Effecient Implemementation of Mean Shift Algorithm on Multi-Core Heterogeneous Computing Platform

Abbas Jhabuawala
4621883
a.m.jhabuawala@student.tudelft.nl

Anant Semwal
4630734
a.semwal@student.tudelft.nl

Minfeng Li
4608348
m.li-10@student.tudelft.nl

Xiaoming Wen
4608682
x.wen@student.tudelft.nl

Anand Haridas
4619153
A.Haridas@student.tudelft.nl

Anurag Kulkarni
4627342
a.kulkarni-1@student.tudelft.nl

***Abstract*—Object tracking is a popular topic in the field of computer vision and embedded applications used for object tracking must meet real time constraints ensuring all computations are completed within time and track object effectively. Improving speed of any application requires an environment which is capable of utilizing parallelism in the code and execute as much code as possible on available processing units. In this report we have optimized Mean Shift Algorithm on a multi-core heterogeneous processor.**

I. Introduction

Optimization of an embedded application is a deterministic process which is undertaken to:

- reduce program memory by reusing code and remove redundant code blocks,
- reduce runtime memory by using appropriate variable assignment
- improve execution time of the code by carefully scheduling tasks on processor
- approximations in calculations as and when required.

This involves in-depth analysis of application code. Profiling tools like grpof and mcprof are used which help in completion of an exhaustive code review. In this report we will discuss our approach and experiments to produce the application code which provides minimum 16x speedup in track function and overall speedup of 4x to application. In the next section we analyze features of the multi-core heterogeneous computing environment. In section three, we will discuss techniques analyzed for baseline evaluation. Section four will discuss about the design space exploration, where we explain in detail about the approaches used for ARM+DSP , ARM+NEON and ARM+DSP+NEON. In section five we have compiled results from our experiments and error analysis done on the tracking videos generated by different approaches.

II. Heterogeneous Computing Environment

To obtain the best performance from the modification made to the program it is important to be aware of the underlying hardware, in particular from a programming aspect, be aware of processing power, hardware resources, pipelining, scheduling issues, memory accesses and hazards that may occur due to parallel operation. Details on the hardware and the resources are elaborated in this section.

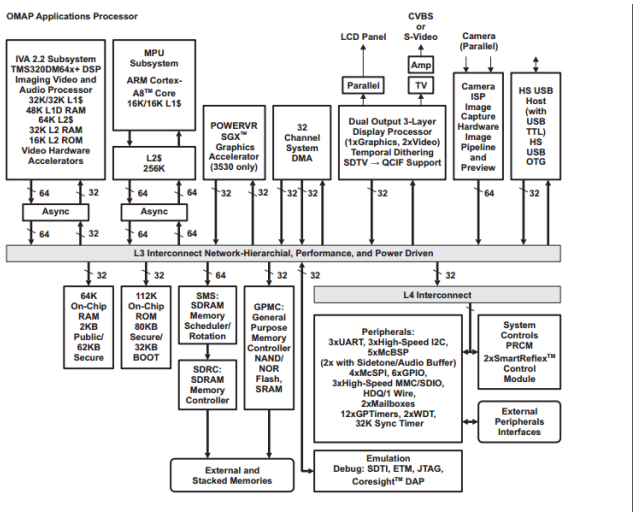

Fig. 1: OMAP Architecture

The implementation of the Mean shift Algorithm is done on the OMAP3530 high performance application processor which is based on enhanced the OMAP 3 architecture. The 3530 system on chip contains the following sub systems:

- ARM® Cortex™-A8 microprocessor unit
- IVA2.2 subsystem with a C64x+ Digital Signal Processor Core
- Graphics Accelerator for 3D graphics acceleration A block diagram of the complete subsytems is shown in Fig 1[1]

A. MPU Subsytem

1) Main Core: The MPU subsystem contains a ARM cortex A8 core which is a low power powerful cached processor that has a full implementation of the ARM v7-A instruction set and also integrates a NEON coprocesor for executing advanced Single Instruction Multiple Data and ARM Floating Point architecture(VFP) instruction set. It embeds two levels of cache in which level one is a 16KB to 32KB configurable instruction and data cache, and level two is a 128KB-1MB configurable unified cache. The communication with the other subsystems are done by a 64 bit high speed Advanced Microprocessor Bus Architecture(AMBA) with Advanced Extensible Interface (AXI) which is asynchronous. A block diagram of the internal structure on the ARM core is shown in Figure 4.

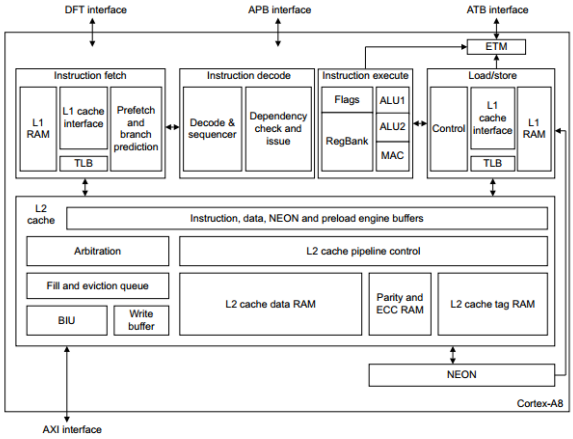

Fig. 2: The structure of the Cortex-A8 Processor

2) ARM Pipeline: The ARM contains a thirteen stage integer pipeline. The main components of the pipeline are the Instruction Fetch, Decode and Instruction Execute. The fetch stage fetches the instruction from the cache and places it an buffer which is then accessed by the decode stage which decodes and sequences all ARM and Thumb-2 instructions. The decoded instructions are then passed to the execution unit which consists of two symmetric Arithmetic Logical Unit (ALU) pipelines, an address generator for load and store instructions, and the multiply pipeline. The execute pipelines also perform register write back. Figure 3 shows the pipeline stages and its internal pipelines.[1] Furthermore some ARM and SIMD instructions types can dual issue therefore in one cycle the ARM can fetch two instructions (more in Thumb-2), decode two instructions, and issue two instructions to the execution unit.[1] The dual instruction issuing in the cortex A8 is as the following:

- Two ALU instructions
- One ALU instruction and one load/store instruction
- One multiply/MAC instruction with one ALU instruction load/store instruction

Keeping these A8 processor structure and capabilities in mind the optimizations on the baseline code were implemented.

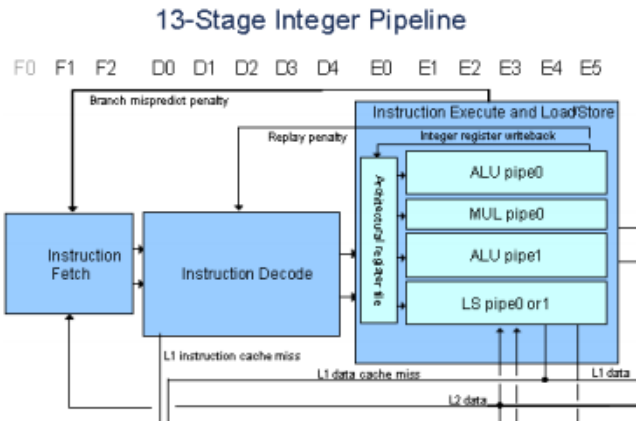

Fig. 3: ARM Pipeline Stages

Section three and four discuss in detail the optimizations which were performed.

3) ARM Instruction: The ARM issues a series of micro operations to the execution pipeline for each ARM instruction executed, though for simple ARM instruction just a single micro operation is executed and only complex ARM instruction such as load and stores use multiple micro ops.[1]

B. NEON

1) Background: Neon co processor is an Advanced Single Instruction Multiple Data (SIMD) architecture extension of the ARM Cortex A8 processor. It is integrated in the processor and shares the processor resource for integer operations, loop control and caching.[3] This is an advantage as compared to other hardware accelerators as it significantly reduces power and area. NEON is aimed toward improving the speed of processing integer and floating point operation by using a single instruction that can operate on all data values in a wide register at the same time. An example is shown in Figure 4.

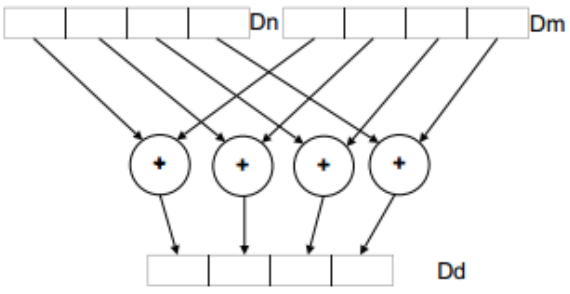

Fig. 4: Single Instruction operating on multiple data simultaneously

Neon has thirty two 64 bit registers which are also visible as sixteen 128 bit registers. Each NEON register are vectors of elements of the same data type and each vector is divided into lanes which contains a data value called elements.[2] The

number of lanes in a vector depends on the size of the vector and the size of the elements in the vector. Figure 5 shows the possible lane divisions in the 64 and 128 bit NEON registers.

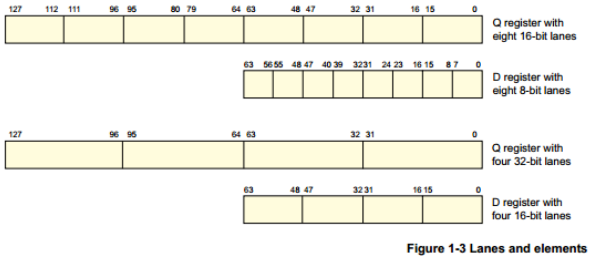

Fig. 5: Lane Division

2) NEON Pipeline: NEON unit is an extension to the ARM pipeline. It consist of a ten stage pipeline that decodes and executes the SIMD media instruction set. The Neon unit includes the following:

- Sixteen instruction deep Instruction queue
- Twelve entry deep Load Data Queue
- Two pipelines for Neon Instruction Decode Logic
- Three execution pipelines for SIMD Integer operation (ALU,Shift,Multiply & Accumalate)
- Two execution pipelines for SIMD Floating Point operation (Single Precision)
- One execution pipeline for SIMD and VFP load/store instructions
- Non pipelined VFP engine for full execution of the VFPv3 data-processing instruction set

The Neon pipeline fetch stage can receive up to two instructions per clock cycle but from the decode stage to the execution stage the dual issue capabilities are limited to certain types of instruction. Only a load/store or permute instruction can be paired with a SIMD data processing instruction and can only happen on the first cycle and the last cycle of a multi cycle operation.[1][3] All intermediate instructions are single issued. Transfer from the NEON to the ARM takes a minimum of twenty clock cycles and therefor by issuing multiple back to back transfer instruction this store latency can be hidden. All this information was taken in account while performing optimizations using the NEON and are detailed in section dedicated to Design Space Exploration.

3) Neon Instruction: NEON instructions travel from the ARM pipeline to the NEON pipeline. The instruction then gets decoded and scheduled on the execution stage of the NEON. As mentioned above only sixteen and twelve instructions and data respectively can be held in the queues of the NEON. After these queues are full, the processor will stall execution of the next NEON instruction in the queue until there is room.In this manner, the cycle timing of the NEON instruction can have a significant impact on the total time if there are a lot of NEON instructions used.

4) Neon Intrinsics: NEON intrinsics are similar to c programming function calls that the compiler replaces with an appropriate NEON instruction or sequence of NEON instructions.[2] Implementing SIMD instruction using the Neon intrinsics eliminates the trouble of doing register allocation and instruction scheduling to eliminate pipeline stall; these are handled by the compiler.[3] The disadvantage of using intrinsics is that the compiler does not generates the exact code that is required. Only NEON intrinsics were used for the experiments. To start programming with NEON intrinsic a NEON vector is created . The general syntax of vector data type is as follows:

$$< type >< size > x < number\_of\_lanes > \_t$$

For example:

- 'int16x4_t' defines a vector of four elements, each sixteen bits wide.

Upon creation of a vector, the NEON intrinsics are implemented on top of it.The general syntax of a NEON intrinsic is as follows:

$$< opname >< flags > \_ < type >$$

For example:

- 'vmul_s16' defines a multiply operation on four elements each sixteen bits wide.

In the end combining these two definition we can write a complete intrinsic such as:
int16x4 result = vmul_s16(int16x4 a, int16x4 b);

which translates to doing a simultaneous multiplication of four elements that are sixteen bits wide and storing the result in a four lane vector named 'result'. The compiler then converts this to the appropriate NEON assembly and sends it on to the pipeline. Following are some among the most commonly used intrinsic operations that are relevant to the optimization of the application .

- vmul(q)- multiplication for integer and single precision floating point values.
- vadd(q) - addition for integer and single precision floating point values.
- vrecpeq - gives an approximate reciprocal value for each element in the vector.
- vld1(q) - loading from ARM scalar regoster to a NEON vector.
- vget(q)_lane - Extracting individual elements from a NEON vector
- vst1(q) - Storing complete neon vectors back in memory
- vdup(q)_n - duplication scalar in all lanes of a vector
- vmul(q)_n - multiplication of all elements of a vector with a scalar.

The 'q' modifier in the intrinsics are used if the vector utilized are quadword in size.[3]

The instruction and load queue, dual issue capabilities, instruction cycle timing were the deciding factors in the choosing the appropriate intrinsic.

C. Digital Signal Processor Subsystem

The OMAP3530 uses a TMS320C64x+ DSP core. This is a fixed-point DSP, an early indication that floating-point operations would be performed in software, and hence would be slow. However, in the following sections, explanation about conversion of floating-point operations to fixed-point operations is given. The core has 64 general purpose registers of 32-bit length and 8 functional units which can run independently - 6 ALUs and 2 multipliers[7]. If the code exhibits enough parallelism for add and multiply operations, these units can be utilized well. The TMS320C64x+ core is capable of executing up to 5760 million instructions per second (MIPS) at a clock rate of 720 MHz[7]. Thus, it finds applications in high-performance DSP requirements where cost is a major concern[7].

The core has two high-performance embedded coprocessors [Viterbi Decoder Coprocessor (VCP) and Turbo Decoder Coprocessor (TCP)] that significantly speed up channel-decoding operations on-chip.The TCP operates at half the CPU clock speed and can decode up to forty-three 384-Kbps or seven 2-Mbps turbo encoded channels (assuming 6 iterations)[7]. The TCP implements the max*log-map algorithm and is designed to support all polynomials and rates required by Third-Generation Partnership Projects (3GPP and 3GPP2), with fully programmable frame length and turbo interleaver. Decoding parameters such as the number of iterations and stopping criteria are programmable[7]. Communications between the VCP/TCP and the CPU are carried out through the EDMA controller[7].

The DSP also has application-specific hardware logic, on-chip memory, and additional on-chip peripherals[7]. Though these architectural strengths, which also include four 16-bit multiply-accumulates (MACs) per cycle for a total of 2880 million MACs per second (MMACS), or eight 8-bit MACs per cycle for a total of 5760 MMACS, support high performance HD video processing, at low power, the application to be used in the assignment makes use of the FLV1 codec to write the video file, along with some algorithmic computations involving OpenCV libraries, which restricts the full utilization of this powerful DSP core. Basically, the algorithm concentrates on the backend processing (such as histogram computation from the pixel values, the weight of the pixels in the tracking region, and generating the tracking rectangle from other computation results), and not so much so on the video encoding/decoding. The tracking function optimization is the main focus of this assignment, hence the basic arithmetic functionalities is what the DSP will be utilized for. The main advantage for the provided SoC is that the DSP can operate in parallel with the ARM. So, to achieve a speedup, arithmetic computation areas in the baseline code, which can be executed in parallel, need to be identified (and indicate this explicitly to the compiler if required), and the code structure should be modified accordingly.

III. BASELINE EVALUATION

The migration of application from homogeneous processor to a heterogeneous processor, baseline benchmarking is essential. This requires identification of the bottlenecks and hypothesizing the possible solution for mapping the tasks to heterogeneous processing units. In our case we have an opportunity to work on ARM, NEON and DSP. Therefore we the first step in our project was to measure and profile the baseline code

A. Profiling

1) GPROF: GPROF is a profiling software which analyzes the code given to it as input and provides information about where the program spent its time and how much time each function takes in its self and in the functions it calls. This information helps identify the bottle necks in the program and helps programmer optimize the code in terms of speed. GPROF saves all these raw timing information in a file called gmon.out which is created when the program is executed.These file can be viewed by running gprof which displays the timing information nicely in the form of flat profiles and call graphs.

To enable GPROF the -pg compiler flag was used which tells the compiler to generate the gmon.out file for the respective code.

As mentioned above, GPROF profiling provides crucial information about the program's timing information and provided us information about the functions that took the most amount of time and also the function call hierarchy. These information are available in the flat profile and the call graph respectively as shown below.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
50.08	0.01	0.01	166	60.34	60.34	MeanShift::pdf_representation(cv::Mat const&, cv::Rect_<int> const&)
50.08	0.02	0.01				MeanShift::track(cv::Mat const&)
0.00	0.02	0.00	166	0.00	0.00	MeanShift::Epanechnikov_kernel(cv::Mat&)
0.00	0.02	0.00	165	0.00	0.00	MeanShift::CalWeight(cv::Mat const&, cv::Mat&, cv::Mat&, cv::Rect_<int>&)
0.00	0.02	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN9MeanShiftC2Ev
0.00	0.02	0.00	1	0.00	0.00	_GLOBAL__sub_I_main

Fig. 6: Flat Profile

Call graph

granularity: each sample hit covers 2 byte(s) for 49.92% of 0.02 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	99.7	0.01	0.01		MeanShift::track(cv::Mat const&) [1]
		0.01	0.00	165/166	MeanShift::pdf_representation(cv::Mat const&, cv::Rect_<int> const&) [2]
		0.00	0.00	165/165	MeanShift::CalWeight(cv::Mat const&, cv::Mat&, cv::Mat&, cv::Rect_<int>&) [12]
		0.00	0.00	1/166	MeanShift::Init_target_frame(cv::Mat const&, cv::Rect_<int> const&) [3]
		0.01	0.00	165/166	MeanShift::track(cv::Mat const&) [1]
[2]	50.0	0.01	0.00	166	MeanShift::pdf_representation(cv::Mat const&, cv::Rect_<int> const&) [2]
		0.00	0.00	166/166	MeanShift::Epanechnikov_kernel(cv::Mat&) [11]
					<spontaneous>
[3]	0.3	0.00	0.00		MeanShift::Init_target_frame(cv::Mat const&, cv::Rect_<int> const&) [3]
		0.00	0.00	1/166	MeanShift::pdf_representation(cv::Mat const&, cv::Rect_<int> const&) [2]
		0.00	0.00	166/166	MeanShift::pdf_representation(cv::Mat const&, cv::Rect_<int> const&) [2]
[11]	0.0	0.00	0.00	166	MeanShift::Epanechnikov_kernel(cv::Mat&) [11]
		0.00	0.00	165/165	MeanShift::track(cv::Mat const&) [1]
[12]	0.0	0.00	0.00	165	MeanShift::CalWeight(cv::Mat const&, cv::Mat&, cv::Mat&, cv::Rect_<int>&) [12]
		0.00	0.00	1/1	__libc_csu_init [24]
[13]	0.0	0.00	0.00	1	_GLOBAL__sub_I__ZN9MeanShiftC2Ev [13]
		0.00	0.00	1/1	__libc_csu_init [24]
[14]	0.0	0.00	0.00	1	_GLOBAL__sub_I_main [14]

Fig. 7: Call Graph

The flat profile is a table created which gives information about the total amount of time the program spent executing

each function and arranges them in decreasing run-time spent in them. From the flat profile it was identified that functions $MeanShift :: pdf_representation$ and $MeanShift :: track$ took the maximum number of time and therefore optimizing these function will significantly improve performance in terms of speed.GPROF isn't perfect, sometimes it is unable to accurately measure all the function in the program as they might run too fast to appear in the program counter histogram and will be indistinguishable with function that were never called. The sampling period at the top of the flat profile table provides the margin of error on the values presented in each of the time figures provided.

In order to have a deeper understanding about the time spent in these function the call graph was interpreted. Call Graphs provide information about how much time is spent in each function as well as how much time is spent in the functions called by that function (Children Function). This is very useful as a function may consume a lot of time but not in itself but in the Children function. For our analysis it was noted that $MeanShift :: track$ takes 99.7% of the total program time but most of this time is divided in the two function it calls namely $MeanShift :: pdf_representation$ and $MeanShift :: CalWeight$. Therefore, reducing the time spent in either of these functions will speed up the program.

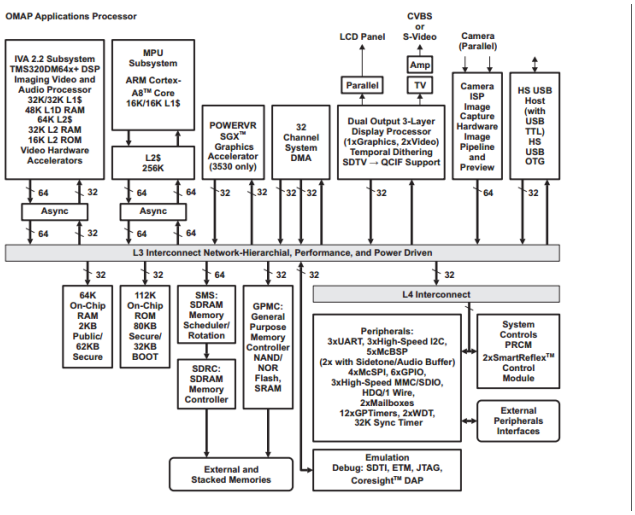

Fig. 8: OMAP Architecture

2) MCPROF: After timing analysis, it was important to understand memory and communication profile of the application. For this MCProf was used. Execution profile and memory profile were considered for evaluation.

We observed the data read and writes in each function which enabled us to understand the memory accesses made by the code. This helped us in removing unnecessary or redundant memory access. This data was available in $memProfile.dat$

Next we observed $execProfile.dat$ which gave us information about the number of execution of each function. So we can select appropriate method while optimizing when we have accurate timing results from the independent code blocks within a function.

Function Name	Accesses Total	Reads	Writes	Allocation Path
main	932161669	698313044	233848625	NA:0
MeanShift::pdf_representation	43635468	33058350	10577118	meanshift.cpp:40
MeanShift::track	3684596	3605921	78675	main.cpp:55
MeanShift::CalWeight	123261333	115077942	8183391	meanshift.cpp:223

Fig. 9: Memory Profile

Total Calls	%Exec.Instr.	Name
1	80	main
168	10	MeanShift::CalWeight
31	4	MeanShift::track
169	3	MeanShift::pdf_representation
1	0	cv::Mat::~Mat
1	0	MeanShift::~MeanShift
2	0	Timer::Print
1	0	MeanShift::Epanechnikov_kernel
1	0	MeanShift::Init_target_frame
1	0	MeanShift::MeanShift
2	0	Timer::Timer

Fig. 10: Execution Profile

B. Helping the Compiler

Compilers are built such that they schedule tasks on processing units in an optimized fashion. But the programmer is supposed to resolve as much data dependencies and provide sufficient hints to help compiler do its job. Thus, we inspected the code for independent calculations and scheduled the instructions to get a better results from compiler.

1) Loop Unrolling: Loop unrolling is the way to reduce loop overhead. The image resolution MxN where M is number of rows and N is number of columns. Since M and N are multiples of two it was safe to assume that unrolling loop by the factor of two would help in performance of the application. This is achieved by means of $\#PRAGMAS$ during compile time or the programmer can unroll while writing the code. For our application we identified inner loops in pdf_representation(), CalWeight() and track() as a suitable code blocks for unrolling. As explained above, unroll factor was selected as 2.

2) Load/Store Optimizations: Memory operations are expensive. The program uses cache memory for fast access of program data. But due to limited size of cache, all data is stored in main memory and is loaded onto cache before program can use the data. The overhead could be between 8-20 CPU cycles and is an expensive operation.
To avoid unnecessary load/store we have eliminated cv::split() method which was splitting multi-channel frame into three mono-channels ie RGB planes. Instead we directly accessed values from the frame and avoided significant bottleneck related to memory operations.

3) Instruction Scheduling & Data Hazards: As mentioned in the previous section, the ARM has the capabilities to issue dual commands simultaneously and is equipped with two ALU, one Multiply unit and one Load/Store unit. Thus, it has an enhanced capabilities to process instructions by making use of parallelization within core. However, during the execution of a program operations depend on the data from preceding instructions. This data may be unavailable to the CPU if its computation is still in progress. For instance, if result of division is required in the next instruction, the pipeline may

get stalled until the divide operation is complete.
This motivates us to analyze the instructions which can be regrouped and avoid data hazards to maximize utilization of processing units and minimize the stalls which lead to delay.

4) Compiler Flags: Compiler can also be directed to apply several optimizations by passing available flags. Several flags are incompatible with arm-gcc and thus we highlight flags that are part of our makefile.

1) -msoft-float, Generate output containing library calls for floating point.
2) -ffast-math, Sets -fno-math-errno, -funsafe-math-optimizations, -fno-trapping-math, -ffinite-math-only, -fno-rounding-math and -fno-signaling-nans.
3) -fno-strict-aliasing
4) -fno-common
5) -fno-omit-frame-pointer
6) -O3, turns on all optimizations specified by -O2 and also turns on the -finline-functions, -fweb and -frename-registers options

C. Optimizing Floating Point Operations

1) Fixed-Point Arithmetic : The *track* function uses many floating point computations, directly and through calls to functions *pdf_representation* and *calWeight*. The ARM Cortex A-8 core of the OMAP3530 provided is configured here to use the NEON's floating point unit. Even though the ARM is capable of doing 32-bit floating point operations using this unit, floating point operations in general are computationally expensive. Keeping this in mind, an approach has been sought to change some floating point operations to fixed-point operations. Specifically, inside the *track* function, the variables representing the center of the frame, x deviation, y deviation, and normalization factors along the horizontal and vertical, which were defined as floating-point initially, are changed to integers using an appropriate scaling factor. The calculation for *center* variable now uses a scaling factor of 10, as the values were obtained correct up to one decimal place. In fact, the computation for this variable is now done only once, during initialization. Whereas, the scaling factors for the horizontal and vertical are chosen as 1000. A compromise is observed in the tracking accuracy, which degrades on reducing the scaling factor further, but a lower scaling factor results in a faster execution of the application in general.

The TMS320C64x+ DSP core in OMAP3530 does not support floating point operations in hardware, hence these operations are performed in software, which results in a poorer performance. For better performance on the DSP side, before executing on DSP the floating point numbers are transformed into fixed point number of Q format on ARM side, which is done by multiplying the original floating point with 2^N and storing as an integer. For example, the floating number 3.14 can be represented as 3215 in Q10 format ($3.14*2^{10}$=3215.36). Again, the larger the chosen N, the better the accuracy we achieve, but the smaller range of the number represented.

IV. Design space exploration

A. Optimized code on ARM - optimization of the baseline code

A lot of scope for optimization has been found within the baseline code itself, apart from the techniques discussed in the above section, for example, the calculation of the sum of elements of the *Epanechnikov* kernel need not be done repeatedly. The value of this sum (to be used for normalization) can be calculated during the initialization using the first frame. The same goes for the *centre* value of the tracking rectangle. Also, since division operations are more expensive than the multiplication ones, the inverse of some variables can be found during the initialization, and this figure can be multiplied repeatedly. The variable *bin_width* lies in this category, and also, this value is now calculated once (during initialization, from the frame and rectangle information known) and used repeatedly, instead of repeated divisions by this value. Optimizations performed in specific functions are mentioned below.

1) Pdf_representation: The inner loop in *Pdf_representation* does not contain complex computations and can be improved a lot by the loop unrolling. In the histogram generation process, the access to the certain bin of output histogram depends on the input domain and can not be parallelized directly. The approach is that creating multiple sub-histograms being maintained and finally merge into a final histogram. But this approach works better if performed on DSP, because of its architecture, but since parallelization was to be achieved, the version of *pdf_representation* has been kept intact on the ARM, and parallelism was exploited though some other computations, as explained in the ARM+DSP section.

2) CalWeight: In the original *CalWeight* function,when calculating the weight, there are three square root, three division and three multiplication operations. The square root and division operations are very costly ,compared to the multiplication, thus we reshape the formula to an efficient one which has one square root, one division and six multiplications, as shown below.

$$\sqrt{a/e}\times\sqrt{b/f}\times\sqrt{c/g} \quad -> \quad \sqrt{(a\times b\times c)/(e\times f\times g)} \quad (1)$$

B. Optimized code on $ARM + DSP$ - parallelizing using DSP

Apart from the optimizations for ARM in general as mentioned above, optimizations for DSP operations have also been sought. In the track function, due to the data dependency between *Pdf_representation* and *CalWeight*, these two function are called sequentially. However, inside each function, computations can be done in parallel. Because of the complex computation in the *CalWeight*, this function is decided to be executed on ARM and DSP in parallel. ARM and DSP are responsible for calculating different parts of the weight matrix for the tracking rectangle. As soon as the DSP is notified, starting its calculation on lower part, ARM starts computing the weights of pixels in the upper part parallely. Once DSP

finishes its work, the calculated weights of pixels are transfered to ARM where the final result is merged.

1) CalWeight: The afore-mentioned division and square root operations in the *CalWeight* function are done by the *IQmath* library, _IQdiv and IQmpy specifically) on the DSP. Since the computation resources on the DSP is limited and there are already complex computations inside the loop, the unrolling does no help much. To optimize the *CalWeight* function further, those variables that stay consistent in the loop are moved outside that loop, and loop parameters is added before the loop for compiler optimization.

The matrices to be sent to the DSP are linearized as shown in Fig. 11. This involves forming the array before sending it to the DSP and loading data from array to merge into the final weight matrix to be written. These load and retrieve operations require considerable computation, and hence the speedup achieved is not as expected. Apart from

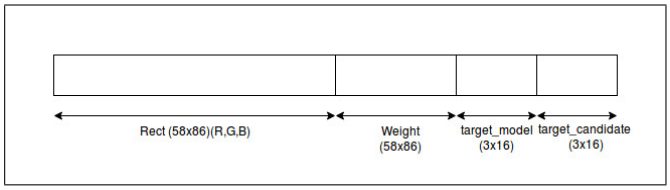

Fig. 11: The linearized data buffer for DSP

the computations, the communication between DSP and ARM also takes time. A improvement is done with regard to cache writeback function. Instead of writing back the cache to the whole pool buffer, DSP only write back the updated variable. According to the timing information, writing back the cache memory to pool buffer of a size of 3*58*86*4 Bytes takes 60 us approximately. In consideration of the large amount of total iterations, this change is expected to give the whole application considerable improvement.

C. Optimized code on $ARM + NEON$

In addition to the ARM processor, the Neon co-processor is utilized to execute instructions that perform the same operation over multiple data. Neon intrinsics are utilized to perform vectorized operations and produce a possible speed up to the application. The following functions are vectorized in the `meanshift.cpp`.

1) Pdf_representation: The calculation of 'bin_value' from the current_pixel_value is vectorized, with the red, green and blue values of the current pixel and the blue value of the next pixel being stored in an array and loaded in to a float quadword sized vector. Using the Neon intrinsic to multiply a 4 laned quadword sized floating point vector with a scalar quantity - 'vmulq_n_f32', the bin values for B,G,R and B (of the next pixel) are obtained. The G 'bin_value' and the R 'bin_value' of the next pixel are calculated in the ARM. The ARM and the Neon processor share the same integer pipeline. As the Neon instructions are executed, the integer pipeline can still execute the ARM instructions, thus enabling parallelization in execution. This helps achieve an increased speed up.
The multiplication *'m_kernel.at<float>(i,j) * normalized_C'* is performed on the Neon. The value of *'m_kernel.at<float>(i,j)'* and *'m_kernel.at<float>(i,j+1)* are stored in an array and are loaded into a vector using the Neon intrinsic - 'vld1_f32'. Using the Neon intrinsic - 'vmul_f32' , that performs vector multiplication but on 64 bit d registers, the multiplication is performed. The addition and updation on *pdf_model.at<float>(int,int)* are performed by extracting the added result from the vector using the Neon intrinsic 'vgetq_lane_f32', adding it with the *pdf_model.at<float>(int,int)* and updating it back. This is performed in the ARM.

2) CalWeight: The algorithm implemented under this function in the optimized code is vectorized with Neon instructions. As in the optimized code in the section above, the current pixel value and corresponding bin_value are calculated, however the calculations are performed for B,G and R for 4 pixels at a time. The target_model.at<float>() values for all blue, green and red components of 4 pixels are stored in *target_model_vec_0*, *target_model_vec_1* and *target_model_vec_2* respectively. These are Neon quadword floating point vectors.Similarly, *target_candidate.at<float>()* values for the blue, green and red components for 4 pixels are stored in the Neon quadword vectors *target_candidate_vec_0*, *target_candidate_vec_1* and *target_candidate_vec_2* Neon vectors, respectively. This particular approach maximizes the usage of all places in a quadword Neon vector. The multiplication, division and the square root in the algorithm are further performed on these two vectors using the Neon intrinsics.The 'vmulq_f32' performs the multiplication of the two quadword registers. Division is performed by multiplying the reciprocal values. The reciprocal estimate is obtained using the intrinsic 'vrecpeq_f32, whose precision is further refined by multiplication intrinsics. Finally, the vrsqrteq_f32 provides a reciprocal square root estimate, which is then assigned to the weight of the corresponding pixel position.

D. Optimized code on $ARM + NEON + DSP$ - parallel execution on the NEON and the DSP

This approach merges all previous approaches and contains the parallel computation between NEON and DSP in *CalWeight*. The same as Approach `B`, lower rectangle are sent to DSP but the *CalWeight* for upper rectangle is done with NEON rather than ARM. A division factor *F* is chosen to divide the rectangle as the upper *58-58/F* rows and the lower *58/F* rows. Lower part is handled by DSP Considering the computation ability of NEON and DSP in our implementation and the communication overhead, the division factor *F* should be properly setup.

V. Experiments & Results

A. Experiment for ARM+NEON+DSP Approach

By changing the division factor *F*, the workload for assigning values into pool varies a lot. Balancing the DSP and NEON workload requires an appropriate *F* to divide the task. The

IQmath library is not as fast as we expected, which makes the calWeight execution on DSP much slower than the execution on NEON. It is envisioned that the optimal splitting factor F would be relatively large, in other words, less workload will be done by DSP. Besides, with more rows sent to DSP, more assignment instruction are needed. Experiments were performed with F varying from 2 to 15 and $F = 9$ is decided based on its relative better performance.

TABLE I: Speedup Results

Approach	Total Time ms	Speedup	Track Time ms	Speedup
ARM Baseline	12794	1x	10509.4	1x
ARM Optimized	2907	4.4x	696.889	15.08x
ARM + DSP	2843	4.5x	608.535	17.27x
ARM + NEON	2667	4.8x	495.726	21.20x
ARM + NEON + DSP	2819	4.5x	602.505	17.5x

Compared to the ARM + NEON, the approach ARM+NEON+DSP should provide a better speed up since the computations are shared with DSP, but it does not in fact. Because the communication overhead takes about 80ms according to timing information.

B. Error Analysis

To determine the quality of tracking we identified deviation in the brightness of Red channel in our video and compared the value of standard deviation with the tracking result generated by the baseline code on PC. $ffmpeg$ was used to split the video into individual images. These images were loaded one-by-one in MATLAB and red channel values were extracted from that. Later the values of red channel were used to identify the standard deviation in the brightness of the image. As our tracker rectangle is of red color, deviation in brightness of red will determine the position of tracking rectangle. We have plotted frame-wise value of standard deviation of red channel for original and final approach.

TABLE II: Error Analysis: Standard Deviation of R-Channel

Frame Number	Baseline	Final	% deviation
Frame 3	51.2175	51.6825	0.9078928101

VI. Conclusion

Thus, we obtained a speedup in track of 17.5x and 4.5x for the total application by using DSP+NEON+ARM and parallelized operations. Also, it is identified that the *calWeight* execution on DSP is much slower than that with NEON, and on DSP, most of the time is consumed by _IQdiv() and _IQsqrt(). We also find that the tracking is still successful even when the square root operation is removed. That is to say, the accuracy of the weight of pixels does not matter so much, thus, some linear approximation can be used for division and square operation, aiming to speed up the execution on DSP side.

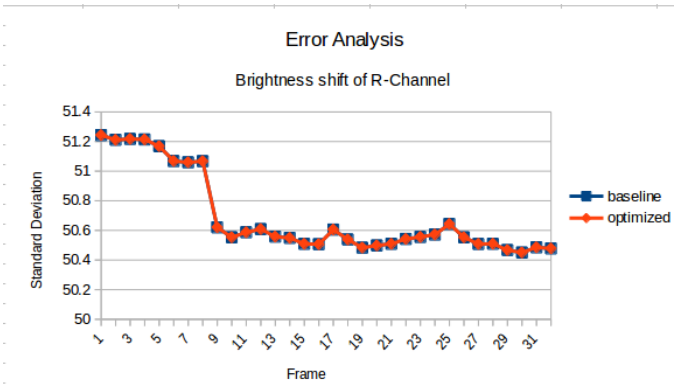

Fig. 12: Standard Deviation of Brightness in R-Channel of Video The baseline and the optimized graph overlap each other indicating a minimal deviation of the implemented tracking from the baseline tracking

.

References

[1] Arm Cortex A8 Technical Reference Manual. *http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_*

[2] ARM NEON Official Website. *https://developer.arm.com/technologies/neon*.

[3] Neon Programmers Manual. *https://developer.arm.com/technologies/neon/intrinsics*.

[4] Arm Cortex A8 Technical Wiki. *http://processors.wiki.ti.com/index.php/Cortex-A8*.

[5] A Guide to Vectorization with Intel® C++ Compilers. .

[6] An Introduction to GCC for the GNU Compilers gcc and g++. .

[7] TMS320C64+ Fixed-Point Digital Signal Processor Product Guide. *http://www.ti.com/product/TMS320C6414*.