C6000 Optimization Basic

Optimization Fields

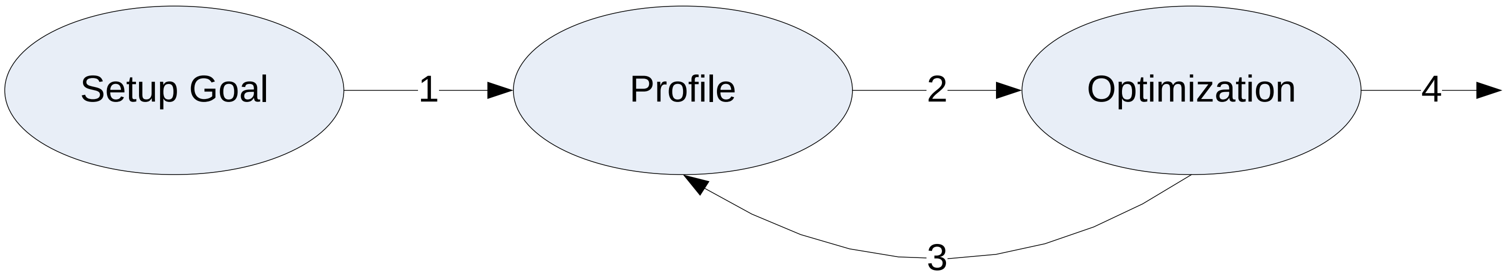
- **Optimization is a system project, there are many fields should be considered:**
  - **Framework optimization**
  - **Algorithm optimization**
  - **Code optimization**
  - **Memory optimization**

Framework and Algorithm optimization are application dependent. They are not covered here.

Agenda

- Optimization preparation
- Basic Optimization methods
- Software Pipeline
- Memory/Cache Optimization
- Other optimization tips and tricks

Optimization Procedure



1. **After decide the optimization goal, current runtime characteristics should be determined to decide whether the implementation can meet the goal.**
2. **After getting the benchmark data, the DSP capability need to be analyzed. Then the optimization direction can be determined.**
3. **For each optimization field, relative optimization methods can be applied to do optimization. This is a multi-loop procedure between profile and optimization.**
4. **If the goal is met, current procedure can terminate. If not, other optimization methods should be used or the goal should be adjusted accordingly.**

Profiling

- **Manually measure the cycles for a function**

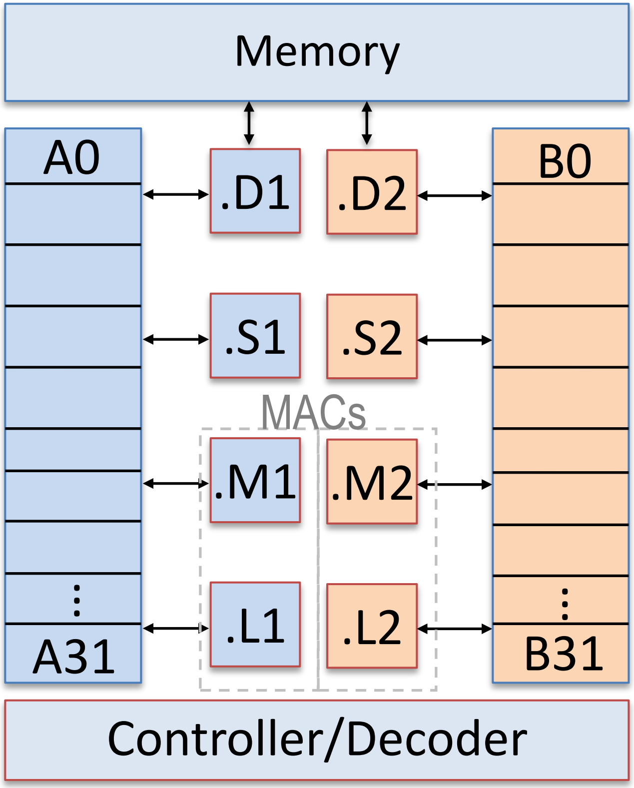
```
#include <c6x.h>
……
preTSC= TSCL;
foo();      //function under test
cycles= ((unsigned int)((0xFFFFFFFFl+TSCL)- (unsigned long long) preTSC)+ 1);
```

- **Profiling with function hook of cgtools**
- **CCS Profiler**

2	Address	ExecuteTimes	InclusiveCycles	ExclusiveCycles	AverageCycles	Percentage	FunctionName
3	c001eb8	9	8383166	8383166	931462	0.3108577	LDDWTest
4	c009a40	34	7331505	7331505	215632	0.27186087	MemCopy8
5	c001300	9	6041932	6041932	671325	0.22404198	STDWTest
6	c000000	125	3721869	3721869	29774	0.1380113	edma_Throughput_Test
7	c001604	34	7955610	624105	18356	0.02314255	MemCopy8Test
8	c001744	1	8364938	409328	409328	0.01517837	MemCopyTest
9	c005360	1	257305	257305	257305	0.00954117	edma_Init

C66x DSP VLIW Architecture

- Two (almost independent) sides, A and B
- 8 functional units, M, L, S, D
- Up to 8 instructions sustained dispatch rate



Basic capabilities of C64x+ core

Operation	Precision	Operations per cycle	Function Units	Notes
MAC	Real 8 x 8	2 x 4 = 8	M1, M2	1
	Real 16 x 16	2 x 2 = 4		
	Real 32 x 32	2 x 1 = 2		
	Complex (16,16)x(16,16)	2 x 1 = 2		
	Complex (32,32)x(32,32)	N/A		
Memory Access	8 bit 16 bit 32 bit 64 bit	2 x 1 = 2	D1, D2	2, 3
Arithmetic Logical	8 bit	4 x 4 = 16	L1, L2, S1, S2	
	16 bit	4 x 2 = 8		
	32 bit	4 x 1 = 4		

Notes

1, if no multiplication, M1, M2 may be used for some other operations.

2, if no memory read/write, D1, D2 may be used for other operations.

3, try to used 64 bit read/write as much as possible.

4, division and module is not supported with single instruction, try to avoid it in your algorithm

Basic capabilities of C66x core

Operation	Precision	Operations per cycle	Function Units	Notes
MAC	Real 8 x 8	2 x 8 = 16	M1, M2	1
	Real 16 x 16	2 x 8 = 16		
	Real 32 x 32	2 x 4 = 8		
	Complex (16,16)x(16,16)	2 x 4 = 8		
	Complex (32,32)x(32,32)	2 x 1 = 2		
Memory Access	8 bit 16 bit 32 bit 64 bit	2 x 1 = 2	D1, D2	2, 3
Arithmetic Logical	8 bit	4 x 8 = 32	L1, L2, S1, S2	
	16 bit	4 x 4 = 16		
	32 bit	4 x 2 = 8		

Notes

1, if no multiplication, M1, M2 may be used for some other operations.

2, if no memory read/write, D1, D2 may be used for other operations.

3, try to used 64 bit read/write as much as possible.

4, division is supported by floating point instruction

5, single precision floating point operation capabilities is same as 32 bit fixed point operation

How do we determine the Optimum

- Analysis is specific to the algorithm.
- Raw performance for a given computation loop depends on:
  a) Number of loads and stores needed.
  b) Number of multiply operations needed.
  c) Number of Arithmetic and logical operations needed.
  d) Size of the data that is being worked upon (shorts, bytes).
- Given 'this' many operations and the capabilities of the architecture how long should it take to perform this algorithm?

Operations Required by an loop

- Key loops of algorithms need be analyzed to identify the operations required
- For example:

```
short a[BUF_SIZE], b[BUF_SIZE];
int i, sum;
for (i=0; i < count; i++)
 {
      sum = sum + a[i] * b[i];
 }
```

Operation	Operations per loop	Function Units
Multiplication	1 (16 bit)	M1, M2
Memory Read/Write	2 (16 bit: a[i], b[i])	D1, D2
Other operations	1 (32 bit: sum)	L1, L2, S1, S2

Estimate cycles for the loop on C64x+

- Without optimization, the cycles for the loop is 1

Operation	Operations per loop	Cycle
Multiplication	1 (16 bit)	1
Memory Read/Write	2 (16 bit)	1
Other operations	1 (32 bit)	1

- To fully utilized the DSP core capabilities, the cycle can be optimized to ¼, i.e. execute 4 loops in a cycle.

Operation	Operations per loop	Operations per cycle	Cycle per loop
Multiplication	1 (16 bit)	4 (16 bit)	¼
Memory Read/Write	2 (16 bit)	2 (64 bit)	¼
Other operations	1 (32 bit)	4 (32 bit)	¼

Create small project for optimization

- **Small project should be created for key functions to verify the optimized code easily and quickly.**

- **The small project should:**

1. **Prepare input**
2. **Use original correct code to generate test results**
3. **Call optimized code**
4. **Compare the results of optimized code with original results**

Example of Optimization Verification

```
#include <c6x.h>
#define TSC_getDelay(preTSC) ((unsigned int)((0xFFFFFFFFl+TSCL)-\
        (unsigned long long)preTSC)+ 1)
……
TSCL= 0;            //enable TSC
t1 = TSCL;
tsc_overhead= TSC_getDelay(t1);
……
t1 = TSCL;
ref_result = foo();
t1= TSC_getDelay(t1)-tsc_overhead;
printf("%8d    cycle for foo\n", t1);

t1 = TSCL;
result = foo_opt();
t1= TSC_getDelay(t1)-tsc_overhead;
printf("%8d    cycle for foo_opt\n", t1);
if (result!=ref_result)
    printf("error with optimization, result = %d, expected result = %d \n",
        result, ref_result);
```

Agenda

- Optimization preparation
- Basic Optimization methods
- Software Pipeline
- Memory/Cache Optimization
- Other optimization tips and tricks

Basic Optimization Directions

- Use compiler option correctly
- Tell compiler more information
- Manually optimize C code
  - Intrinsic
  - SIMD (Single Instruction Multiple Data)
  - Manually unroll loop

Optimization Example

- The following example code will be used to illustrate the optimization effect of each optimization method (count = 40)

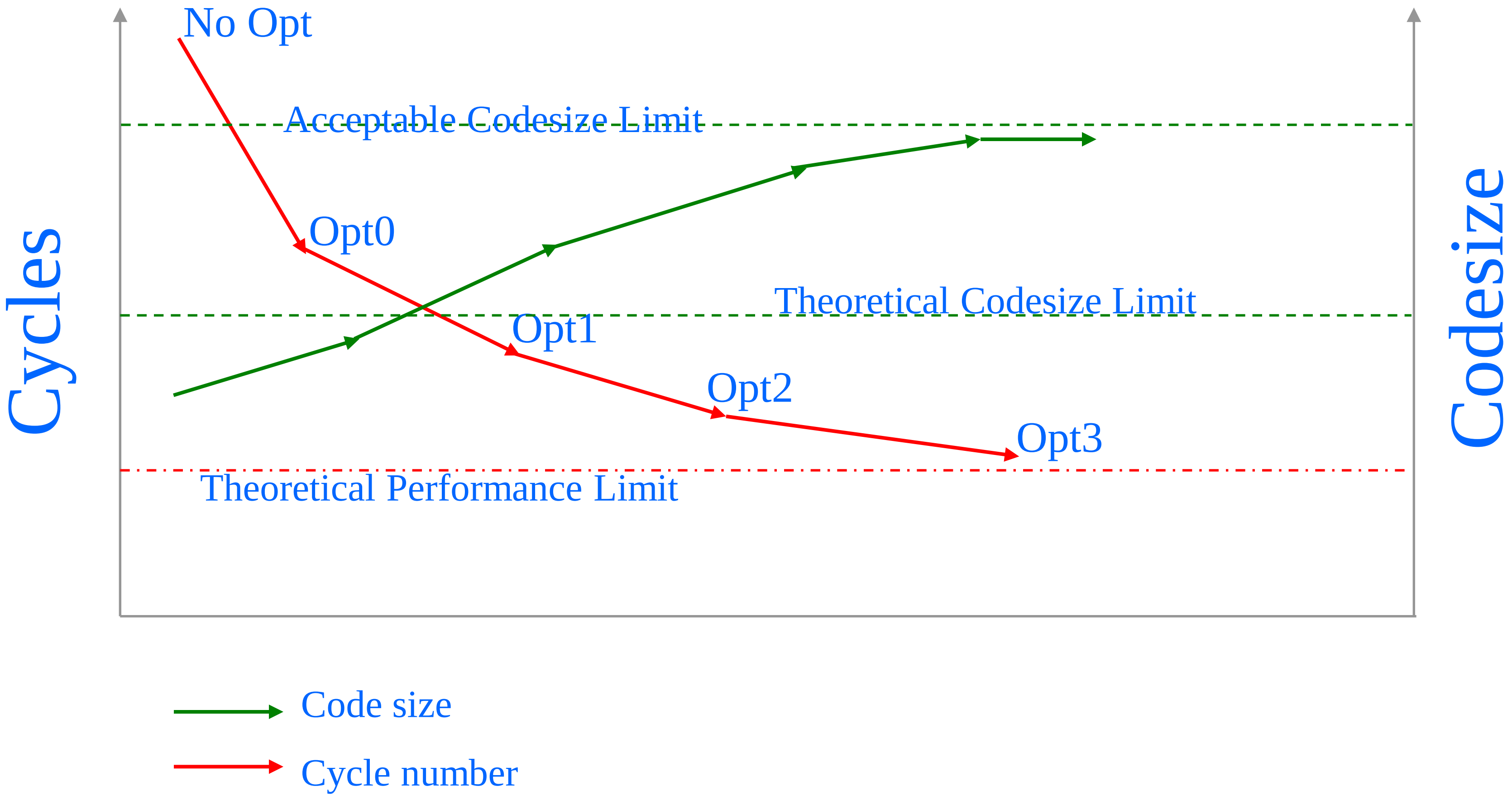
```
int dotp_c(short *a, short *b, int count)
{
        int i;
        int sum = 0;

        for (i=0; i < count; i++)
        {
            sum = sum + a[i] * b[i];
        }
        return(sum);
}
```

Choosing the "Right" Build Options

- –mv6600 enables 6600 ISA
- –o[2|3] = Optimization level. Critical!
  - –o2/-o3 enables SPLOOP (c66 hardware loop buffer).
  - –o3, file-level optimization is performed.
  - –o2, function-level optimization is performed.
  - –o1, high-level optimization is minimal
- –ms[0-3] is used if codesize is a concern:
  - Use in conjunction with –o2 or –o3.
  - Try –ms0 or –ms1 with performance critical code.
  - Consider –ms2 or –ms3 for seldom executed code.
  - NOTE: Improved codesize may mean better cache performance.

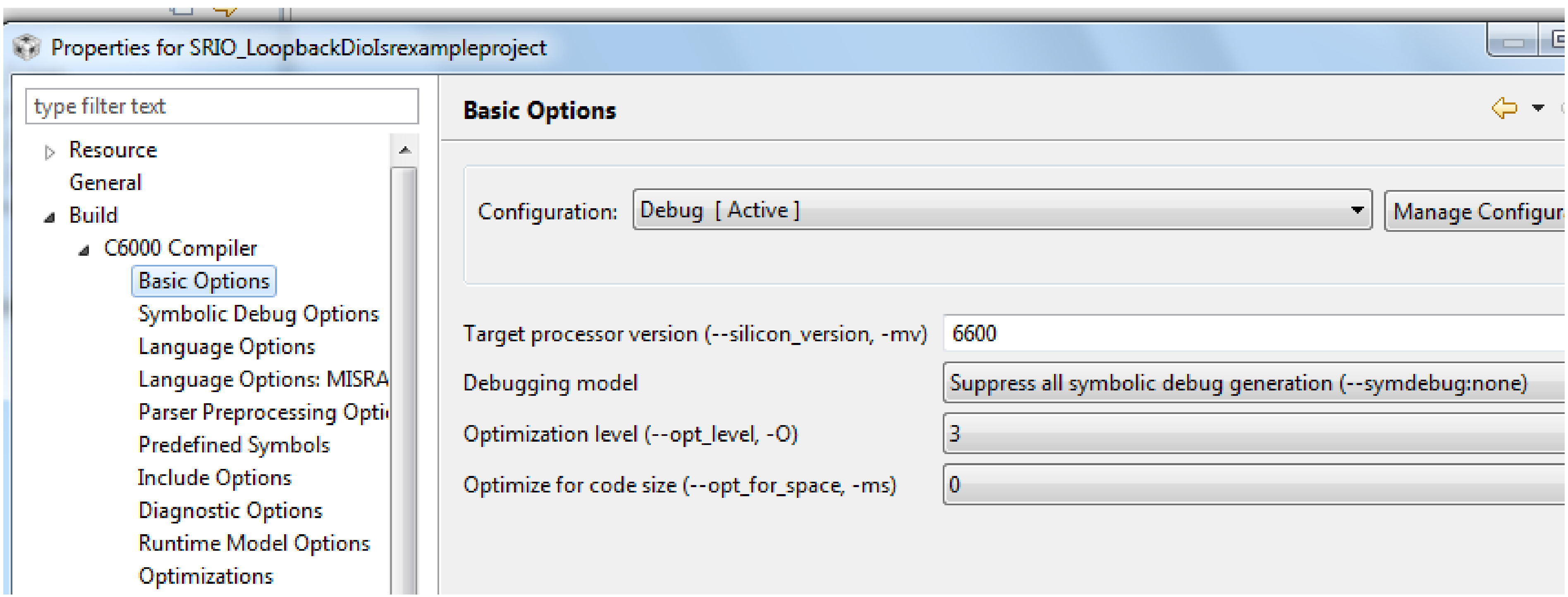
Compiler Optimization



The balance point is decided by the application integration goal.

Build Options for Optimization

- Select the “best” build options.
  - More than just “turn on –o3”!
- DO NOT use –g



Build Options to Avoid

- **–g** generates full symbolic debug. While it is great for debugging, it should **not** be used in production code.
  - Inhibits code reordering across source line boundaries
  - Limits optimizations around function boundaries
  - Can cause a 30-50% performance degradation for control code
  - Basic function-level profiling support now provided by default

- **–ss** generates interlist source code into assembly file.
  - As with –g, this option can negatively impact performance.

Optimization Result

Optimization Method	Cycles for Loop codes	Cycles for total Function
No optimization	1360	1397
-o3 option	35	93
No debug	35	83

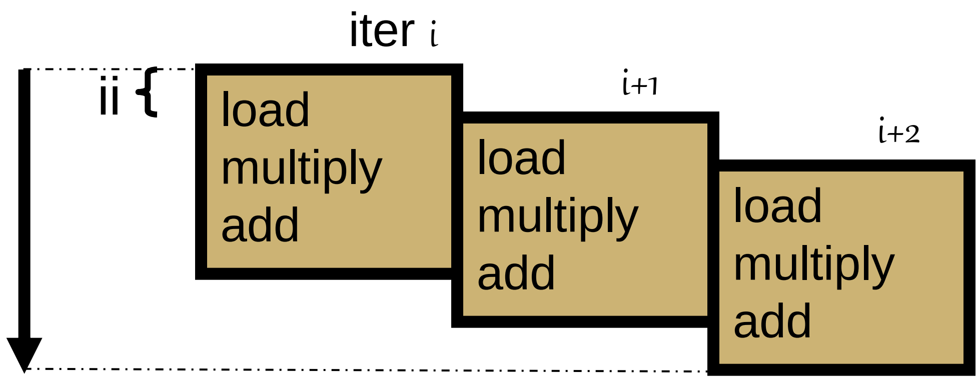
Tell Compiler more information

- **Memory Disambiguation**
  - **Restrict Keyword: Int32 * restrict ipDataAddr**
- **Knowing minimum/maximum loop iterations:**
  - **#pragma MUST_ITERATE(1, 8, 1);**
- **Aligning pointers on boundaries:**
  - **#pragma DATA_ALIGN(variable, 2^n alignment);**
  - **_nassert(((int)ipBuf & 7) == 0);**

Tell Compiler more information

```
int dotp_c(short *restrict a, short b[restrict], int count)
{
        int i;
        int sum = 0;
        _nassert(((int)a % 8) == 0);
        _nassert(((int)b % 8) == 0);

       #pragma MUST_ITERATE(8, 1000, 8);
        for (i=0; i < count; i++)
        {
              sum = sum + a[i] * b[i];
        }
        return (sum);
}
```



Optimization Result

Optimization Method	Cycles for Loop codes	Cycles for total Function
No optimization	1360	1397
-o3 option	35	93
No debug	35	83
Memory Alignment	35	75
MUST_ITERATE	28	41

Manually optimize C code

- Using intrinsic to do SIMD (Single Instruction Multiple Data) processing
  - Double word load/store
  - Packed data operation intrinsic: _dotp2, _add2, _shr2…
- Manually unroll loop if speed is most critical

```
#pragma UNROLL(4);
```

Intrinsic

- C code can directly use instructions by calling intrinsic function
  - Think of intrinsic functions as a specialized function library written by TI
  - Compiler will instantiates an instruction directly
  - Direct control over instruction selection
- All instructions that is not directly represented by C operator can be used in this way.
- #include <c6x.h>
  - has prototypes for all the intrinsic functions

Intrinsic Example

Intrinsics

int **_add2 (**int *src1*, int *src2* **);**
int **_dotp2 (**int *src1* **,** int *src2* **);**
unsigned **_norm (**int *src2* **);**

```
int x;
unsigned int y;
y = _norm(x);
```

Refer to *C Compiler User's Guide* for more information

Data Access Intrinsic

Data Access	Assembly Instruction
unsigned & **_amem4 (**void **ptr* **);**	`LDW, STW`
unsigned & **_mem4 (**void * *ptr* **);**	`LDNW, STNW`
const unsigned & **_amem4_const (**const void **ptr* **);**	`LDW`
long long & **_amem8 (**void **ptr* **);**	`LDDW, STDW`
long long & **_mem8 (**void * *ptr* **);**	`LDNDW, STNDW`
const long long & **_amem8_const (**const void **ptr* **);**	`LDDW`
double & **_amemd8 (**void **ptr* **);**	`LDDW, STDW`
double & **_memd8 (**void * *ptr* **);**	`LDNDW, STNDW`
const double & **_amemd8_const (**const void **ptr* **);**	`LDDW`

Refer to *C Compiler User's Guide* for more information

32 ←→ 64

Intrinsic	Description
unsigned **_loll (**long long *src***);**	Returns the low (even) register of a long long register pair
unsigned **_hill (**long long *src***);**	Returns the high (odd) register of a long long register pair
long long **_itoll (**unsigned src2, unsigned src1**);**	Builds a new long long register pair by reinterpreting two unsigned values, where src2 is the high (odd) register and src1 is the low (even) register
float **_lof (**double src**);**	Returns the low (even) register of a double register pair
float **_hif (**double *src***);**	Returns the high (odd) register of a double register pair
double **_fod (**float src2, float src1**);**	Builds a new double register pair by reinterpreting two float values, where src2 is the high (odd) register and src1 is the low (even) register

Example: Using Intrinsics with DOTP2

```
for (i = 0; i < len; i += 4)                              SUB
                                                       || B
{
   a3_a2_a1_a0 = _amem8_const(&a[i]);                  || LDDW

   b3_b2_b1_b0 = _amem8_const(&b[i]);                  || LDDW

   sum_high += _dotp2(_hill(a3_a2_a1_a0,               || DOTP2
                      _hill(b3_b2_b1_b0);              || ADD

   sum_low  += _dotp2(_loll(a3_a2_a1_a0,               || DOTP2
                      _loll(b3_b2_b1_b0);              || ADD
}
```

Optimization Result

Optimization Method	Cycles for Loop codes	Cycles for total Function
No optimization	1360	1397
-o3 option	35	93
No debug	35	83
Memory Alignment	35	75
MUST_ITERATE	28	41
Single Instruction Multiple Data	19	37

- **Other optimization method, such as UNROLL doesn’t help for this loop code. But for other more complex code, you can try it to use them to balance resources usage.**

Agenda

- Optimization preparation
- Basic Optimization methods
- Software Pipeline
- Memory/Cache Optimization
- Other optimization tips and tricks

Software Pipeline

```
    LDH
||  LDH
    MPY
    ADD
```

How many cycles would it take to perform this loop 5 times?

(Disregard delay-slots).

$5 \times 3 = 15$

________________ **cycles**

Let's examine hardware (functional units) usage ...

Non-Pipelined Code

Cycle								
1	ldh	ldh	.M1	.M2	.L1	.L2	.S1	.S2
2			mpy					
3					add			
4	ldh	ldh						
5			mpy					
6					add			
7	ldh	ldh						
8			mpy					
9					add			

Pipelining Code

Cycle								
1	ldh	ldh	.M1	.M2	.L1	.L2	.S1	.S2
2	ldh	ldh	mpy					
3	ldh	ldh	mpy		add			
4	ldh	ldh	mpy		add			
5	ldh	ldh	mpy		add			
6	No LDH's?		mpy		add			
7					add			

Pipelining these instructions took 1/2 the cycles!

Pipelining Code

Prolog

Staging for loop

Loop Kernel

Single-cycle "loop" iterated three times

Epilog

Completing final operations

1	ldh	ldh	.M1	.L1
2	ldh	ldh	mpy	
3	ldh	ldh	mpy	add
4	ldh	ldh	mpy	add
5	ldh	ldh	mpy	add
6			mpy	add
7				add

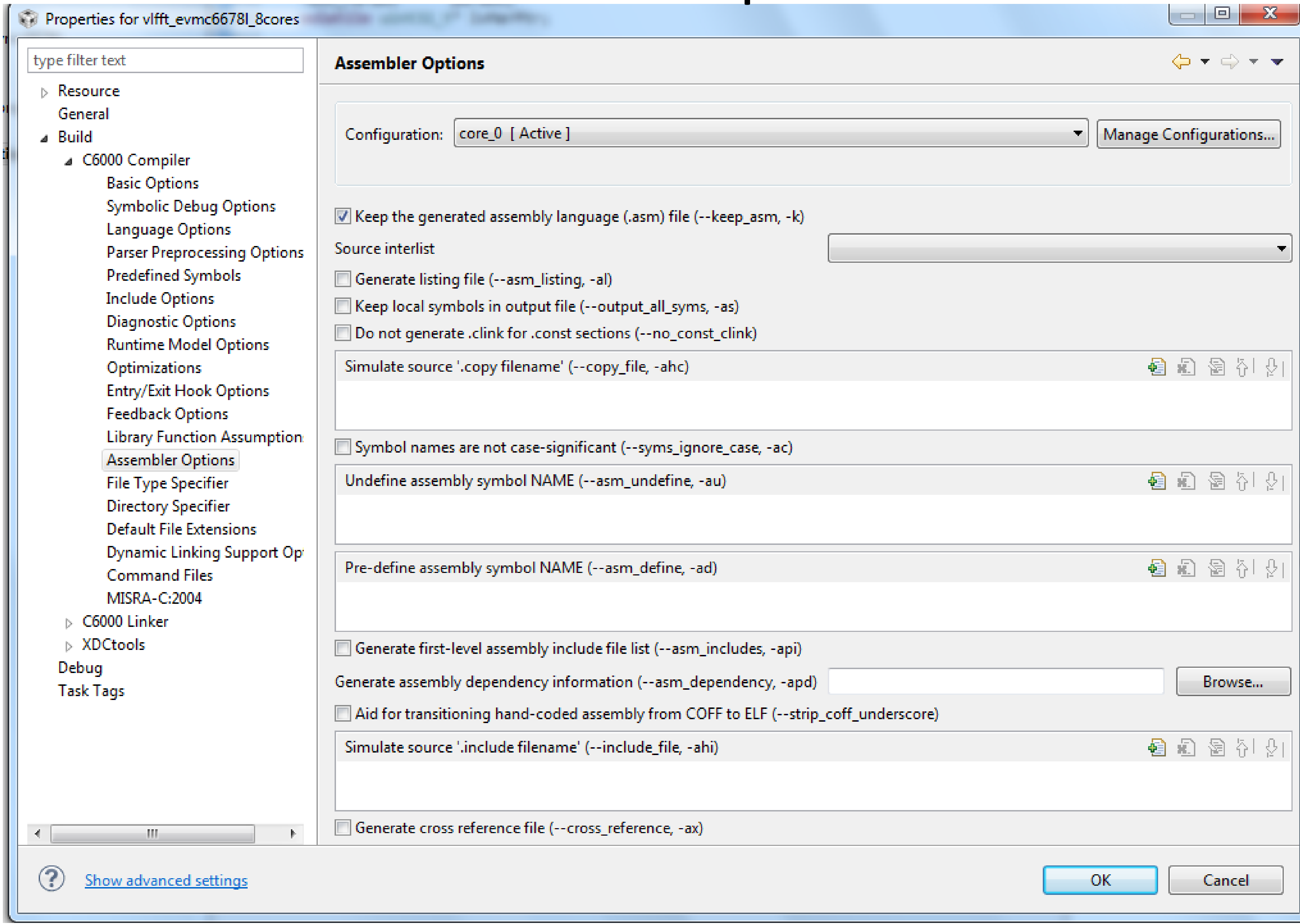
Compiler Feedback

- Compiler Gives important feedbacks for optimization in the assembly file generated with –k option
- Always compile with –s and –mw, as they extra information to the resulting assembly file
- It is very helpful to determine your optimization direction.
- Take following simple code as example:

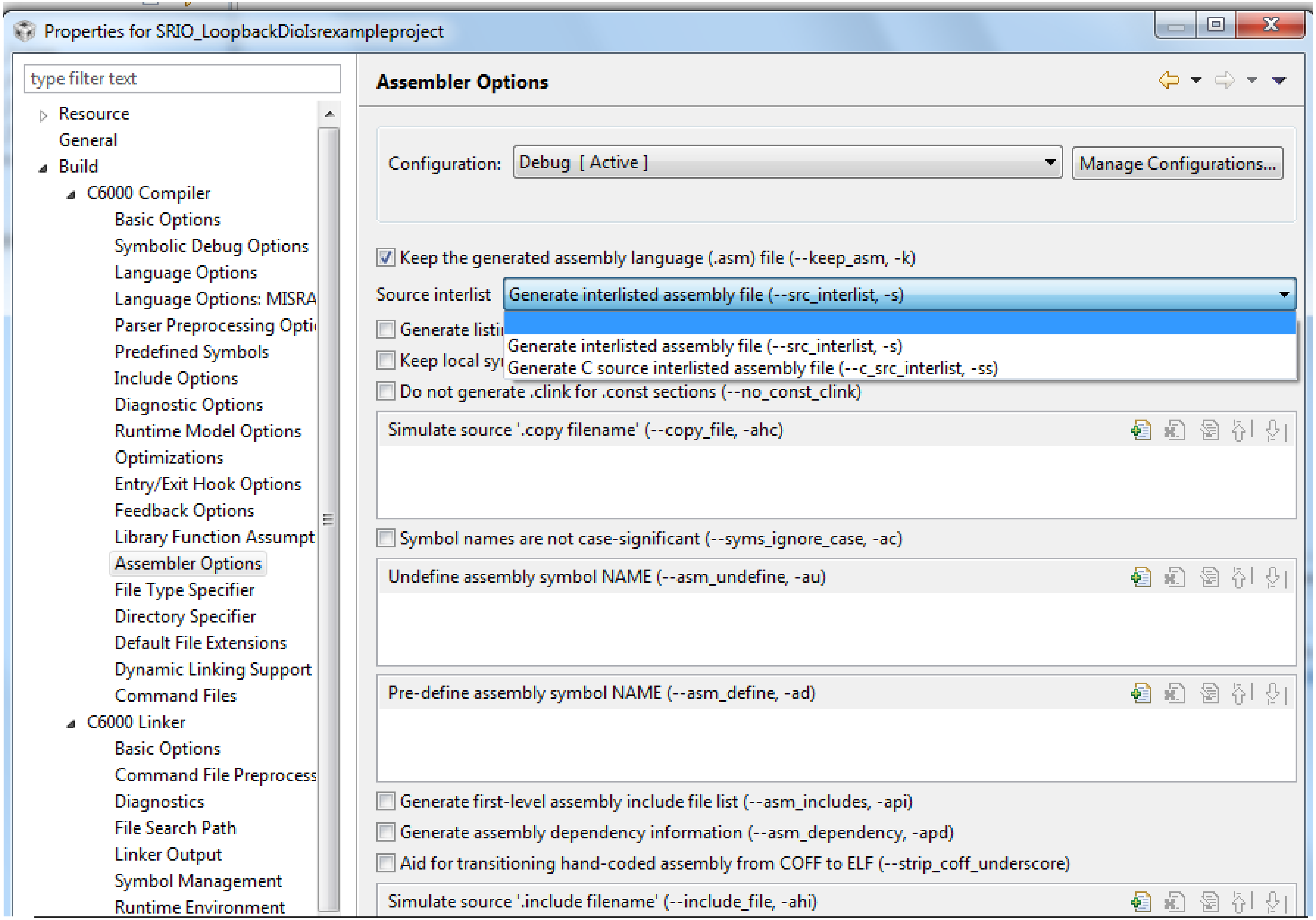
```
int dotp_c(short *a, short *b, int count)
{
    int i;
    int sum = 0;

    for (i=0; i < count; i++)
    {
        sum = sum + a[i] * b[i];
    }
    return(sum);
}
```

Assembler Options



-S and -MW Setting



Software Pipeline Feedback

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 6
;*      Loop opening brace source line   : 7
;*      Loop closing brace source line   : 9
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 1
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        0
;*      .D units                     1*       1*
;*      .M units                     1*       0
;*      .X cross paths               1*       0
;*      .T address paths             1*       1*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          1        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        0
;*      Bound(.L .S .D .LS .LSD)     1*       1*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 1  Schedule found with 8 iterations in parallel
;*      Done
;*
;*      Loop will be splooped
;*      Collapsed epilog stages       : 0
;*      Collapsed prolog stages       : 0
;*      Minimum required memory pad   : 0 bytes
```

Optimize the example with basic methods

```
int dotp_c(short * restrict a, short * restrict b, int count)
{
  int i;
  int sum = 0;

  _nassert((int) a % 8 == 0);
  _nassert((int) b % 8 == 0);
  #pragma MUST_ITERATE(8, 400, 8);
  #pragma UNROLL(4)
  for (i=0; i < count; i++)
  {
        sum = sum + a[i] * b[i];
  }
  return(sum);
}
```

Compiler feedback for the optimized code

```
;*      Loop source line                 : 10
;*      Loop opening brace source line   : 11
;*      Loop closing brace source line   : 13
;*      Loop Unroll Multiple             : 4x
;*      Known Minimum Trip Count         : 2
;*      Known Maximum Trip Count         : 100
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 1
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        0
;*      .D units                     1*       1*
;*      .M units                     1*       1*
;*      .X cross paths               1*       1*
;*      .T address paths             1*       1*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          1        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        0
;*      Bound(.L .S .D .LS .LSD)     1*       1*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 1  Schedule found with 10 iterations in parallel
;*      Done
;*
;*      Loop will be splooped
;*      Collapsed epilog stages       : 0
;*      Collapsed prolog stages       : 0
;*      Minimum required memory pad   : 0 bytes
;*
;*      Minimum safe trip count       : 1 (after unrolling)
```

The –mh Compiler Option

- –mh<num>. Speculative loads. Permits compiler to fetch (but not store) array elements beyond either end of an array by <num> bytes. Can lead to:
  - Better performance, especially for “while” loops
  - Smaller code size for both “while” loops and “for” loops
  - Not needed if SPLOOP buffer is used

- Software-pipelined loop information in the compiler-generated assembly file suggests the value of <num>

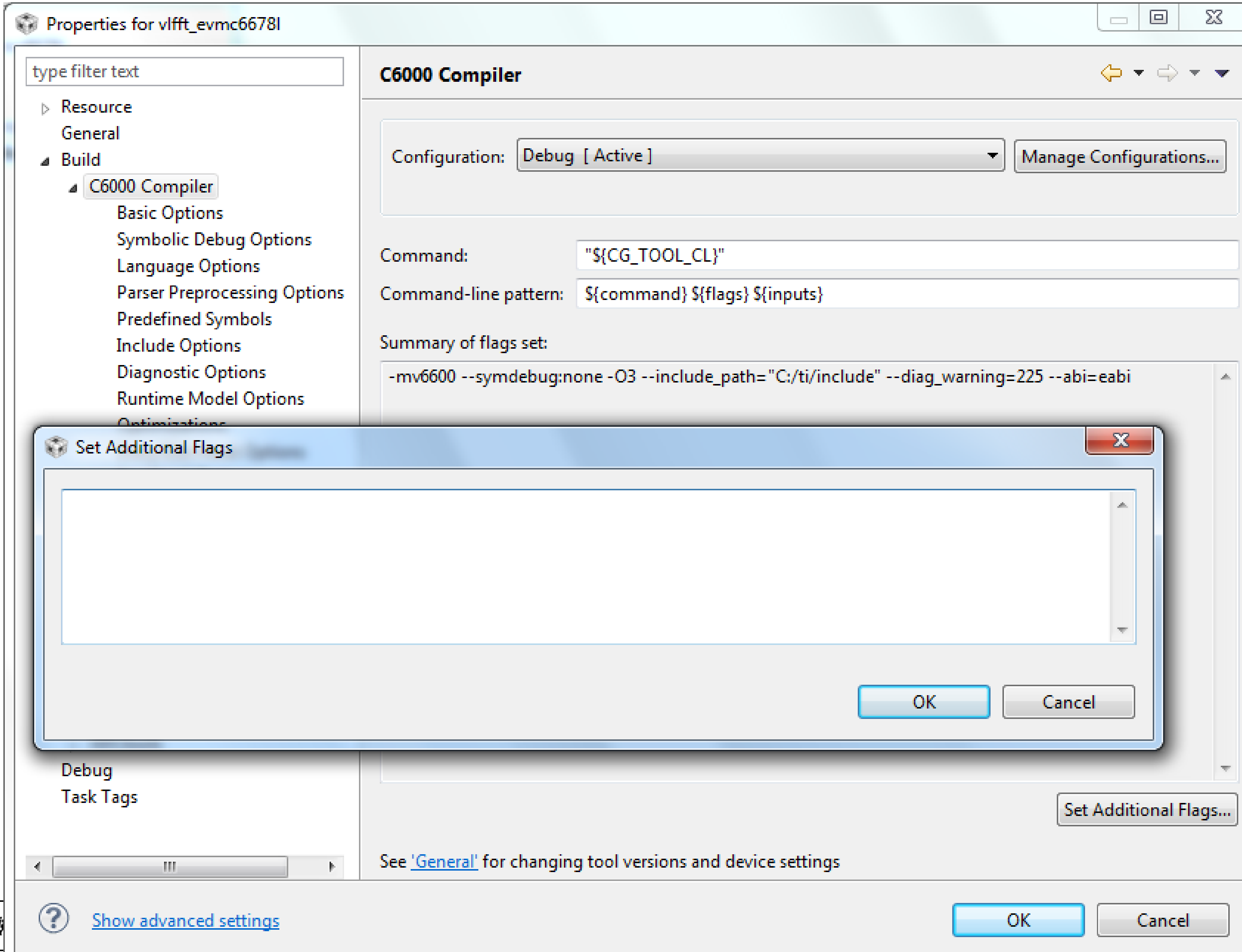
```
;* Minimum required memory pad : 0 bytes
;*
;* For further improvement on this loop, try option -mh56
```

- Indicates compiler is fetching 0 bytes beyond the end of an array.
  - If loop is rebuilt with –mh56 (or greater), there may be better performance and/or smaller code size.
  - NOTE: Need to pad buffer of <num> bytes on both ends of sections that contain array data

```
MEMORY {
   /* pad (reserved): origin = 1000, length = 56 */
   myregion: origin = 1056, length = 3888
   /* pad (reserved): origin = 3944, length = 56 */
}
```

- Alternatively, other memory areas (code or independent data) can be used as pad regions.

And if You Don’t Find the GUI?



Agenda

- Optimization preparation
- Basic Optimization methods
- Software Pipeline
- Memory/Cache Optimization
- Other optimization tips and tricks

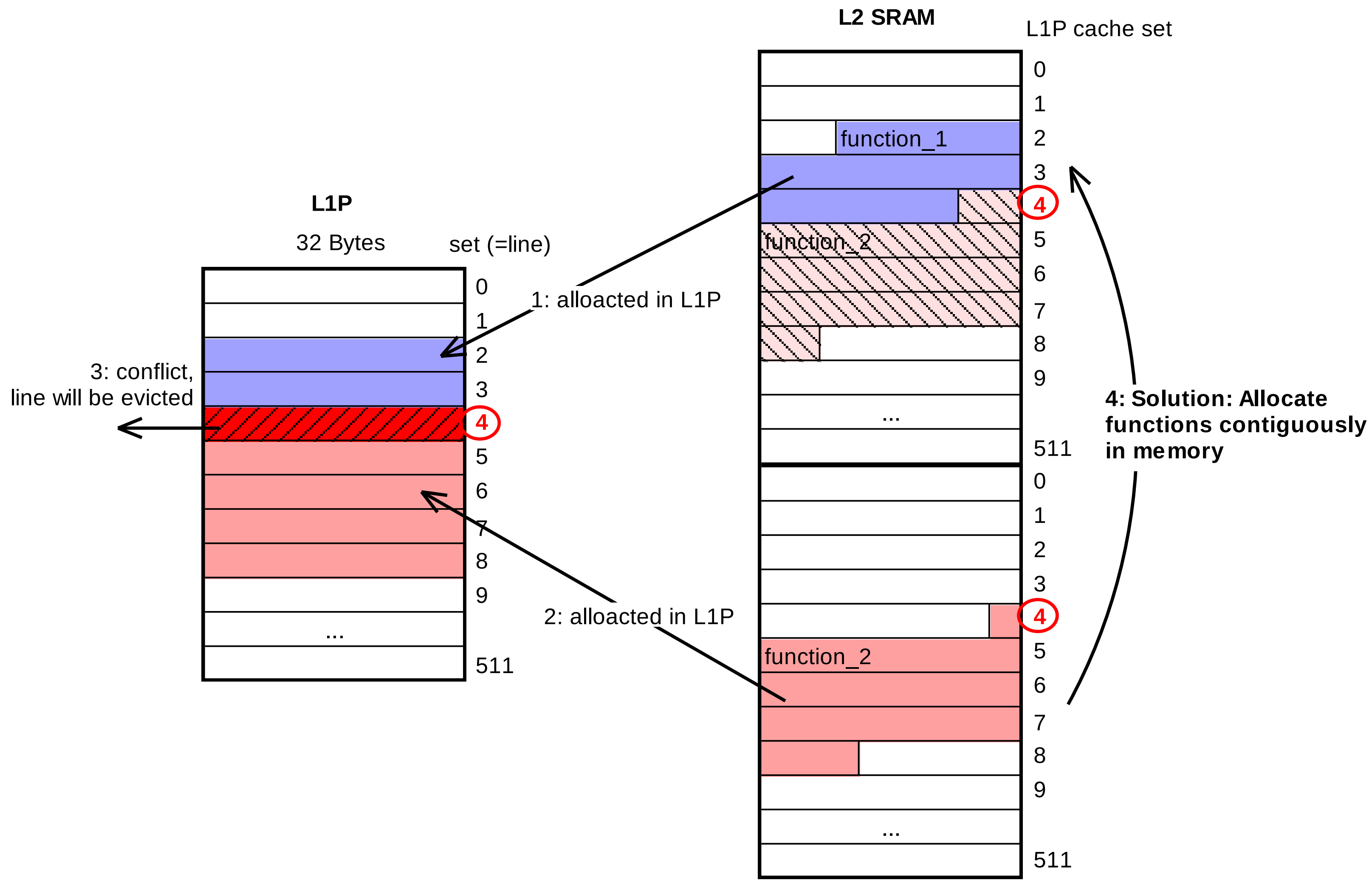
Memory Optimization

- If Possible, Put all code / data on-chip
  - Best performance, Easiest to implement
  - Shared L2 is the best for code and read-only data
- Use Multiple Sections
  - Keep critical code and data on-chip
  - Put non-critical code and data off-chip
    - #pragma CODE_SECTION(dotp, “critical”);
    - #pragma DATA_SECTION (x, “myVar”);

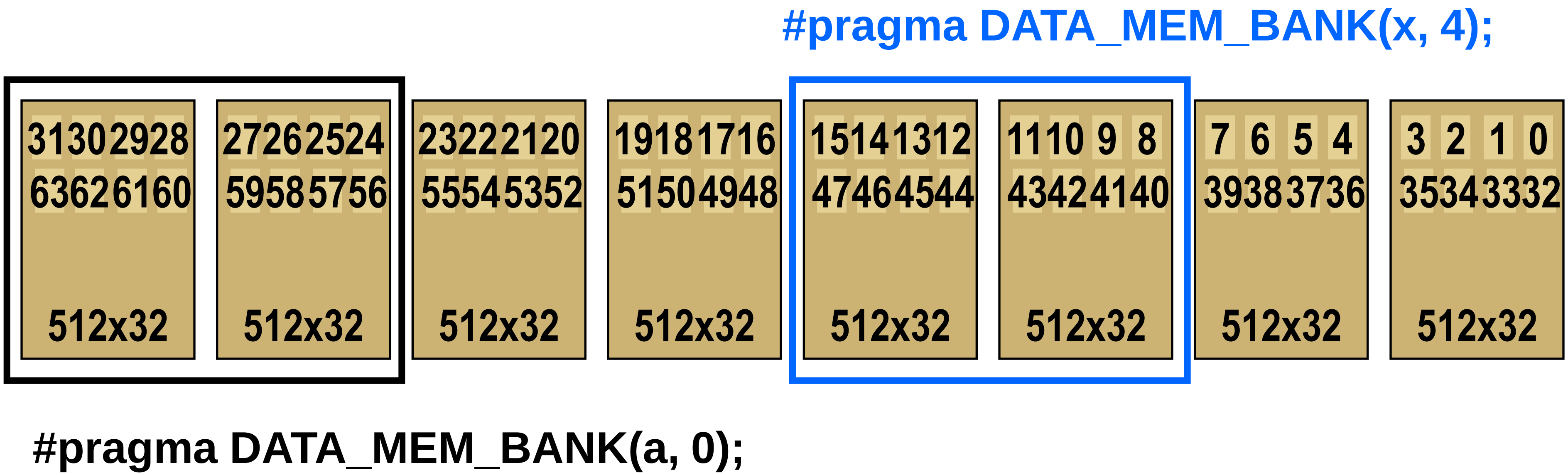
Cache Optimization

- ***maximize line reuse before eviction***
- **Reduce the number of cache misses**
  - **Reduce amount of memory accessed during algorithm**
  - **Improve spatial locality of memory accesses**
  - **Improve temporal locality of memory accesses**
- **Reduce the penalty (DSP core stall) cycles associated with misses**
  - **Making use of miss pipelining**
  - **use touch loop, that is, read one word from each line continuously**
- **Avoid L1D bank conflict**

contiguous allocation



L1D Memory Banks



- Only one access allowed per bank per cycle
- Use DATA_MEM_BANK to make sure that arrays that will be accessed in parallel start in different banks

DATA_MEM_BANK(*var*, *0 or 2 or 4 or 6*)

```
#pragma DATA_MEM_BANK(a, 0);
short a[256] = {1, 2, 3, …
#pragma DATA_MEM_BANK(x, 4);
short x[256] = {256, 255, 254, …
#pragma UNROLL(2);
#pragma MUST_ITERATE(10, 100, 2);
for(i = 0; i < count ; i++) {
   sum += a[i] * x[i];
}
```

- **An internal memory specialized Data Align**
- **Optimizes variable placement to account for the way internal memory is organized**

Avoid memory bank issue in code

Potential bank conflict with below code

```
   LDDW x[i]
|| LDDW a[i]
```

No bank conflict with below code

```
   LDDW a[i]
|| LDDW a[i+4]
   LDDW x[i]
|| LDDW x[i+4]
```

Original code must be unrolled x8 to generate above code

```
#pragma UNROLL(8)
for(i = 0; i < count ; i++) {
    sum += a[i] * x[i];
}
```

Agenda

- Optimization preparation
- Basic Optimization methods
- Software Pipeline
- Memory/Cache Optimization
- Other optimization tips and tricks

Generic Optimization Advice

- No "printf" in your key code!
- No "if", branch, and call in key loop!
- Use peripherals (and coprocessors) to offload unnecessary tasks from the CorePacs.
- Make sure the loop trip counters are (unsigned) int or long (32 bit) … and <u>not</u> short (16 bit).

Golden Rule of Software Pipeline

The larger the loop,

the less efficient the optimizer.

If your application code contains very long loops … break the loop into multiple loops … even if it means storing intermediate results in L1

Volatile

```
int *ctrl;
while (*ctrl == 0);
```

This code may be eliminated by optimizer

Add volatile qualifier to keep it

```
volatile int *ctrl;
while (*ctrl == 0);
```

volatile qualifier will disable optimization on corresponding data

Array and structure

- **keep array dimensions to no more than 2 x[][] is enough**
- **Try to keep structure depths as small as possible**
  - **x->y->z[i]->w is very inefficient to access**
- **structure of arrays** struct->a[i] **are better than arrays of structure** struct[i]->a

Pointers in Structures

- Create local pointers *at top-level of function* and restrict qualify pointers instead.
- Use local pointers in function/loop instead of original pointers.

```
typedef struct {int *p} _str;
myfunc(_str * restrict s)
{
    // create local pointers at top-level of function
    int * restrict sp = s->p;
    // use sp instead of s->p
    *sp = …
        = *sp
}
```

Writing Efficient Code with Structure References

General Tips:

- **Avoid dereferencing structure elements in loop control and loops.**
- **Instead create/use local copies of pointers and variables when possible.**

Original loop:

```
while (g->y < 25)
{
    g->p->a[i++] = …
}
```

Hand-optimized Loop:

```
int    y  = g->y;
short *a  = g->p->a;

while (y < 25)
{
    a[i++] = …
}
```

If Statements

- **Compiler will if-convert loops with small if statements:**

Original C code:

```
if (p)
     x = 5
else
     x = 7
```

After if conversion:

```
   [p] x = 5
|| [!p] x = 7
```

If Statements (cont.)

- **Compiler will** not **if convert large if statements.**
- **Compiler will** not **software pipeline loops with if statements that are not if-converted.**

```
;*----------------------------------------------------
;*   SOFTWARE PIPELINE INFORMATION
;*      Disqualified loop: Loop contains control code
;*----------------------------------------------------
```

- **For software pipelinability, user must transform large if statements because compiler does not know if this is profitable.**

Structural Improvements

- Some program and data structures are better than others
  - if-for is better than for-if
  - one software pipelines, one doesn't

```
for (i=0; i < N; i++)
{
   if (case0)
     do_exec0[i];
   else if (case1)
     do_exec1[i];
   else
     do_exec2[i];
}
```

```
if (case0)
   for (i=0; i < N; i++)
     do_exec0[i];
else if (case1)
   for (i=0; i < N; i++)
     do_exec1[i];
else
   for (i=0; i < N; i++)
     do_exec2[i];
```

- Complex control blocks or compromises pipelining

asm(“ ”)

◆asm() function can be used to execute an assembly statement in C language.

◆Normally, it is not recommended to be used. It will break many optimization methods of compiler.

◆Below are several special cases:

```
for(i=0; i<0x100000; i++)  /*delay*/
{/* without asm(), this loop may be eliminated  by optimizer*/
        asm("    nop 5");
}
asm("    IDLE");   /*make the DSP core idle to save power*/
asm("    dint");  /*disable interrupt*/
asm("    rint");  /*restore interrupt*/
```

◆Pay attention to the space between the “ and instructions, it is compulsory

Optimization Literature

- SPRU187, “TMS320C6000 Optimizing Compiler User’s Guide”, provides a complete description of the C/C++ compiler option and how to tune compiler to generate optimized code.
- SPRU198, “TMS320C6000 Programmer’s Guide”, provides step by step procedures to tune code performance using compiler options and manually optimization technology on C6000 DSPs.
- SPRA666, “Hand-Tuning Loops and Control Code on the TMS320C6000”, provides useful tips and to tune loop and control code manually.