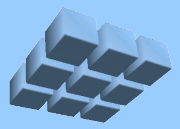
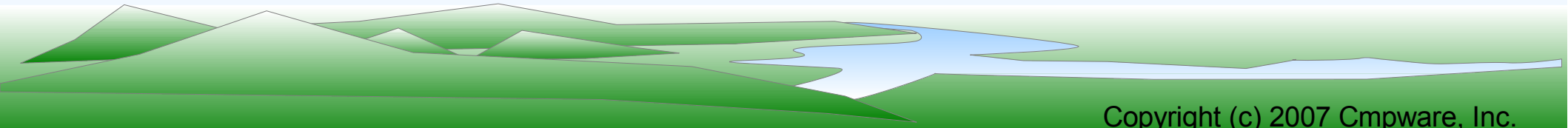
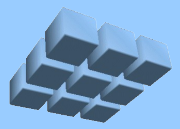


The Cmpware CMP-DK 3.0: Cell BE Version

Cmpware, Inc.

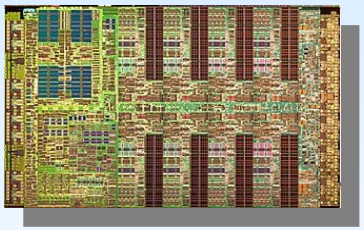


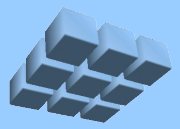


- I. Introduction
- II. Installing the *Cmpware CMP-DK*
- III. Application I: A Simple Program
- IV. Application II: Shared Memory
- V. Application III: Mailboxes
- VI. Optimization I: SPU Dependencies
- VII. Optimization II: SPU Dual Issue
- VIII. System Level Analysis
- IX. Overview

Introduction

- *Cmpware CMP-DK* for the *Cell BE*
  - Uses *Cmpware's* powerful multicore modeling technology
  - A simulation based software development enviroment for the Cell BE
  - Special Cell BE displays to aid in code partitioning and optimization of
  - Complements existing tools



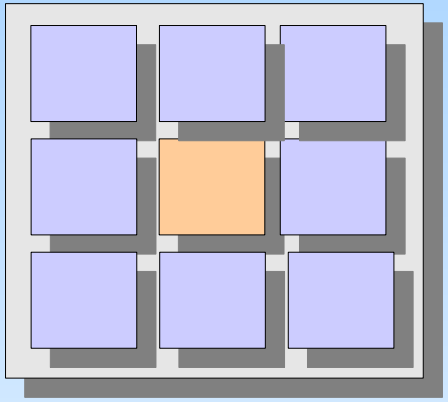


The Cmpware Approach

- Multicore simulaton
  - Software development on simulator
  - High level of execution control
  - Run-time execution statistics
  - Simpler (and often cheaper) than HW
- Powerful multicore IDE
  - Easy access to all system data
  - Specifically designed for multicore

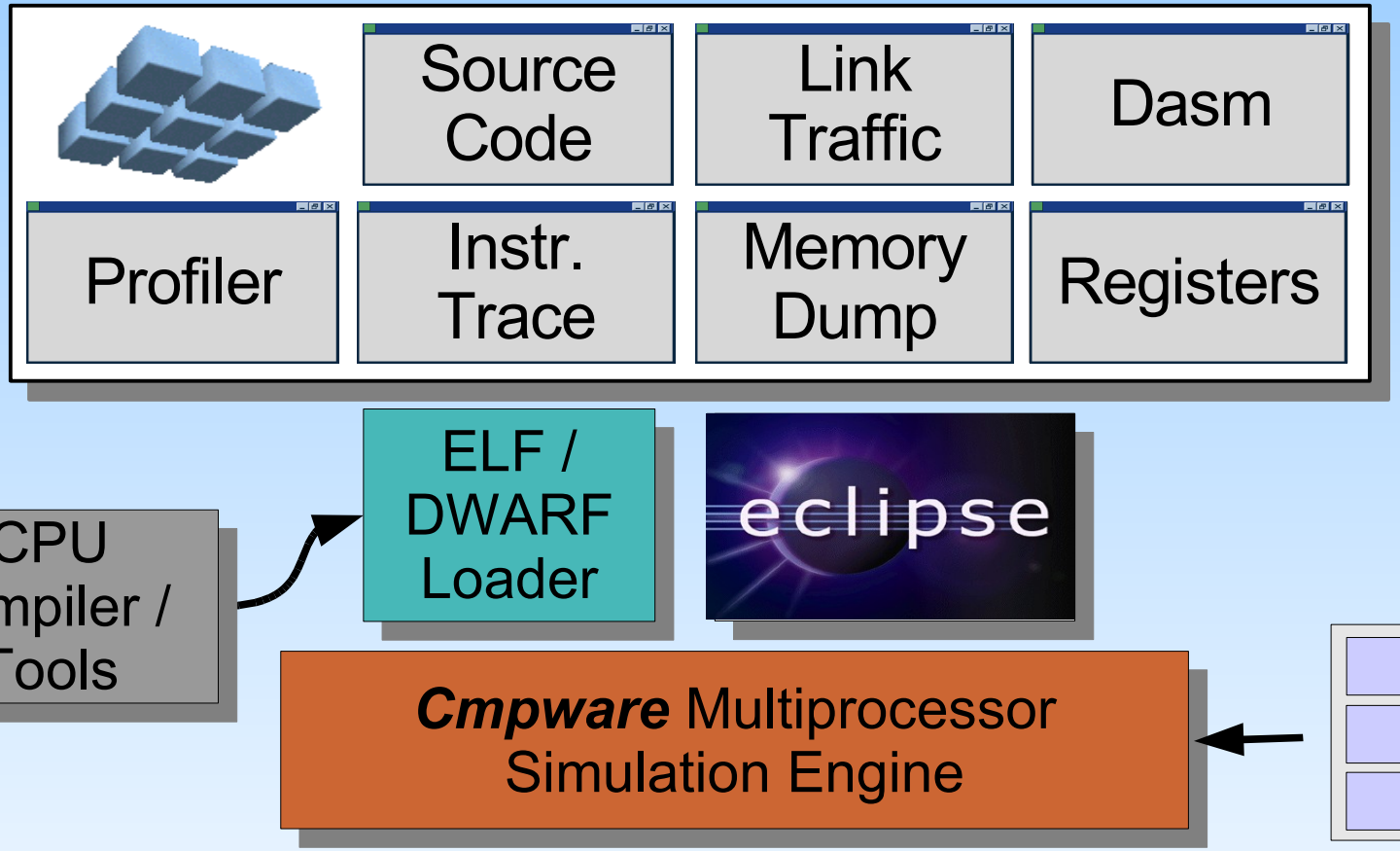
The Cell BE Simulation Model

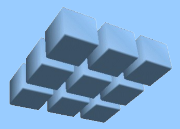
- Cmpware *Cell BE* simulation model:
  - 64-bit *PowerPC* core
  - *PowerPC* FP extensions
  - Eight *SPU* cores
  - System level model ('glue')



- Supplies all IDE display data
- Built-in assemblers and disassemblers
- **4M+** operations per second

Cmpware CMP-DK IDE





I. Introduction

- II. Installing the *Cmpware CMP-DK*

III. Application I: A Simple Program

IV. Application II: Shared Memory

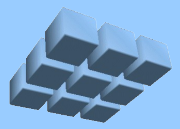
V. Application III: Mailboxes

VI. Optimization I: SPU Dependencies

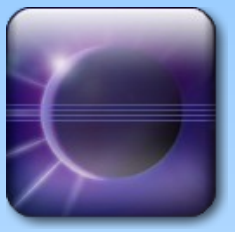
VII. Optimization II: SPU Dual Issue

VIII. System Level Analysis

IX. Overview



Installing the Cmpware CMP-DK

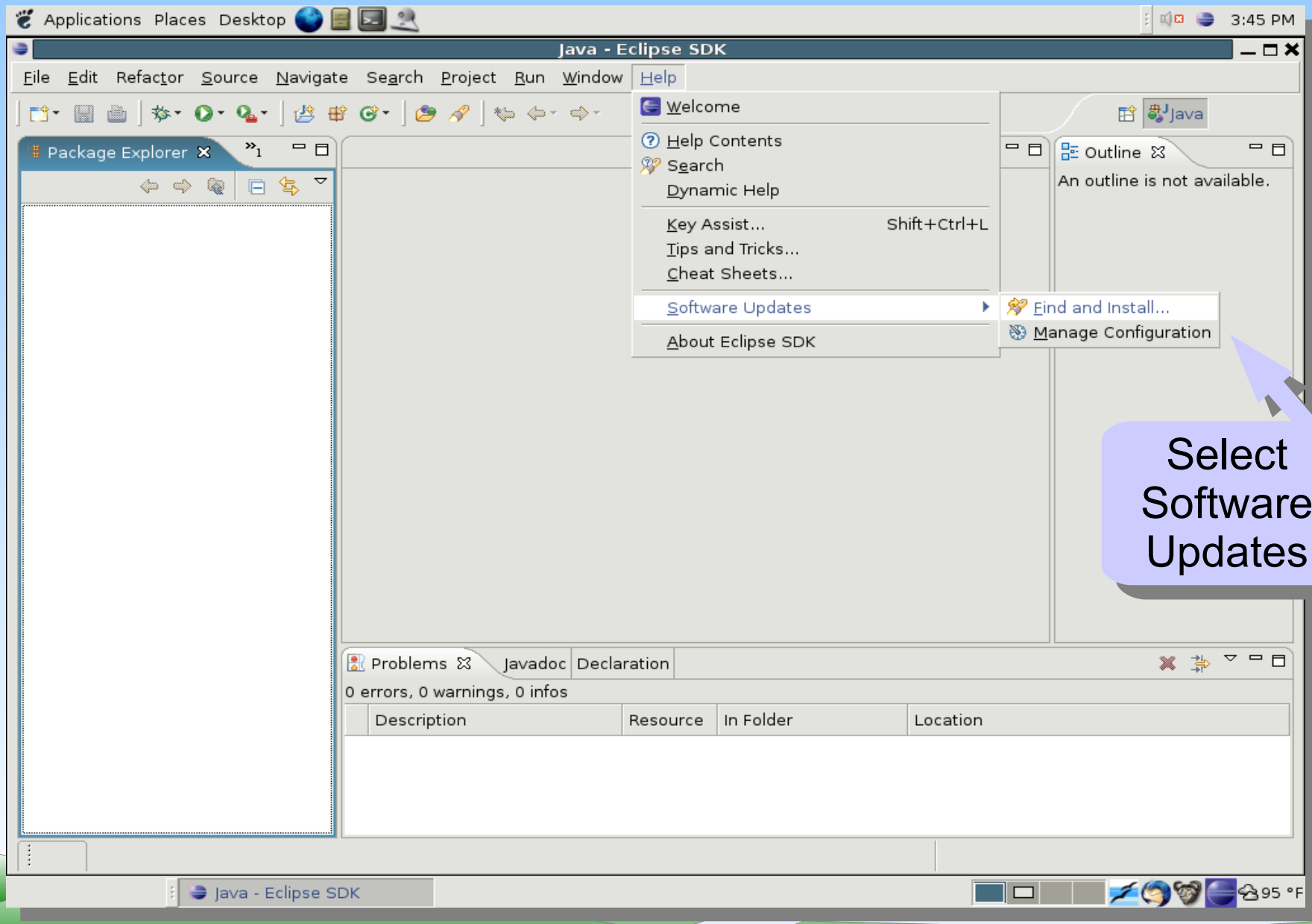


- An *Eclipse* 'plugin'
- Approximately 1.5 MB (total)
- Installs from an *Eclipse 'Update Site'*:

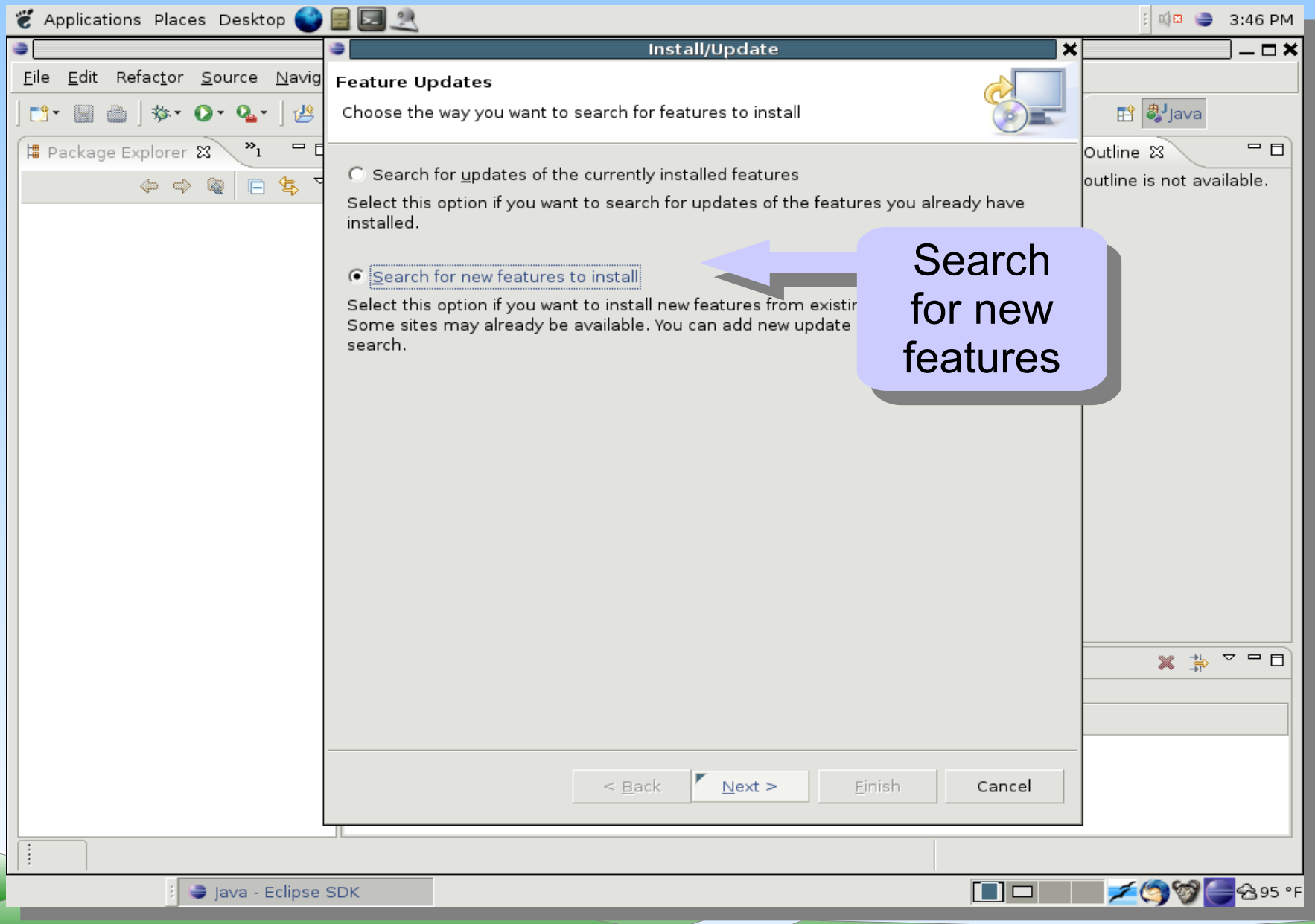
  http://www.cmpware.com/cellbe/
- Up and running in seconds*
- Works with other existing *Eclipse* installations
- 'C' Development Toolkit (CDT) recommended

* requires license code from *Cmpware*

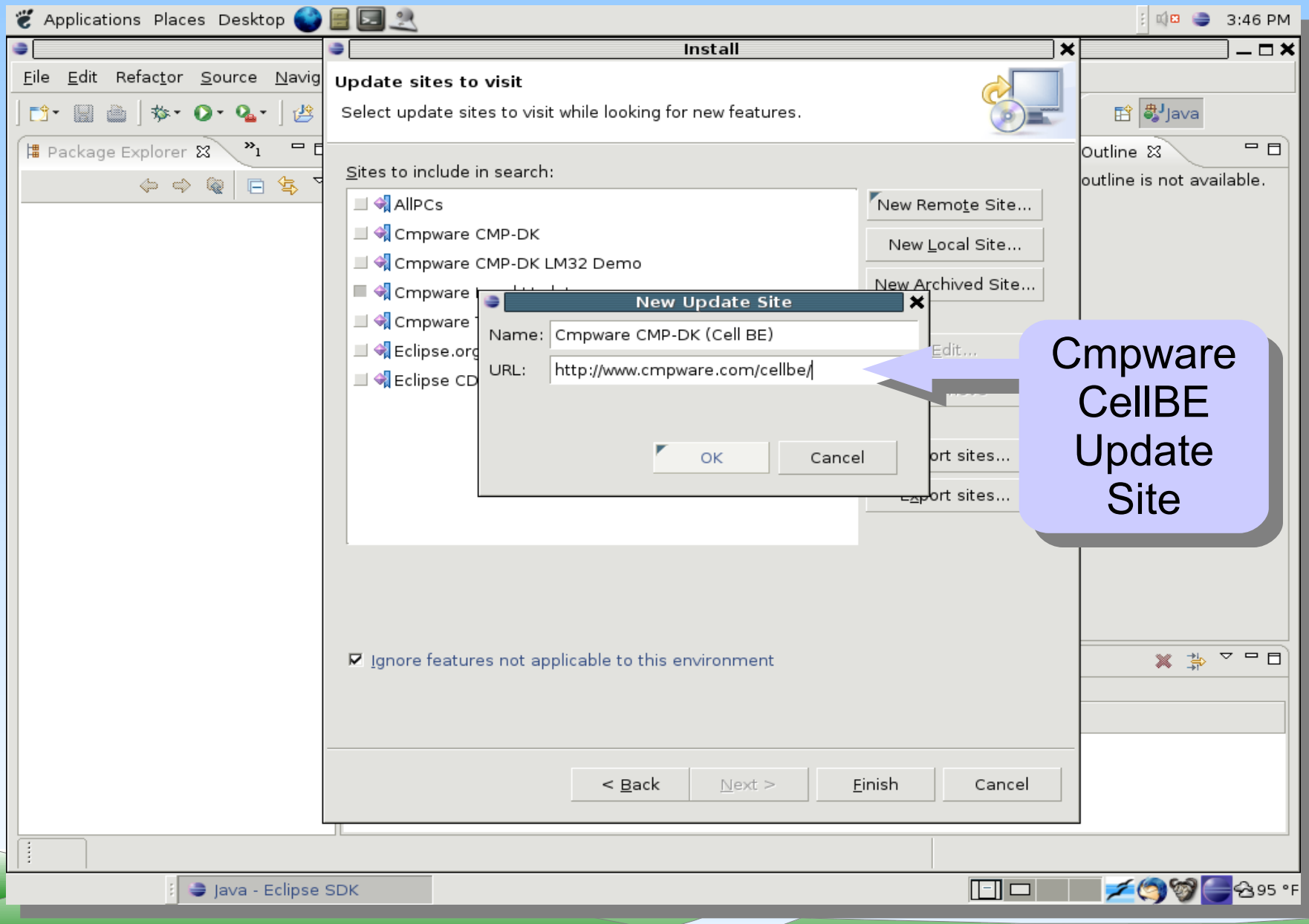
Installation: Step 1



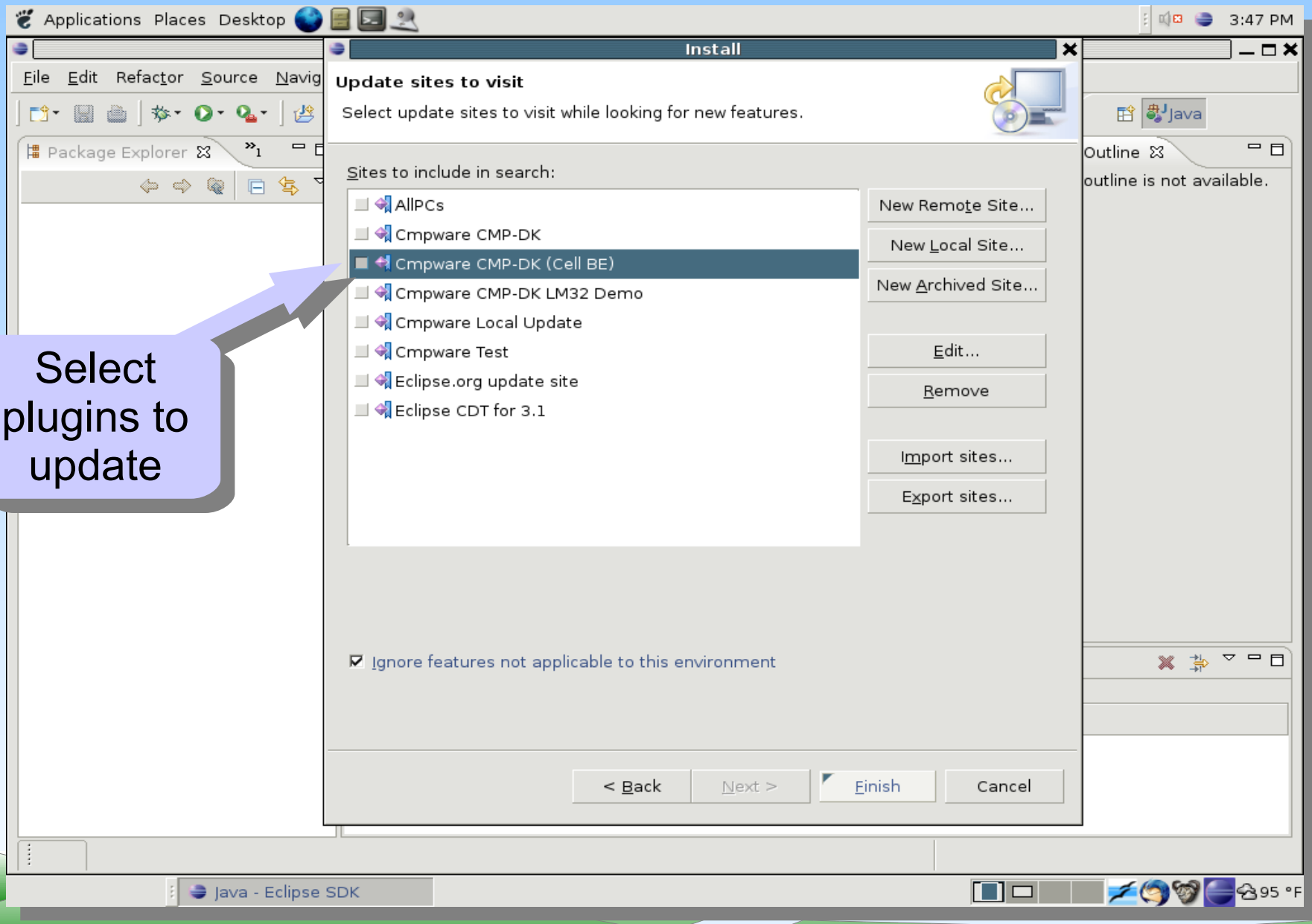
Installation: Step 2



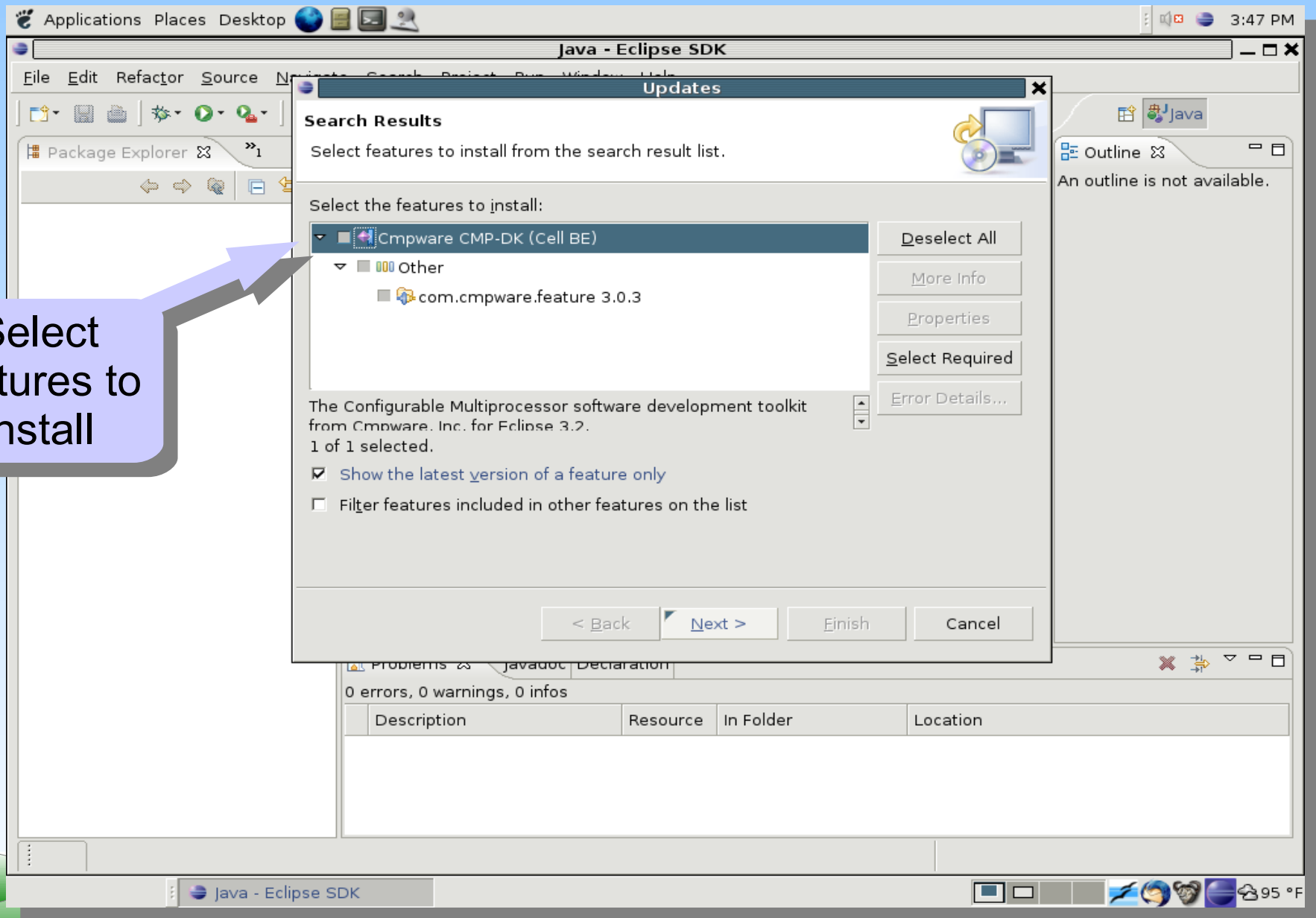
Installation: Step 3



Installation: Step 4

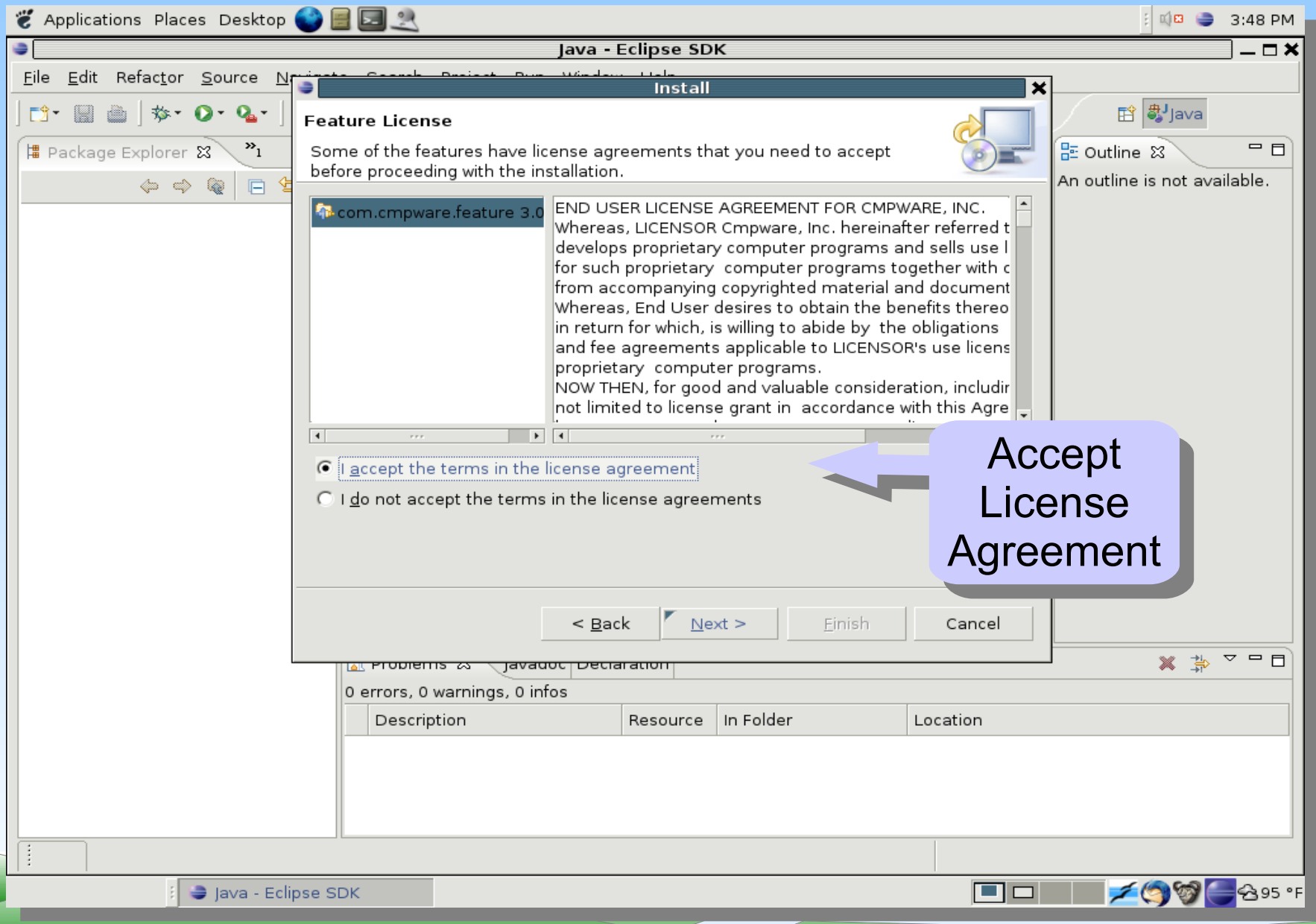


Installation: Step 5

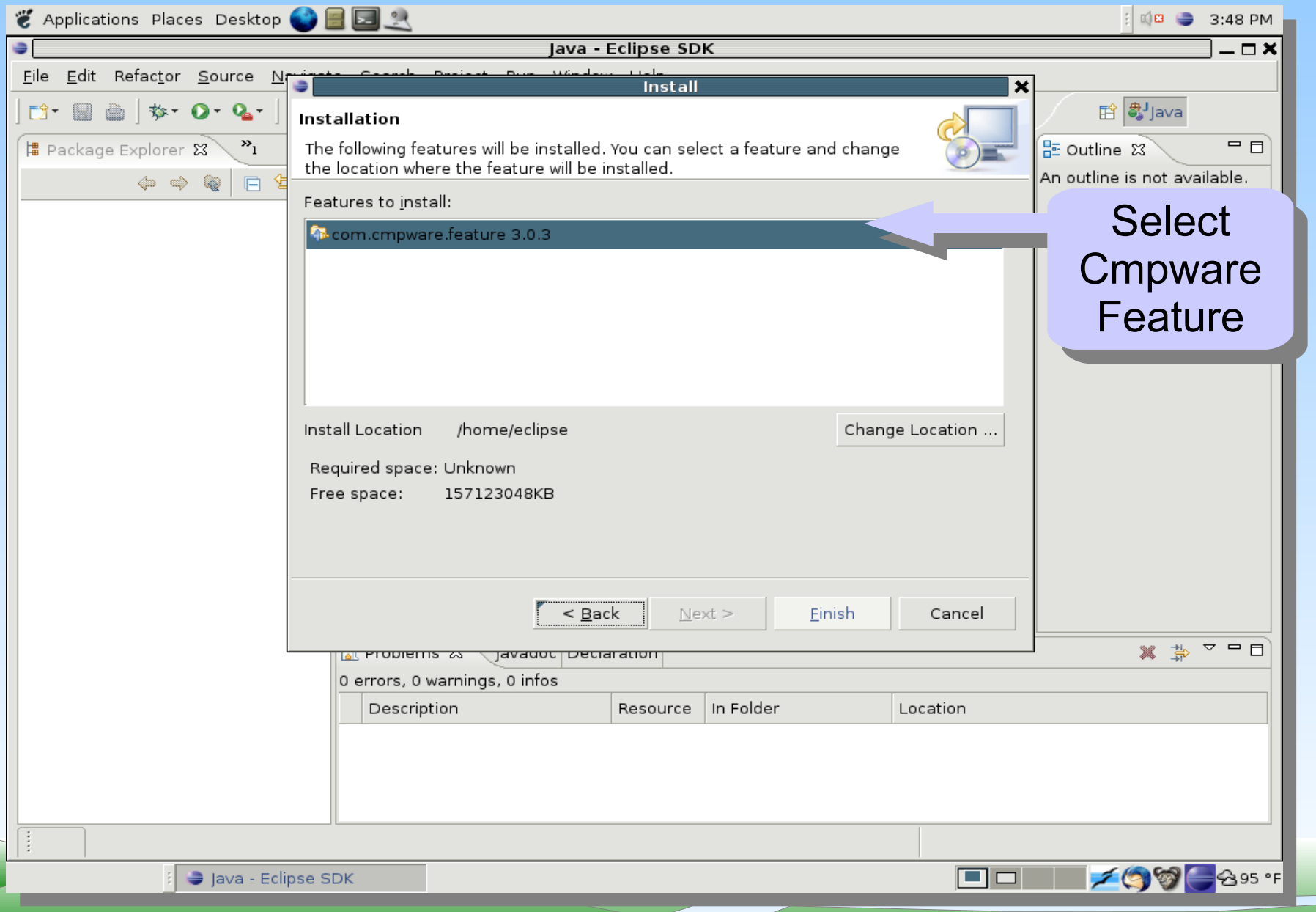


Select features to install

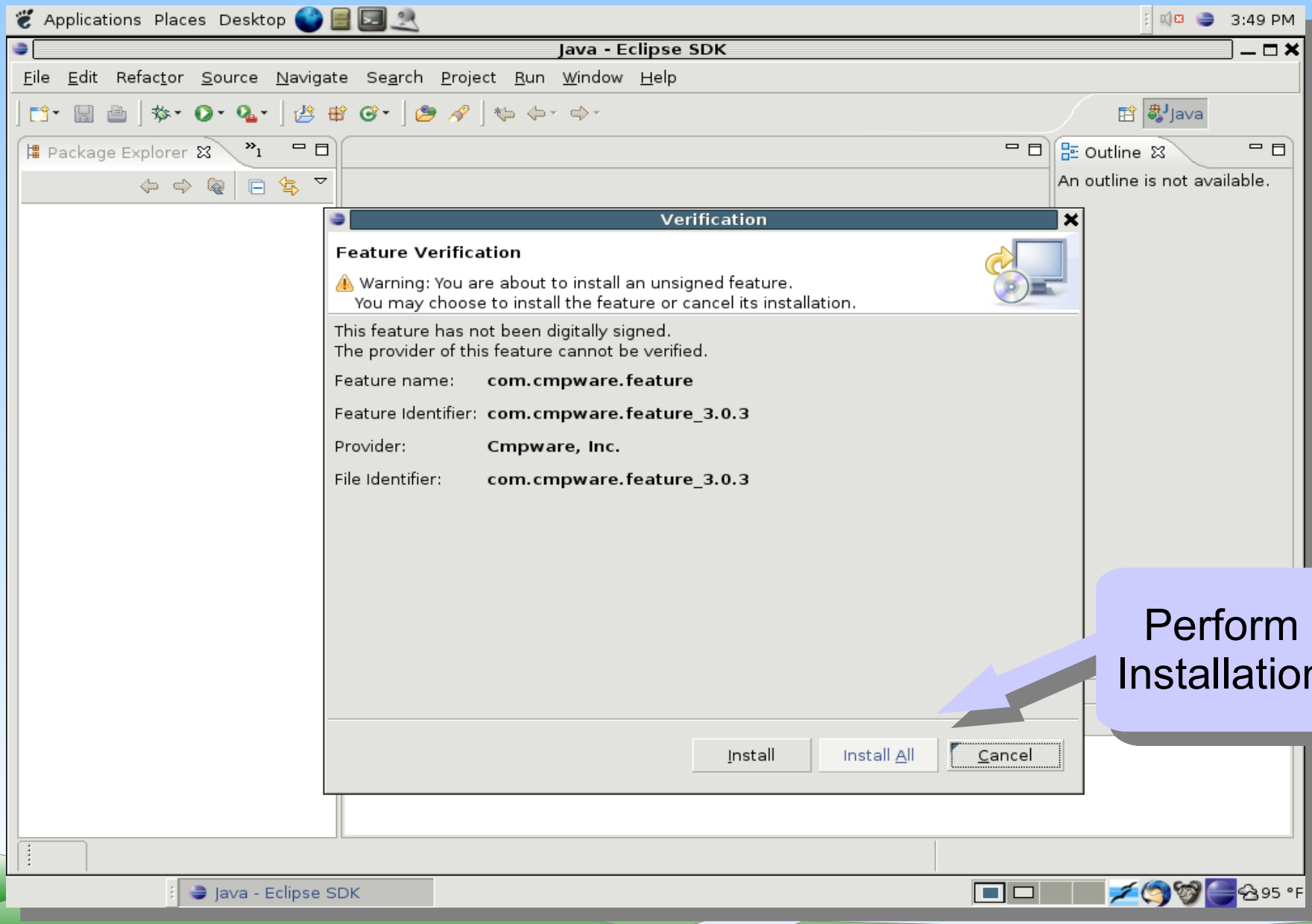
Installation: Step 6



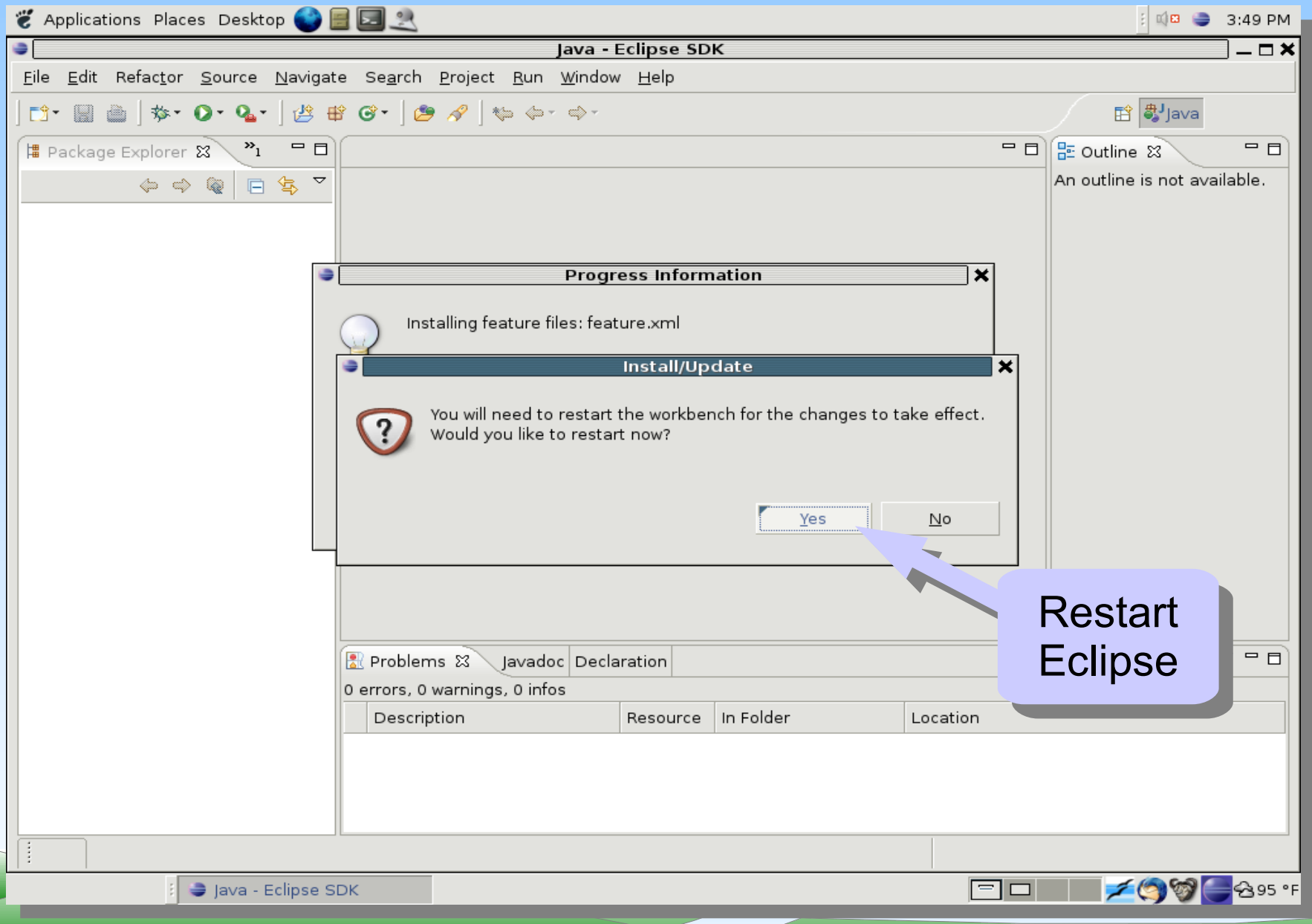
Installation: Step 7



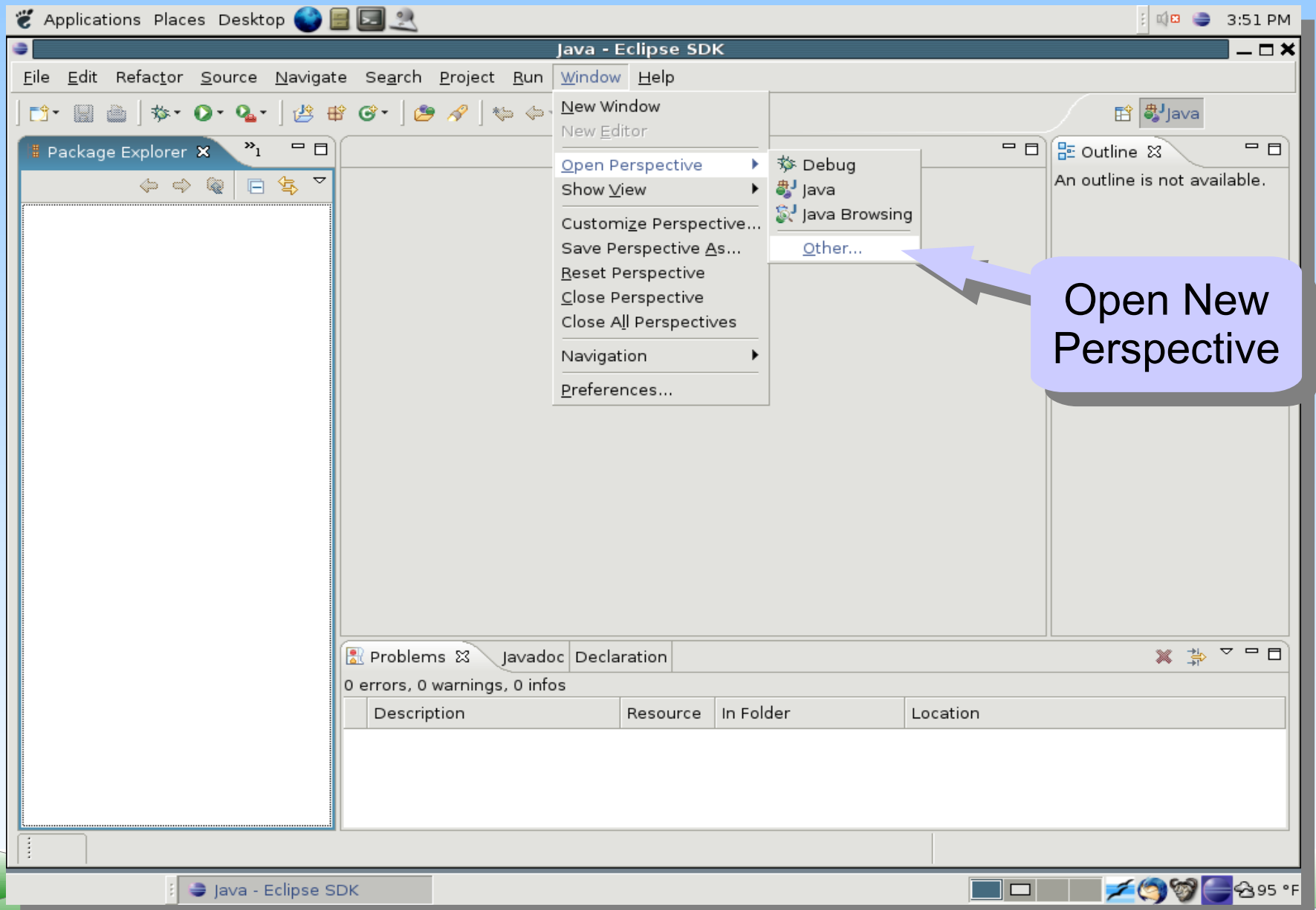
Installation: Step 8



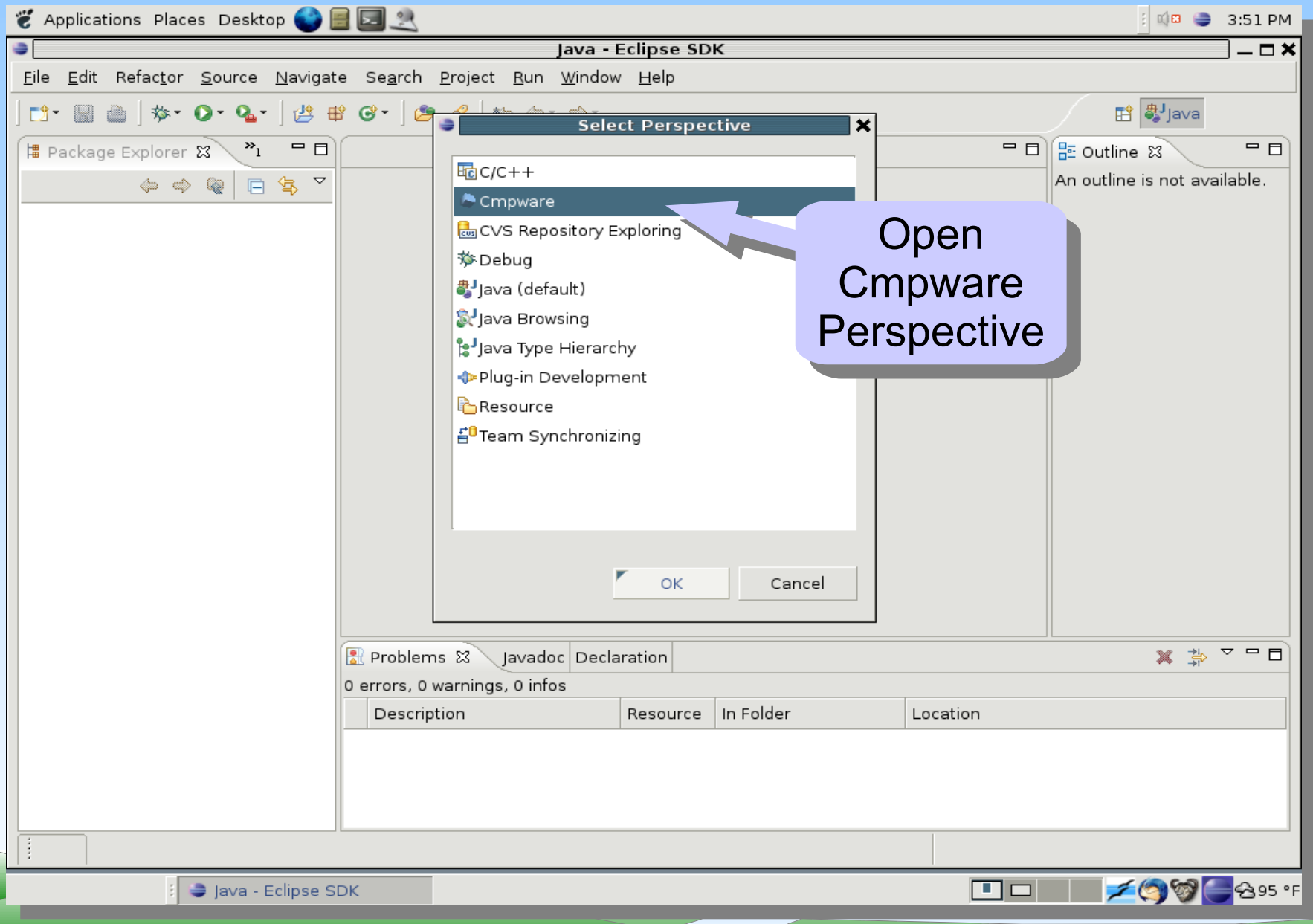
Installation: Step 9



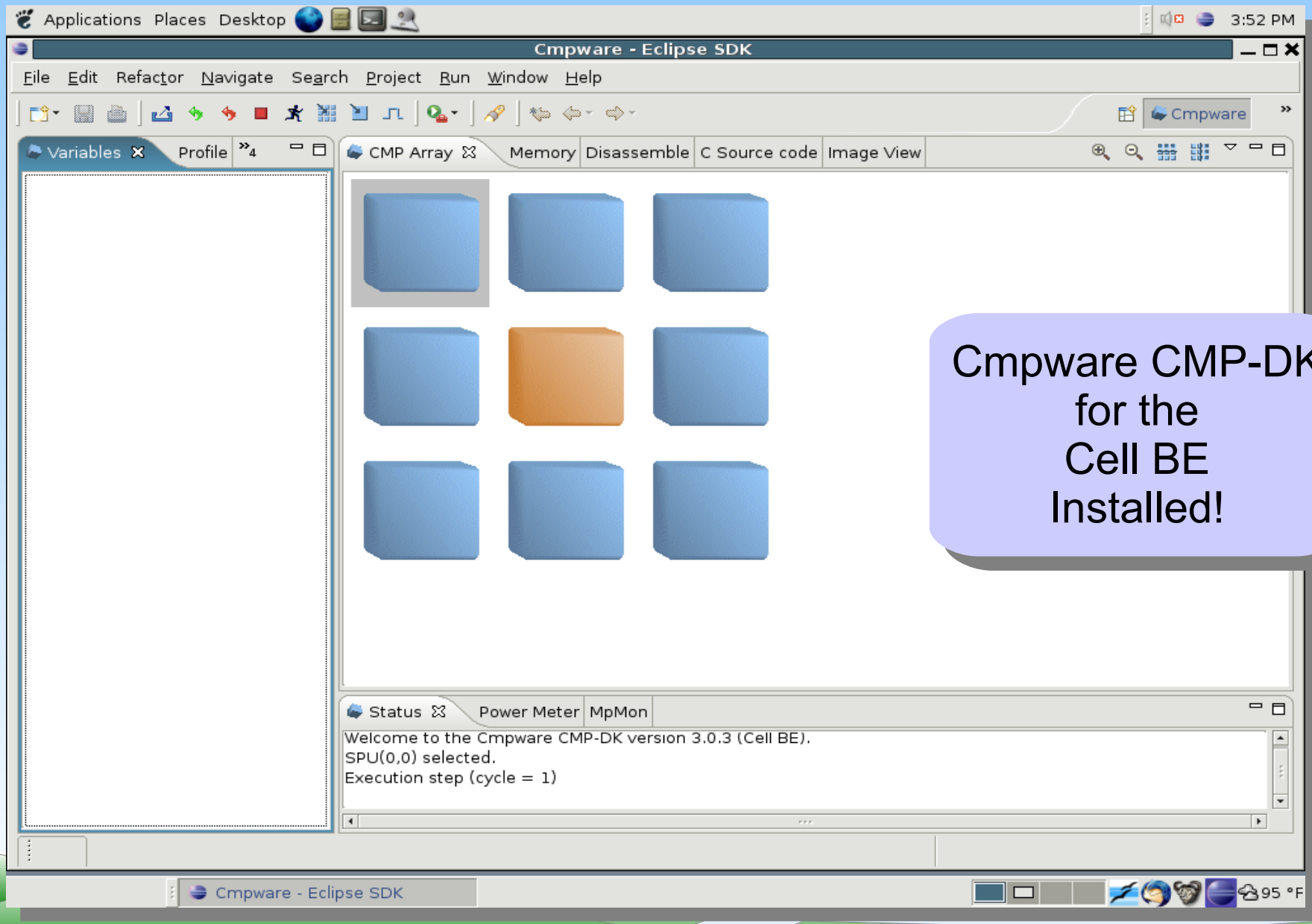
Installation: Step 10



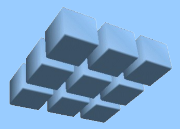
Installation: Step 11



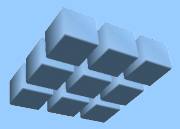
Installation: Cmpware CMP-DK



Cmpware CMP-DK for the Cell BE Installed!



I. Introduction
II. Installing the *Cmpware CMP-DK*
III. Application I: A Simple Program
IV. Application II: Shared Memory
V. Application III: Mailboxes
VI. Optimization I: SPU Dependencies
VII. Optimization II: SPU Dual Issue
VIII. System Level Analysis
IX. Overview



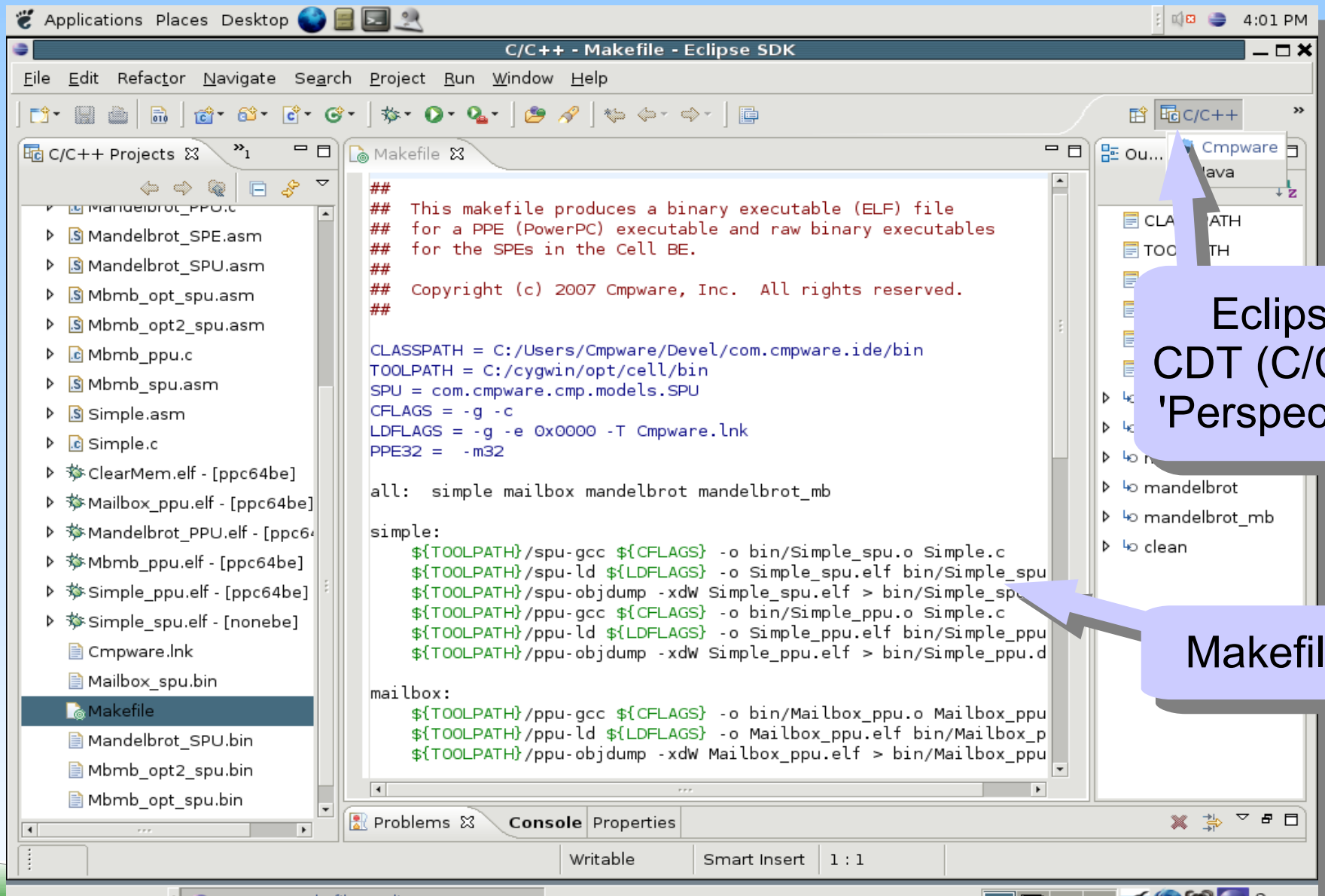
Application I: A Simple Program

- Compiled for both PPU and SPUs
- Tests compiler and *Cmpware* installation
- Uses IBM Cell tools for Windows / Cygwin

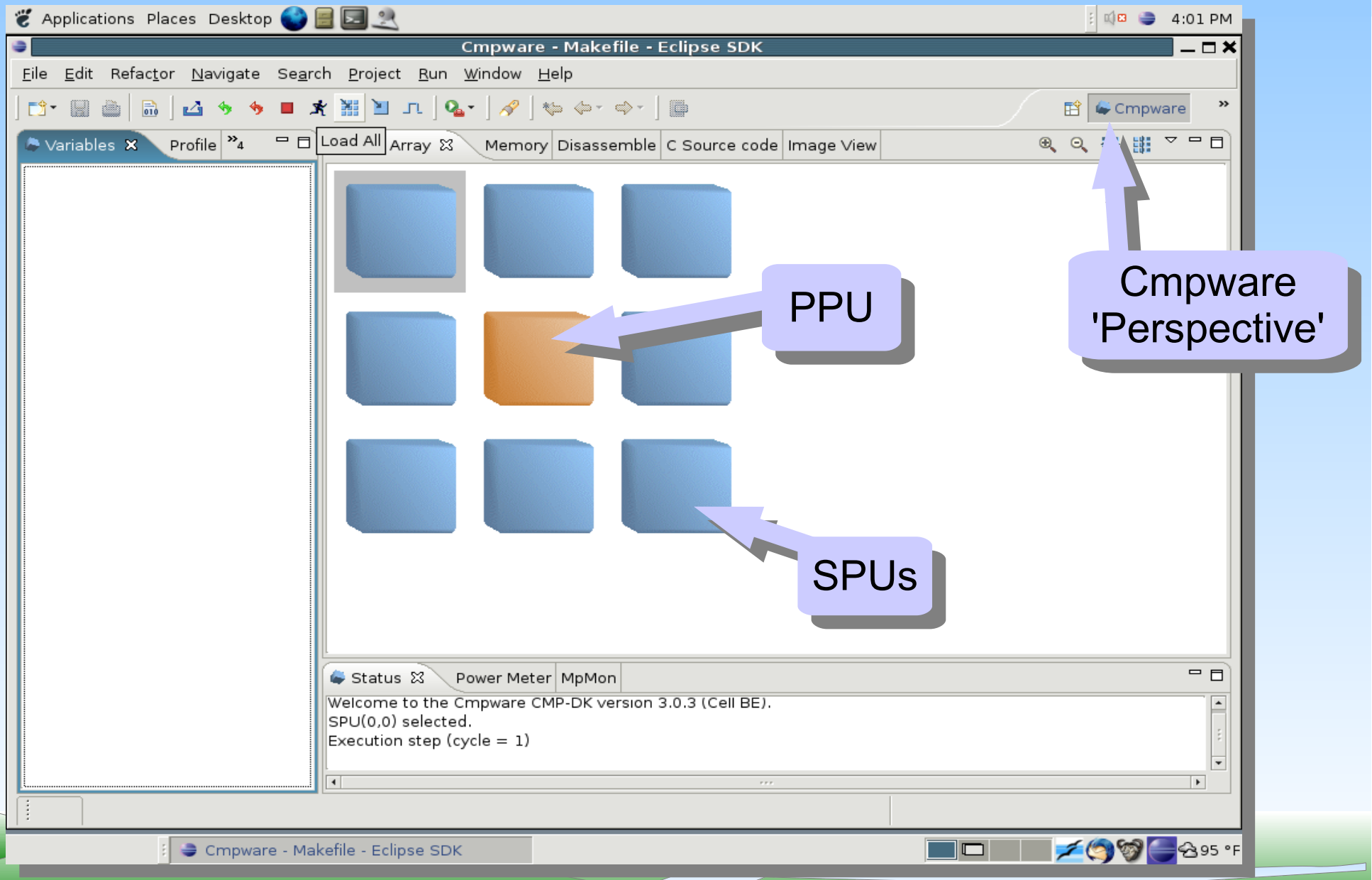
(http://cellbe-cygwin.cvs.sourceforge.net/cellbe-cygwin/cellbe-cygwin/)

```
/*
**  A simple test program
**
** Copyright (c) 2007 Cmpware, Inc. All Rights Reserved.
**
*/

int main(int argc, char *argv[]) {
     int  i;
     float a = 1.0;

     for(i=0; i<1000; i++)
        a = a + 1.0;

}  /* end main() */
```

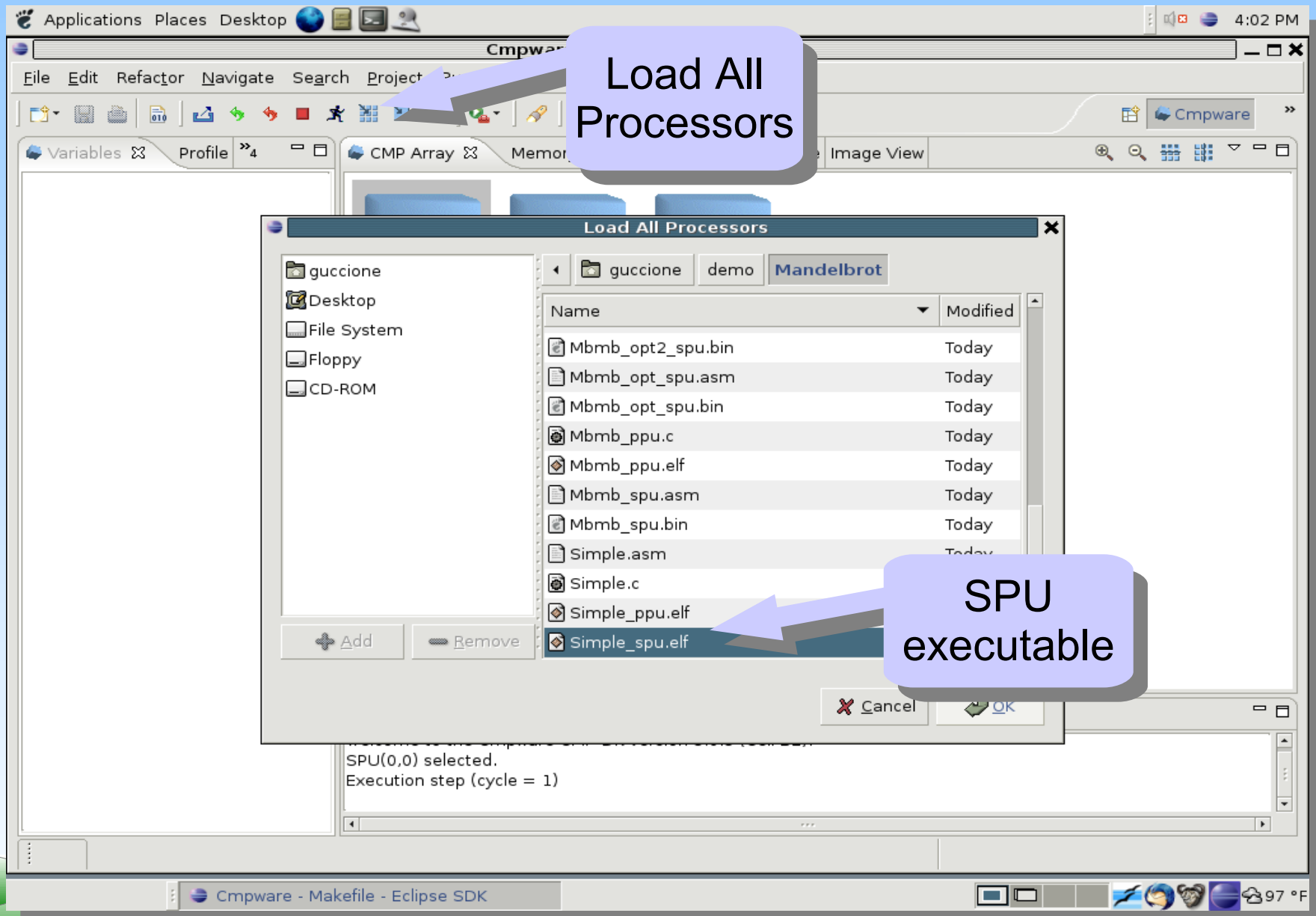
Application I: CDT Makefile



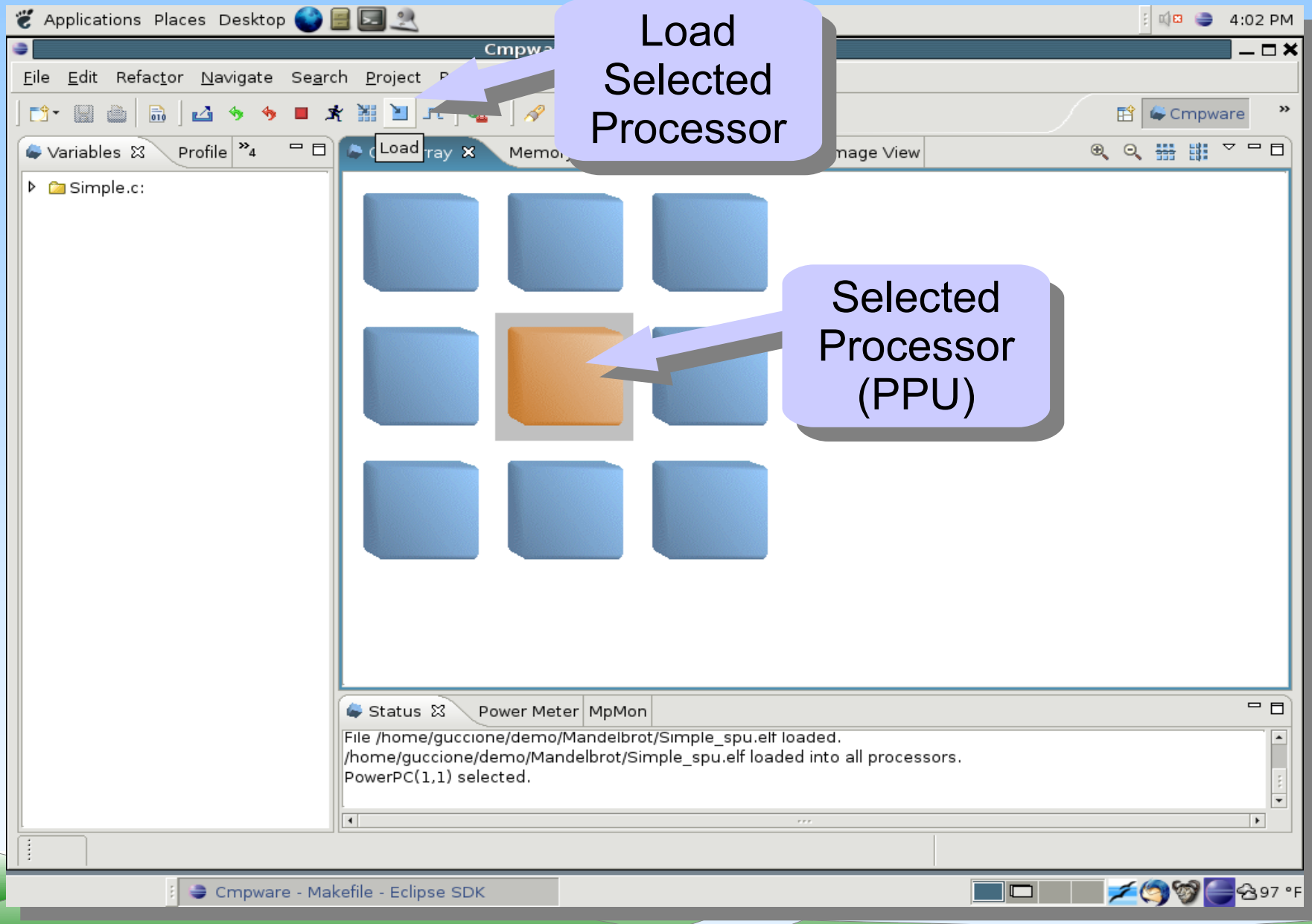
Application I: The Main Display



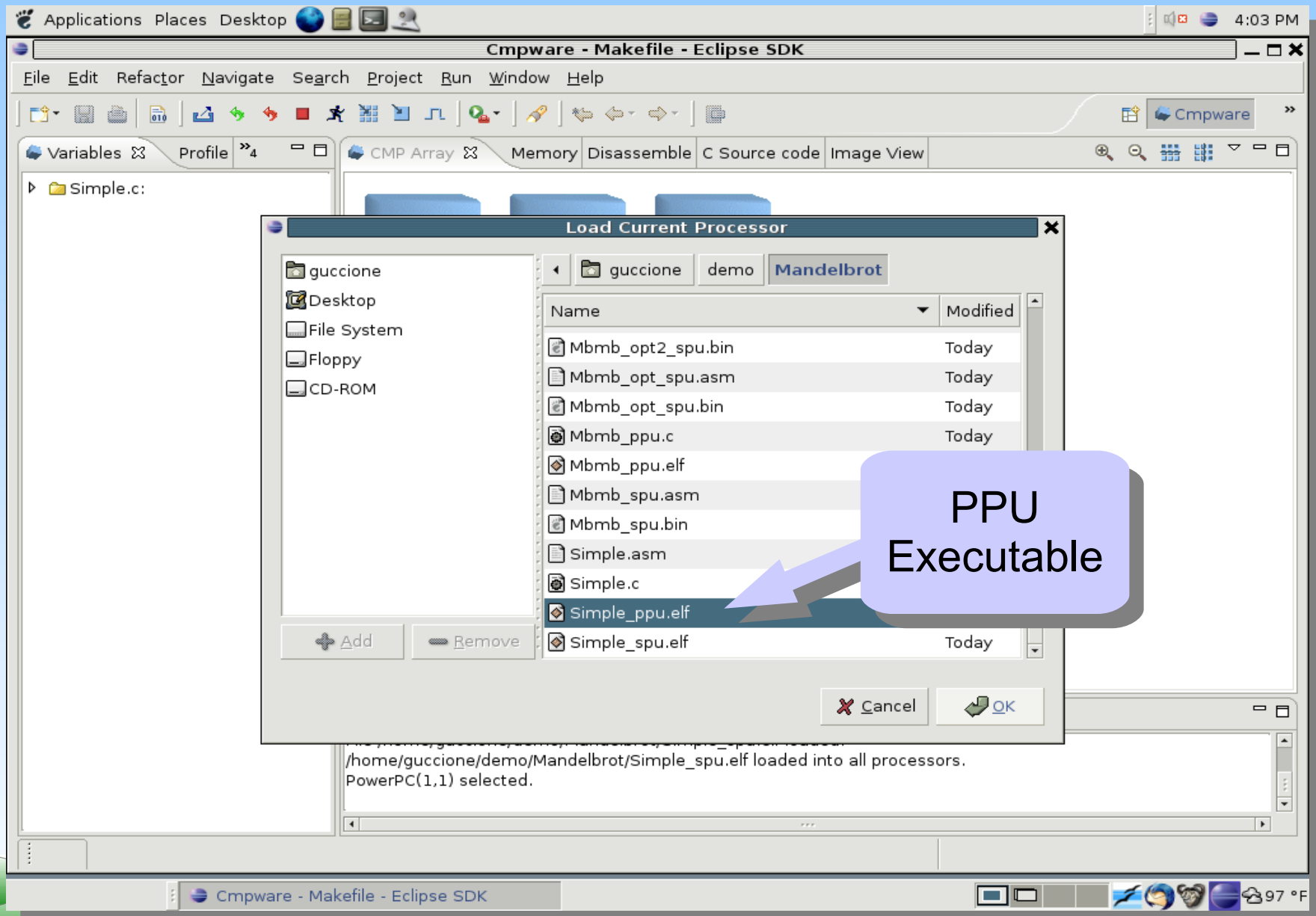
Application I: Loading SPU Code



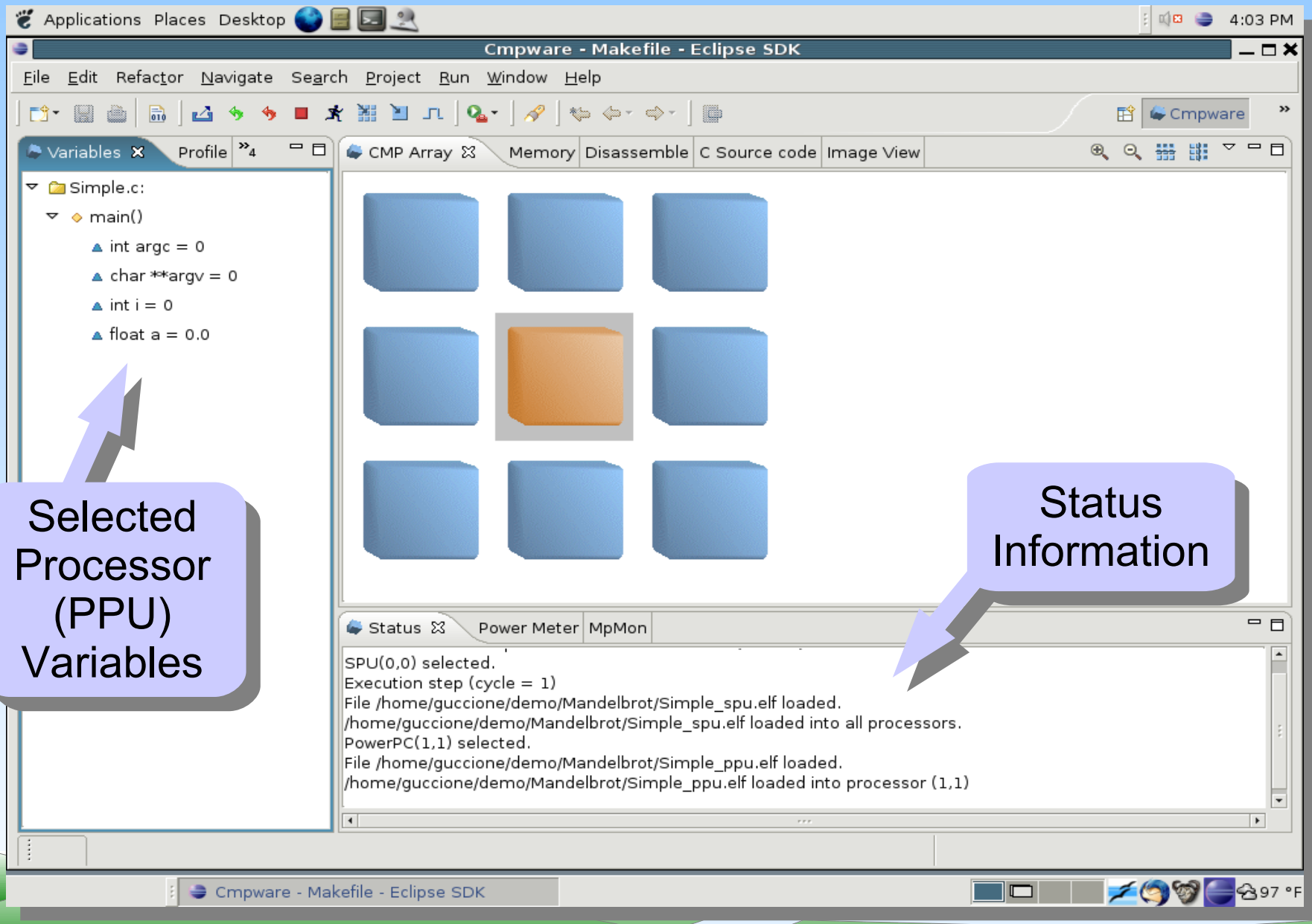
Application I: Loading PPU Code



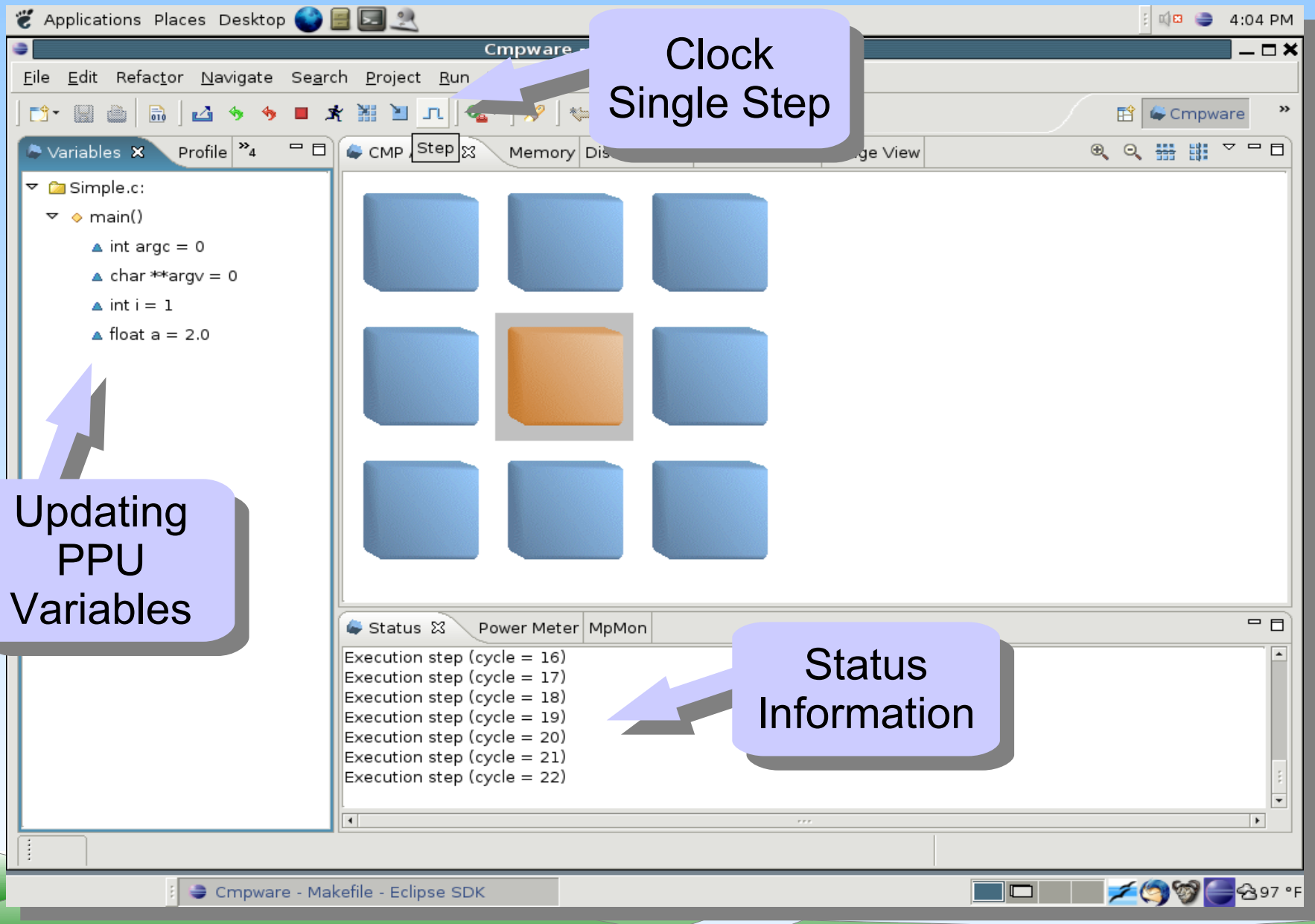
Application I: Loading PPU Code



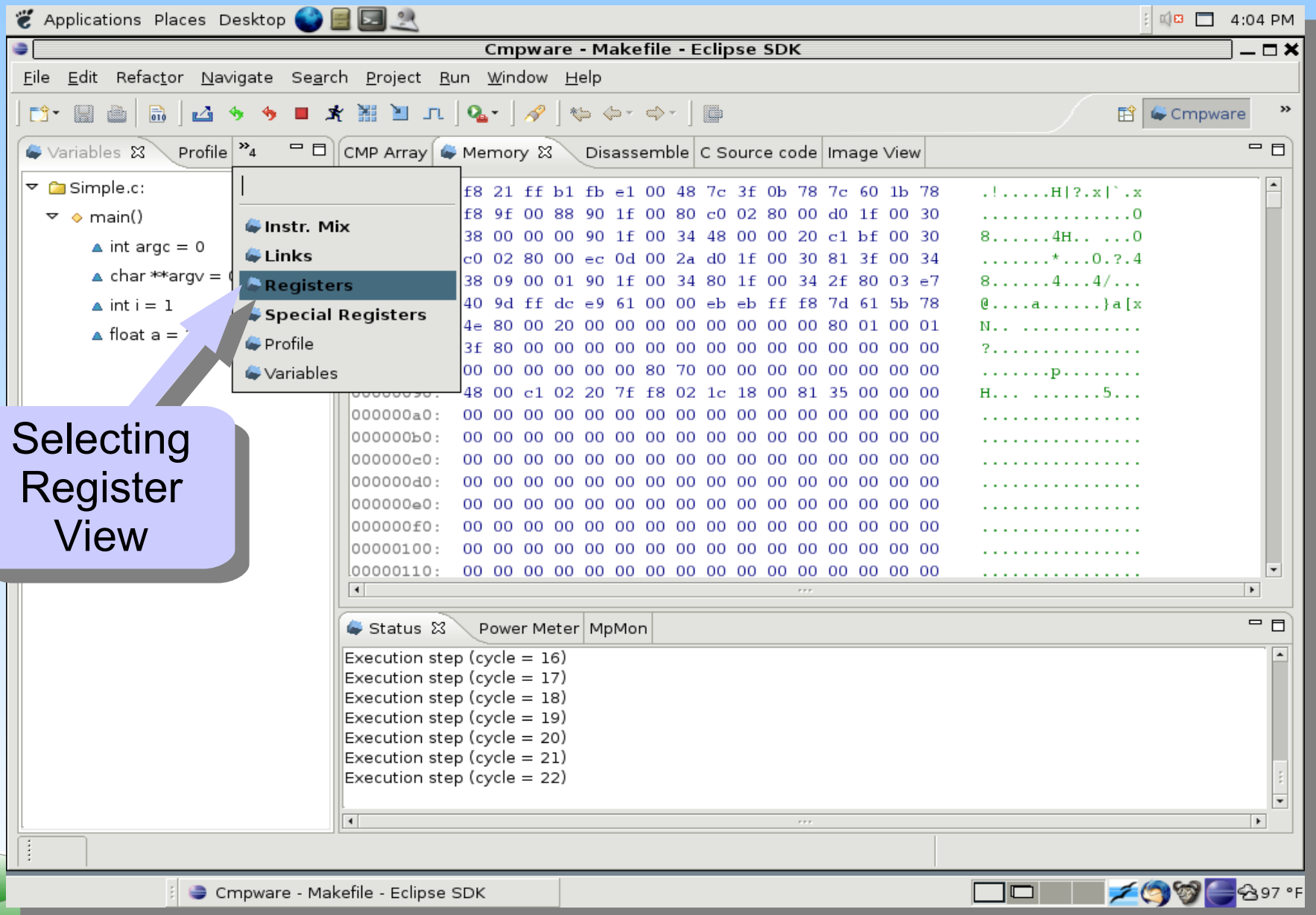
Application I: All Code Loaded



Application I: Executing Code

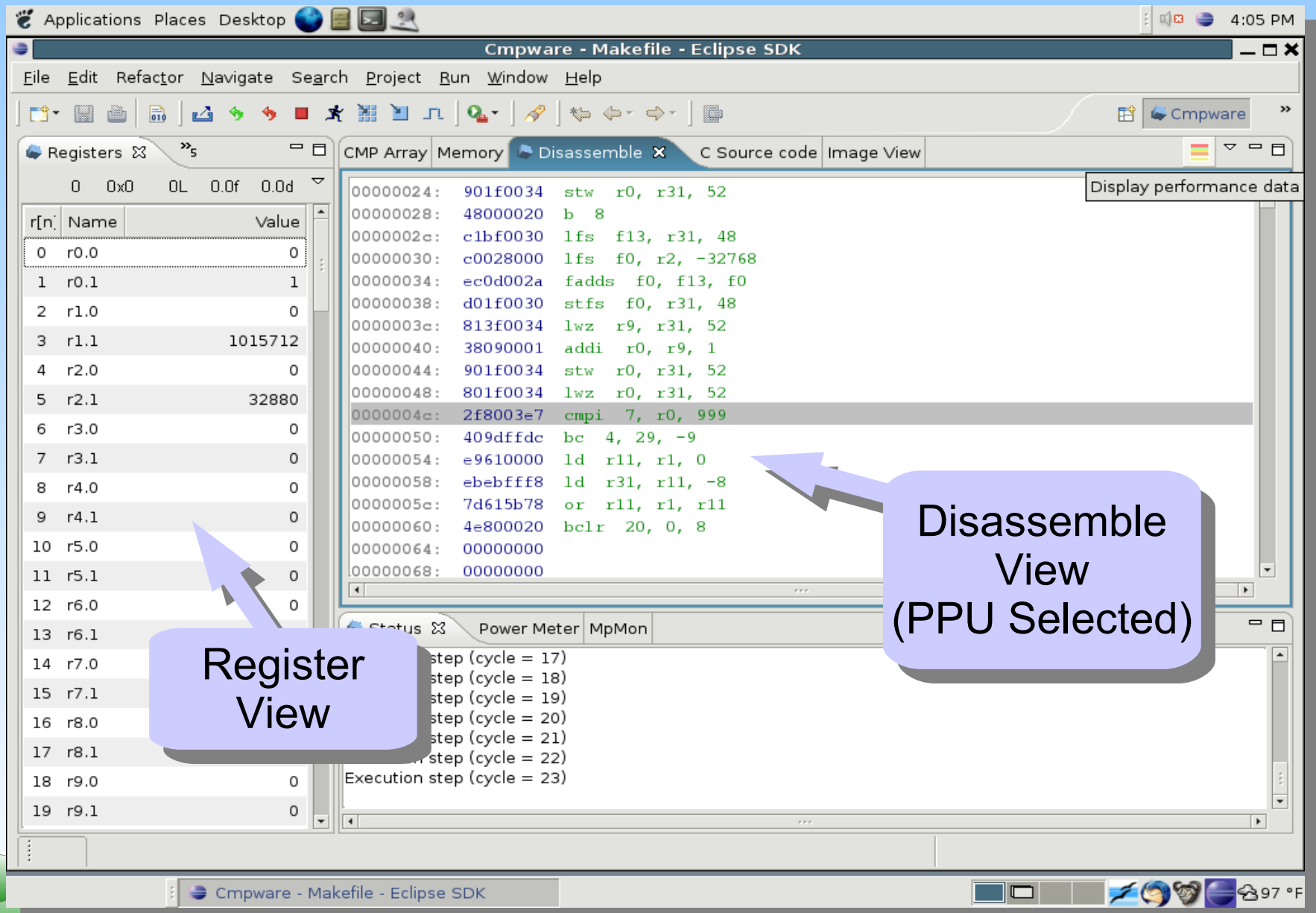


Application I: Memory View

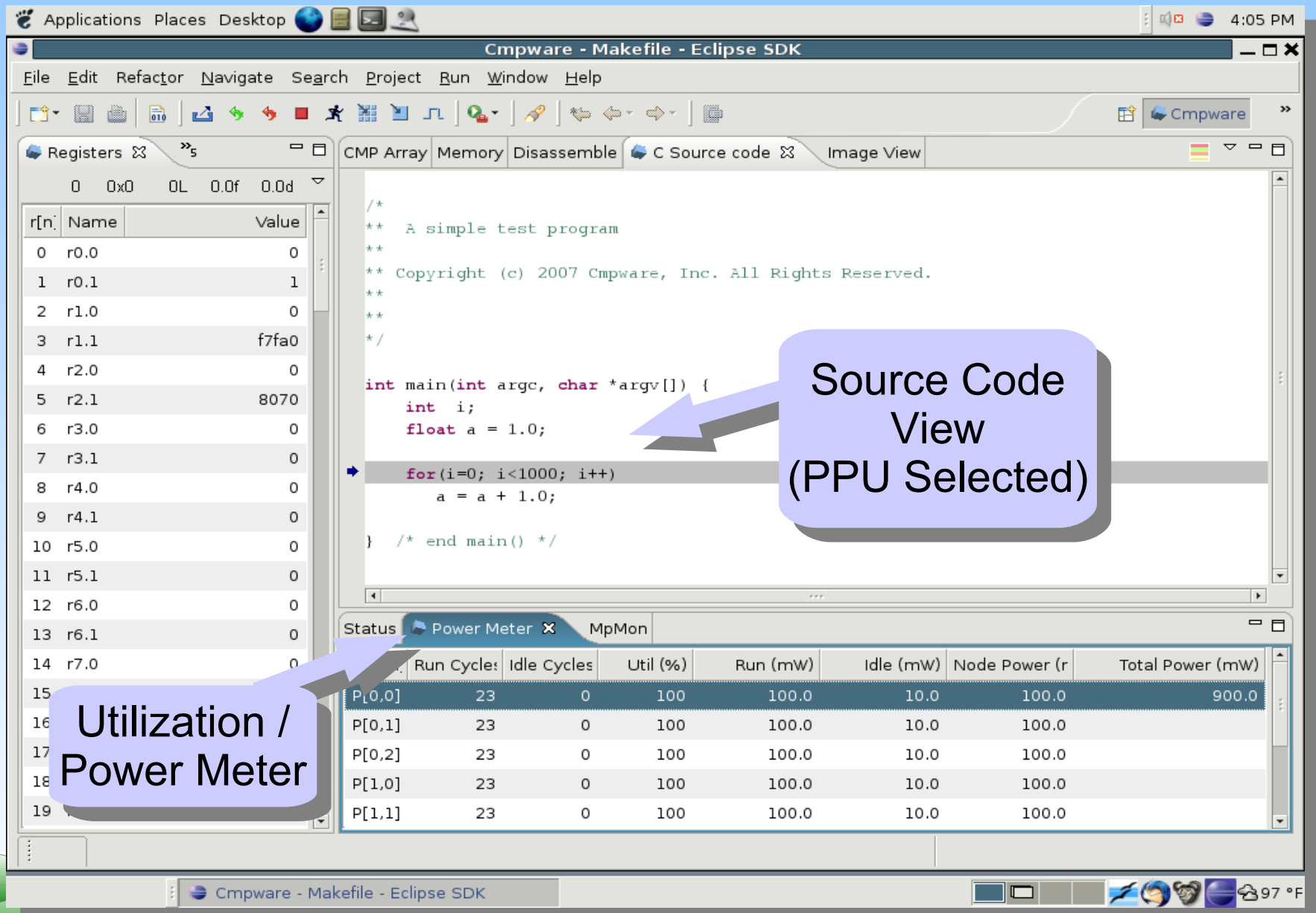


Selecting Register View

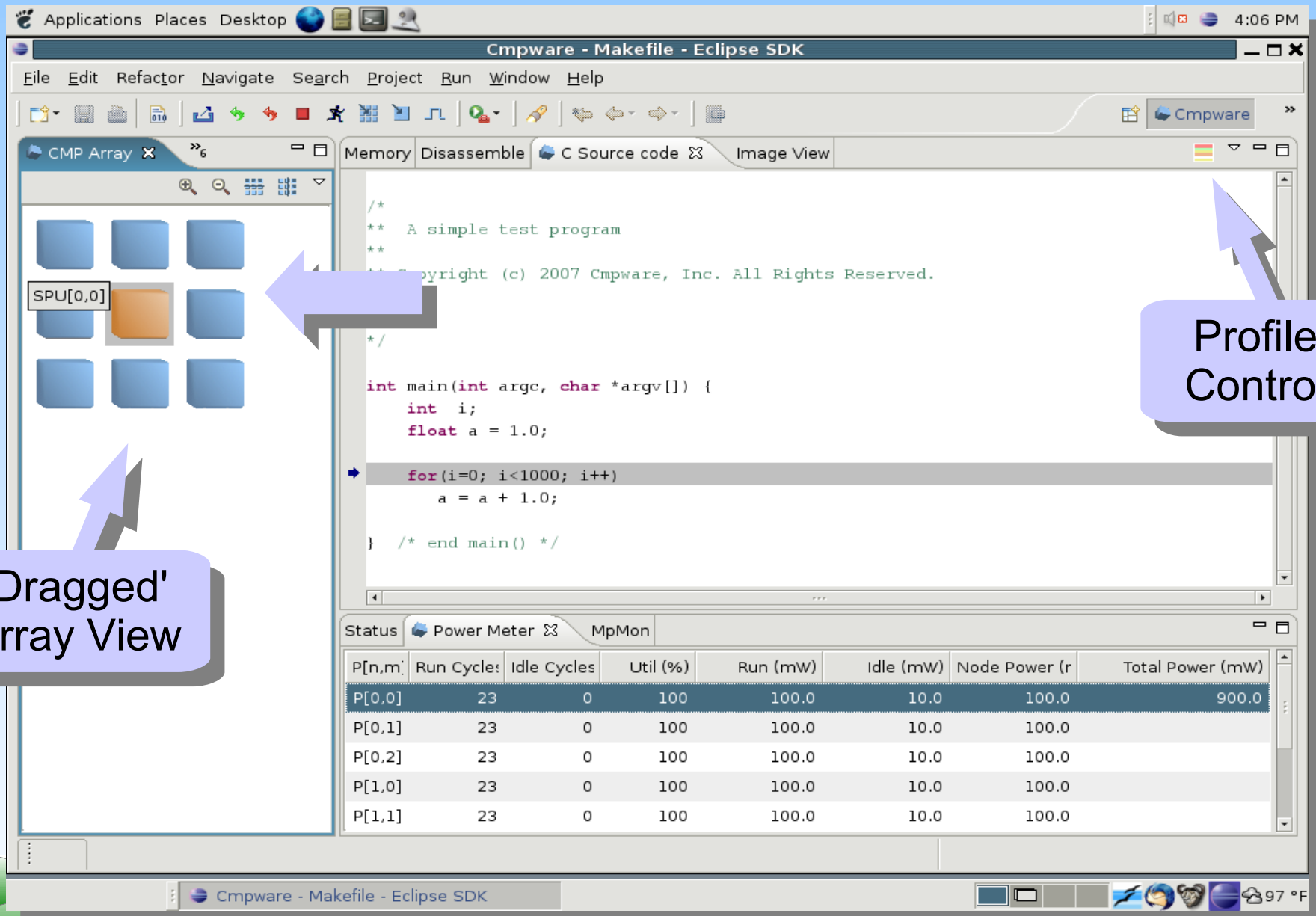
Application I: Disassemble View



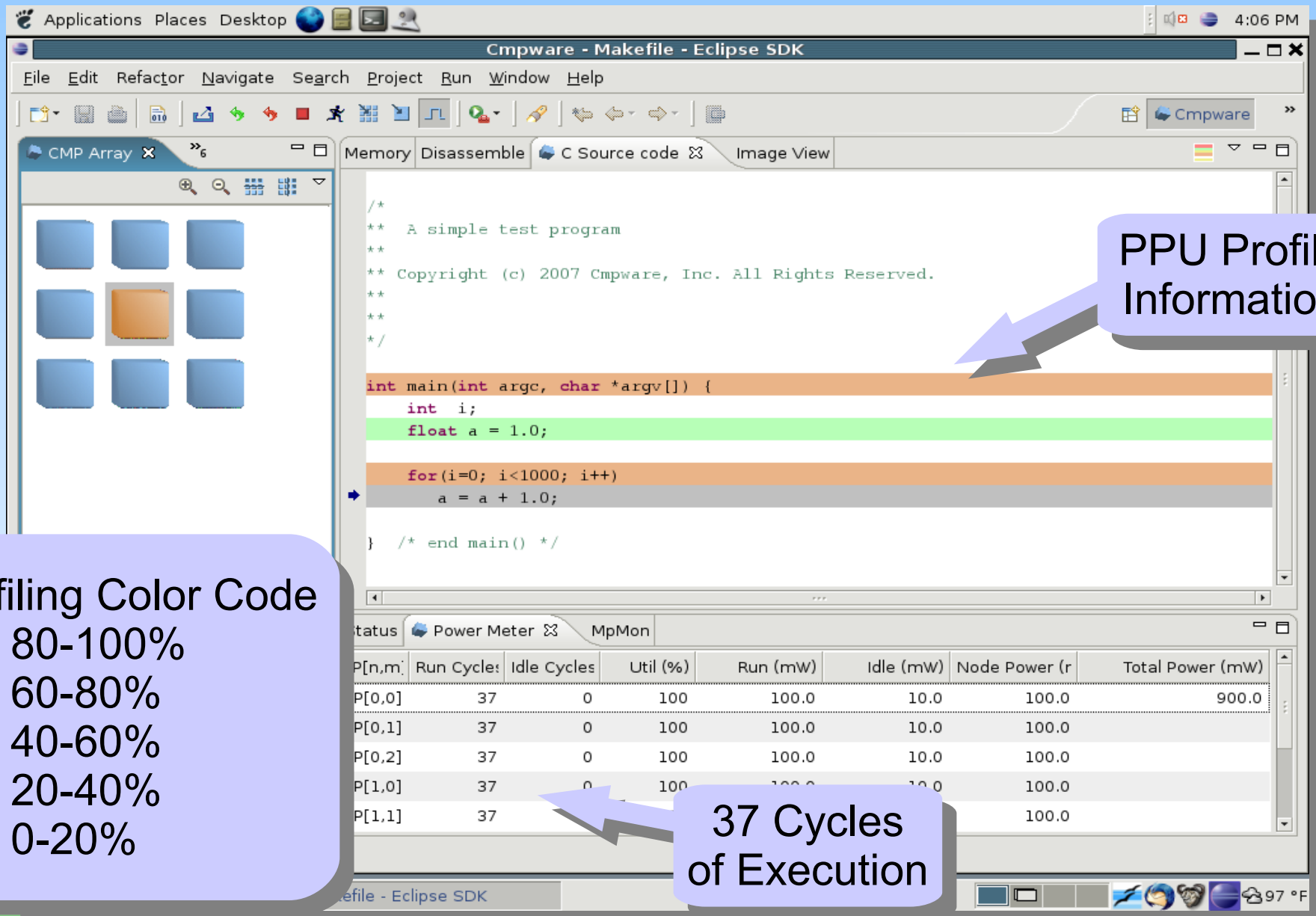
Application I: Source Code View



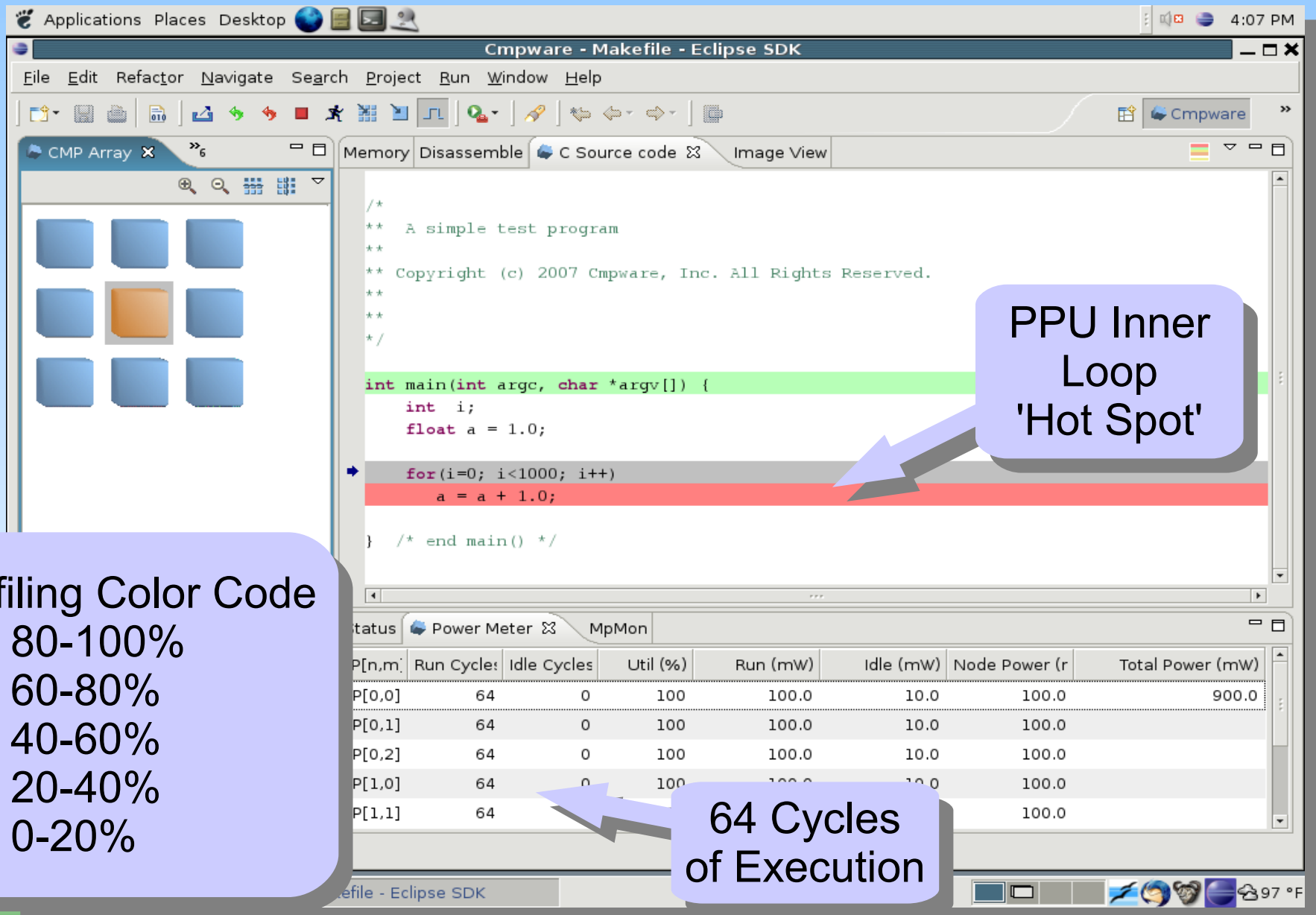
Application I: Moving Windows



Application I: Source Profiling



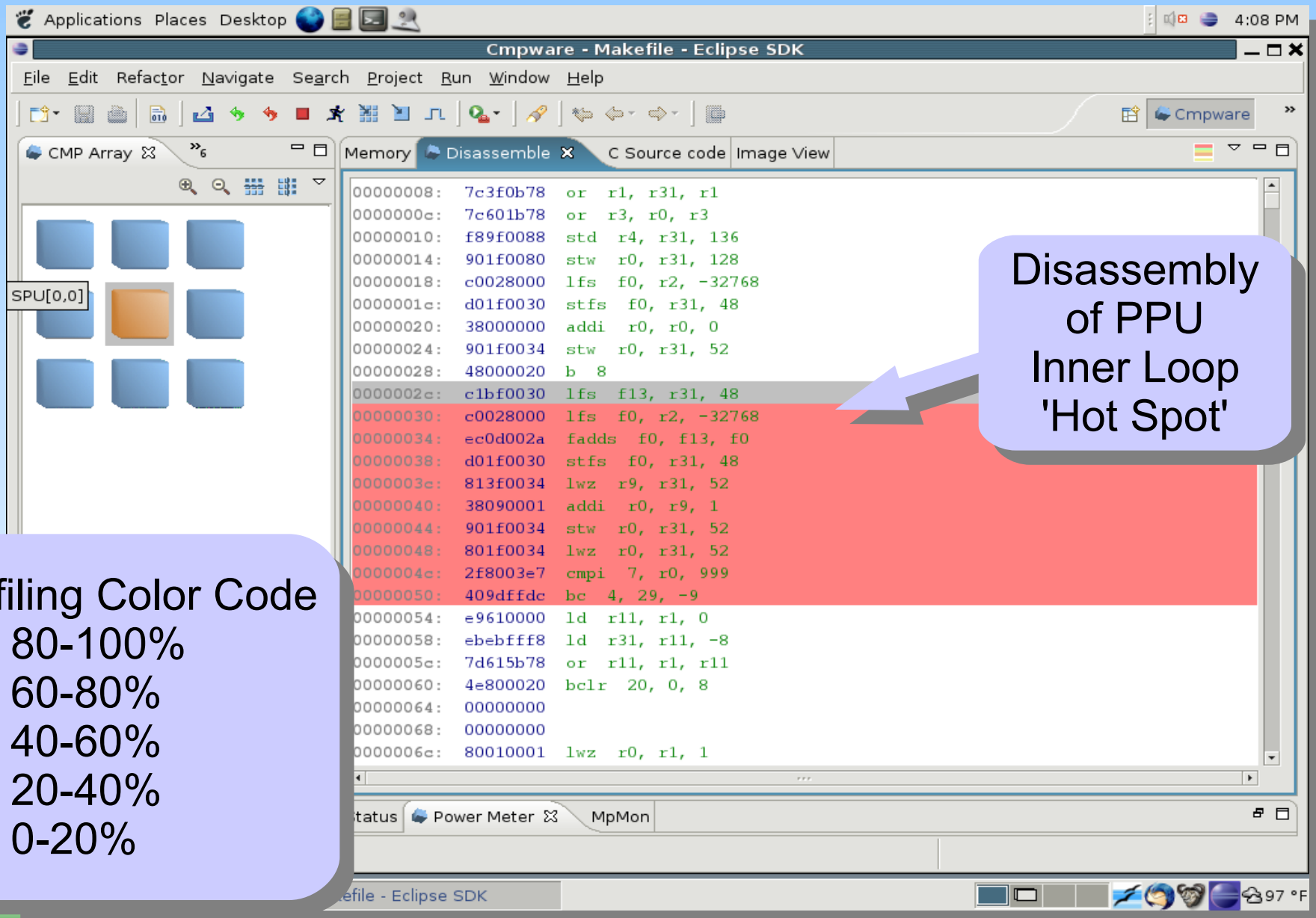
Application I: Source Profiling



Profiling Color Code

- 80-100%
- 60-80%
- 40-60%
- 20-40%
- 0-20%

Application I: Assembly Profiling

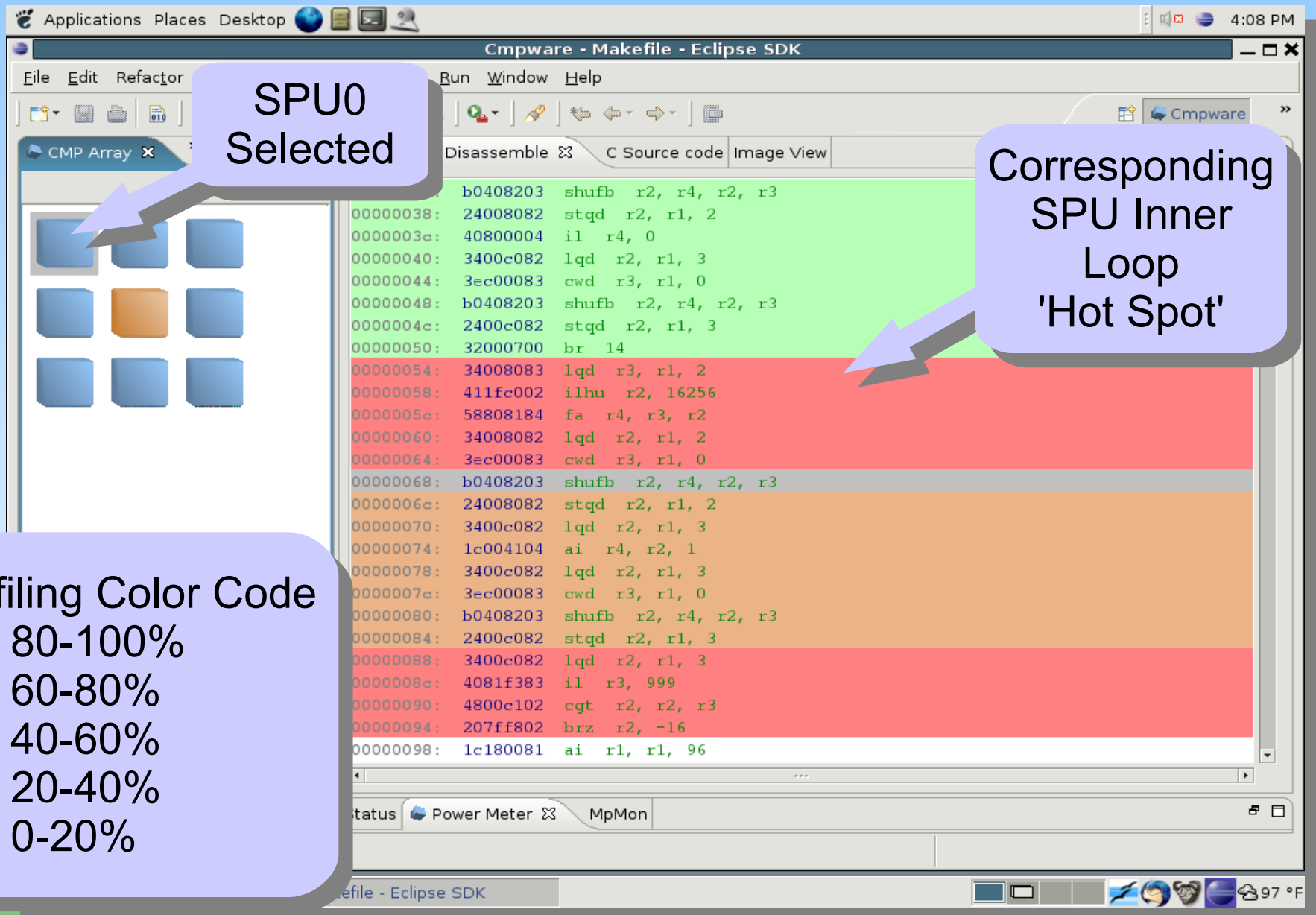


Disassembly of PPU Inner Loop 'Hot Spot'

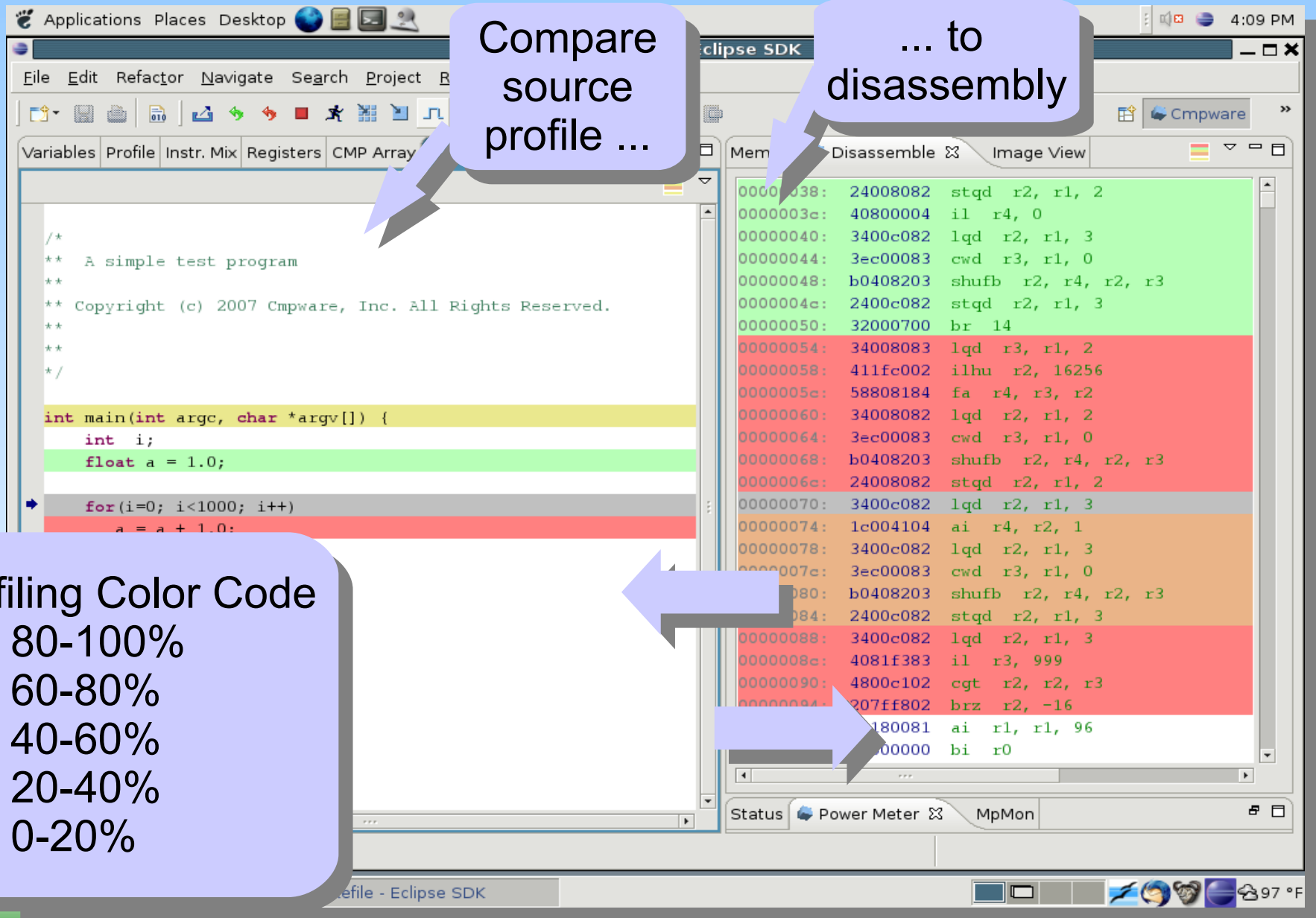
Profiling Color Code

- 80-100%
- 60-80%
- 40-60%
- 20-40%
- 0-20%

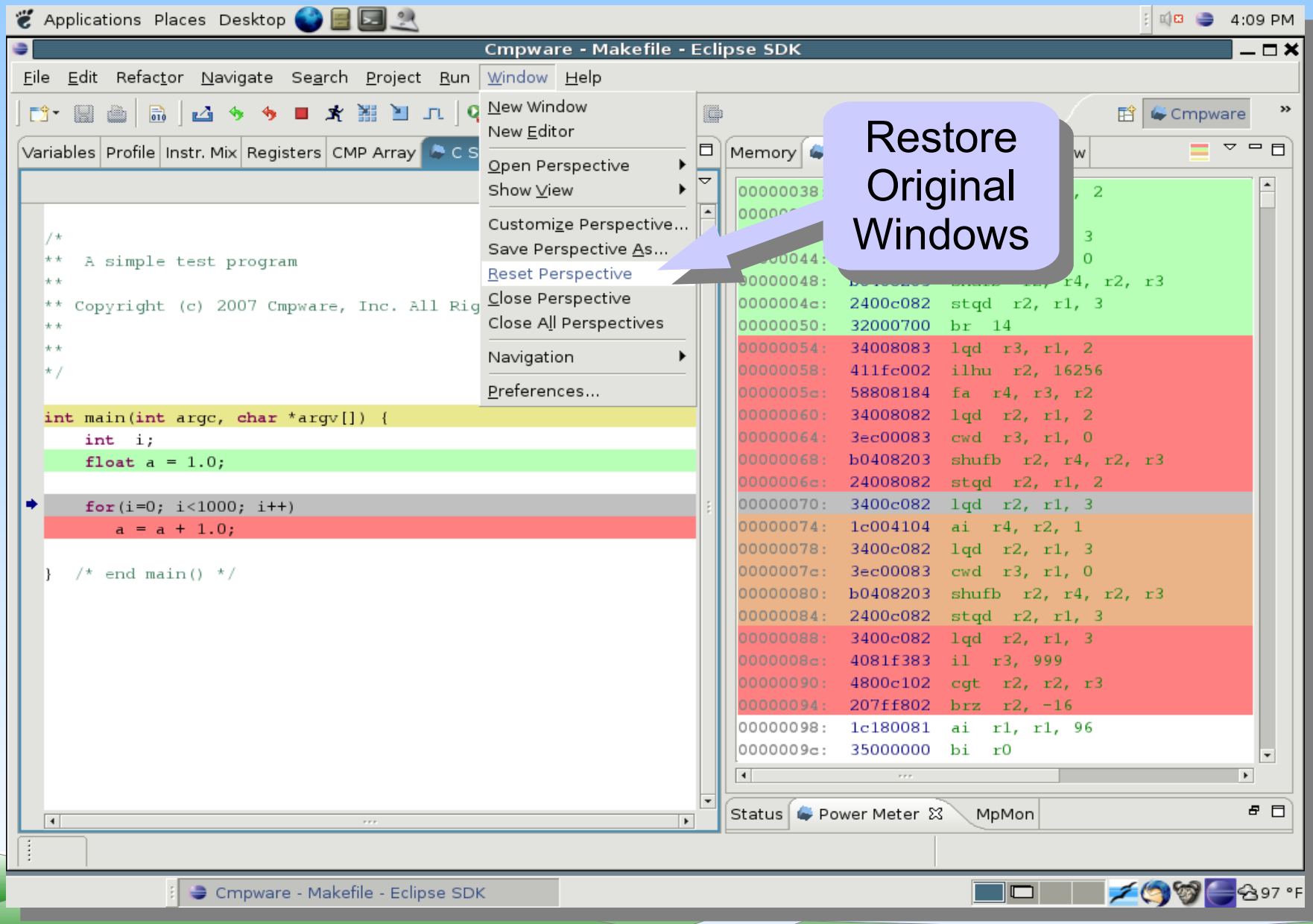
Application I: Assembly Profiling



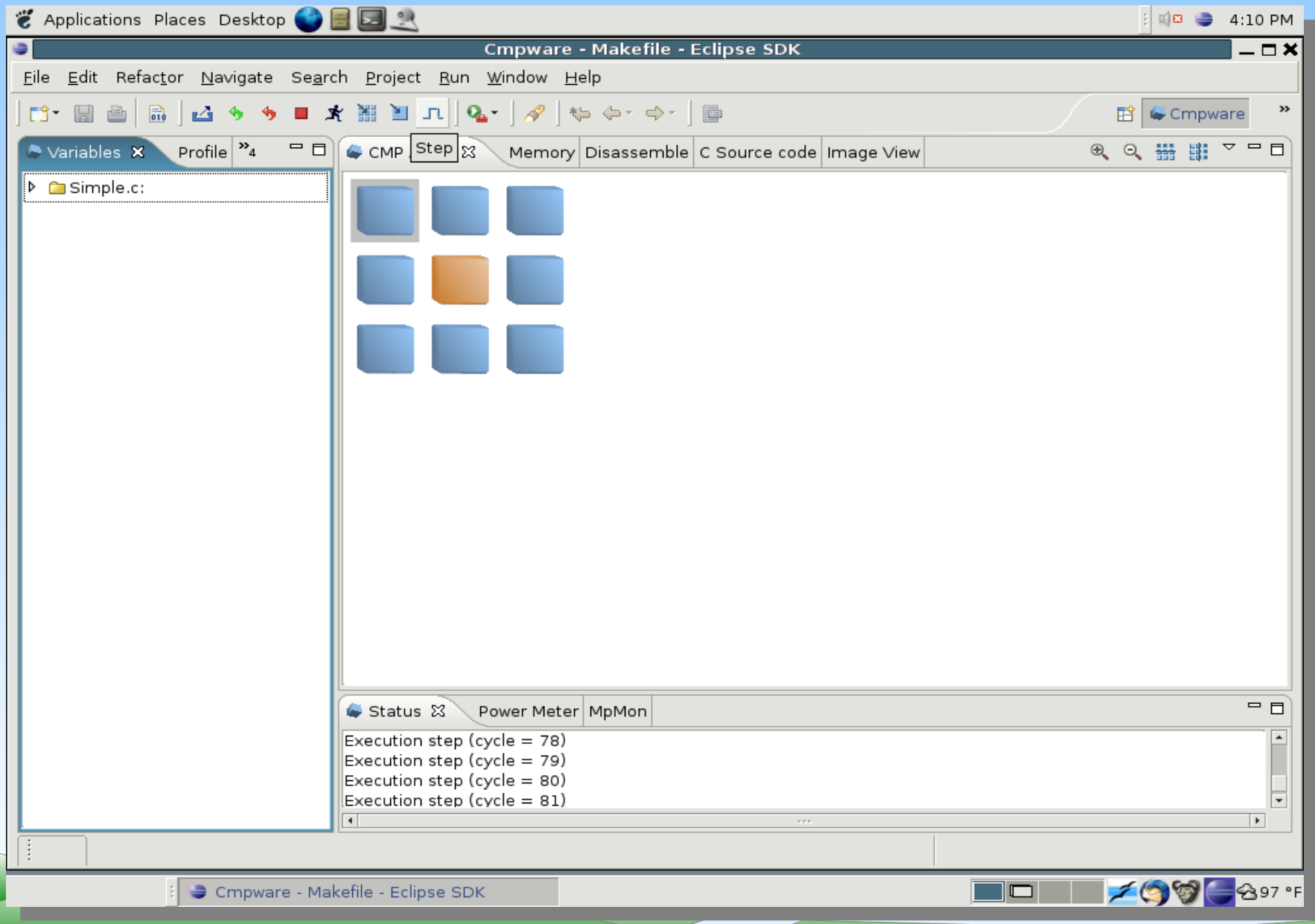
Application I: Moving Windows

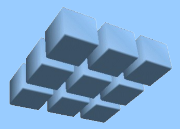


Application I: Reset Perspective



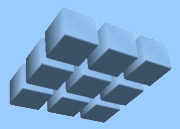
Application I: Default Perspective





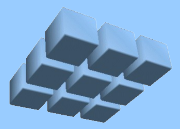
Application I: Overview

- Introduces the *Cmpware CMP-DK*
- Loading and execution of *Cell BE* code
- Interactive graphical profiler
- Data displays (memory, registers, etc.)
- Code displays (source, disassembly)
- Changing views (moving, resizing)
- Interacting with other *Eclipse* plugins (CDT)



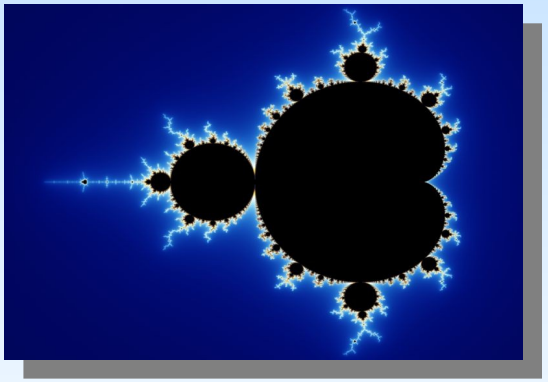
I. Introduction
II. Installing the *Cmpware CMP-DK*
III. Application I: A Simple Program
- **IV. Application II: Shared Memory**

V. Application III: Mailboxes
VI. Optimization I: SPU Dependencies
VII. Optimization II: SPU Dual Issue
VIII. System Level Analysis
IX. Overview

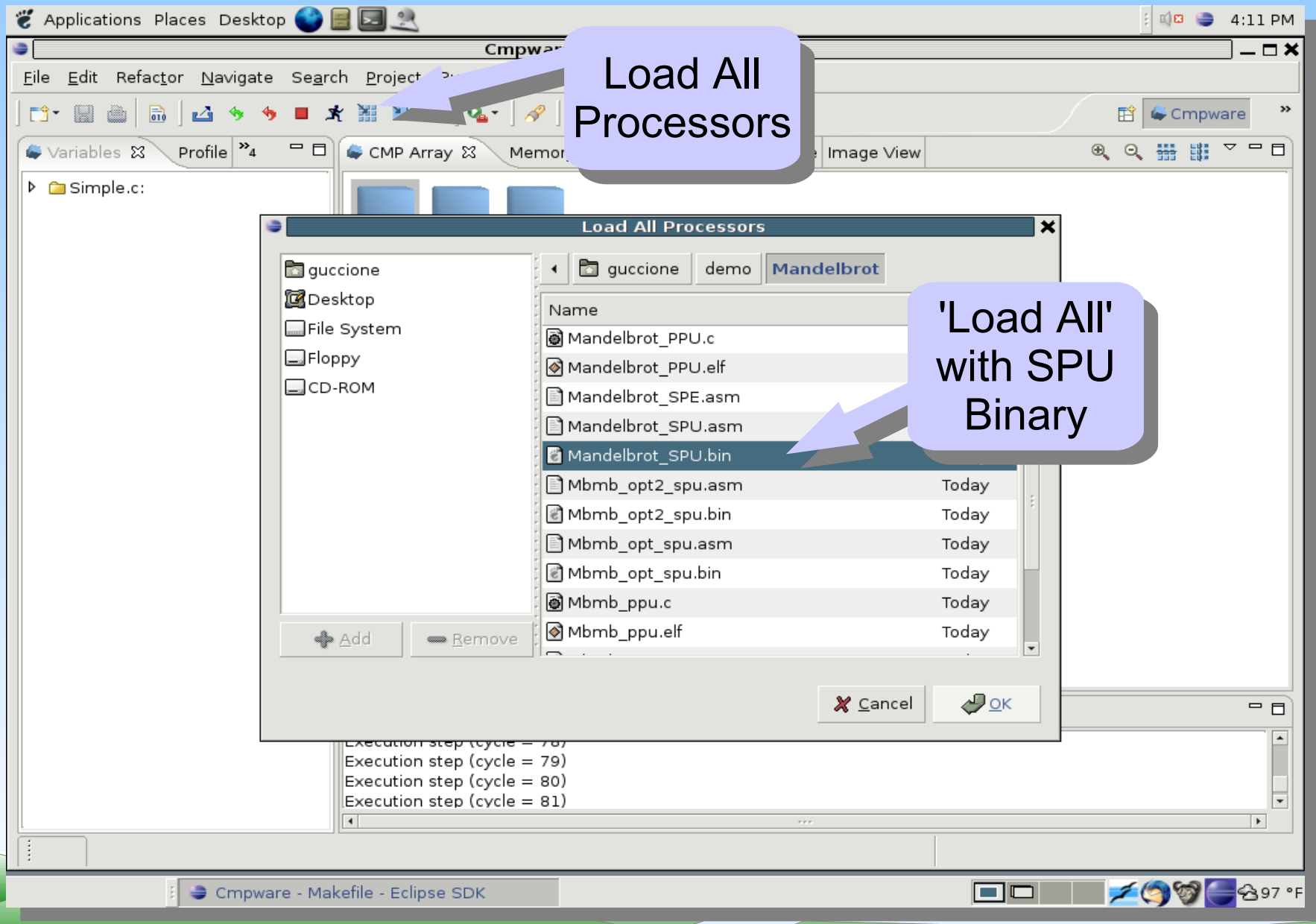


Application II: Shared Memory

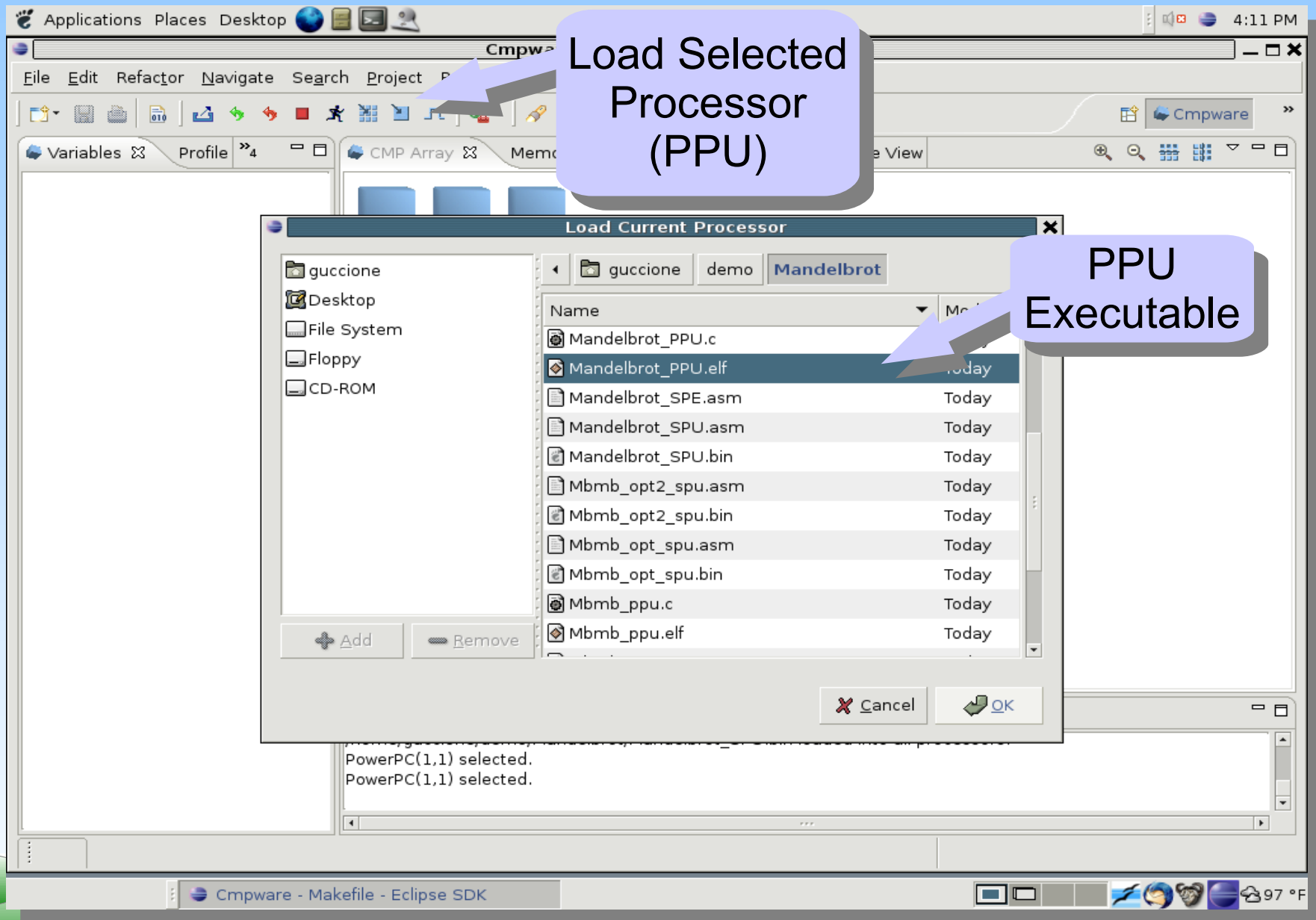
- Application II: the Mandelbrot Set
- All communication via shared memory
- Floating point intensive, highly parallel
- Demonstrates *Cell BE* architecture
- PPU controls SPUs
- All work done on SPUs
- SPUs coded in assembly



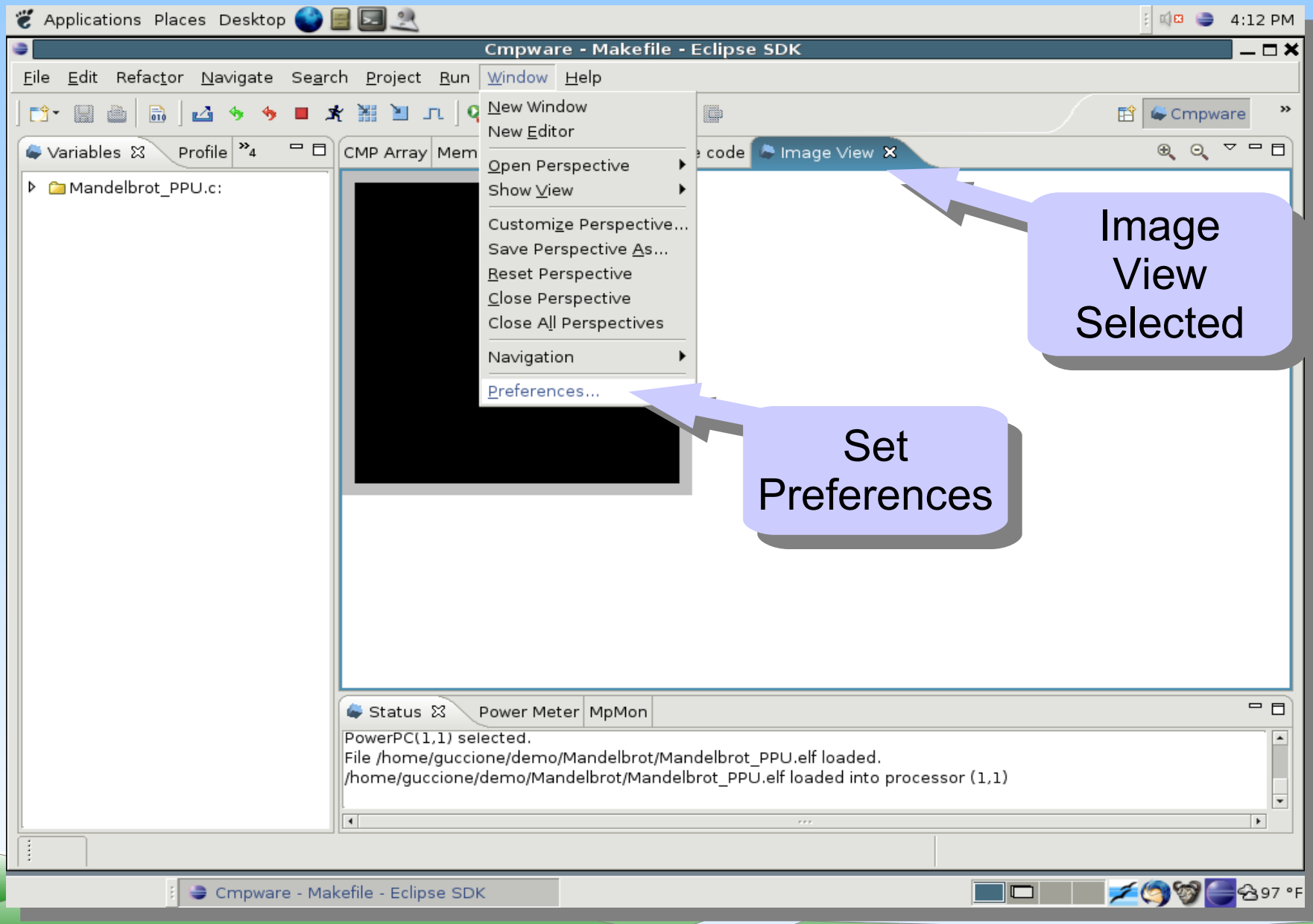
Application II: Load SPU Code



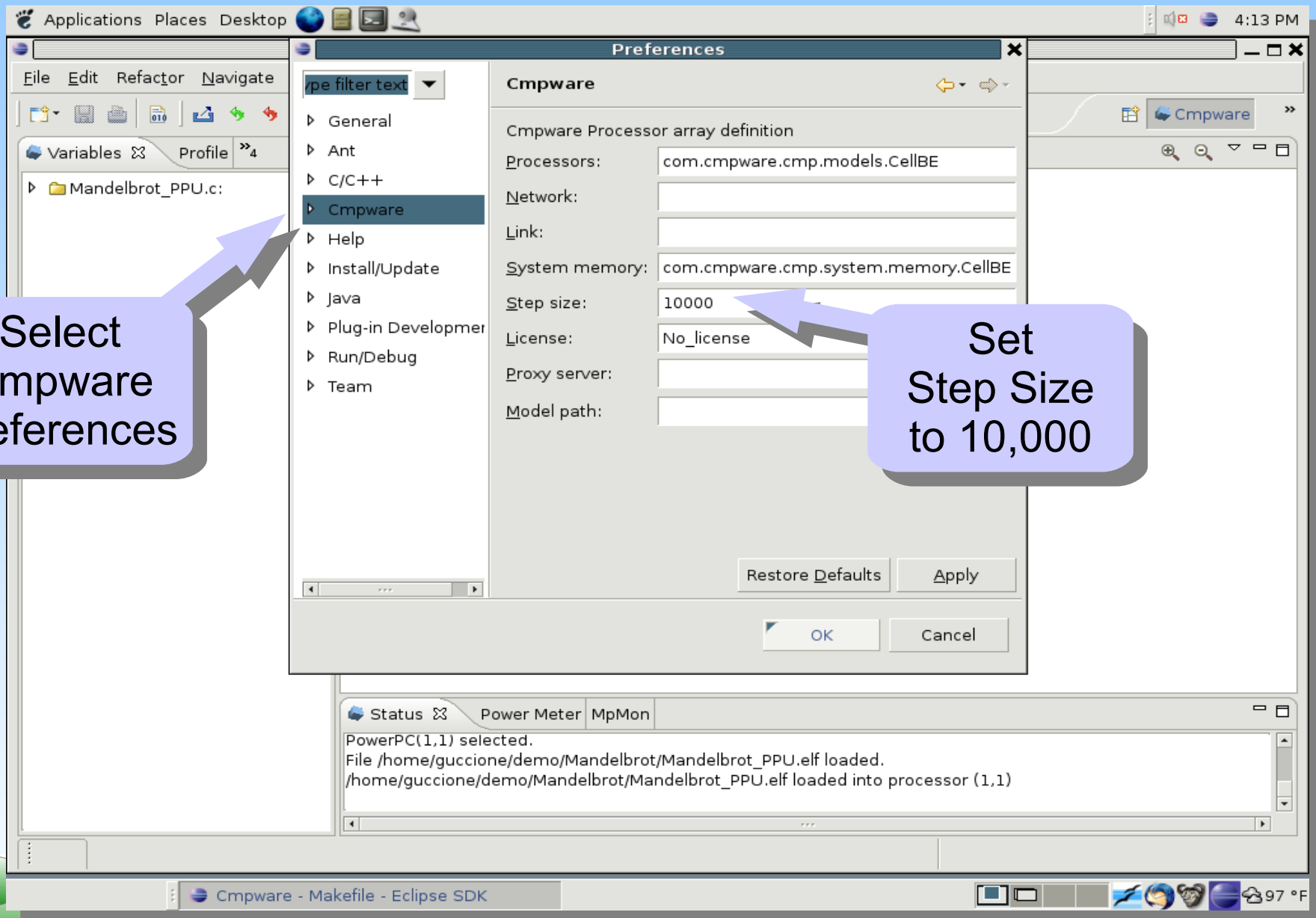
Application II: Load PPU Code



Application II: Set Preferences



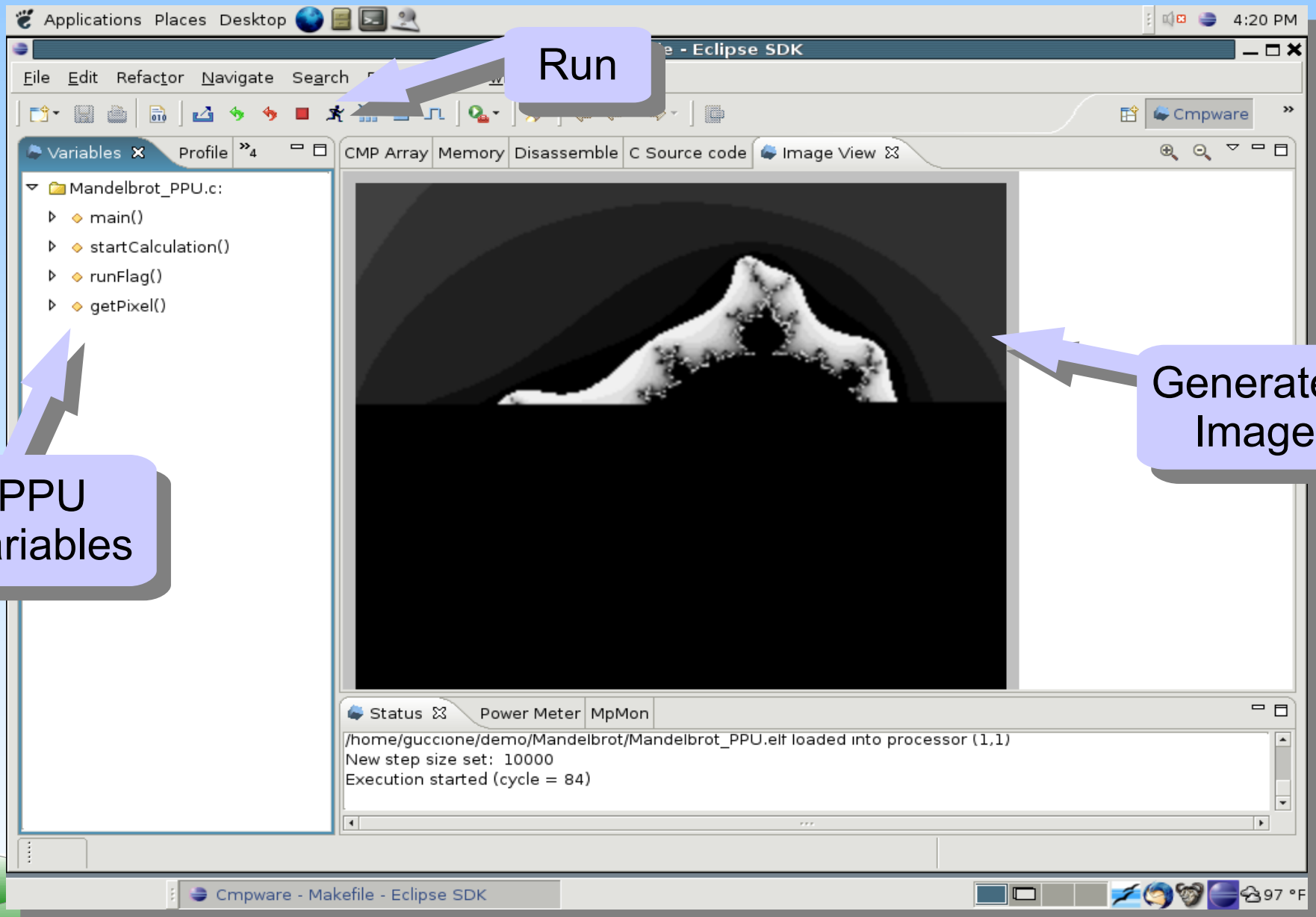
Application II: Set Step Size



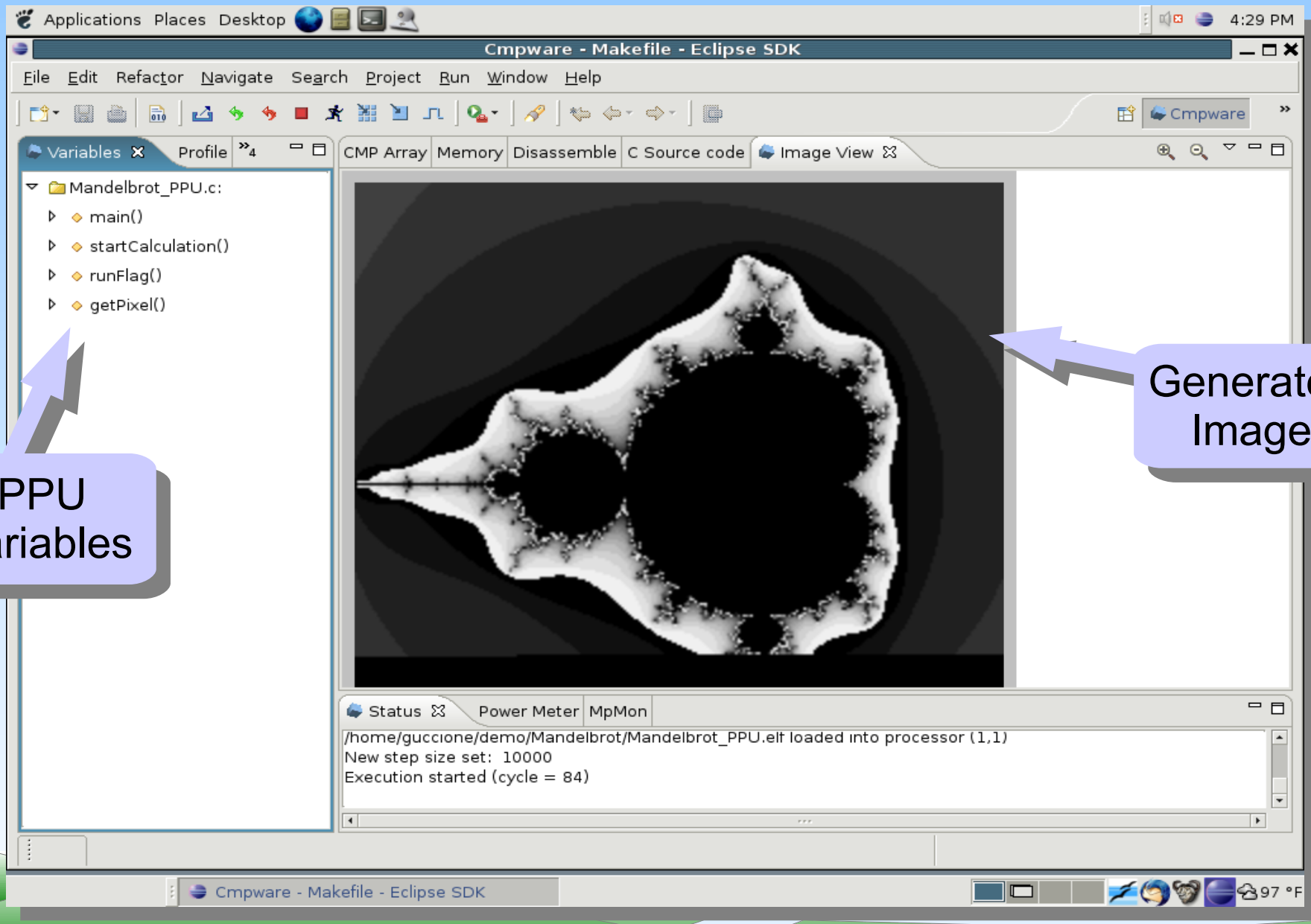
Select Cmpware Preferences

Set Step Size to 10,000

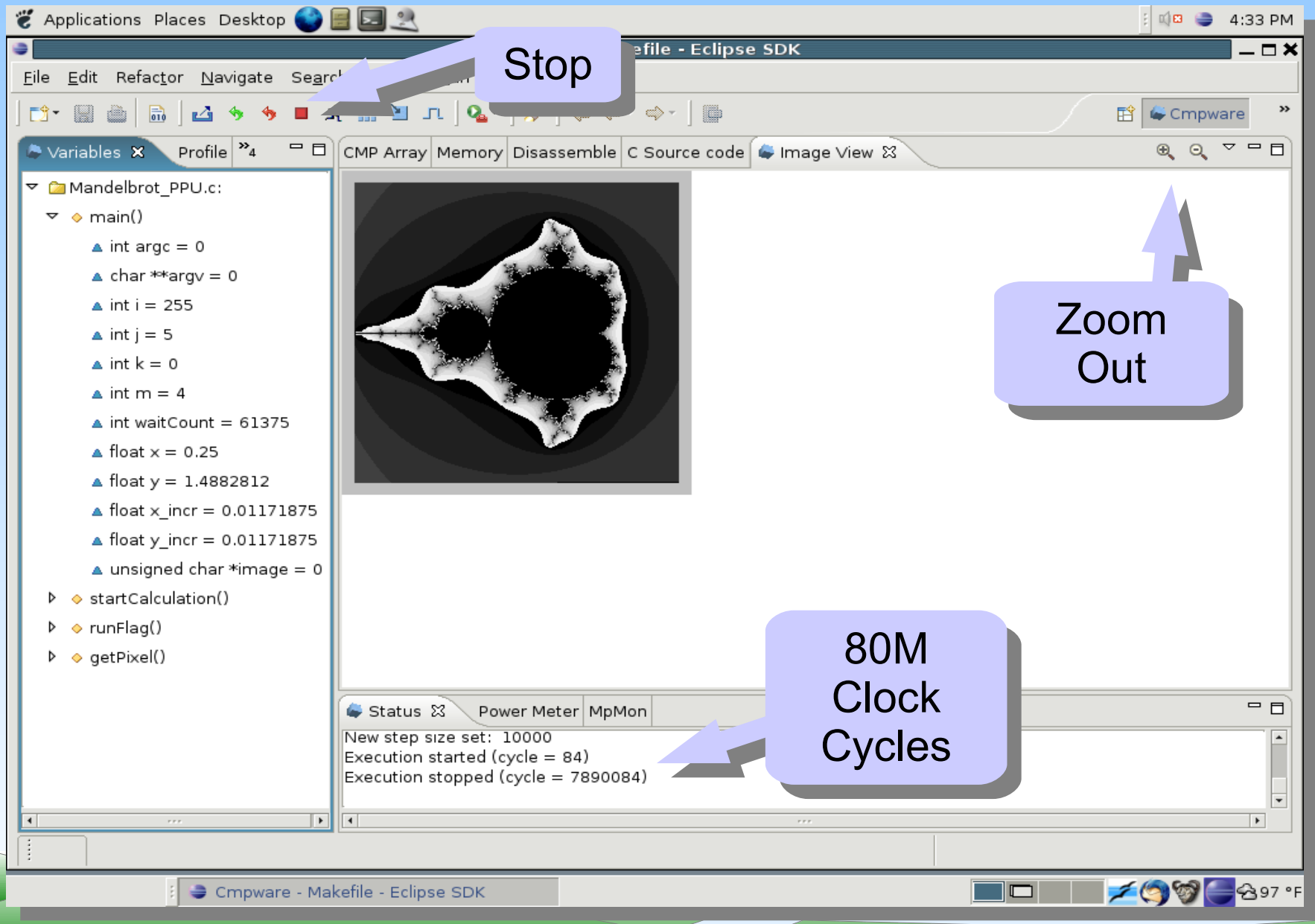
Application II: Running the Code

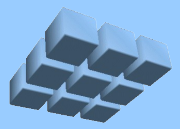


Application II: Running the Code



Application II: Stopping Execution

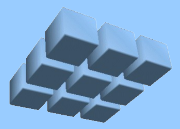




SPU Assembly Code

```
--
--  This is the inner loop of the
--  Mandelbrot algorithm for the
--  CellBE SPU.  It is used to
--  generate the data used by
--  Mandlebrot_PPE.c
--
--  Copyright (c) 2007 Cmpware, Inc.
--   All Rights Reserved.
--


-- Useful constants
#define  zero    r0
#define  one     r1

-- The (shared memory) parameters
#define  flag     r8
#define  x        r9
#define  y       r10
#define  cutoff  r11
#define  imax    r12
#define  pixel   r13

-- Other variables
#define  params  r14
#define  tmp0    r15
#define  tmp1    r16
#define  tmp2    r17
#define  tmp3    r18
```

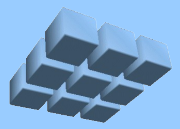
```
#define  z_re    r20
#define  z_im    r21
#define  c_re    r22
#define  c_im    r23
#define  done_mask  r24
#define  icount     r25

-- Initialize constants
il    zero, 0
il    one, 1
il    done_mask, 0
il    icount, 0
il    params, 0x1000

-- Wait for the 'go' flag
lqx   flag, params, zero
brz   flag, -1

-- Load parameters
il    tmp0, 16
lqx   x, params, tmp0
il    tmp0, 32
lqx   y, params, tmp0
il    tmp0, 48
lqx   cutoff, params, tmp0
il    tmp0, 64
lqx   imax, params, tmp0
```



SPU Assembly Code (cont.)

```
--  Load Z and C initial values
a     z_re, zero, zero
a     z_im, zero, zero
a     c_re, x, zero
a     c_im, y, zero

--  z^2 (re):  (z.re * z.re) - (z.im * z.im)
fm    tmp1, z_re, z_re
fm    tmp2, z_im, z_im
fs    tmp3, tmp1, tmp2

--     z^2 (im):  (z.re * z.im) +
--                 (z.re * z.im)
fm    tmp1, z_re, z_im
fa    z_im, tmp1, tmp1
fa    z_re, tmp3, zero

--  z = z^2 + c
fa    z_re, z_re, c_re
fa    z_im, z_im, c_im

--  Is ((z.re^2) + (z.im^2)) > cutoff
fm    tmp2, z_re, z_re
fm    tmp3, z_im, z_im
fa    tmp2, tmp3, tmp2
fcgt  tmp3, tmp2, cutoff
```

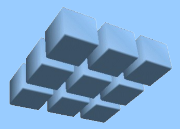
```
--  Increment iteration count for values
--  still less than cutoff
or    done_mask, done_mask, tmp3
and   tmp2, done_mask, one
a     icount, icount, tmp2

--  imax = imax - 1
sf    imax, one, imax
brnz  imax, -16

-- Copy results to shared memory
il    tmp0, 80
stqx  icount, params, tmp0

-- Set 'ready' flag
stqx  zero, params, zero

-- Go back to start
-- (and wait for another request)
bra   0
nop
nop
```



PPU 'C' Code

```
/*
**  This program is used to generate
**  the Mandelbrot set.  For higher
**  resolution, increase the ITERATIONS
**  to several thousand.  This will greatly
**  increase the computation time, but is
**  necessary for zooming in to higher
**  resolutions.  The other constants control
**  the region of the Mandlebrot set being
**  mapped.
**
**  This file is the PPU (PowerPC) portion
**  of the code which runs on the PPU unit
**  of the Cell BE.  It uses shared memory
**  to drive the computation taking place
**  on the SPEs.  The SPEs each calculate
**  one set of four pixels in the Mandlebrot
**  set and return the data via shared memory.
**
**  Copyright (c) 2007 Cmpware, Inc.
**  All rights reserved.
*/

#include  "CellBE.h"

/*  Functions */
unsigned char  getPixel(int  spe,
                        int  pixelNum);
int  runFlag(int spe);
void  startCalculation(int  spe, float x,
            float y, float x_incr);
```

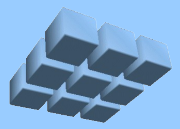
```
/*
**  Constants
*/

#define  ITERATIONS  100  // Default:   100
#define  CUTOFF       5.0  // Default:   5.0
#define  X_START     -2.0  // Default:  -2.0
#define  X_END        1.0  // Default:   1.0
#define  Y_START     -1.5  // Default:  -1.5
#define  Y_END        1.5  // Default:   1.5
#define  X_PIXELS    256
#define  Y_PIXELS    256

#define  SPU_READY   0
#define  SPU_BUSY    1

/* The parameters for the SPE (quadrwords) */
typedef struct {
   int    flag[4];   // The control flag
   float  x[4];      // X (c.re)
   float  y[4];      // Y (c.im)
   float  cutoff[4]; // 'cutoff'
   int    imax[4];   // Max iterations
   int    pixel[4];  // Pixel data
   } params;
```



PPU 'C' Code (cont.)

```
int
main(int  argc, char *argv[]) {
   int    i, j, k, m;
   int    waitCount = 0;
   float  x = X_START;
   float  y = Y_START;
   float  x_incr = (X_END - X_START) /
              ((float) X_PIXELS);
   float  y_incr = (Y_END - Y_START) /
               ((float) Y_PIXELS);
   unsigned char *image =
     (unsigned char *) 0x40000L;

   for (i=0; i<Y_PIXELS; i++) {
     for (j=0; j<(X_PIXELS/(SPUS*4)); j++) {

       /* Start calculations */
       for (k=0; k<SPUS; k++) {
          startCalculation(k, x, y, x_incr);
          x = x + (4 * x_incr);
          }  /* end for(k) */
```

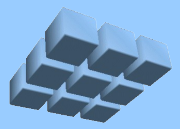
```
       /* Get pixel results */
       for (k=0; k<SPUS; k++) {
          while (runFlag(k) != SPU_READY)
             waitCount++;
             for (m=0; m<4; m++)
                *image++ = getPixel(k,m);
          }  /* end for(k) */

       }  /* end for(j) */

       x = X_START;
       y = y + y_incr;

       }  /* end for(i) */

for(;;) ;

}  /* end main() */
```



PPU 'C' Code (cont.)

```
/**
**  This function sends the required
**  parameters to the SPE and
**  sets the shared memory flag to
**  begin the computation.
**
**  @param  spu  the SPU number.
**          Should be 0 to 7.
**
**  @param  x  the x coordinate.
**          This is the real portion
*           of the complex number 'C'
**           used in the calculation.
**
**  @param  y  the y coordinate.
**          This is the imaginary
**          portion of the complex
**          number 'C' used in the
**          calculation.
**
**  @param  x_incr  the x coordinate
**          increment.  This is the amount
**          x is incremented per pixel.
**
*/
```

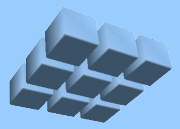
```
void  startCalculation(int  spu, float x,
        float y, float x_incr) {
   params *p = (params *) (long) (BP_BASE +
        (spu * SPU_MEMORY_RANGE) + 0x1000);

   p->x[0] = x;
   p->x[1] = x + x_incr;
   p->x[2] = x + (2 * x_incr);
   p->x[3] = x + (3 * x_incr);
   p->y[0] = y;
   p->y[1] = y;
   p->y[2] = y;
   p->y[3] = y;
   p->cutoff[0] = CUTOFF;
   p->cutoff[1] = CUTOFF;
   p->cutoff[2] = CUTOFF;
   p->cutoff[3] = CUTOFF;
   p->imax[0] = ITERATIONS;
   p->imax[1] = ITERATIONS;
   p->imax[2] = ITERATIONS;
   p->imax[3] = ITERATIONS;
   p->flag[1] = 0;
   p->flag[2] = 0;
   p->flag[3] = 0;
   p->flag[0] = SPU_BUSY;  // Start

   }  /* end startCalculation() */
```



PPU 'C' Code (cont.)

```
/**
**  This function returns the 'run'
**  flag from shared memory.  It
**  should have a value of WAIT,
**  GO or DONE.
**
**  @param  spu  the SPU number.
**          Should be 0 to 7.
**
**  @returns  the value of the SPE
**            'run' flag is returned.
**
*/

int runFlag(int spu) {
   params *p = (params *) (long) (BP_BASE +
        (spu * SPU_MEMORY_RANGE) + 0x1000);
   return (p->flag[0]);
   }  /* end runFlag() */
```

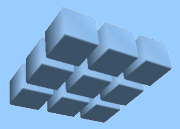
```
/**
**  This function gets a pixel
**  value from a PE.  There are
**  four pixels produced per
**  calculation.
**
**  @param  spu  the SPE number.
**          Should be 0 to 7.
**
**  @param  pixelNum  the pixel
**          number.  Should be 0 to 3.
**
**  @returns  A 32 bit value
**            representing the number
**            of iterations is
**            returned.  This value is
**            always less than or
**            equal to the 'cutoff'
**            parameter.
**
*/

unsigned char getPixel(int  spu,
   int  pixelNum) {
     unsigned char pixel;
   params *p = (params *) (long) (BP_BASE +
      (spu * SPU_MEMORY_RANGE) + 0x1000);
   pixel = (unsigned char)
      ((p->pixel[pixelNum]& 0x0f) << 4);
   return (pixel);
   }  /* end getPixel() */
```



Application II: Overview

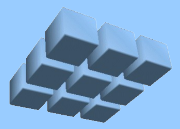
- Introduces PPU / SPU communication
- Demonstrates shared memory programming
- Introduces *Eclipse* 'Preferences'
- Demonstrates Image View

A Note on Synchronization: This communication scheme eliminates the need for 'semaphores'. This is not generally true in shared memory software.



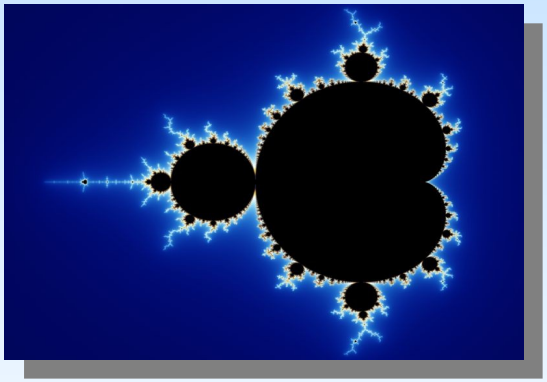
I. Introduction
II. Installing the *Cmpware CMP-DK*
III. Application I: A Simple Program
IV. Application II: Shared Memory
- **V. Application III: Mailboxes**

VI. Optimization I: SPU Dependencies
VII. Optimization II: SPU Dual Issue
VIII. System Level Analysis
IX. Overview

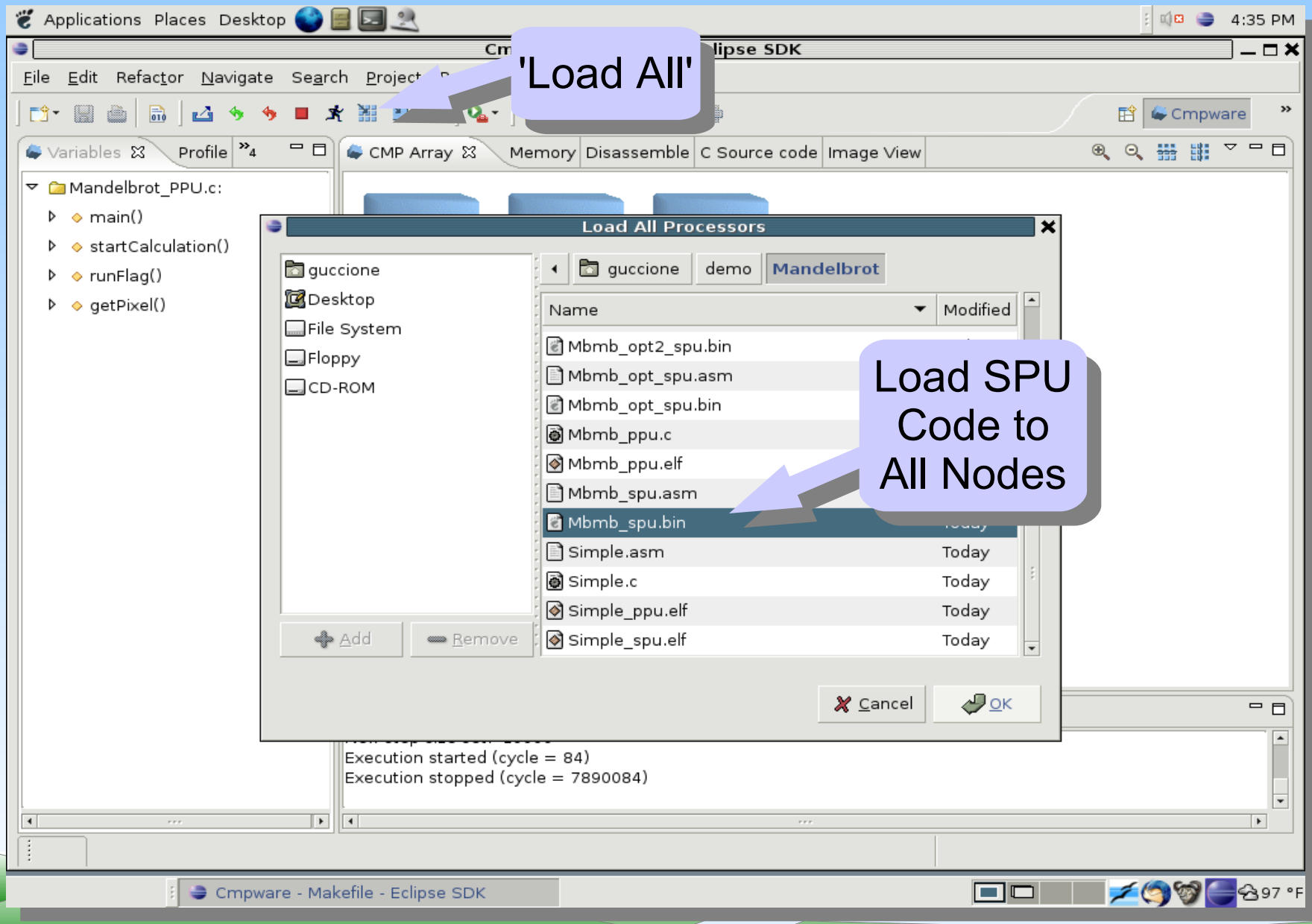


Application III: Mailboxes

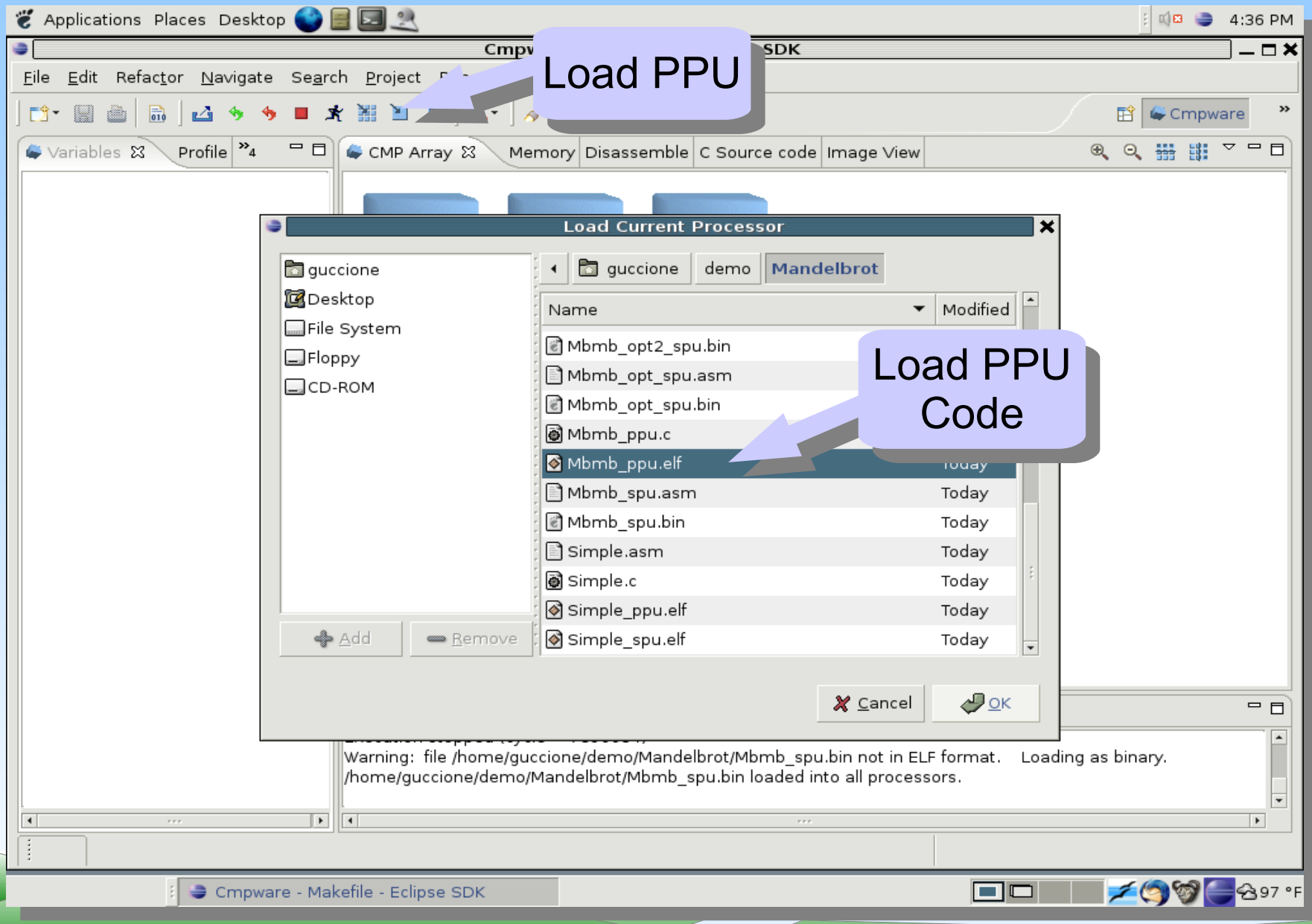
- SPU / PPU communication via 'mailboxes'
- Mandelbrot application converted from shared memory to mailboxes
- Mailboxes: 32-bit communication channels
  - Point to point
  - Synchronized
  - Blocking



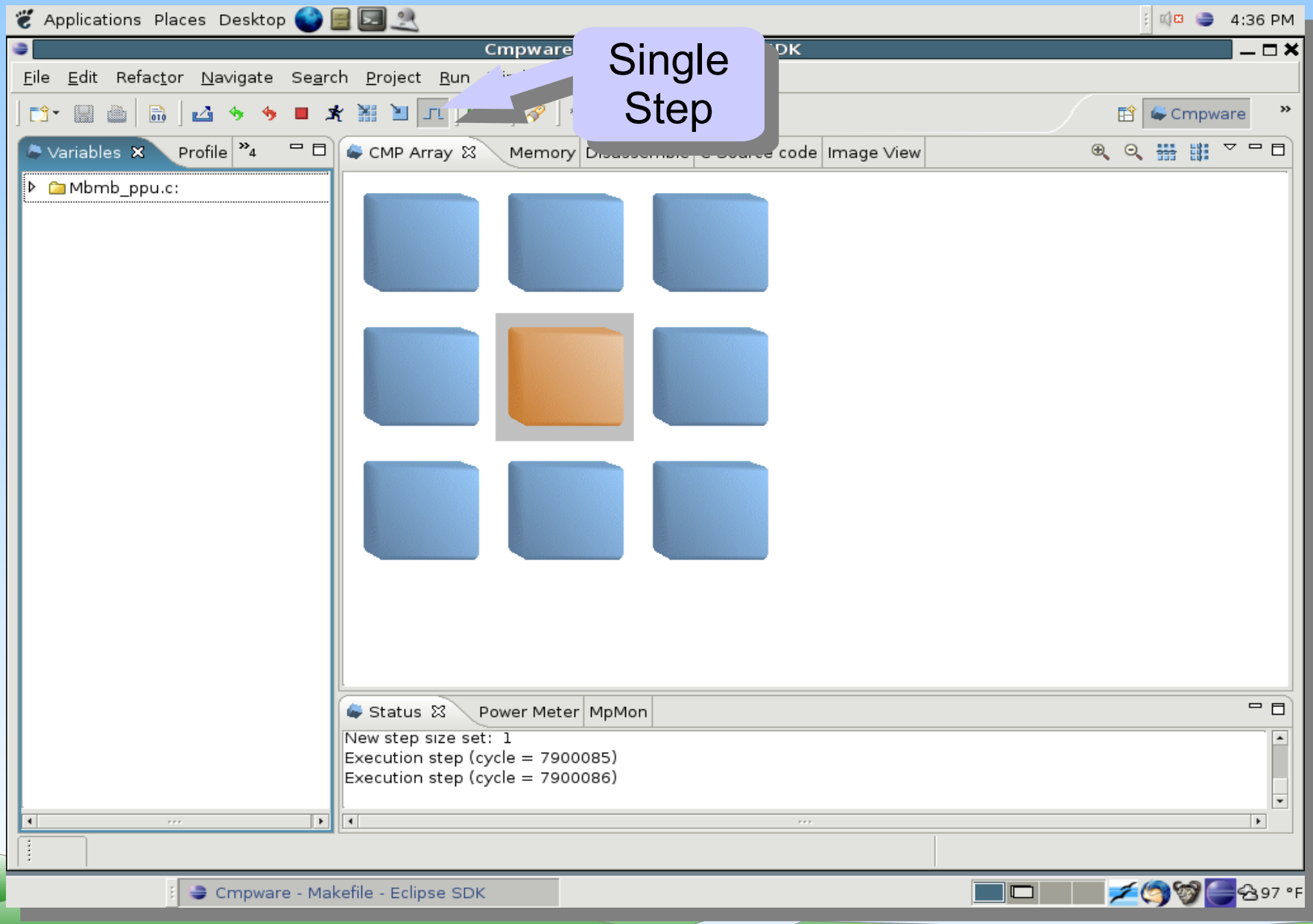
Application III: Load SPU code



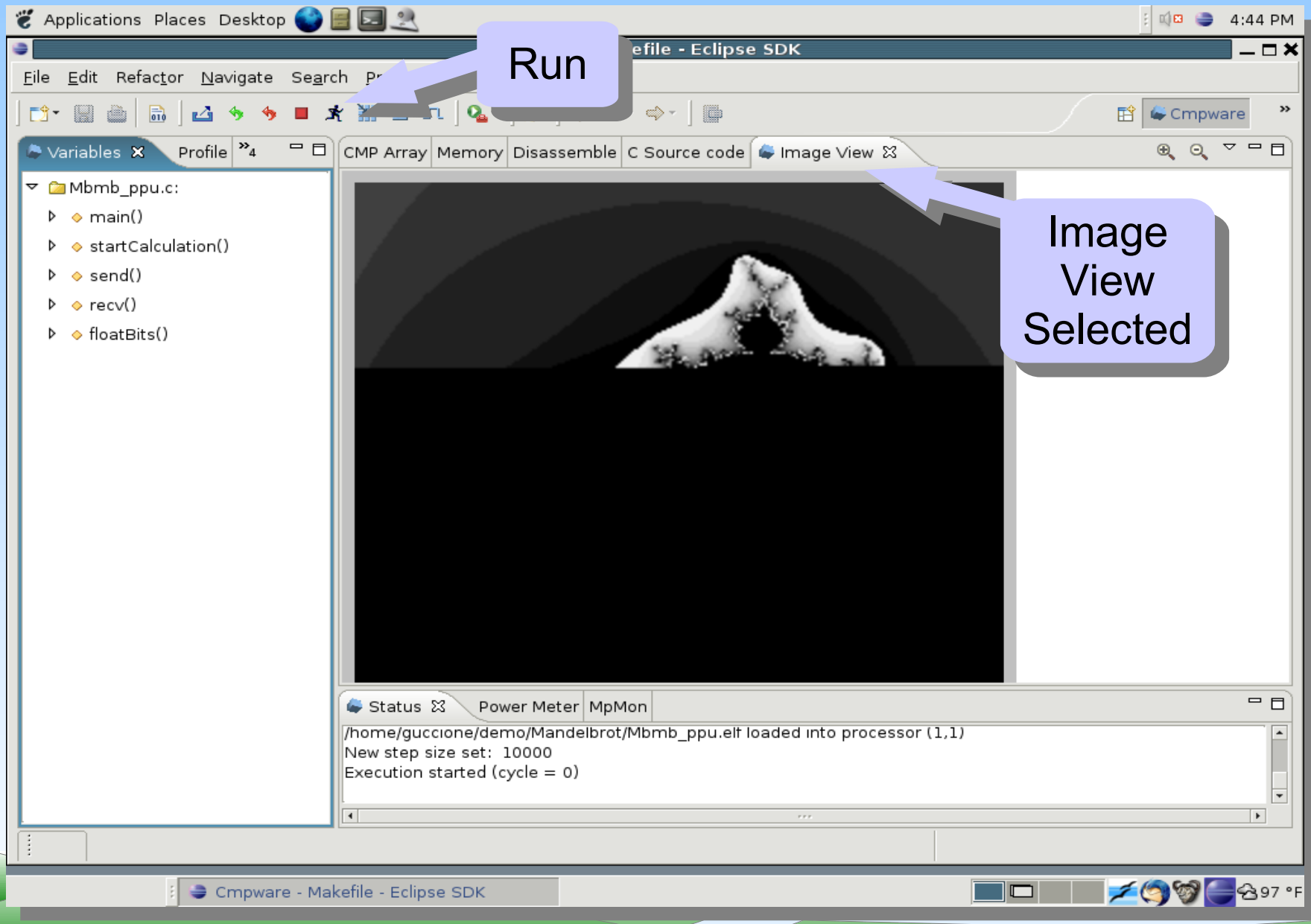
Application III: Load PPU code

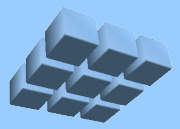


Application III: Executing Code



Application III: Executing Code

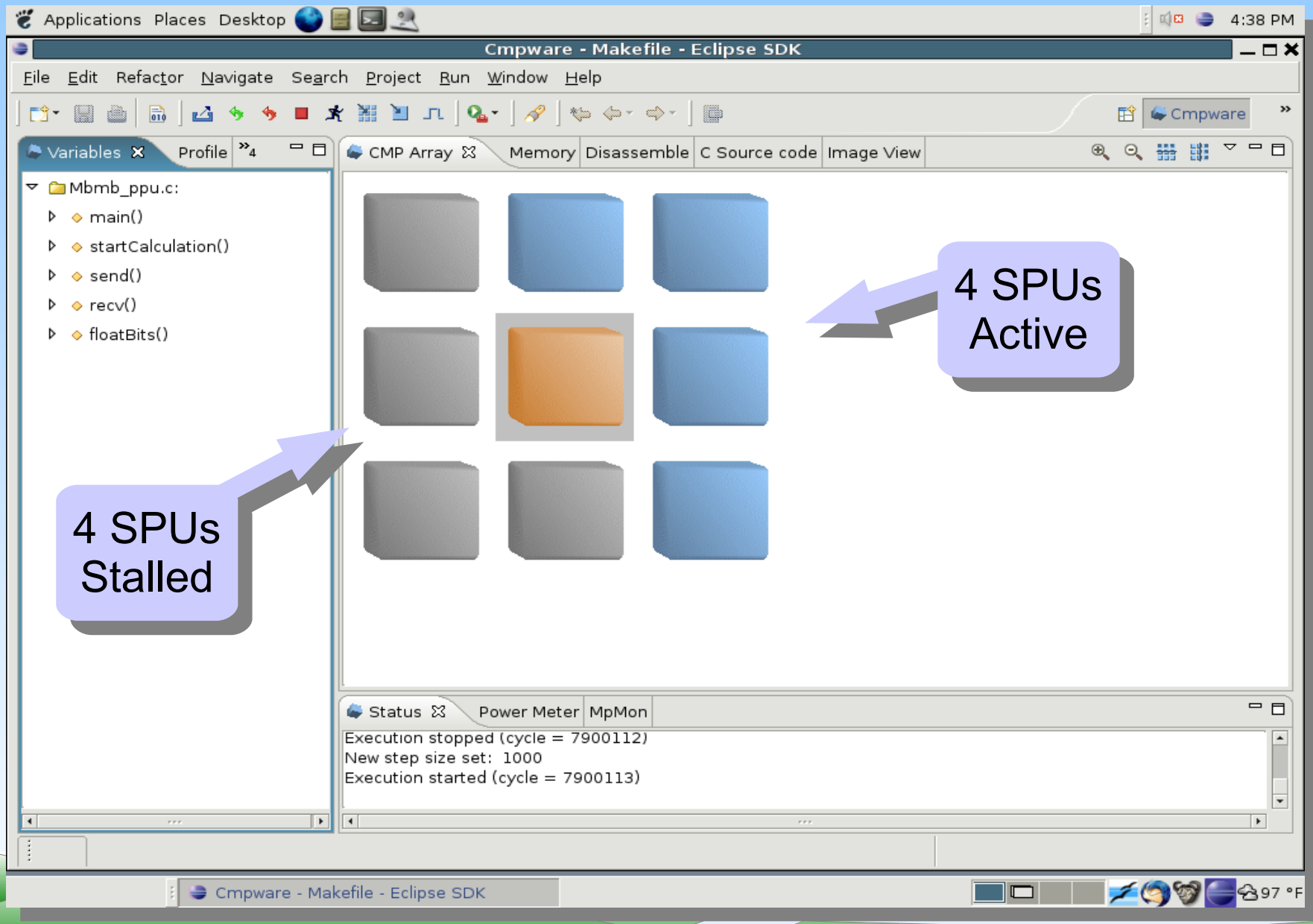


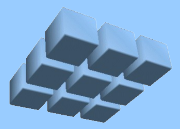


Application III: Stalling

- SPU mailboxes:
  - **`rdch`** and **`wrch`** instructions
  - Wait for data to become available ('stall')
- Alternative to 'busy waiting' in loop
- *Cmpware* detects stalls
  - Colors processors acording to activity
  - Tracks stalls in Utilization / Power Meter

Application III: SPU Stalls





Application III: SPU Stalls

Applications Places Desktop 4:38 PM

Cmpware - Makefile - Eclipse SDK

File Edit Refactor Navigate Search Project Run Window Help

Cmpware

Variables Profile

Mbmb_ppu.c:
- main()
- startCalculation()
- send()
- recv()
- floatBits()

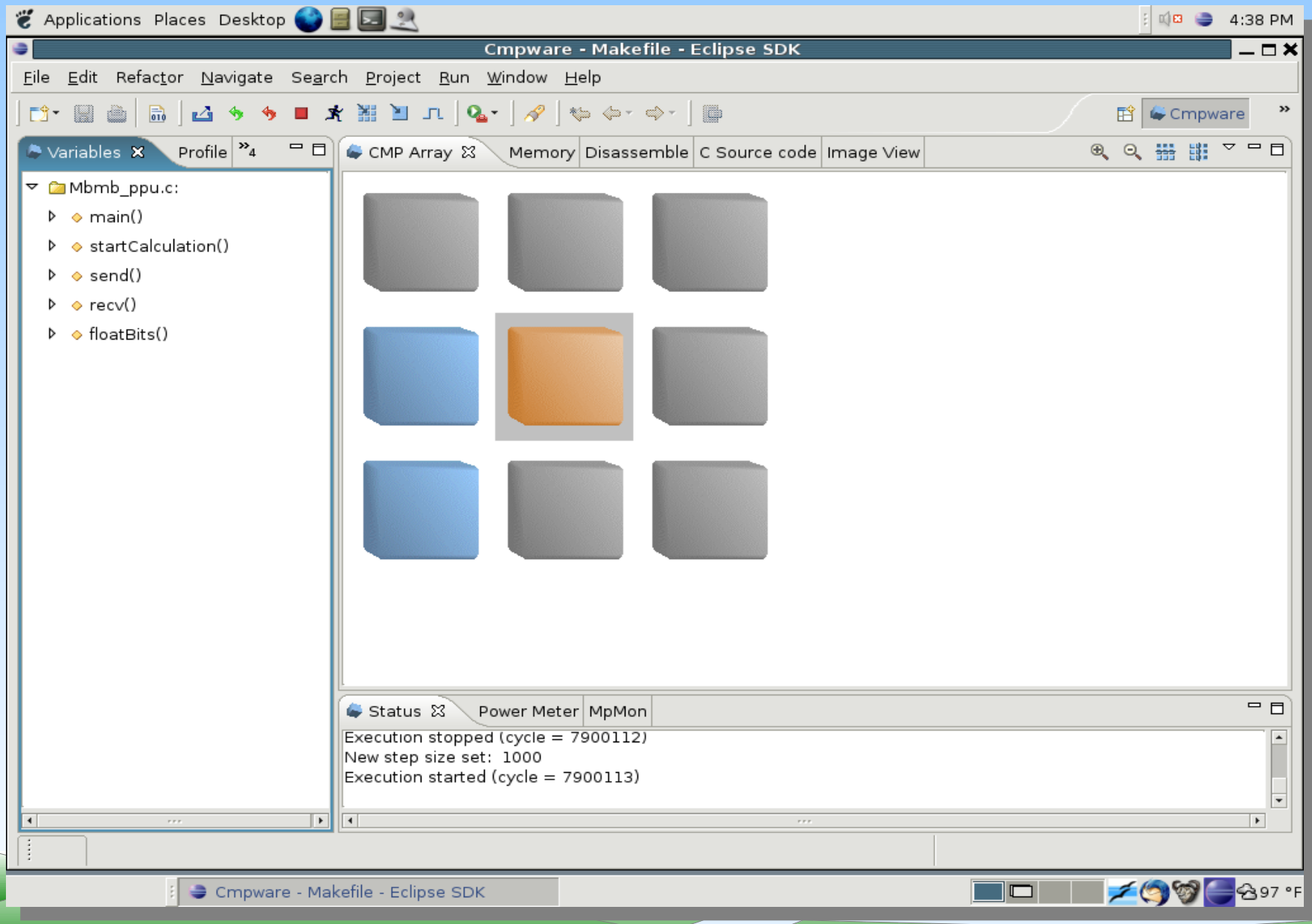
CMP Array Memory Disassemble C Source code Image View

Status Power Meter MpMon

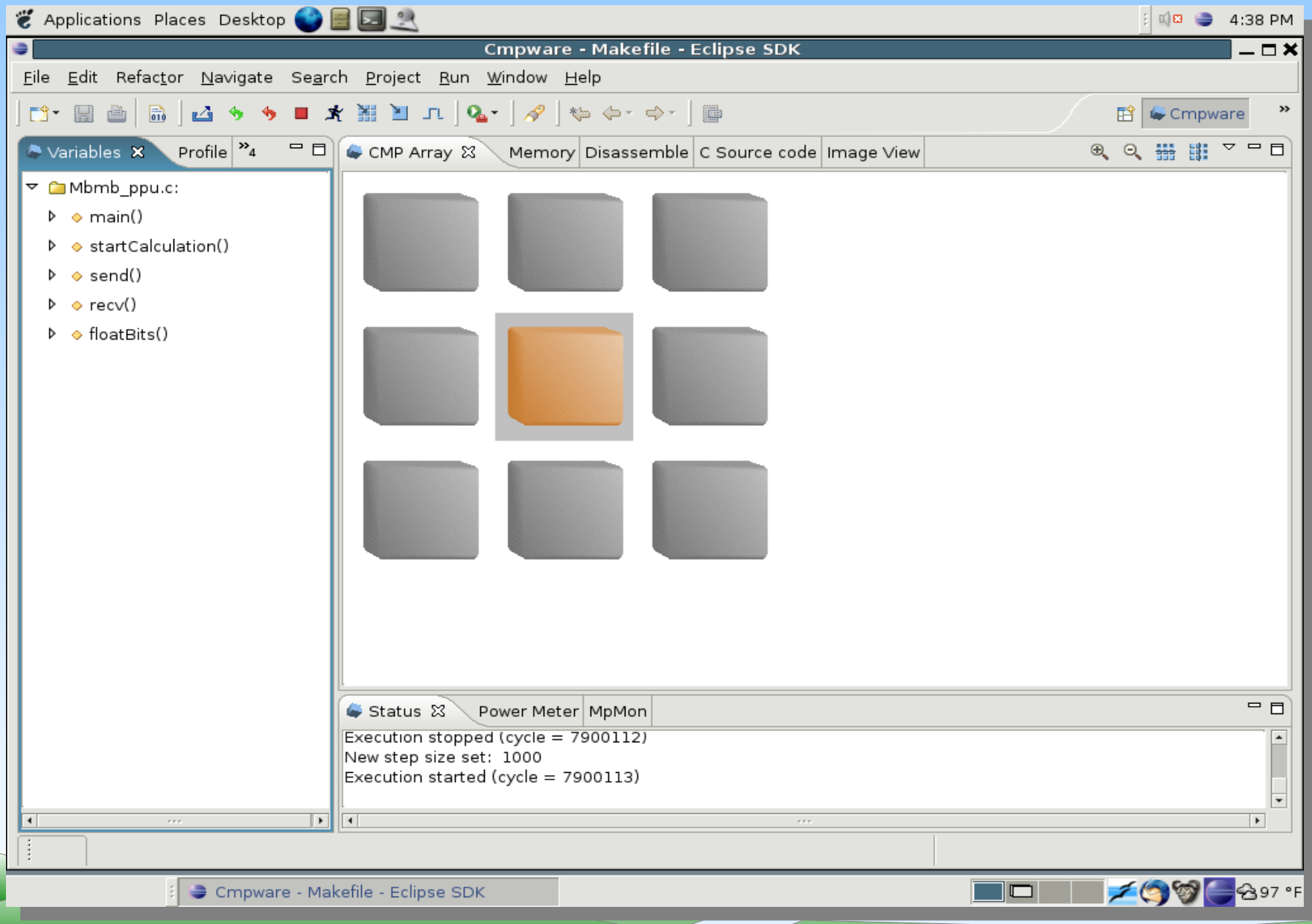
Execution stopped (cycle = 7900112)
New step size set: 1000
Execution started (cycle = 7900113)

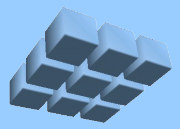
Cmpware - Makefile - Eclipse SDK 97 °F

Application III: SPU Stalls

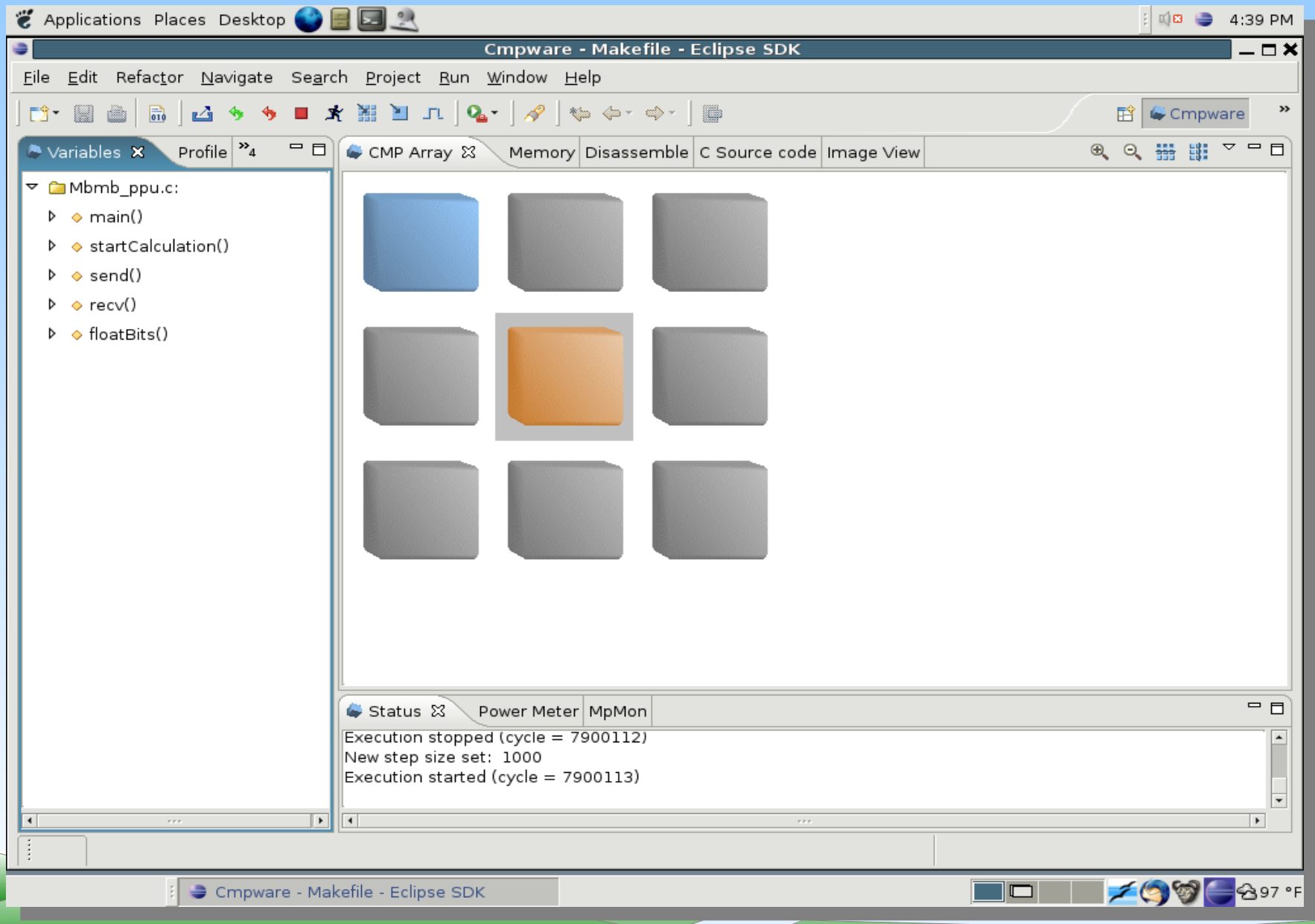


Application III: SPU Stalls

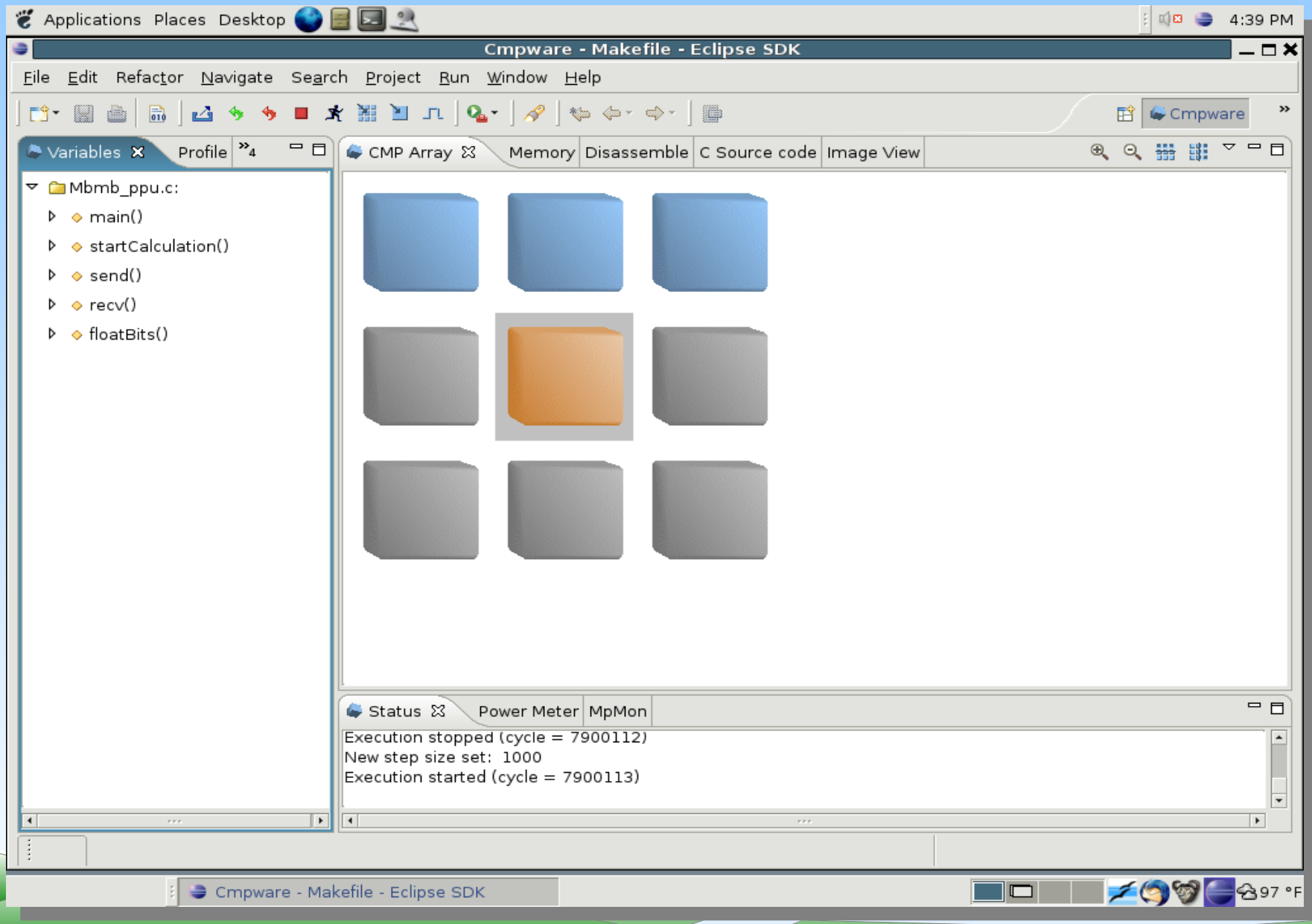


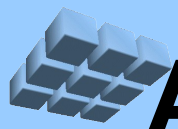


Application III: SPU Stalls



Application III: SPU Stalls

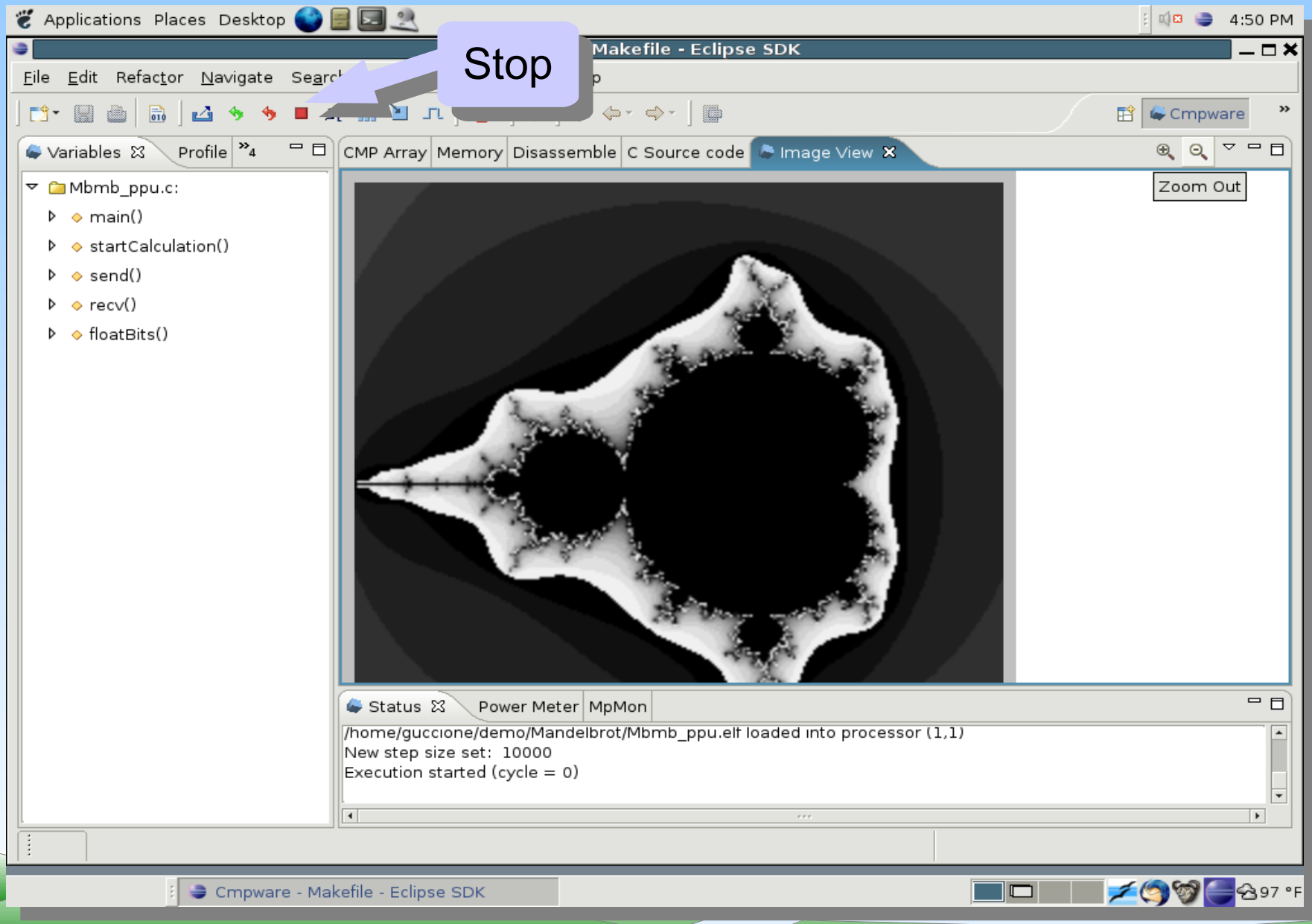


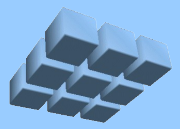


Application III: Execution Patterns

- Animated execution patterns in main display
- PPU overwhelmed 'feeding' 8 SPUs
- Approximately 30% utilization in SPUs
- Simulator defaults to one instruction per cycle
  - Verifies functional correctness
  - Gives good indication of bottlenecks
  - Identifies / animates execution patterns
  - Helps balance loads

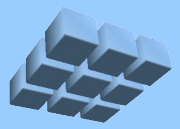
Application III: Executing Code





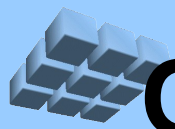
Application III: Overview

- Mailboxes provide synchronized, blocking, serialized communication channels
- Easier to debug and analyze
- *Cmpware* tools:
  - Identify channel stalls
  - Provide animated display data
  - Track processor utilization
  - Assist in load balancing and optimization



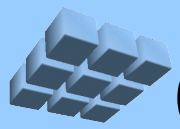
I. Introduction
II. Installing the *Cmpware CMP-DK*
III. Application I: A Simple Program
IV. Application II: Shared Memory
V. Application III: Mailboxes
- **VI. Optimization I: SPU Dependencies**

VII. Optimization II: SPU Dual Issue
VIII. System Level Analysis
IX. Overview



Optimization I: SPU Dependencies

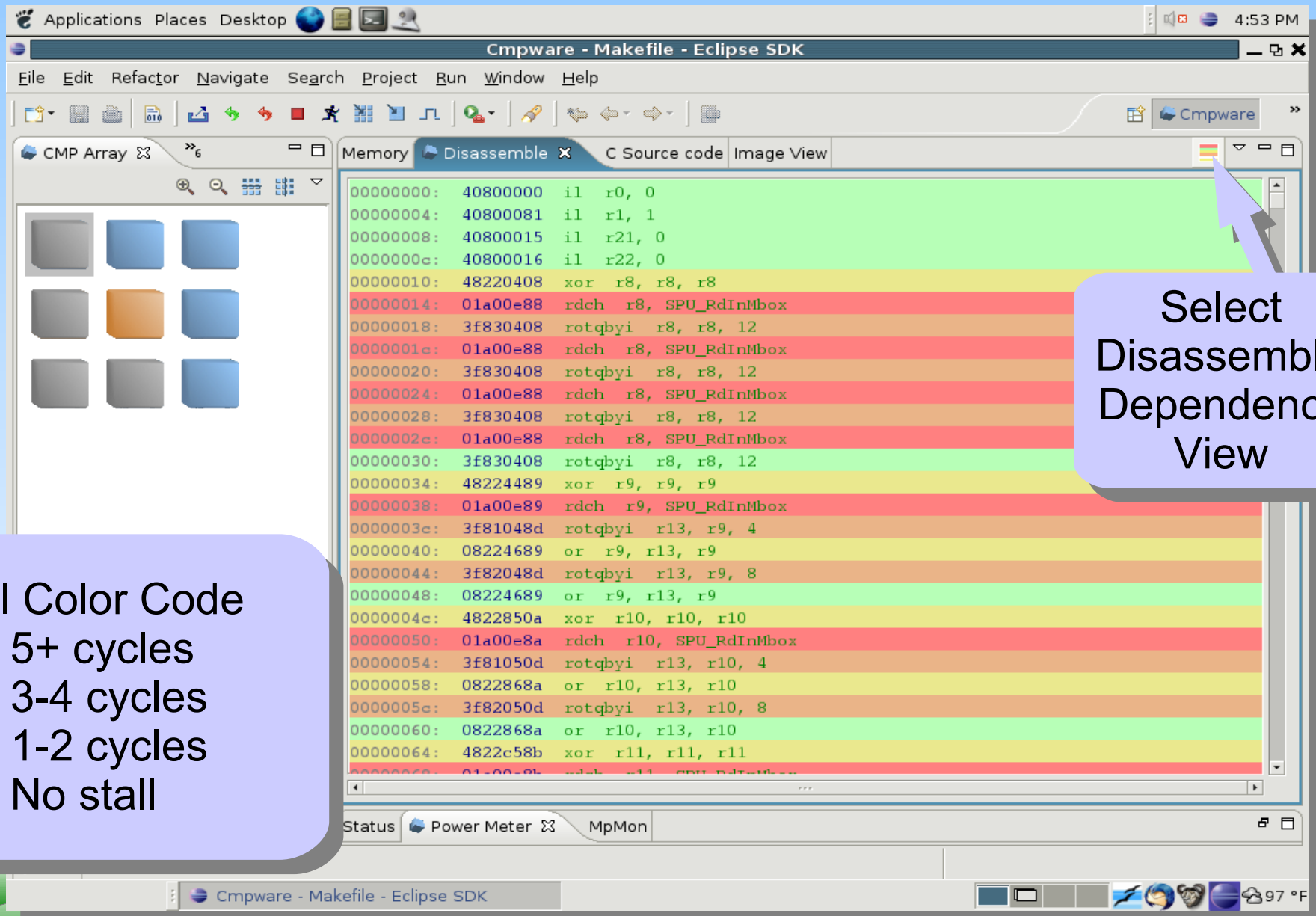
- Correctly executing hand-coded assembly is not the end of the road for SPU coding
- Performance issues depend heavily on the ordering of instructions ('scheduling')
- Results not immediately available to downstream instructions
- Instructions can wait from 2 to 6 cycles for data (longer for double precision)
- Performance can be **6x** worse than expected



Optimization I: Dependency View

- SPU Instruction scheduling is complex
  - Different instructions take different number of cycles – difficult to remember
  - Keeping track of register usage tedious
- *Cmpware* SPU Dependency View:
  - A quick 'snapshot' of SPU schedule
  - Intuitive color-coding of instructions
  - Interactive support for code scheduling

Optimization I: SPU Stalls

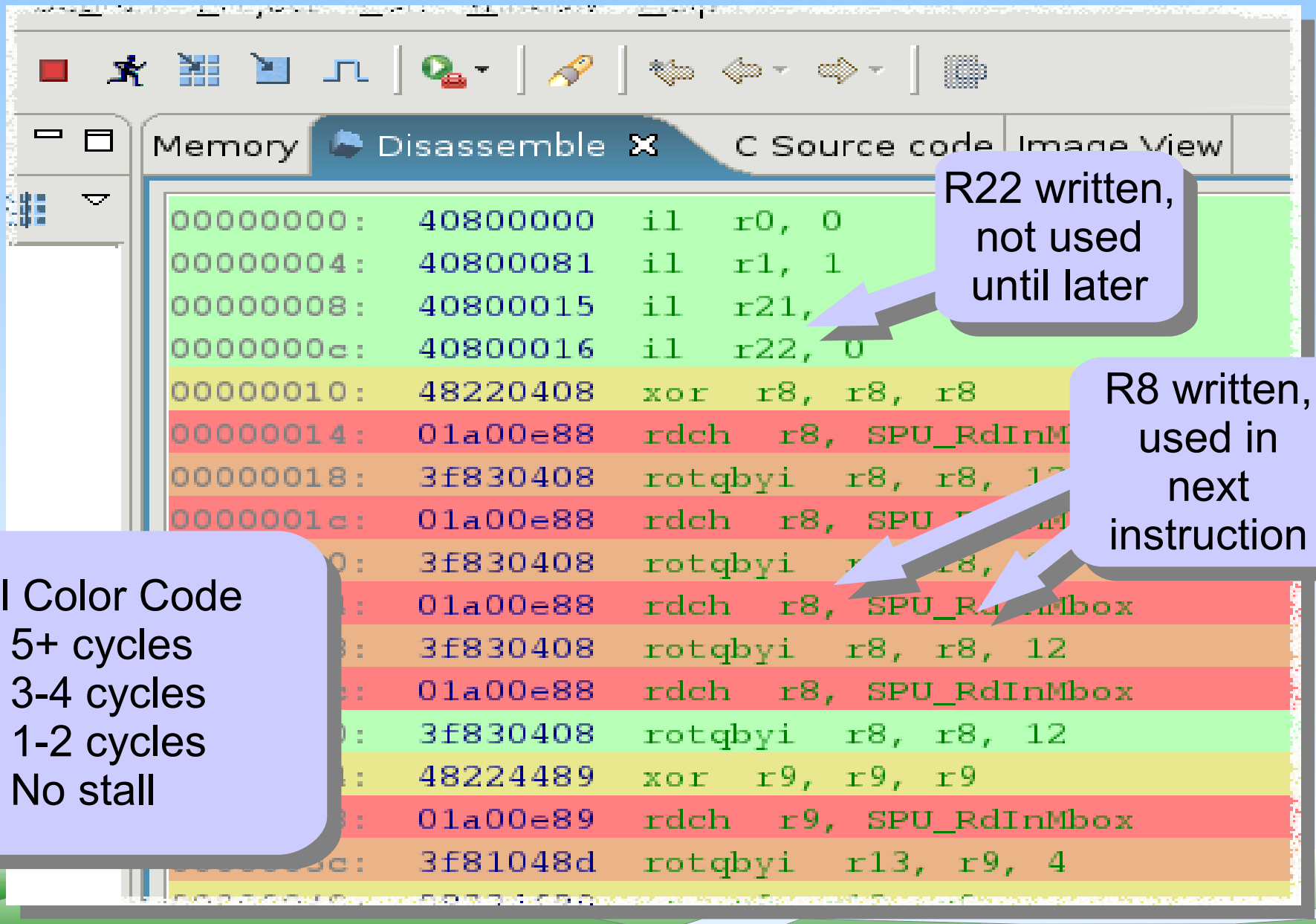


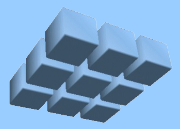
Select Disassemble Dependency View

Stall Color Code

- 5+ cycles
- 3-4 cycles
- 1-2 cycles
- No stall

Optimization I: SPU Stalls





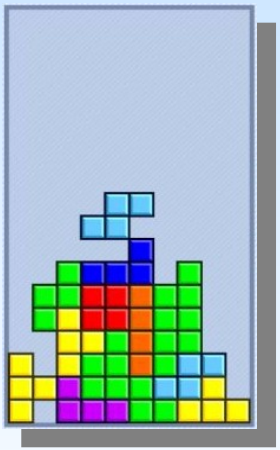
Optimization I: Statistics

- 60 instructions, 167 - 190+ cycles
- **2x - 3x** penalty for dependency stalls

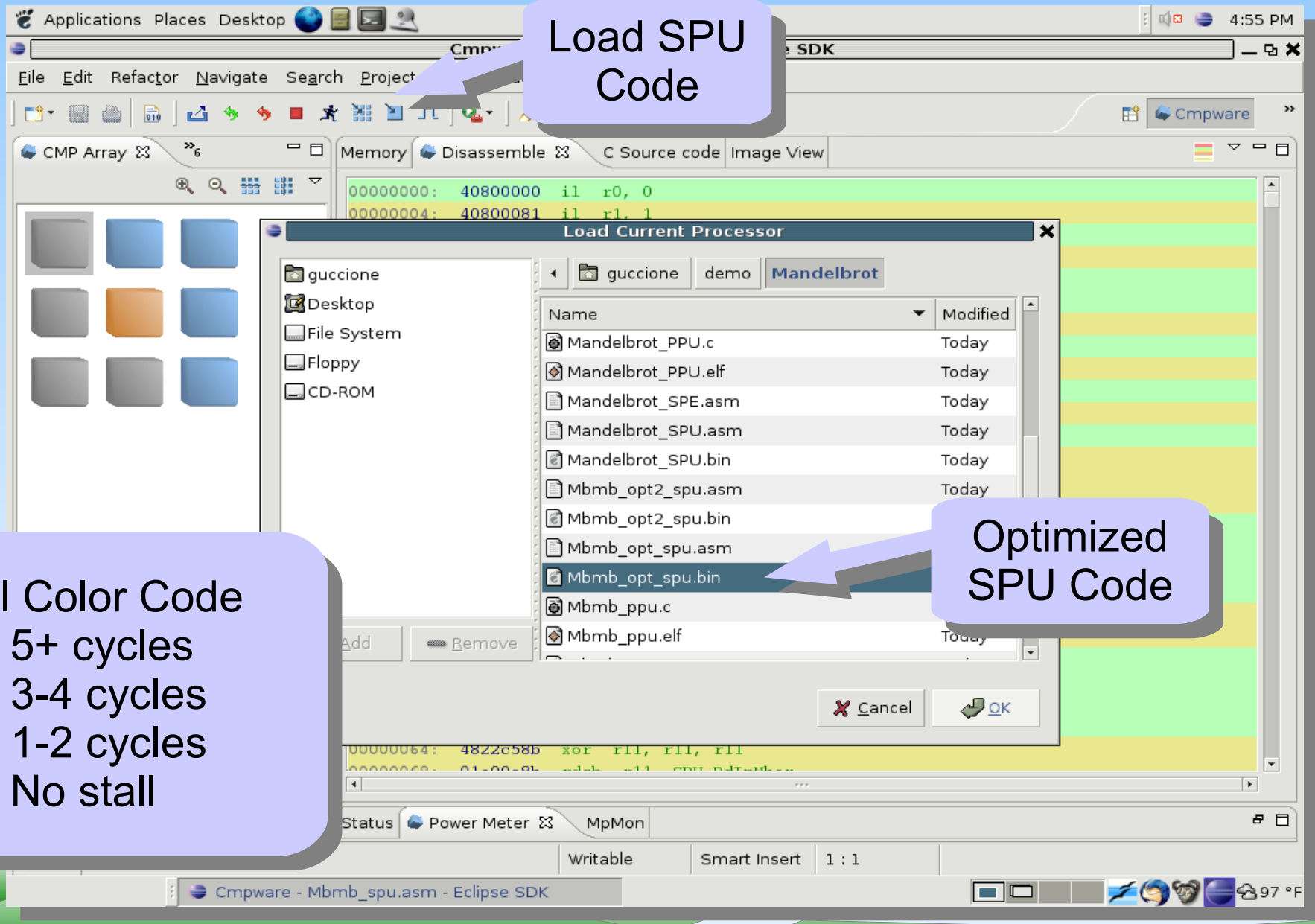
Instructions	Stalls	Stall Cycles
25	No stalls (green)	0
11	1-2 stalls (yellow)	11 – 22
12	3-4 stalls (orange)	36 – 48
12	5+ stalls (red)	60
Total: **60**		**107 – 130+**

Optimization I: Optimization Tips

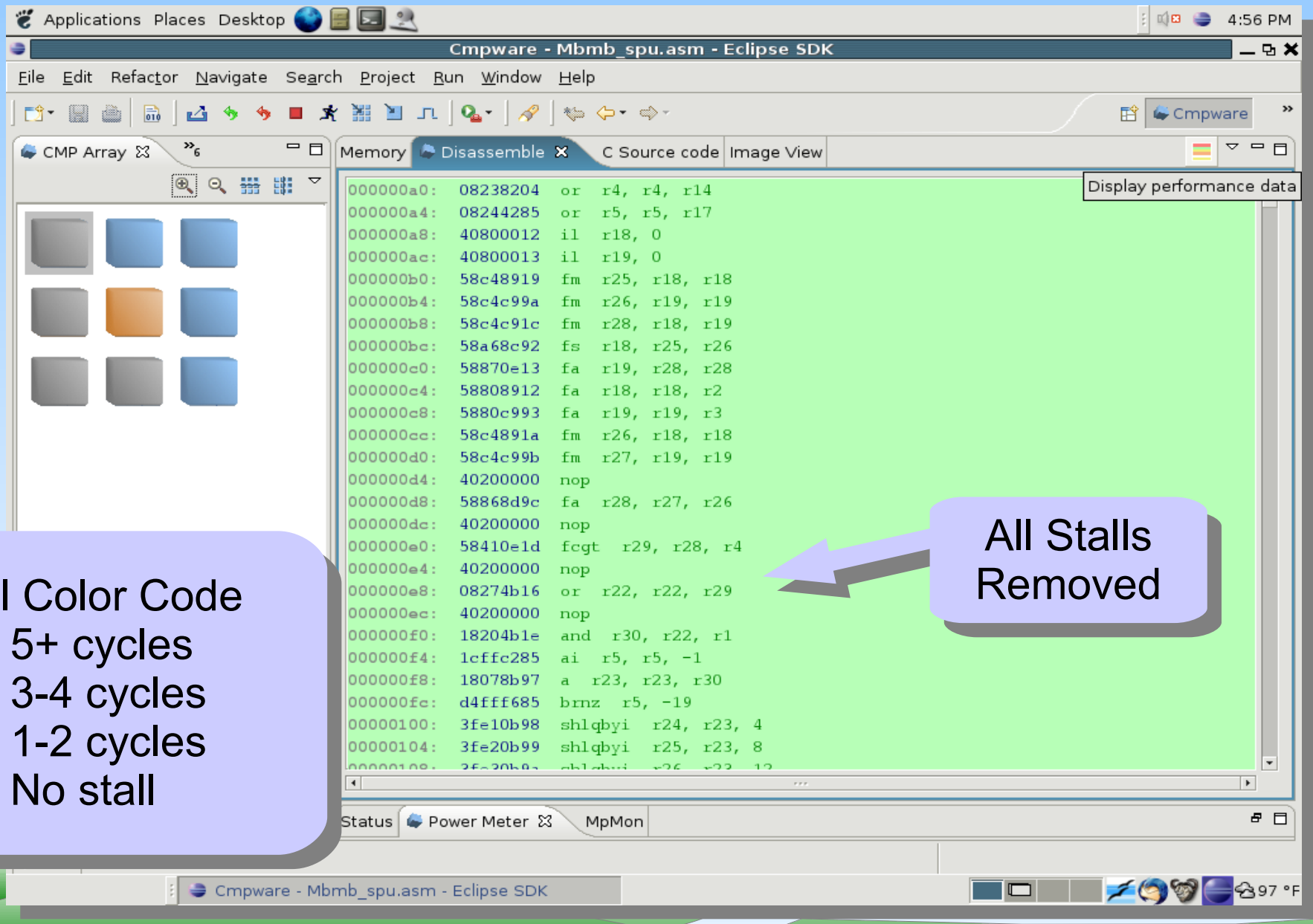
- Load data early
- Use lots of registers – register reuse can make scheduling difficult
- Do multiple interleaved calculations at a time
- 'Abstract' away from the algorithm – just think registers and cycles
- It's sort of like 'Tetris' :^)

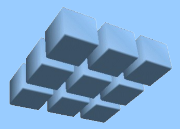


Optimization I: Optimized Code



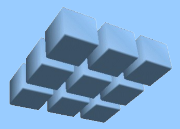
Application III: Optimized Code





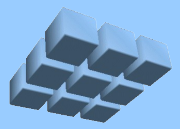
Optimization I: Overview

- SPU performance highly dependent on dependencies ('schedule')
- An unusual idea for many programmers
- *Cmpware* SPU Dependency View:
  - Give a 'snapshot' of depencies
  - Fast and interactive
  - Permits lots of experimentation
- Can dramatically improve SPU performance
- **2x - 3x** speedup in Mandelbrot SPU code



I. Introduction
II. Installing the *Cmpware CMP-DK*
III. Application I: A Simple Program
IV. Application II: Shared Memory
V. Application III: Mailboxes
VI. Optimization I: SPU Dependencies
- **VII. Optimization II: SPU Dual Issue**

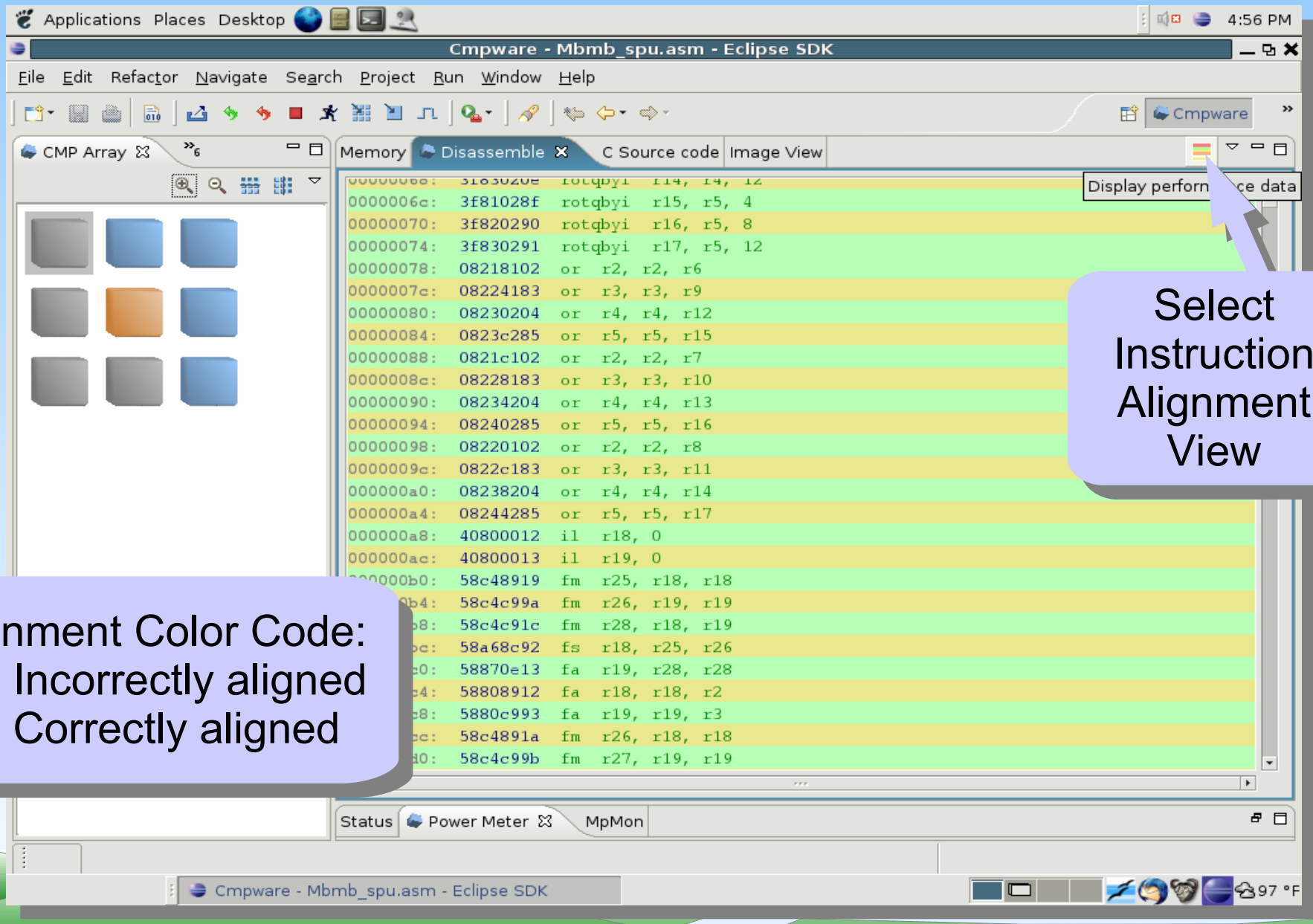
VIII. System Level Analysis
IX. Overview



Optimization II: SPU Dual Issue

- SPU can execute two instructions in a cycle
- Two pipelines not identical
  - **'Pipe 0' instructions:** arithmetic, logical, others
  - **'Pipe 1' instructions:** branches, loads, stores, others
- Pipe 0 instructions should be aligned on 'even' addresses (ending in 0x0 or 0x8)
- Pipe 1 instructions should be aligned on 'odd' addresses (ending in 0x4 or 0xc)

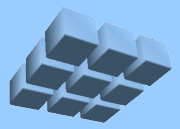
Optimization II: Alignment View



Select Instruction Alignment View

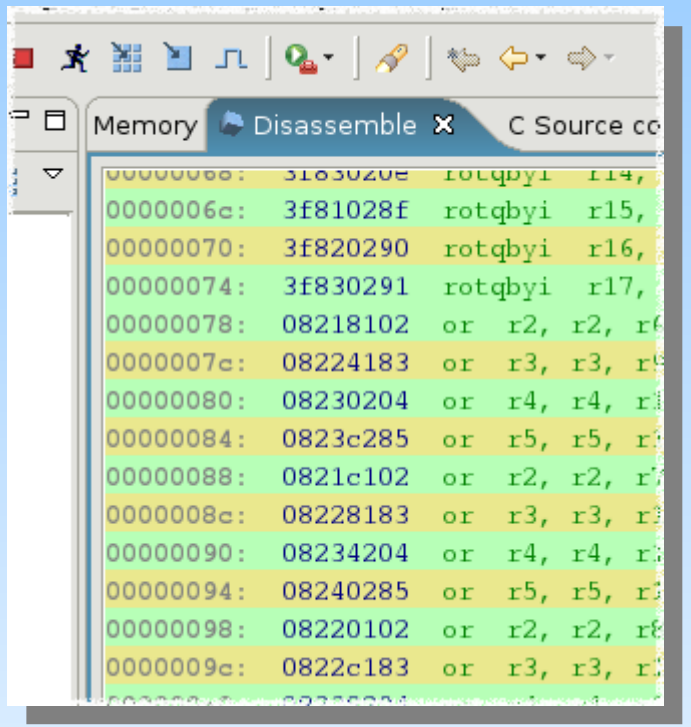
Alignment Color Code:

- Incorrectly aligned
- Correctly aligned

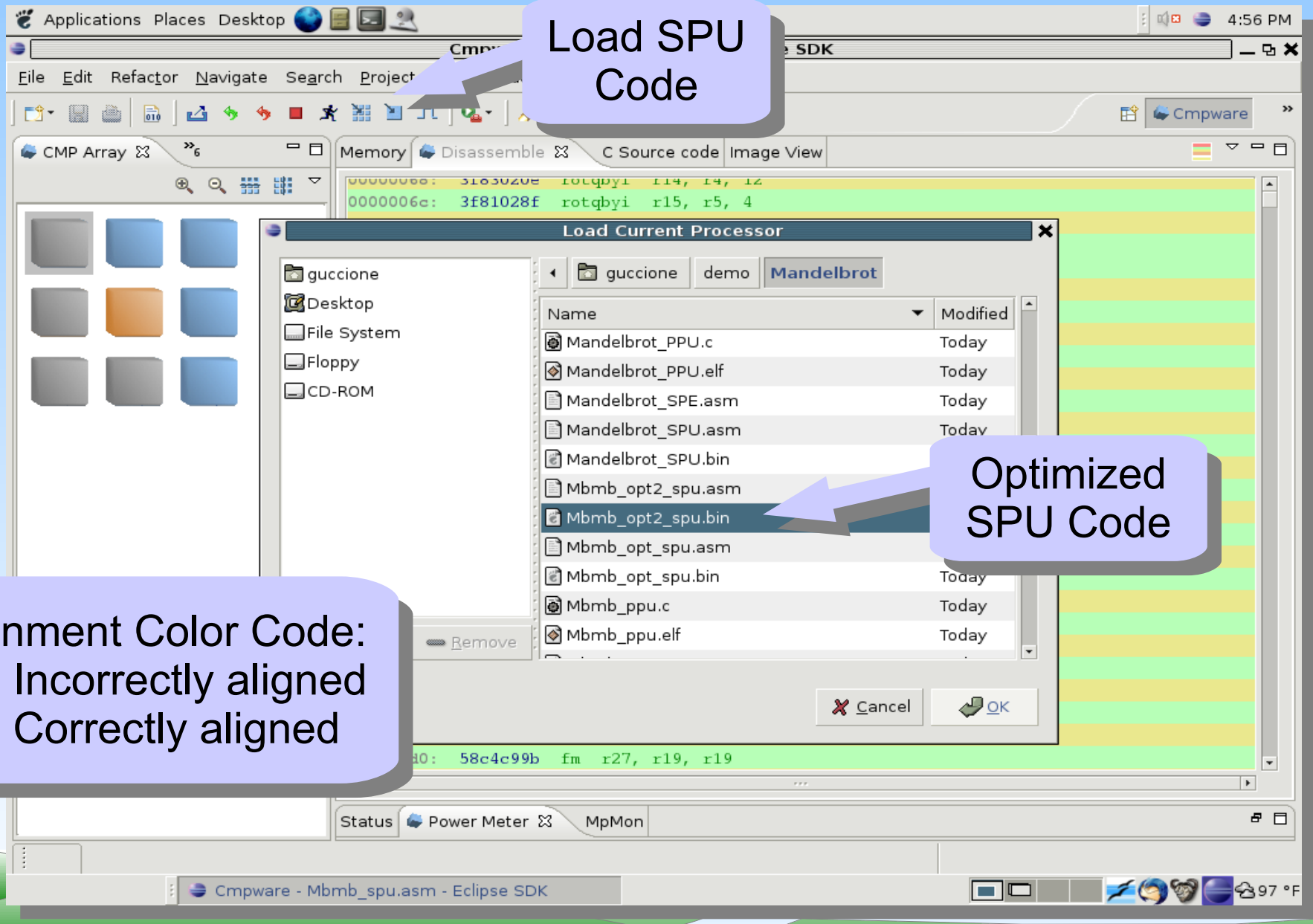


Optimization II: Alignment

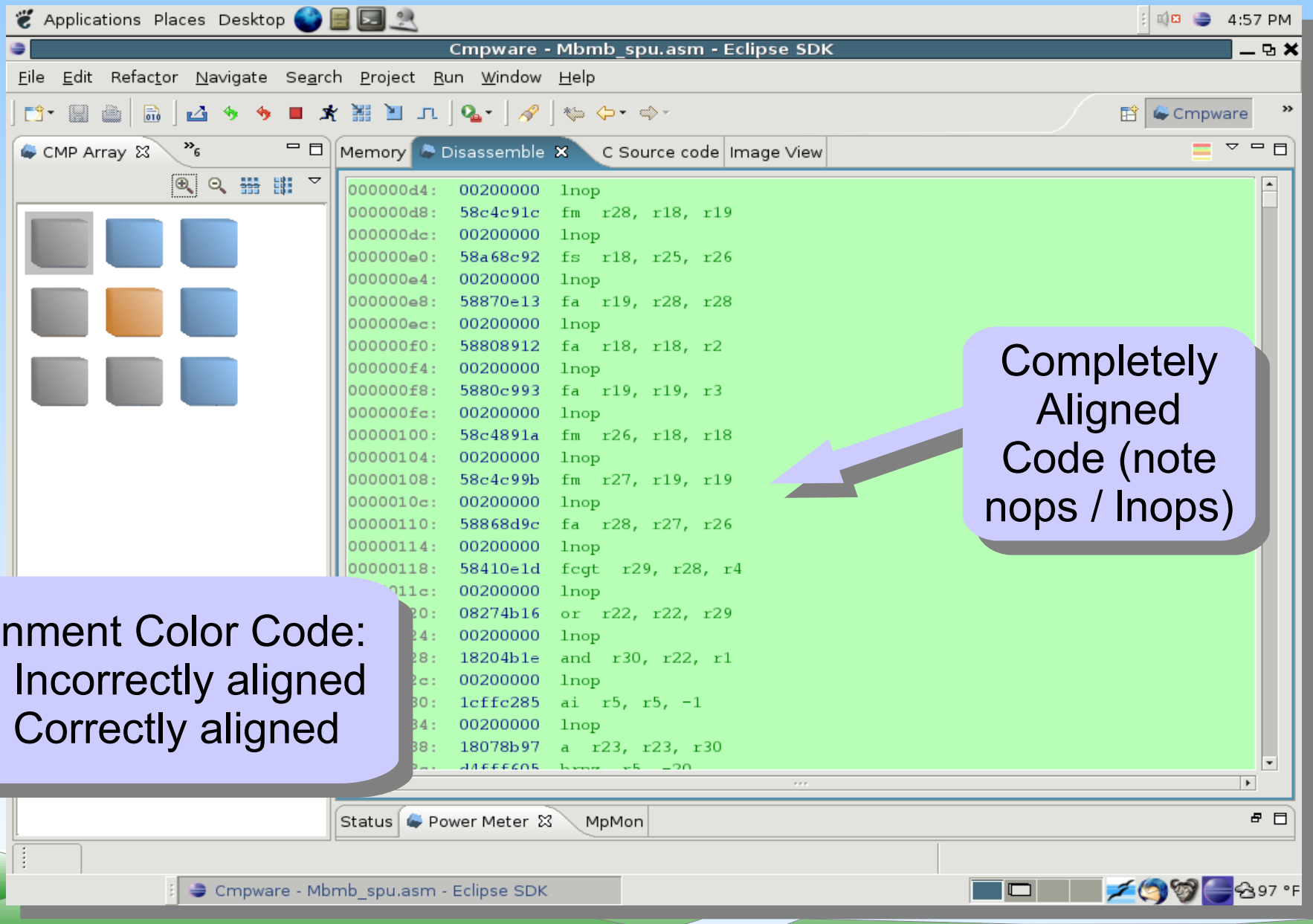
- Worst case:
  - One instruction per cycle
  - yellow / green 'stripes'
  - 50% SPU utilization
- Best case:
  - All green
  - Completely aligned
  - 100% SPU utilization (2x speedup)
- All yellow: only one '**nop**' away from 100%

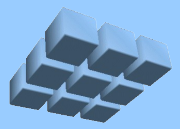


Optimization II: Aligned Code



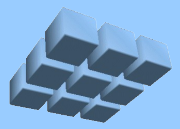
Optimization II: Aligned Code





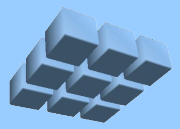
Optimization II: Alignment Tips

- Start from the top and work down
- Move instructions upward
- Don't break existing dependencies!
- Insert **`nop/lnop`** as a last resort
- **`Nop/lnops:`**
  - Increase code size
  - But may reduce overall cycle count
  - Nice to have **`nop`**s in code as 'documentation'

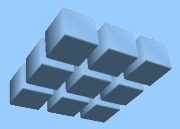


Optimization II: Overview

- SPU performance highly dependent on instruction alignment
- Another unusual idea for many programmers
- *Cmpware* SPU Alignment View:
  - Give a 'snapshot' of instruction alignment
  - Fast and interactive
  - Permits lots of experimentation
- Can quickly improve SPU performance
- Nearly **2x** speedup in Mandelbrot SPU code



I. Introduction
II. Installing the *Cmpware CMP-DK*
III. Application I: A Simple Program
IV. Application II: Shared Memory
V. Application III: Mailboxes
VI. Optimization I: SPU Dependencies
VII. Optimization II: SPU Dual Issue
- VIII. System Level Analysis
IX. Overview



System Level Analysis: Optimization Statistics

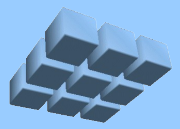
- Each optimization phase adds instructions ...

  ... but reduces cycle count

- Approx. **4x** performance increase

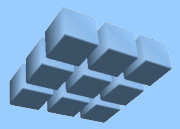
	Instructions	Stalls	I.P.C.*	Cycles
Naïve	60	107 – 130+	1	167 – 190+
Opt. I	73	0	1	73
Opt. II	98	0	2	49

* I.P.C. is 'Instructions Per Clock' for SPU dual issue pipeline.



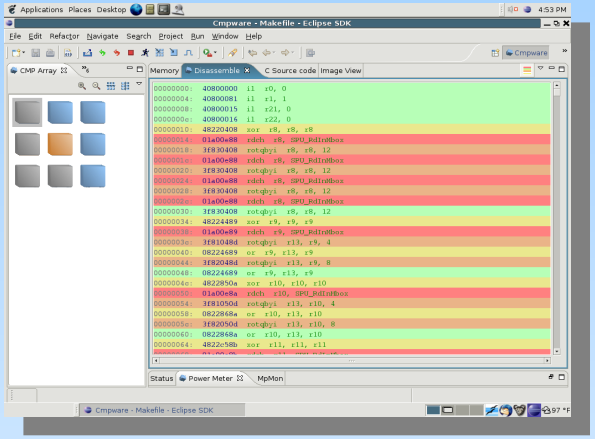
System Level Analysis

- Use various *Cmpware* tools together:
  - Processor Utilization indicates which nodes to optimize
  - Profiling indicates areas on nodes to optimize
  - Stall data / profiling aids in partitioning
- Customize the IDE:
  - Any view can be accessed in <2 mouse clicks
  - Moving / resizing windows streamlines development



Putting It All Together

- In the *Cmpware CMP-DK* for the Cell BE:
  1. Edit ...
  2. Compile ...
  3. Execute ...
  4. and Debug multicore code
  5. Find bottlenecks and 'hot spots'
  6. Optimize SPU dependencies
  7. Optimize SPU instruction alignment



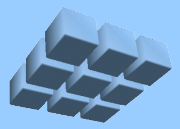
... all in one Eclipse-based IDE

Cmpware CMP-DK for the Cell BE

- *Eclipse* / *Java* based
- Runs 'everywhere'
- Completely self-contained
- Compact: 1.5 MB 'plugin'
- Easy to install (seconds)




- ***Increase programmer productivity***
- ***Increase Cell BE software performance***



Resources

- Cmpware, Inc: http://www.cmpware.com/
- Cmpware Cell BE Update Site:
  http://www.cmpware.com/cellbe/
- All code from this presentation avaliable at:
  http://www.cmpware.com/cellbe/CmpwareCellBE_demo.zip
- Cmpware License Request: info@cmpware.com
- IBM Cell tools for Windows / Cygwin
  http://cellbe-cygwin.cvs.sourceforge.net/cellbe-cygwin/cellbe-cygwin/