



Porting and Optimization Techniques for C++ Based Image Processing Algorithms on TMSC62x DSP

Ketul Patel
Project Manager – DSP Group
Embedded Division
einfochips Inc.
ketul@einfochips.com
www.einfochips.com

Overview

- ◆ Porting Methodology for C++ based Algorithms to TMS320C62x DSP
- ◆ Optimization Techniques



Agenda



◆ Image Processing Algorithm – *Need for a Porting Methodology*



◆ Methodology

- Converting C++ based architecture to C
- Optimizing Memory and Stack Usage
- Using DSP Specific Features
- Optimizing critical Loops and Logic
- Optimizing Floating-Point Calculations



◆ Performance Statistics

Image Processing Algorithm

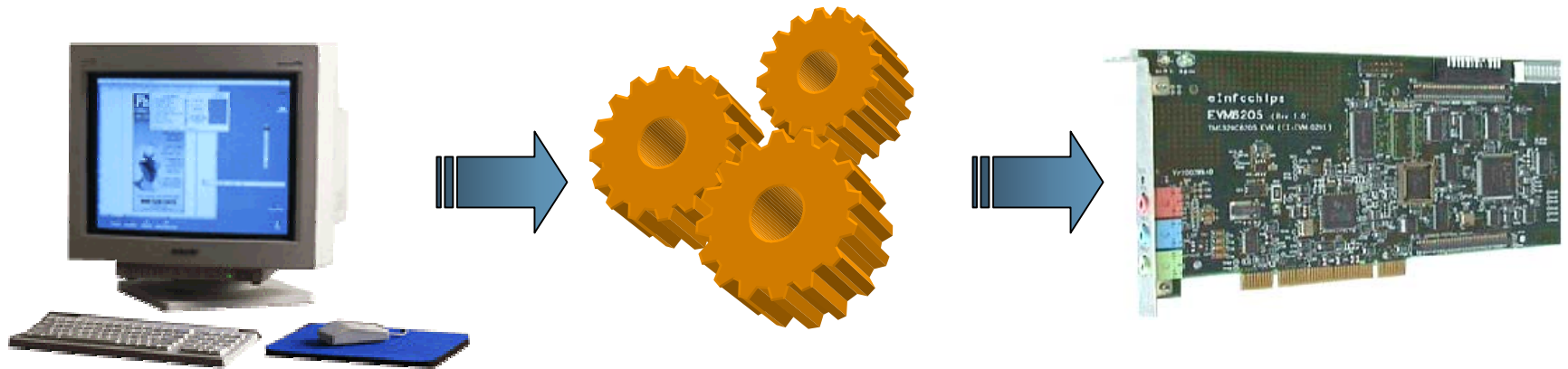


Image Processing Algorithm

Need for a *Porting Methodology* for an Embedded Platform

- **Algorithm Characteristics:**
 - Large Data Transfer
 - Computation Intensive Processing

Need for a Porting Methodology ...

◆ Platforms

	PC Platform	Embedded Platform
Processor Speed	In GHz	In MHz
System Memory	Large (100 MB)	Small (10 MB)
Program Memory	Large (in MB)	Small (in KB)
Stack	Deep call tree	Limited Depth
Algorithm Code	Generic – Research work	Custom – Application Specific

Algorithm Porting Challenge

- ◆ Porting C++ based Image Processing Algorithm with **50 times** performance improvement on TI C6205 DSP for IDEO Inc. (www.ideo.com)
- ◆ Original Algorithm is a Research work on PC platform
- ◆ Ported Algorithm
 - Should be Compatible to PC and DSP platforms
 - Should not use processor specific assembly code
 - Should Maintain Readability of the Optimized 'C' code

Performance on TI DSP TMS320C6205	Original	Targeted
	5 Second / Frame	100 mS / frame

Porting Methodology

It's an Art.....



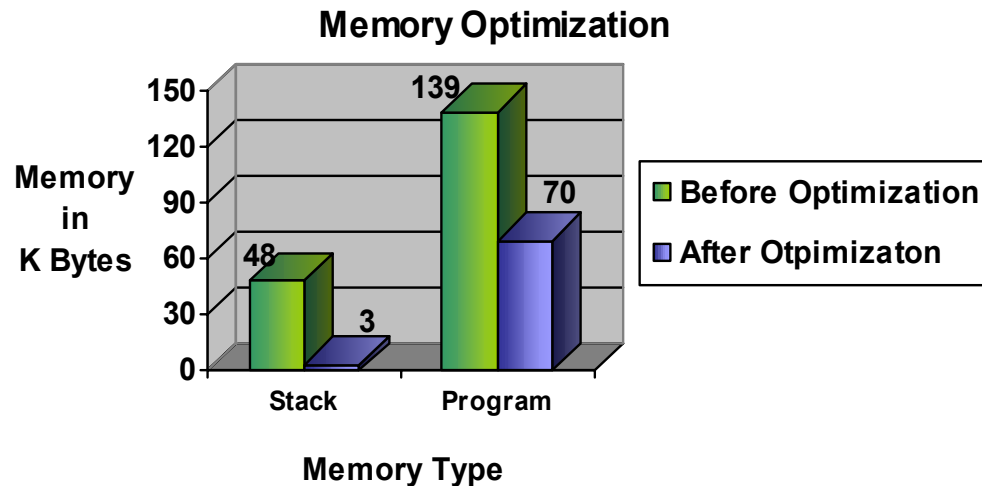
- ◆ Migration: C++ to C
- ◆ Optimizing Memory and Stack Usage
- ◆ Use of DMA and Internal Memory
- ◆ Use of Platform Specific Features
- ◆ Optimizing critical Loops and Logic
- ◆ Optimizing Floating-Point Calculations

◆ Migrating from C++ to C

- Class to Structure
- Initialize / Destroy Structure Members
- Remove Parameters passed by reference in C++ methods
- Simplify Class Hierarchy: To reduce referencing levels
- Remove C++ Templates

Performance Improvement Factor: ~2

◆ Optimizing Memory and Stack Usage

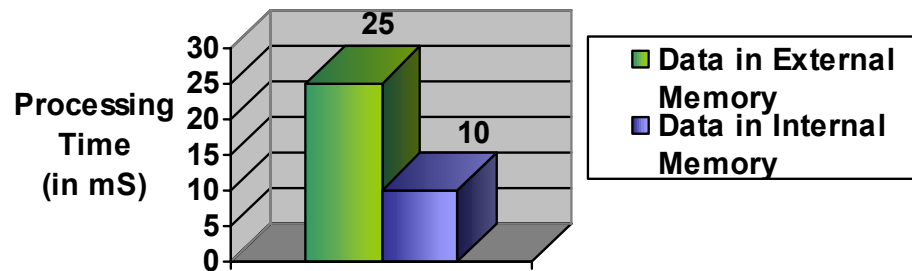


- Dynamic Vs Static Memory Allocation
- Local Vs Global Variables
- Reduce Parameter Passing
- Structure and Array Parameters as Pointers

Performance Improvement Factor: **~1.4**

◆ DMA and Internal Memory Usage

Image processing - Performance Improvement



- Replace standard 'memcpy', 'memset' APIs with similar functionality implemented using DMA
- Use of 32-bit aligned DMA transfers
- Process Image Data in Internal Memory

Performance Improvement Factor: **~2.5**

◆ Using TI DSP Specific Features

- **Efficient Memory Configuration**
 - **Code Segment – Use IPRAM as Cache for large Programs**
 - **Data – Use IDRAM for Global Variables**
 - **Stack – Use IDRAM for Faster Code Execution**
- **Powerful DMA**
 - **Block Transfer for moving Image Data**
 - **Trigger DMA start on Event**
- **Parallel Execution**
 - **Execute multiple instructions in parallel**

◆ Using Platform Specific Features

■ Choice of Data Types

- Size of DSP registers (or data bus) Vs variable data type
- Avoid use of 'Bit Field' variables
- Use of Data values Vs Pointers in high frequency execution areas

■ Use of Global Vs Local Variables

- Number of references of the Variables
- Use of 'register' for Local Variables

Performance Improvement Factor: ~3

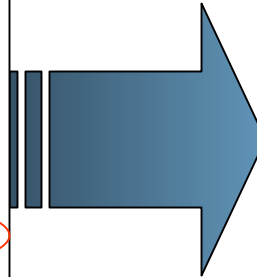
◆ Loop Optimization

- **Loop Fission: Split large loop into multiple small loops (Increases instruction level parallelism and decreases loop overhead)**
- **Loop Fusion: Combining multiple loops into a single loop (Increases cache locality of big loops, reduce the data copy overhead to internal memory)**
- **Avoid Function call within loop**

◆ Inline Function

```
void function()  
{  
    int r, c;  
    int x = 100;  
    int outA[480][640];  
    int outB[480][640];  
  
    for(r = 0; r < 480 ; r++ )  
    {  
        for( c = 0; c < 640 ; c++ )  
        {  
            outA[r][c] = x - inA[r][c];  
            outB[r][c] = function_1( outA[r][c]  
, inB1[r][c] );  
        }  
    }  
  
    int function_1(short a, short b)  
    {  
        return ( (a*a) + (a*b) + (b*b) );  
    }  
}
```

Execution Time: **450 mS**



```
void function()  
{  
    int r, c;  
    int x = 100;  
    int outA[480][640];  
    int outB[480][640];  
  
    for( r = 0; r < 480 ; r++ )  
    {  
        for( c = 0; c < 640 ; c++ )  
        {  
            outA[r][c] = x - inA[r][c];  
        }  
        for( c = 0; c < 640 ; c++ )  
        {  
            outB[r][c] = ( outA[r][c] * outA [r][c] )  
                + ( outA[r][c] * inB1[r][c] )  
                + ( inB1 [r][c] * inB1[r][c] );  
        }  
    }  
}
```

Execution Time: **380 mS**

◆ Logic Optimization

- Process Image Data in Blocks instead of Pixel (e.g. a Line)
- Process on ROI (Region Of Interest) instead of Full frame (Minimizes data transfer and data processing)

Note: Algorithm specific

Performance Improvement Factor: **~2.7**

◆ Floating-Point to Fix-Point

Code with Floating-Point arithmetic

```
void function( )
{
    double x, y, z;
    unsigned char result[100];
    unsigned char i = 0;

    for ( ; i < 100 ; i++ )
    {
        y = i * 0.0356;
        z = i * 0.1595;

        x = 0.5 + ( y * ( z - 1.0 ) );

        result[ i ] = (unsigned char) x;
    }
}
```

Execution Time: **2809 μS**

Code without Floating-Point arithmetic

```
void function( )
{
    long x, y, z;
    unsigned char result[100];
    unsigned char i = 0;

    for ( ; i < 100 ; i++ )
    {
        y = i * 356; /* 0.0356 * 10000 */
        z = i * 1595; /* 0.1595 * 10000 */

        x = 5000 + ( y * ( z - 10000 ) ) / 10000 );

        result[ i ] = (unsigned char) ( x /
        10000 );
    }
}
```

Execution Time: **1420 μS**

◆ Division Optimization

- Use multiplier and divisor in **power of 2 instead of power of 10**

```
void function( )  
{  
    long x, y, z;  
    unsigned char result[100];  
    unsigned char i = 0;  
  
    for ( ; i < 100 ; i++)  
    {  
        y = i * 584; /* 584 = 0.0356 * 2^14 */  
        z = i * 2614; /* 2614 = 0.1595 * 2^14 */  
  
        z = z - 16384; /* 16384 = 2^14 */  
        x = (y * z) >> 14; /*now x has only 2^14 multiplier*/  
  
        x = 8192 + x; /* 8192 = 0.5 * 2^14 */  
  
        result[ i ] = (unsigned char)( x >> 14);  
    }  
}
```

Execution Time: **39.4 μS**

◆ Performance at each Stage

Original Algorithm on TI C6205 DSP	~5 Seconds / Frame
C++ to C Conversion	2.5 Seconds / Frame
Optimizing Memory and Stack Usage	1.8 Seconds / Frame
DMA and Internal Memory Usage	800 mS / Frame
Logic and Loop Optimization	300 mS / Frame
Floating point to Fix point conversion	99 mS / Frame

Conclusion

“By systematic porting and careful optimization, the algorithms developed for PC processors - running at GHz speed, can be ported to DSP platforms - running at MHz speed with a real time system performance.”



Thank You...

Porting and Optimization Techniques for C++ Based Image Processing Algorithms on TMSC62x DSP

Ketul Patel
Project Manager – DSP Group
Embedded Division
eInfochips Inc.
ketul@einfochips.com
www.einfochips.com