Advanced Multi-Threaded Programming

by Aaron Coday

April 2005

The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation Intel on the date of publication. Intel makes no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN IS PROVIDED AS IS. INTEL MAKES NO REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL OR THIRD PARTIES. INTEL EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL DOES NOT WARRANT THAT THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL DISCLAIMS ALL LIABILITY THEREFOR.

INTEL DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIMS ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel, the Intel logo, Pentium, Intel Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2005 Intel Corporation

Introduction

In this course and its accompanying labs, you will become familiar with intermediate to advanced techniques for explicit threading and OpenMP* threading. You'll demonstrate your understanding of explicit threading by adding GUI responsiveness to the Apfel* application. You'll demonstrate your understanding of OpenMP threading by improving the performance of the fractal calculation that the program is executing.

To complete the Advanced Multi-Threading labs included in this course, you'll need the following tools:

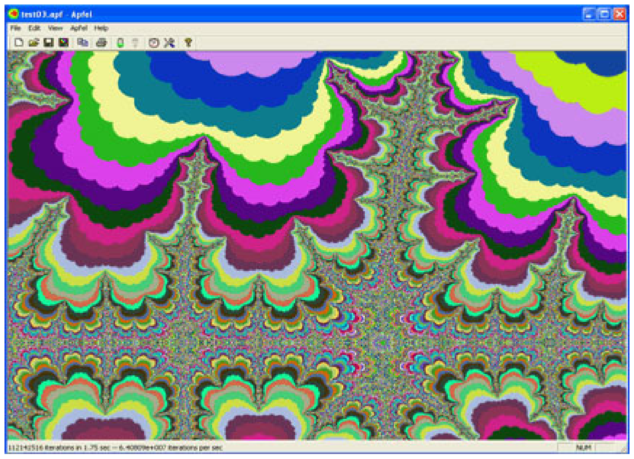
- Microsoft Visual Studio*
- Intel® Compiler 7.1 or later
- Intel® Threading Toolkit (recommended, but not required)

You can download the Apfel source code and other files you will need for the labs by clicking this link. ftp://download.intel.com/software/products/college/advMT/Lab/ Please download and extract these files before continuing with the course.

This course is divided into two parts, each structured around one of the multi-threading labs. Each section starts with a description of the multi-threading problem to be addressed, followed by detailed instructions as to a possible solution. Each of the two sections then concludes with a lab activity, in which you will implement the proposed multi-threading solution.

Overview

Apfel* is a fractal benchmark program written by Andreas Stiller from *C'T Computer* magazine in Germany. It is used to test computation performance of various systems by generating fractals, a task requiring repetitive and recursive floating point computations.



112142516 iterations in 1.75 sec -- 6.40009e+007 iterations per sec

We have two goals for enhancing the program. The first is to use multi-threading to add greater GUI responsiveness by separating the GUI from the fractal computations. Any computer system can benefit from this first part. The second goal is to improve the speed of the fractal computation itself by utilizing multiple threads. Note that this technique will only improve performance in Hyper-Threading Technology, dual core, or dual processor systems. However, thanks to our implementation, single processor systems will not experience any loss of performance.

After you download and extract the Apfel source code and other files you will need for the labs, run the Apfel program in hard, medium, and easy profiles. You'll notice when you click on the green button to calculate the fractal that the GUI becomes unresponsive; you can't move the window, or even click on the cancel button (red button). In the first part, you'll change the functionality to allow GUI responsiveness while calculating the fractal.

Also, note the number of iterations and the time elapsed, both of which are displayed in the lower-left corner of the application. The number of iterations can be used for verifying correctness, and the time is a measure of performance. The shorter the time, the better the performance. This will be the focus of the second part of the labs.

Guidelines

Apfel* is a big program. In order to complete the labs in this course, you will only need to modify the `CApfelRun` and `CApfelView` classes.

The Apfel software architecture is based on Microsoft* Foundation Classes (MFC). `CApfelRun` is the class which performs the fractal computation. `CApfelView` represents the GUI. When the GUI `Update()` call is made, it eventually calls the `CApfelRun::Run` method, which in turn calls the `CApfelRun::DoRun` method. The `Update()` method is called whenever it is necessary to update the GUI. The `DoRun` method does the actual fractal computation.

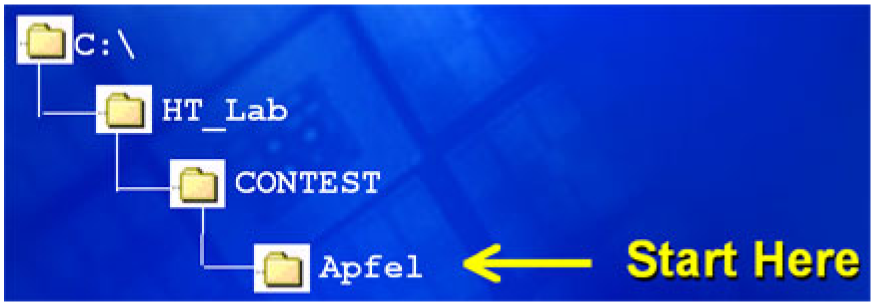
- CApfelRun – class that does the work
- CApfelView::Update ->
    - CApfelView::RenderToDIB ->
        - CApfelRun::Run ->
        - CApfelRun::DoRun – performs the calculation

Hint: It would be good software architecture to introduce multi-threading in the CApfelRun class and not in the CApfelView class. This way, the GUI doesn't need to know too much about threading. It will also facilitate future program enhancements.

The **DoRun** method basically calculates one row of the screen at a time. Each iteration is essentially independent from every other one. We'll be working with this method in the second lab when we learn how to speed up the fractal calculation. Since iterations are independent, we'll split them up among multiple threads. More on this in Lab Activity 2.

- DoRun (ScreenBuffer, Stats)
    - For each row
        - Buf <- Calculate row (CalcRow)
        - Look up in color reference and output to ScreenBuffer

Files and Structure



- Visual Studio*: Apfel.dsw
- Focus
    - CApfelRun = computation
    - CApfelView = GUI
- Sample data sets: hard, medium, simple

This describes the tree structure of the files on the computer. It may vary depending on where the class files were installed.

Part 1: Responsiveness

- Goals:
    - Add multi-threading to enhance GUI responsiveness
    - Use Win32* threading
- Tasks:
    - Add thread function to `CApfelRun` to execute `DoRun` asynchronously
    - After calculation is complete, send a `WM_DONE` message back to the GUI so that the screen updates

There are two main tasks that need to be completed to separate the GUI from the calculation – that is, to do the calculation asynchronously.

Task 1: Add thread function to `CApfelRun` class to execute `DoRun` method asynchronously. To call `DoRun` asynchronously, you'll need to launch it on a second thread.

- Win32 requires that we have a thread function in order to create a new thread, so we need to wrap the `DoRun` method in a thread function. Win32 requires this to be of a special type, such as:
    - `static DWORD WINAPI ApfelThreadFunc(LPVOID param)`
- Notice that this type is static, so in order to call the `DoRun` method, you'll need to use a trick similar to that described in the section called "Hint #1" in the appendix to this course.
- You'll also have to figure out how to pass the parameters of `DoRun`, because the thread function only takes a single `LPVOID`. (You can create a `struct` yourself, or store it in the `CApfelRun` class and pass the `this` pointer.)
- So the `CApfelRun::Run` method will create the thread and pass the parameters and return immediately, allowing the GUI to continue. However, how do we know when the calculation is done?

Task 2: After the thread function finishes (i.e., the calculation is done), we need to send a message back to the GUI to alert the GUI that the calculation is done.

- We'll need to change the `CApfelView::RenderToDIB` to comment out the call to the `OnDone` method. Do you know why? (Hint: possible race condition.)

- You'll use `PostMessage` to send a `WM_DONE` message back to the GUI, so that it can finally draw the bitmap. Pay attention to what you need in order to call `PostMessage`, since you may need to pass it to the thread function, as well.

Click here [link to Appendix: Win32 Threads] for a review of common Win32 threading functions you'll need for Lab Activity 1: Add GUI Responsiveness. You can check out more details on MSDN.microsoft.com.

Extra Credit

As an "Extra Credit" exercise, make the "Abort" or "Cancel" feature (red button) function correctly. Hint: Consider synchronization and race conditions.

Lab Activity 1: Add GUI Responsiveness

Asynchronous Thread Function in CApfelRun

Setup

1. Launch Microsoft Visual Studio.
2. Open workspace from C:\Lab\CONTEST\apfel\apfel.dsw.
3. Make sure that Intel Compiler is selected.
4. Build the application by selecting **Release build** and then **Build** (F7).
5. Press **Ctrl-F5** to run the application.

Adding thread function to CApfelRun

Basically you add a thread function to CApfelRun and then take care of starting and passing the necessary information into the thread. The new thread is responsible for performing the DoRun method.

1. Add a thread function to the CApfelRun class (that is, in the header file); leave the body empty for now (that is, in the cpp file).

The function prototype to add to CApfelRun (this signature is required by CreateThread):

```
static DWORD WINAPI ApfelThreadFunc(LPVOID
param);
```

The function body for now:

```
DWORD WINAPI CApfelRun::ApfelThreadFunc(LPVOID param)
{
/* nothing for now */
};
```

2. In this step you figure out how to communicate the DoRun parameters from the calling thread to the thread function and then on to DoRun. CApfelRun::Run method will be calling CreateThread with CApfelRun::ApfelThreadFunc and then CApfelRun::ApfelThreadFunc will call CApfelRun::DoRun. For this to work, you need a way to communicate the CWnd, the DIB, and the CApfelStatistics objects for the parameters to DoRun. You also need the `this` pointer, so that you can call DoRun inside of ApfelThreadFunc (it is a static method of CApfelRun).

   One way to do this is to store these objects in the CApfelRun class itself. So, add the following to the CApfelRun class (in the header file):

```
CWnd* m_pWnd;
 HDIB m_hDIB;
 CApfelStatistics* m_pStat;
```

3. Modify the CApfelRun::Run method to call CreateThread and also to store the parameter objects for ApfelThreadProc to access later.

   Modify the Run method to look like this:

```
OOL CApfelRun::Run(CWnd* pWnd, HDIB hDIB,
CApfelStatistics* pStat /*= NULL*/)
{
     LPSTR lpDIB = (LPSTR)::GlobalLock(hDIB);
     if (!lpDIB)
     {
          return NULL;
     }
     DWORD ymax = ::DIBHeight(lpDIB);
     ::GlobalUnlock(hDIB);

     m_pWnd = pWnd;
     m_hDIB = hDIB;
```

```
	m_pStat = pStat;

	CreateThread(NULL, 0, ApfelThreadProc,
			(LPVOID)this, 0, NULL);
	// Notice we pass this pointer, so that we can call
the DoRun
	// method on the CApfelRun object

	BOOL bRes = TRUE;

	// No longer call DoRun synchronously.
	//BOOL bRes = DoRun(hDIB, pStat);
	return bRes;
}
```

4. Fill in the body of the CApfelRun::ApfelThreadProc function so that it uses the this pointer to call DoRun and to pass it the parameters.

Modify ApfelThreadProc to look like the following:

```
/*static*/ DWORD WINAPI CApfelRun::ApfelThreadProc(LPVOID
pParam)
{
	// Get the CApfelRun pointer
	CApfelRun* pThis = (CApfelRun*)pParam;
	ASSERT(pThis);
	ASSERT_VALID(pThis);

	// Call the method on CApfelRun::DoRun
	BOOL bRes = pThis->DoRun(pThis->m_hDIB, pThis-
>m_pStat);

	return (DWORD)TRUE;
}
```

Posting WM_Done message

Post a WM_DONE message after the calculation is finished. You also need to prevent the CApfelView::OnDone method being called in the CApfelView::RenderToDIB method.

5. In the CApfelView::RenderToDIB method, comment out the call to OnDone. This is to prevent a race condition because the CApfelRun::Run method now immediately returns even before the calculation has finished computing.

6. After the DoRun call has completed, you must post the message manually from the CApfelRun::ApfelThreadProc function.

   After the DoRun call, add the following:

   ```
   // Now we post a done message so that we render
   pThis->m_pWnd->PostMessage(WM_DONE, 100,
   (LPARAM)bRes);
   ```

7. Build and then run the lab. When you click to start the computation of the fractal, you should be able to move the window around and the GUI should be responsive.

Part 2: Performance

In the second lab, our goal is to speed up the fractal calculation by using multi-threading to improve the performance of the `DoRun` method.

Our task is to use OpenMP to parallelize the `DoRun` method. As you do this, be aware of the following issues:

- You need to convert the main loop to a for-loop. OpenMP works only with for-loops.
- Make sure you synchronize where necessary.
- Each thread may need separate resources (memory for each row...).

You can confirm that you have the correct result by comparing the number of iterations displayed in the GUI with the number of iterations before multi-threading. They should be the same.

Open MP items you may need for the Adding Performance lab include:

- `#pragma omp parallel`
- `#pragma omp for`
- `#pragma omp critical`
- `omp_get_thread_num()`
- `private clause`

If you are not familiar with these, you can check out the OpenMP specification in the docs directory, or get the full spec from www.openmp.org.

Lab Activity 2: Adding Performance with OpenMP

Parallelize the CApfelRun::DoRun Method

The following describes in high-level detail the steps to parallelize the DoRun Method. It is followed by **Listing 1**, which shows an example solution. Again there are actually several ways to solve the problem.

Setup

Make sure to enable OpenMP compiler support. By default, this is not enabled in the Intel Compiler. It can be enabled through the settings menu or by adding /Qopenmp to the compiler command line.

Convert do-while loop to for loop

- OpenMP parallel constructs do not work on do-while loops. Therefore, convert the `do-while loop` into a `for loop`.

Parallelize for loop

- Use the "`#pragma omp parallel for`" construct to parallelize the `for` loop.

Thread private data

Make sure to use the private clause as necessary for thread private variables.

1. Create thread private versions of the buf data structure. Each thread should have a separate copy of buf. This can be done in one of two ways:

   a. Create an array of bufs and access the correct one with omp_get_thread_id()

   b. Create the buf data structure inside the parallel region.

   In the second case, you must protect the AlignedAlloc function call because it is not thread-safe.

2. Make sure you protect the addition to dwIterations variable. You do not want to put an "omp critical" around the whole line because the computation is down in the function call. Solve this by returning the CalcRow into a temporary and then with "omp critical" add that value to dwIterations.

Speed up the loop

To speed up the loop, you can use:

1. The schedule clause on the "pragma omp parallel" construct to choose a better scheduling for the threads.

2. The Intel® Thread Profiler to track the performance of the schedule.

3. The reduction clause on the "pragma omp parallel" construct to replace the "omp critical" on dwIterations.

Verify

- Verify that the application runs faster AND that the number of iterations displayed in the GUI is the same as the non-openmp version.

  Hint: You can also use omp_set_num_threads(1) to see what the non-openmp version would run like.

Listing 1

```
#include <omp.h>
#define MAX_THREADS 32

BOOL CApfelRun::DoRun(HDIB hDIB, CApfelStatistics* pStat)
{
      COLORREF* pcr = new COLORREF[m_t];
      if (!pcr)
            return FALSE;

      // access DIB
      LPSTR lpDIB = (LPSTR)::GlobalLock(hDIB);
      if (!lpDIB)
            return FALSE;

      // currently we work only with hicolor DIBs
      if (!IsHiColorDIB(lpDIB))
      {
            // unaccess DIB
            ::GlobalUnlock(hDIB);
            return FALSE;
      }

      // resolution
      CSize szRes(::DIBWidth(lpDIB), ::DIBHeight(lpDIB));

      ASSERT(szRes.cx % 16 == 0);

      srand(::timeGetTime());
      for (int k = 0; k < m_t; k++)
      {
            pcr[k] = RGB(rand(), rand(), rand());
      }

      if (pStat)
            pStat->Start();

      const float i0s = m_i0s; // 0.0f
      const float r0s = m_r0s; // 0.0f
      const float del = m_del; // 0.01f
      const int t = m_t; // 1024
      const int g = m_g; // 3

      const int xmax = szRes.cx;
      const int ymax = szRes.cy;

      // One buffer for each thread
      unsigned short* buf[MAX_THREADS];

      // Uncomment this if you want to manually set the number
of threads
```

```
	// omp_set_num_threads(2);

	// Need local version, because can't reduce an element of
type  somestruct->variabletoreduce
	DWORD dwIterations = 0;
	int nid;

#pragma omp parallel
	{
		float i, r;
		int x;
		int y;
		int color;
		nid = omp_get_num_threads();

// Could also do this with an omp critical and thread private
buf.  But this is instructive.
#pragma omp single
		for (int tid = 0; tid < nid; tid++)
		{
			buf[tid] = (unsigned
short*)::AlignedAlloc(szRes.cx * sizeof(unsigned short), 32);
			if (!buf[tid])
			{
				// unaccess DIB
				//::GlobalUnlock(hDIB);
				//return FALSE;
			}
		}

		tid = omp_get_thread_num();
```

```
          // Don't forget to use reduction or critical on
dwIterations
#pragma omp for schedule(dynamic, 8) reduction(+: dwIterations)
          for (y = 0; y < ymax; y++)
          {

               i = (i0s - ymax / 2 * del) + del * y;
               r = (r0s - xmax / 2 * del);// + (del * xmax) *
y;

               if (m_bAbort)
               {
                    // OMP doesn't allow breaks, so you have
to cheat to allow breaking
                    y = ymax;
               }

               dwIterations += CalcRow(buf[tid], xmax, del, i,
r, g, t);

               for (x = 0; x < xmax; x++)
               {
                    color = buf[tid][x];
                    ASSERT((color >= 1) && (color <= t));
                    COLORREF cr = pcr[color - 1];
                    ::SetDIBPixel(lpDIB, CPoint(x, y), cr);

               }



          } // for
```

```
        }
        if (pStat)
                pStat->Stop();



        pStat->m_dwIterations = dwIterations;

        // Cleanup
        for (int ktid = 0; ktid < nid; ktid++)
        {
                ::AlignedFree(buf[ktid]);
        }

        // unaccess DIB
        ::GlobalUnlock(hDIB);

        delete[] pcr;



        return TRUE;
}
```

Summary

You can use multi-threading to add extra functionality, to increase performance, or both. You should know and be able to use both explicit threading (Win32*) and OpenMP*.

Supplemental Material

Intel® Threading Toolkit

- Intel® Thread Checker,
- Thread Profiler,
- Intel® VTune™ Performance Analyzer

Intel® Thread Checker

- Locate threading bugs in applications on IA-32 systems running Windows*.
- Use remote collectors to locate threading bugs in applications on IA-32 and Itanium®-based systems running Linux*.

Running Intel® Thread Checker

1. Statistics collected within VTune™ analyzer
    - Compile with icl /Qopenmp_profile (/MD /Qopenmp)
2. Statistics collected outside VTune analyzer
    - Compile with icl /Qopenmp_profile
    - Run program outside VTune environment
    - Import *guide.gvs* statistics file into VTune analyzer

To import guide.gvs files, simply do File/Open File for OpenMP Statistics (*.gvs) files.

Thread Profiler

- For Windows*, locate performance bottlenecks in Win32* and OpenMP* threaded applications
- For Linux*, now you can locate performance bottlenecks in POSIX* and OpenMP threaded applications, from a host Windows system
- View graphic displays that show each thread's state and parallel-serial transitions to confirm that performance is meeting expectations or where it is falling short — helps you decide where to focus optimization efforts

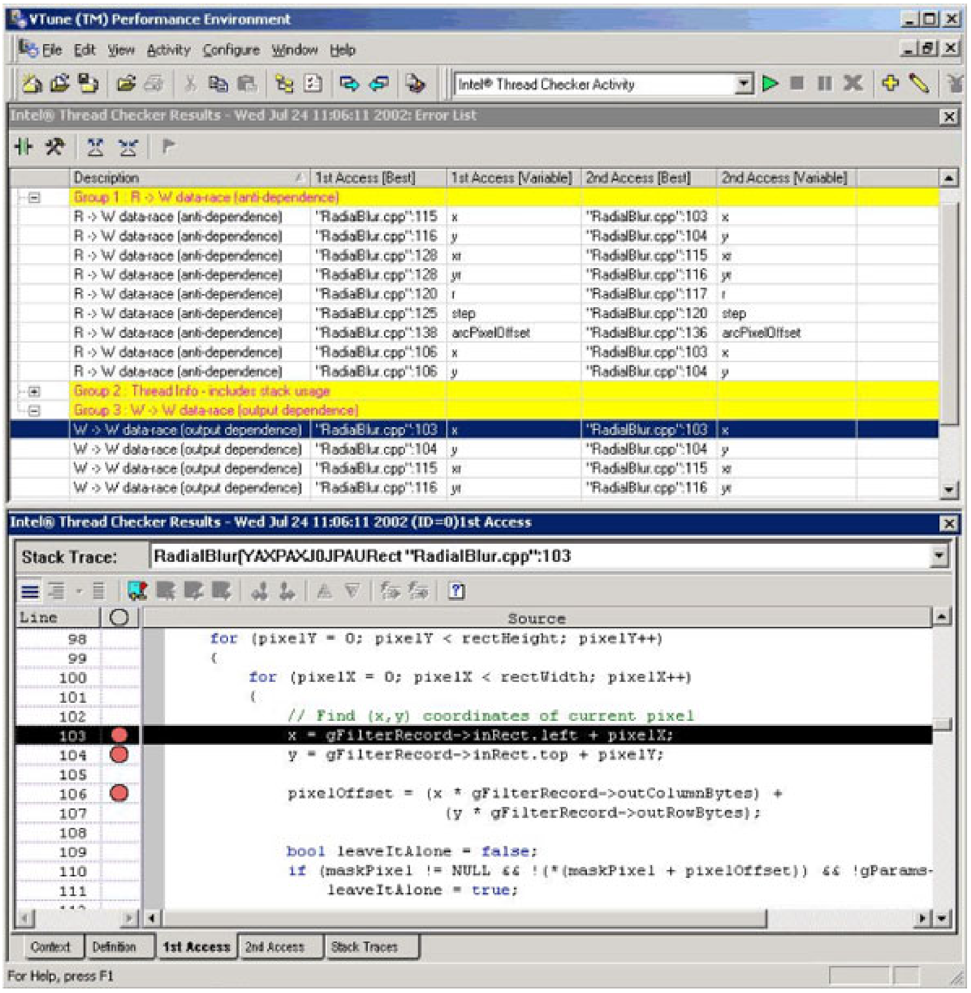
Intel ®VTune™ Performance Analyzer

Error List

- Customizable
- Links to source view

Source View

- Error context
- Error locations
- Stack trace


VTune (TM) Performance Environment
File Edit View Activity Configure Window Help
Intel® Thread Checker Activity

Intel® Thread Checker Results - Wed Jul 24 11:06:11 2002: Error List

Description	1st Access [Best]	1st Access [Variable]	2nd Access [Best]	2nd Access [Variable]
Group 1 : R -> W data-race (anti-dependence)				
R -> W data-race (anti-dependence)	"RadiaBlur.cpp":115	x	"RadialBlur.cpp":103	x
R -> W data-race (anti-dependence)	"RadiaBlur.cpp":116	y	"RadialBlur.cpp":104	y
R -> W data-race (anti-dependence)	"RadiaBlur.cpp":128	xr	"RadialBlur.cpp":115	xr
R -> W data-race (anti-dependence)	"RadiaBlur.cpp":128	yr	"RadialBlur.cpp":116	yr
R -> W data-race (anti-dependence)	"RadiaBlur.cpp":120	r	"RadialBlur.cpp":117	r
R -> W data-race (anti-dependence)	"RadiaBlur.cpp":125	step	"RadialBlur.cpp":120	step
R -> W data-race (anti-dependence)	"RadiaBlur.cpp":138	arcPixelOffset	"RadialBlur.cpp":136	arcPixelOffset
R -> W data-race (anti-dependence)	"RadiaBlur.cpp":106	x	"RadialBlur.cpp":103	x
R -> W data-race (anti-dependence)	"RadiaBlur.cpp":106	y	"RadialBlur.cpp":104	y
Group 2 : Thread Info - includes stack usage				
Group 3 : W -> W data-race (output dependence)				
W -> W data-race (output dependence)	"RadiaBlur.cpp":103	x	"RadialBlur.cpp":103	x
W -> W data-race (output dependence)	"RadiaBlur.cpp":104	y	"RadialBlur.cpp":104	y
W -> W data-race (output dependence)	"RadiaBlur.cpp":115	xr	"RadialBlur.cpp":115	xr
W -> W data-race (output dependence)	"RadiaBlur.cpp":116	yr	"RadialBlur.cpp":116	yr

Intel® Thread Checker Results - Wed Jul 24 11:06:11 2002 (ID=0)1st Access

Stack Trace: RadialBlur(YAXPAXJ0JPAURect "RadialBlur.cpp":103

```
Line     Source
 98      for (pixelY = 0; pixelY < rectHeight; pixelY++)
 99      {
100          for (pixelX = 0; pixelX < rectWidth; pixelX++)
101          {
102              // Find (x,y) coordinates of current pixel
103 ●            x = gFilterRecord->inRect.left + pixelX;
104 ●            y = gFilterRecord->inRect.top + pixelY;
105
106 ●            pixelOffset = (x * gFilterRecord->outColumnBytes) +
107                                (y * gFilterRecord->outRowBytes);
108
109              bool leaveItAlone = false;
110              if (maskPixel != NULL && !(*(maskPixel + pixelOffset)) && !gParams-
111                  leaveItAlone = true;
```

Appendix – Win32 Threads

The following is a review of common Win32 threading functions.

Creating Win32* Threads

```
// Thread handle

HANDLE CreateThread(

      LPSECURITY_ATTRIBUTES ThreadAttributes,

      DWORD StackSize,

      // Functions are explicitly mapped to threads

      LPTHREAD_START_ROUTINE StartAddress,

      // One 32-bit value parameter passed

      LPVOID Parameter,

      DWORD CreationFlags,

      // NULL or CREATE_SUSPENDED

      LPDWORD ThreadId );
```

Waiting for Kernel Objects

This is the hub function for synchronization.

```
DWORD WaitForSingleObject (

      HANDLE hHandle,

      DWORD dwMilliseconds);

      // Timeout (0 .. INFINITE)
```

This function waits for a kernel object to become "signaled." The meaning of "signaled" depends on the object type. For threads, `WaitForSingleObject()` waits for a thread to terminate.

`WaitForMultipleObjects()` waits for more than one kernel object at the same time.

Posting Window Message

```
HANDLE CWnd->PostMessage(

      UINT message,

      // Message (WM_DONE)

      WPARAM wParam,

      LPARAM lParam );

      // Additional Message info
```

`PostMessage` sends a message to the given `CWnd` object. It is useful for signaling done (`WM_DONE`).

Hint #1: CreateThread and C++

This is a trick to create a thread calling an object method. Win32 needs a C style or static method for a `CreateThread` call, but you can work around this by using a static method on the class, then passing the `this` pointer of the object and having the static method just invoke the desired method on the object.

```
Class Foo {

      Static DWORD WINAPI _threadimpl(LPVOID param);

      Bool ThreadProc();

};

…

Foo::_threadimpl(LPVOID param) {

      Foo* pFooObj = (Foo*) param;

      pFooObj->ThreadObj();

}

…

Foo SomeFoo;

HANDLE hThread = CreateThread(NULL, 0, Foo::_threadimpl,
(LPVOID)&SomeFoo, 0, NULL);
```

intel®