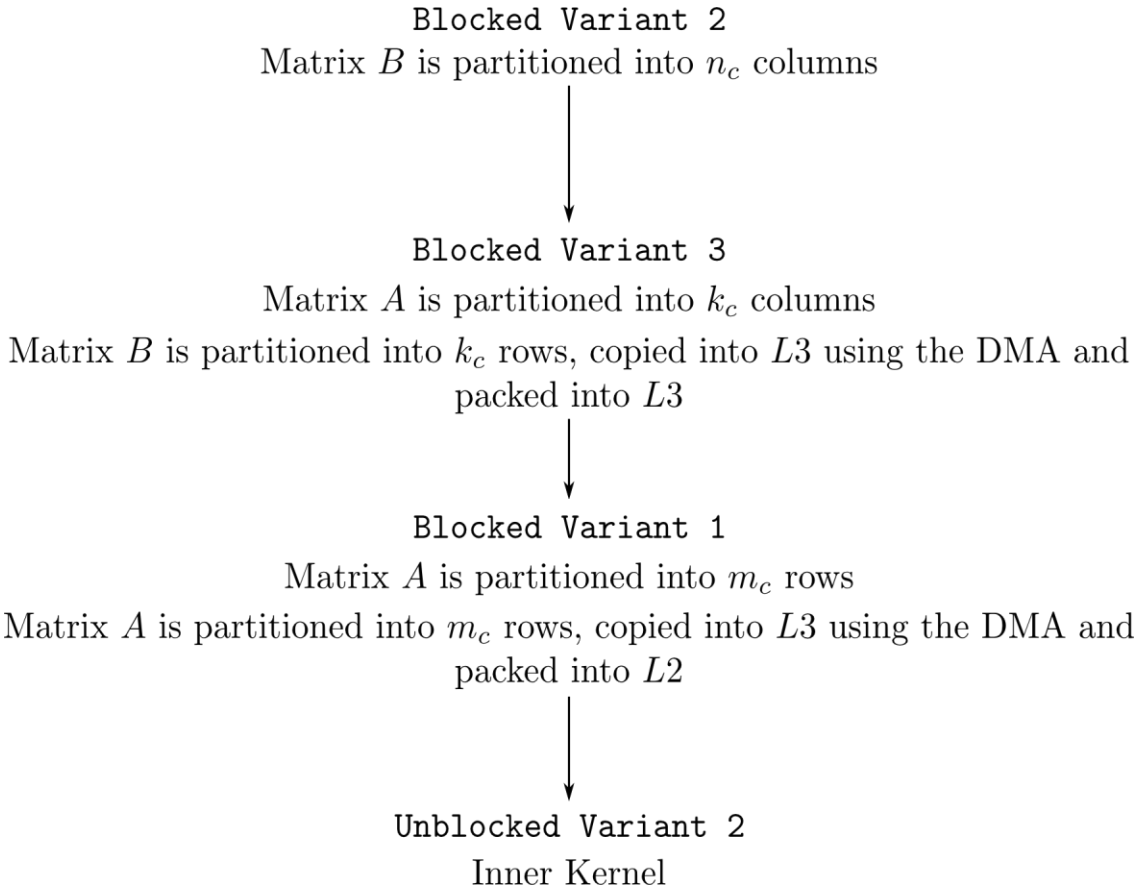
```
1  struct gemm_s
2  {
3    impl_t              impl_type;
4    varnum_t            var_num;
5    blksz_t*            b;
6    func_t*             gemm_ukrs;
7    struct scalm_s*     sub_scalm;
8    struct packm_s*     sub_packm_a;
9    struct packm_s*     sub_packm_b;
10   struct packm_s*     sub_packm_c;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

Control tree definition for `gemm`

```
1  struct packm_s
2  {
3    impl_t              impl_type;
4    varnum_t            var_num;
5    blksz_t*            mr;
6    blksz_t*            nr;
7    bool_t              does_densify;
8    bool_t              does_invert_diag;
9    bool_t              rev_iter_if_upper;
10   bool_t              rev_iter_if_lower;
11   pack_t              pack_schema;
12   packbuf_t           pack_buf_type;
13 };
14 typedef struct packm_s packm_t;
```

Control tree definition for packing routines

Algorithmic Variants for GEMM with DMA Integration



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

```
1  static pool_t pools[3];
2  //...
3  static void* pool_mk_blk_ptrs[ BLIS_NUM_MC_X_KC_BLOCKS ];
4  static char pool_mk_mem[ BLIS_MK_POOL_SIZE ];
5
6  static void* pool_kn_blk_ptrs[ BLIS_NUM_KC_X_NC_BLOCKS ]
7  static char pool_kn_mem[ BLIS_KN_POOL_SIZE ];
8
9  static void* pool_mn_blk_ptrs[ BLIS_NUM_MC_X_NC_BLOCKS ];
10 static char pool_mn_mem[ BLIS_MN_POOL_SIZE ];
```

Allocating intermediate buffers

```
1  static pool_t pools[9];
2  //...
3  static void* pool_mk_blk_ptrs_L1[ BLIS_NUM_MC_X_KC_BLOCKS ];
4  #pragma DATA_SECTION( pL1, ".myL1" );
5  static char pool_mk_mem_L1[ BLIS_MK_POOL_SIZE ];
6
7  static void* pool_mk_blk_ptrs_L2[ BLIS_NUM_MC_X_KC_BLOCKS ];
8  #pragma DATA_SECTION( pL2, ".myL2" );
9  static char pool_mk_mem_L2[ BLIS_MK_POOL_SIZE ];
10
11 static void* pool_mk_blk_ptrs_L3[ BLIS_NUM_MC_X_KC_BLOCKS ];
12 #pragma DATA_SECTION( pMSMC, ".myMSMC" );
13 static char pool_mk_mem_L3[ BLIS_MK_POOL_SIZE ];
14
15 static void* pool_kn_blk_ptrs_L1[ BLIS_NUM_KC_X_NC_BLOCKS ];
16 #pragma DATA_SECTION( pL1, ".myL1" );
17 static char pool_kn_mem_L1[ BLIS_KN_POOL_SIZE ];
18
19 static void* pool_kn_blk_ptrs_L2[ BLIS_NUM_KC_X_NC_BLOCKS ];
20 #pragma DATA_SECTION( pL2, ".myL2" );
21 static char pool_kn_mem_L2[ BLIS_KN_POOL_SIZE ];
22
23 static void* pool_kn_blk_ptrs_L3[ BLIS_NUM_KC_X_NC_BLOCKS ];
24 #pragma DATA_SECTION( pMSMC, ".myMSMC" );
25 static char pool_kn_mem_L3[ BLIS_KN_POOL_SIZE ];
26
27 static void* pool_mn_blk_ptrs_L1[ BLIS_NUM_MC_X_NC_BLOCKS ];
28 #pragma DATA_SECTION( pL1, ".myL1" );
29 static char pool_mn_mem_L1[ BLIS_MN_POOL_SIZE ];
30
31 static void* pool_mn_blk_ptrs_L2[ BLIS_NUM_MC_X_NC_BLOCKS ];
32 #pragma DATA_SECTION( pL2, ".myL2" );
33 static char pool_mn_mem_L2[ BLIS_MN_POOL_SIZE ];
34
35 static void* pool_mn_blk_ptrs_L3[ BLIS_NUM_MC_X_NC_BLOCKS ];
36 #pragma DATA_SECTION( pMSMC, ".myMSMC" );
37 static char pool_mn_mem_L3[ BLIS_MN_POOL_SIZE ];
```

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

Texas Instruments