IN4342- EMBEDDED SYSTEMS LABORATORY
ASSIGNMENT 02 REPORT
GROUP 6

TEAM MEMBERS
Charalampos Papadopoulos(4192141)
Luis Garcia Rosario(4062949)
Preethi Ramamurthy(4190742)
Vasilis Vasilopoulos(4188047)

INTRODUCTION:

This assignment dealt with the porting of the Canny Edge Detection application developed on BeagleBoard, namely an open-source platform featuring the TI OMAP3530 system-on-a-chip. The goal was to improve the performance of the application by at least 25%, employing the multicore heterogeneous architecture of the OMAP3530. The algorithm itself is a purely computational one consisting of four stages as follows:

1. Use of a Gaussian filter to remove the noise - The noise in the image is removed by using the Gaussian filter, where the raw image is convolved with a Gaussian filter. The final result is a slightly blurred version of the original image.
2. Determination of the edge strength - The horizontal, vertical and diagonal edges is detected using four filters. The image gradient is then found to highlight regions with high spatial derivatives.
3. Application of Non-Maximal Suppression – The regions with high spatial derivative is tracked and the pixels which are not at the maximum are suppressed.
4. Application of hysteresis – Canny uses thresholding with hysteresis to further reduce the gradient.

EXECUTION ON THE DSP AND THE ARM:

In the **baseline version**, the canny detection algorithm was executed only on the ARM with -o3 optimisation level. The flow of the functions is as follows:

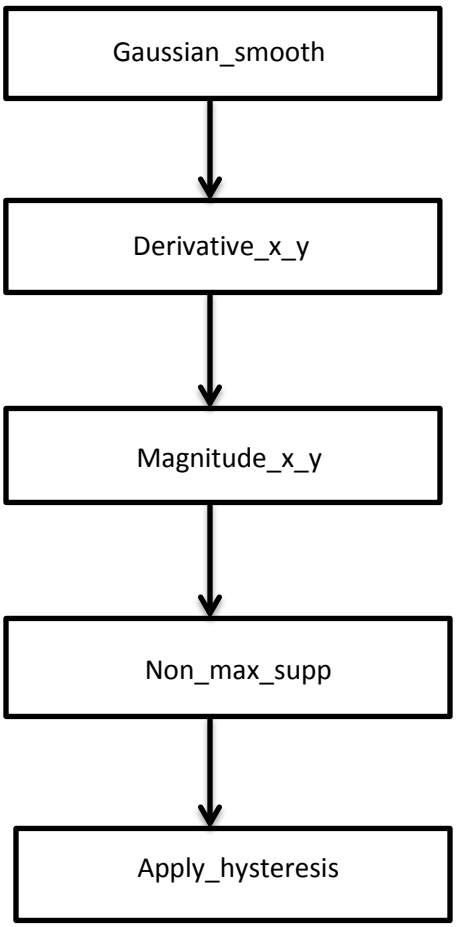


Figure1. Flow chart of the functions executed on the ARM

The execution time on the ARM for the canny edge algorithm was found to be 443360us.

Version	**Execution time in us**		
Baseline version	Klomp image	Shapes image	Square image
	443222	151874	111499
Baseline version with pool_notify	587994	183185	137027

Table 1 Average Execution time of the baseline version

We also executed the baseline version with the pool_notify routine, to get an idea about the overhead introduced due to the communication with the DSP.

Execution time of the functions on the ARM for the klomp image(having activated the POOL and NOTIFY components, without using them though):

Function	Execution time in us
Gaussian_smooth	298370
Derivative_x_y	7111
Magnitude_x_y	211303
Non_max_supp	64789
Apply_hysteresis	9278

In the **optimized version,** the canny algorithm has to be implemented on the DSP and the ARM. To accomplish this, we used the information from profiling data obtained when the canny algorithm is executed on the ARM alone. The profiling data suggests that the software floating point functions like "__adddf3" and "__aeabi_fadd" are the most time consuming function. This is the software library for floating point functions used on the ARM processor since the processor does not have a floating point (hardware) unit in its architecture.

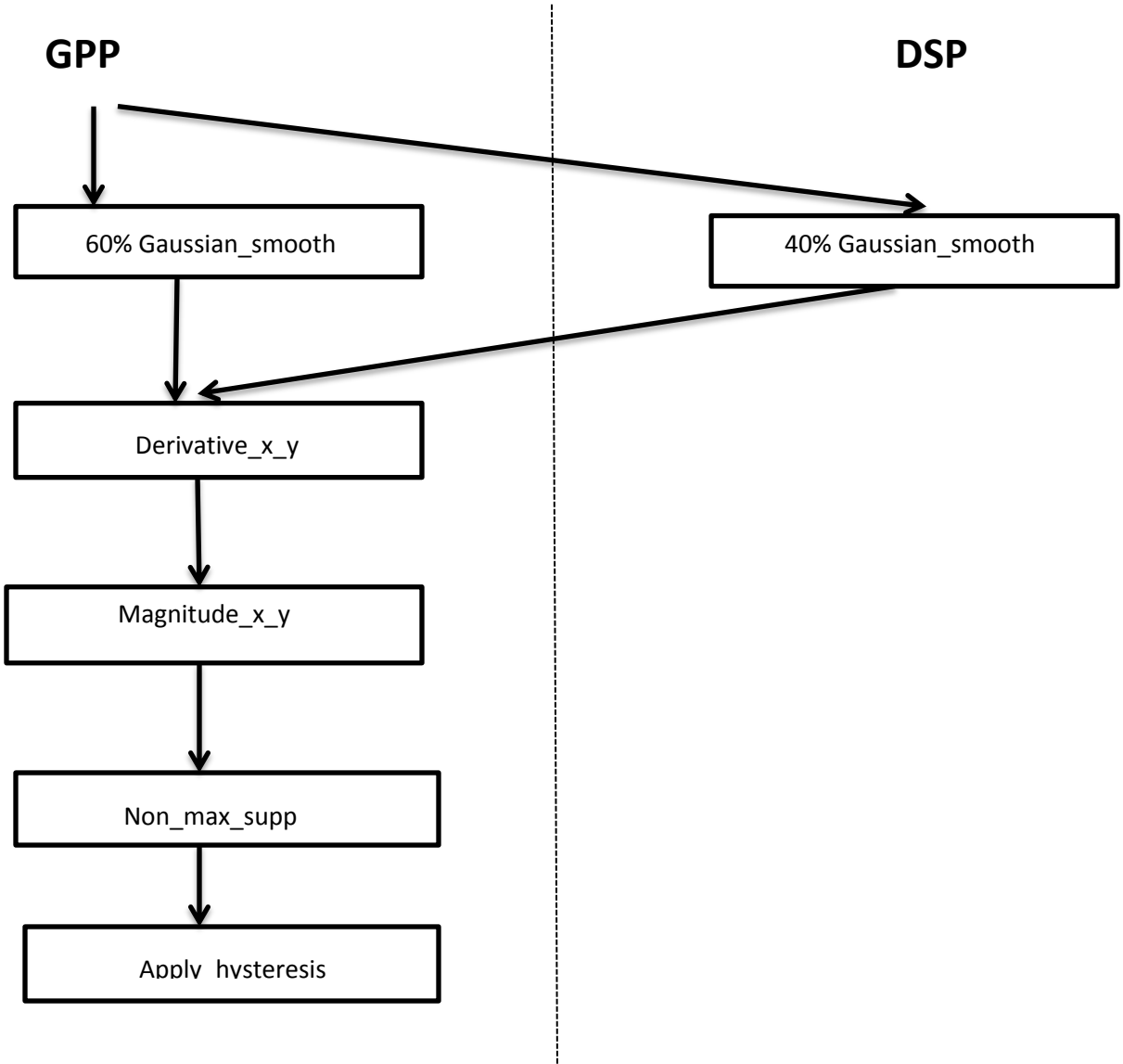


Figure 2: Flow of functions on the ARM and DSP for the optimized version

From the profiling results, it was not clear about which function(s) use the software floating point functions. To obtain this information, we turned our attention on the function that was being called the most. The result of this analysis showed that the functions 'gaussian_smooth' and 'magnitude_x_y' were being called the most.

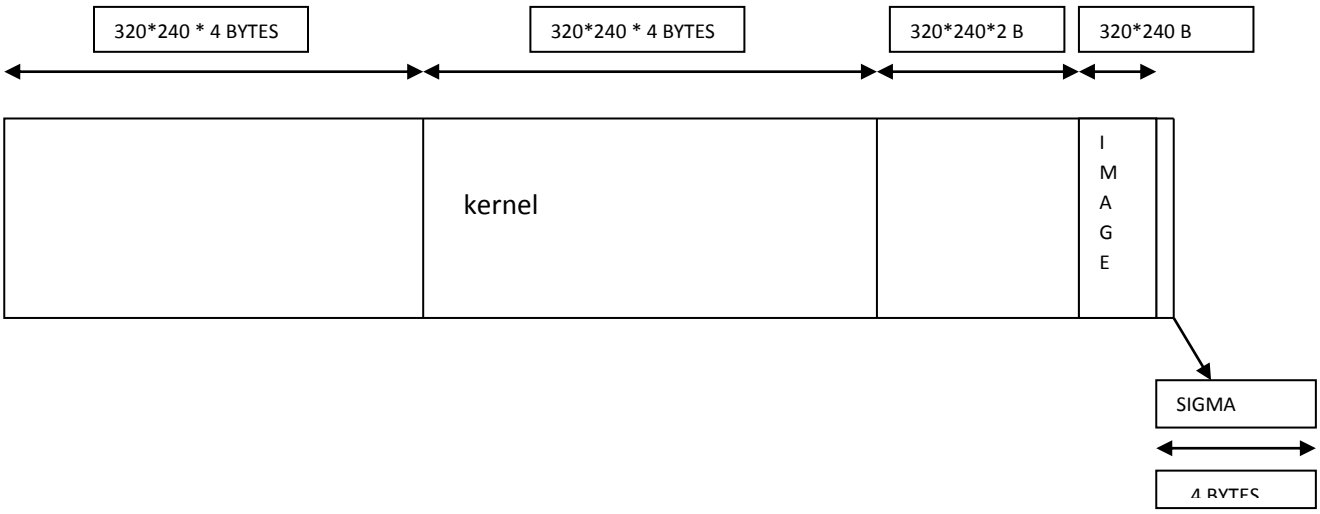
The idea is to execute the functions that do addition/multiplication/division single precision (float) and double precision (double) floating point values on the DSP side. For this purpose, we have ported the 'gaussian_smooth' function to the DSP. 40% of the 'gaussian_smooth' executes on the DSP and 60% of the 'gaussian_smooth' is executed on the GPP. The processing on the GPP and the DSP is done in parallel. And the results from the execution on DSP and GPP is given to the 'magnitude_x_y' function, which continues execution on the GPP side now.

In order to improve the performance, we studied the ranges of the floating point values used in the 'gaussian_smooth' and implemented the floating point to fixed point conversion using the 16-bit fractional part.

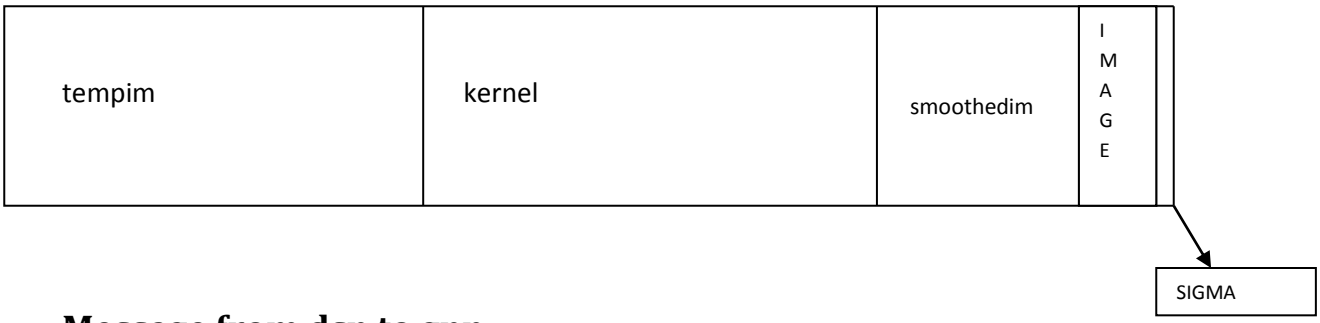
Version	**Execution time in us**		
Optimized version	Klomp image	Shapes image	Square image
	140781	49539	36148

Table 2 Average Execution time on the ARM and DSP for the optimized code

Message inside buffer from gpp to dsp:



Buffer during execution on the dsp side:



Message from dsp to gpp

COMMUNICATION OVERHEAD:

Taking as an example the klomp.pgm image but without posing any threats to the validity of our claims, we would like to mention that when the algorithm executed on the ARM alone, it was observed that the execution time was about 200,000us larger than the execution time of the code running on the ARM without having activated the POOL and NOTIFY components. When we began porting functions on DSP, the execution time was rendered even higher and the communication mechanism more intricate thus adding extra communication overhead in the overall design. This is owing to two reasons: 1) the communication between the ARM and the DSP takes place through the shared memory using the DSP/BIOS, and 2) DSP operates at a lower frequency compared to the ARM processor

LIMITATIONS OF MESSAGE PASSING

The total memory available for a memory pool is 851968 bytes, which is a limitation when the messages of larger size has to be passed. Another limitation is that DSP is not well suited for floating point operations and fixed point implementation works much faster than floating point on DSP.

MEMORY LIMITATIONS

The memory was limited on the DSP side and there was not sufficient space to allocate images. Following from that, we used only the memory pool which proved to be more than enough for the purpose of this assignment.

OPTIMIZATIONS FOR DSP

As the DSP does not support the floating point operations, we had to implement the 16-bit fixed point implementation for converting the floating point to fixed point implementation. We made many versions of the code and studied the trade-off of the performance when the code was implemented on the DSP and GPP. From the result of the study, we concluded that implementing 40% of the 'gaussian_smooth' on the DSP and 60% of the 'gaussian_smooth' on the ARM gave the best performance result.

DSP COMPILER OPTIONS THAT MAY IMPROVE PERFORMANCE

Remove –ml3: The –ml3 flag compiles all calls as far calls. Beginning with CCStudio 3.0 (C6000 compiler version 5.0), the linker automatically fixes up near calls that do not reach by using trampolines. In most cases, few calls need trampolines, so removing –ml3 usually makes code a few percent smaller and faster. By default, scalar data (pointers, integers etc.) are near and aggregates (arrays, structs) are far. This default works well for most applications.

Remove –g: The –g flag supports full symbolic debug. It is great for debugging, but when used in the production code –g inhibits code reordering across source line boundaries and limits optimizations around function boundaries. This results in less parallelism, more nops and generally less efficient schedules. This can cause a 30-50% performance degradation for control code, generally somewhat less but still significant degradation for performance critical code. Moreover, beginning with C6000 compiler version 5.0, basic function-level profiling support is provided by default.

OPTIMZATIONS FOR ARM

On the ARM side, we performed the optimization on the sqrt(), as the square root function using the math.h library takes much of the execution time. The code used for optimization is as follows:

```
unsigned long isqrt(unsigned long x)
{
  register unsigned long op, res, one;
```

```
  op = x;
  res = 0;
  /* "one" starts at the highest power of four <= than the argument. */
  one = 1 << 30;  /* second-to-top bit set */
  while (one > op) one >>= 2;

  while (one != 0) {
    if (op >= res + one) {
      op -= res + one;
      res += one << 1;  // <-- faster than 2 * one
    }
    res >>= 1;
    one >>= 2;
  }
  return res;
}
```

OUTPUT IMAGES:

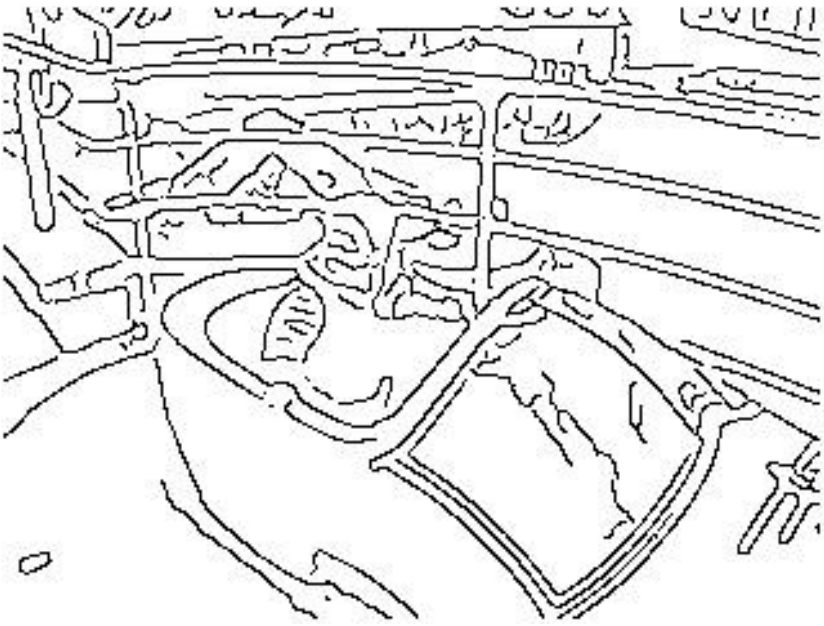


Figure 3: Klomp image output using optimized code

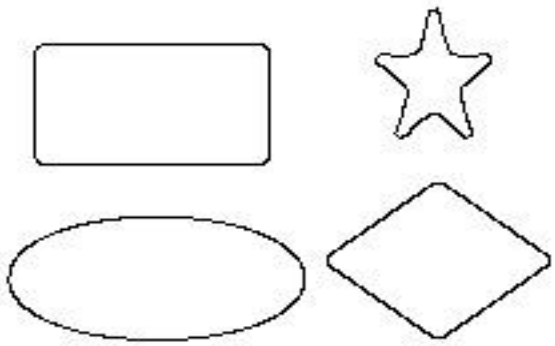


Figure 4: Shapes image output using the optimized code

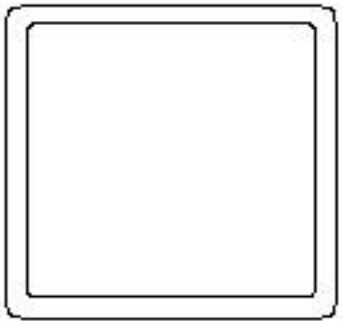


Figure 5: Square image output using the optimized code

By visually inspecting the output images, there seems to be no difference as for the shapes and square images, whereas this is not the case for the klomp image. In order to reach a safer conclusion, we wrote a code which informs the user about the difference between two images. The results have shown that the output images differ by at most 740 pixels which corresponds to less than 1% of the total image sizes.

SPEED UP ACHIEVED:

According to Amdahl's Law, the speedup is calculated according to the following equation:

$$Speedup = \frac{Execution\ time\ for\ entire\ task\ without\ using\ the\ enhancement}{Execution\ time\ for\ entire\ task\ using\ the\ enhancement}$$

In addition, we also calculate the performance enhancement using the following formula:

$$\%Enhancement = \frac{(Execution\ time\ of\ the\ canny\ for\ the\ baseline\ version)}{Execution\ time\ of\ the\ canny\ for\ the\ optimized\ version}$$

Version	Picture	Execution time	Speed up	%Enhancement
Baseline version	Klomp	443222	--	--
	Shapes	151874	--	--
	Square	111499	--	--
Optimized version	Klomp	140781	3.1483	68.24
	Shapes	49539	3.0657	67.38
	Square	36148	3.0845	67.58

Table 3: Speed up of the optimized version over the baseline version

As can be seen from the table, we achieved an average of 67% enhancement for the optimized version over the baseline version. This speed up can be attributed to the fact that we use the correct compiler optimization flags, fixed point implementation and the optimized square root function.

CONCLUSION:

In conclusion, spending more and more time on the assignment, it was becoming more and more clear that the design space exploration of the assignment is quite wide. Doing professional rather than student-wise work, we managed to achieve a performance improvement of approximately 65%. This was done parallelizing the execution of the canny edge detection application, utilizing fixed point instead of floating point values on both processors and finally making use of an efficient integer square root calculation algorithm instead of the standard sqrt() function. Last but not least, we would like to emphasize the learning process which seems to be quite meaningful.

REFERENCES:

[1] http://en.wikipedia.org/wiki/Canny_edge_detector

[2]http://webcache.googleusercontent.com/search?q=cache:bnLWh_BEEkgJ:www.codecodex.com/wiki/Calculate_an_integer_square_root+&cd=1&hl=en&ct=clnk

[3] http://beagleboard.org/

[4] http://www.ti.com/product/omap3530