http://dx.doi.org/10.14400/JDC.2015.13.12.381

A Code-level Parallelization Methodology to Enhance Interactivity of Smartphone Entertainment Applications

Byung-Cheol Kim

Information and Culture Technology Studies, Seoul National University

스마트폰 엔터테인먼트 애플리케이션의 상호작용성 개선을 위한 코드 수준 병렬화 방법론

김병철

서울대학교 정보문화학전공

Abstract One of the fundamental requirements of entertainment applications is interactivity with users. The mobile device such as the smartphone, however, does not guarantee it due to the limit of the application processor's computing power, memory size and available electric power of the battery. This paper proposes a methodology to boost responsiveness of interactive applications by taking advantage of the parallel architecture of mobile devices which, for instance, have dual-core, quad-core or octa-core. To harness the multi-core architecture, it exploits the POSIX thread, a platform-independent thread library to be able to be used in various mobile platforms such as Android, iOS, etc. As a useful application example of the methodology, a heavy matrix calculation function was transformed to a parallelized version which showed around 2.5 ~ 3 times faster than the original version in a real-world usage environment.

Key Words : Smartphone Applications, Parallelization, Thread, POSIX, Matrix Calculation

요 약 스마트폰과 같은 이동형 장치들은 계산 성능이나 메모리 크기, 배터리 전력량 등의 한계로 인해 엔터테인먼트 애플리케이션이 요구하는 상호작용성을 보장하기 어렵다. 이를 해결하기 위해 본 논문에서는 상호작용이 필수적인 애플리케이션의 응답 속도를 개선할 수 있는 코드 수준 병렬화 방법론을 제안한다. 이 방법을 적용하면, 스마트폰 등에서 제공하는 멀티코어 아키텍쳐를 바탕으로 기존 애플리케이션의 모노코어 알고리즘을 복잡한 재설계 없이 코드 수준에서 병렬화 할 수 있다. 특히 플랫폼 독립적인 표준 쓰레드 라이브러리인 POSIX 쓰레드를 활용하면 안드로이드나 iOS등의 다양한 스마트폰 플랫폼에서 본 방법론을 적용할 수 있다. 이의 효과적인 응용 사례로서 수백만 개의 원소를 처리하는 행렬 연산 함수를 병렬화 해보았고 실사용 환경에서 약 3배가량의 성능 향상을 확인하였다.

주제어 : 스마트폰 애플리케이션, 병렬화, 쓰레드, POSIX, 행렬 계산

Received 24 October 2015, Revised 28 November 2015
Accepted 20 December 2015
Corresponding Author: Byung-Cheol Kim
(ITCT Studies, Seoul National University)
Email: clorvie@gmail.com

ISSN: 1738-1916

1. Introduction

This paper describes some strategies and associated techniques to boost the performance of compute- intensive mobile applications without algorithmic re-design. This assumes there is room for improvement beyond algorithms, that is, the parallel architecture of the computer.

Most general and popular ones of such parallel features are multiple computing cores(e.g., dual-core, quad-core, or octa-core)[1,2]. But revising of serial algorithms to parallel ones is complex task of agonizing pain. Thus various approaches and methodologies have been suggested to simplify the process[3,4,5,6,7,8,9]. The proposed methodology is to parallelize already-deployed algorithms at the code level.

Note that the target coding languages of concern are C and C++. More specifically, a language which has full access privileges to memory is required since parallelization means multiplexing on data manipulation. In that point, C/C++ are still the state-of-the-art languages.

2. Using POSIX Threads

In utilizing a multi-core system there are multi-process and multi-thread techniques. Practically, in most cases, the latter is preferred[10]. Besides the fact that a process is a relatively heavier structure than a thread, the critical point is in data sharing over processes or threads. IPC (Inter-Process Communication) is very costly and limited because of the nature of the process and, more importantly, system security. In contrast, inter-thread communication is a little bit trivial since all threads share the same memory space, that is, the memory addresses of one process.

The POSIX thread, so called pthread, is an inter-operable multi-threading technology as POSIX stands for Portable Operating System Interface. For example, source code which uses pthreads on a Linux can be migrated onto the MS Windows without any modification. It only needs re-compilation. Thus, in order to take advantage of portability over mobile platforms, multi-threading will be described based on POSIX threads.

2.1 Basic Procedure

A typical procedure to apply a multi-threading technique to existing compute-intensive code is as follows:

- Specifying and isolating a compute-intensive block,
- Defining a thread function for the block, and
- Replacing the block with the thread function.

Next subsections will explain them in more detail.

2.1.1 Specifying and isolating a compute-intensive block

A compute-intensive block usually resides in a loop due to its nature. It might be a block of nested loops. To specify it is relatively easy because it must be one of the bottleneck points in the entire processing. In practice, we can track them down by examining the elapsed time of each computing block. Then, the outermost loop of the bottleneck block might be a driver of threads, each computation content of which comes from one computation step of the loop. For example, the for loop with variable i in [Fig. 1] would be the thread-driving loop and its inner part including the nested loop would be the thread procedure like in [Fig. 3].

To isolate a computation block means separation of variables of all input, output, and intermediate data from outside the block. Such data isolation is also needed between loop steps. In other words, both input data into and output data from one step of the computing loop should be independent from another step's. If not, each thread that comes from the step should be executed in order and that leads nothing but parallelization.

```
void Foo(float s[], float r[], float a[], float b[], float c[])
{// assumption: float s[100], r[9000], a[9000], b[9000], c[9000];
   int i, j, q;
   // This nesting loop is the block to replace
   for (i = 0; i < 100; i++) {
      s[i] = 0;
      // This nested loop is compute-intensive
      for (j = 0; j < 90; j++) {
            q = i*90;
            r[q+j] = a[q+j] - b[q+j] * c[q+j];
            s[i] += r[q+j];
      }
      s[i] /= 90.0f;
   }
}
```

[Fig. 1] A target code example. s and r are output while a, b, c are input.

In practice, all the related data should be collected and had better be wrapped in one memory structure such as struct or class as in [Fig. 2].

```
typedef struct {
         float *pS, *pR;
         float *pA, *pB, *pC;
} FooThreadData;
```

[Fig. 2] A wrapping structure for all the related data in the block

2.1.2 Defining a thread function for the block

The definition of a pthread procedure basically comes from the code inside the loop. A meaningful difference is in that it uses its own data set separated from another thread's. Thus, the procedure is declared with void pointers to input/output data to transport any kind of wrapped data at once as in [Fig. 3].

Note that the output of each thread task should be delivered to the caller using memory on heap or at least the caller's stack, too. The caller's main job is to collect the result of each thread and release the used memory if needed.

```
void* FooThreadProcedure(void *pData)
{
   // Convert the input from void to a member-accessible pointer
   FooThreadData *pTD = (FooThreadData*)pData;

   // Convenience variables for code readability and writability
   float s = 0, *r = pTD->pR;
   float *a = pTD->pA, *b = pTD->pB, *c = pTD->pC;
   int j;

   // The compute-intensive loop
   for (j = 0; j < 90; j++) {
      r[j] = a[j] - b[j] * c[j];
      s += r[j];
   }
   s /= 90.0f;
   *(pTD->pS) = s;

   return pData;
}
```

[Fig. 3] An example thread definition for the compute-intensive block in function Foo(...)

2.1.3 Replacing the block with the thread function

The final task is to substitute the target compute-intensive block with the defined thread function.

Each thread starts its own lifetime by detaching itself from the calling thread (e.g., function FooThreaded(...) in [Fig. 4]). It implies that the former cannot access the local memory(variables) in the latter's scope any longer. Therefore, at the beginning inside the outermost loop, the wrapped data structure for a thread procedure must be created as a dynamic memory. Then, all the related data should be assigned onto it in accordance with each thread's input coverage as of the comment 'Thread-2' in [Fig. 4].

After setting the data, a thread is created using pthread_create(...) with it. At this point, the created thread will be executed independently from other threads including its caller. It goes on only until the given function (e.g., FooThreadProcedure(...)) ends. The caller can wait for a thread to finish and get the result using pthread_join(...) as of the comment 'Thread-2.5' and 'Thread-3' in [Fig. 4].

```
// the number of cores + alpha (considering overhead)
#define NumOfThreads 5

void FooThreaded(float s[], float r[],
                 float a[], float b[], float c[])
{
// assumption: float s[100], r[9000], a[9000], b[9000], c[9000];

   // Thread-1. declarations
   size_t iThread=0, nThreads=0, nMaxThreads = NumOfThreads;
   // STL's vector could be used instead of a plain array
   pthread_t aThreads[NumOfThreads];
   // each thread's own input/output data
   FooThreadData *pData, *pResult = NULL;
   int i, rc, tCount;

   for (i = 0; i < 100; i++) {
      // Thread-2. distribute tasks among threads
      pData = new FooThreadData();
      pData->pS = s + i;
      pData->pR = r + i*90;
      pData->pA = a + i*90;
      pData->pB = b + i*90;
      pData->pC = c + i*90;

      // create a thread
      rc = pthread_create(&aThreads[iThread++], NULL,
                          FooThreadProcedure, (void*)pData);
      nThreads++;

      // Thread-2.5. inter-rim harvest
      //  to prevent an overflow of aThreads[]
      if (nThreads >= nMaxThreads) {
         if (iThread >= nMaxThreads)
            iThread = 0;
         // wait for the thread to finish
         pthread_join(aThreads[iThread], (void**)&pResult);
         if (pResult) {
            /* collect and process on the result if needed */
            delete pResult; // only if needed
            pResult = NULL;
         }
         nThreads--;
      }
   }

   // Thread-3. harvest threads' result

   // should examine to the end of aThreads[]
   if (nThreads >= nMaxThreads - 1) {
      for (tCount = 0; tCount < nMaxThreads; tCount++) {
         // This equality means the cyclic reuse of the array
         if (tCount == iThread) continue;
         // wait for the thread to finish
         pthread_join(aThreads[tCount], (void**)&pResult);
         if (pResult) {
            /* collect and process on the result if needed */
            delete pResult; // only if needed
            pResult = NULL;
         }
      }
   }
   // if there exist(s) unfinished thread(s)
   else if (nThreads > 0) {
      for (tCount = 0; tCount < nMaxThreads; tCount++) {
         // This equality means the end of threading
         if (tCount == iThread) break;
         // wait for the thread to finish
         pthread_join(aThreads[tCount], (void**)&pResult);
         if (pResult) {
            /* collect and process on the result if needed */
            delete pResult; // only if needed
            pResult = NULL;
         }
      }
   }

   // finally all cleared.
   iThread = nThreads = 0;
}
```

[Fig. 4] A multi-threaded version of function Foo(...)

3. Case Studies

3.1 SIFT Descriptor Extraction

Robust feature extraction techniques like SIFT (Scale-Invariant Feature Transform) are getting an increasing focus for mobile applications such as face detection/recognition, image-based search, and so on. Interactive use of such applications always requires maximum utilization of computing resources. Thus, it could be a target of parallelization discussed above.

A function shown in [Fig. 5] is one of the compute-intensive blocks in computing SIFT descriptors[12]. Since it has a typical structure of repetition of the same operations on a series of massive memory blocks, it could be easily transformed into a threaded version like [Fig. 6] and [Fig. 7].

```
/*
  Computes feature descriptors for features in an array.
  Based on Section 6 of Lowe's paper.

  @param features array of features
  @param gauss_pyr Gaussian scale space pyramid
  @param d width of 2D array of orientation histograms
  @param n number of bins per orientation histogram
*/

void compute_descriptors(CvSeq* features,
                         IplImage*** gauss_pyr, int d, int n)
{
   // The holder of main data
   float*** hist;
   struct feature* feat;
   struct detection_data* ddata;
   int i, k = features->total;

   // For each feature
   for (i = 0; i < k; i++) {
      feat = CV_GET_SEQ_ELEM(struct feature, features, i);
      ddata = feat_detection_data(feat);

      // Evaluate orientation histograms
      hist = descr_hist(gauss_pyr[ddata->octv][ddata->intvl],
          ddata->r, ddata->c, feat->ori, ddata->scl_octv, d, n);

      // Convert the histograms to SIFT descriptors
      hist_to_descr(hist, d, n, feat);
      release_descr_hist(&hist, d);
   }
}
```

[Fig. 5] A function to evaluate SIFT descriptors excerpted from 'OpenSIFT' [12]

One point on which an attention should be given here is the line right below 'Thread-2' comment line in [Fig. 9]. Instead of using new to prepare a thread-specific data memory, it uses an element of an array (aSIFTThreadData[]) which is declared as a global variable outside the function. It is because dynamic memory allocation imposes a non-negligible amount of tasks on the kernel. Thus it is ought to be employed as less as possible.

Note that the array is *global* which can be safely accessed to outside the function (e.g., in SIFTThreadProc()) as discussed in Section 2.1.3. And consequently, it is no longer required to release the allocated memory after each waiting for the end of a thread as '//delete pResult;' in this case.

```
class SIFTThreadData
{
public:
   int d, n;
   struct feature* feat;
   IplImage*** gauss_pyr;
};

void* SIFTThreadProc(void *pData)
{
   SIFTThreadData* p = (SIFTThreadData*)pData;

   struct detection_data* ddata = feat_detection_data(p->feat);

   // Evaluate orientation histograms
   float*** hist =
      descr_hist(p->gauss_pyr[ddata->octv][ddata->intvl],
         ddata->r, ddata->c, p->feat->ori,
         ddata->scl_octv, p->d, p->n);

   // Convert the histograms to SIFT descriptors
   hist_to_descr(hist, p->d, p->n, p->feat);
   release_descr_hist(&hist, p->d);

   return pData;
}
```

[Fig. 6] A thread definition and its data structure for function compute_descriptors(...)

The implementation was on Android 4.3 (Jellybean) and the performance test was conducted on Samsung Galaxy Note 3 of quad-core 2.3GHz application processor(AP). On the input image of 640x480 pixels, the original function was done in 1.8319 seconds while the parallel version was done in 0.6066 seconds, about 3 times faster. Note that the specified number of threads did not make a big difference from 5 to 200.

```
#define _NumOfSIFTThreads_ 100
SIFTThreadData aSIFTThreadData[_NumOfSIFTThreads_];

void compute_descriptors_thr(CvSeq* features,
                             IplImage*** gauss_pyr, int d, int n)
{
   // Thread-1. declarations
   typedef unsigned int size_t;
   size_t iThread = 0, nThreads = 0;
   size_t nMaxThreads = _NumOfSIFTThreads_;
   pthread_t aThreads[_NumOfSIFTThreads_];
   SIFTThreadData *pThreadData, *pResult = NULL;
   int i, k = features->total;

   for (i = 0; i < k; i++) {
      // Thread-2. distribute tasks among threads
      // We just use an array instead.
      //pThreadData = new SIFTThreadData();
      pThreadData = aSIFTThreadData + iThread;
      pThreadData->d = d;
      pThreadData->n = n;
      pThreadData->gauss_pyr = gauss_pyr;
      pThreadData->feat =
                  CV_GET_SEQ_ELEM(struct feature, features, i);

      int rc = pthread_create(&aThreads[iThread++], NULL,
                        SIFTThreadProc, (void*)pThreadData);
      nThreads++;

      // Thread-2.5. inter-rim harvest to prevent an overflow
      if (nThreads >= nMaxThreads) {
         if (iThread >= nMaxThreads)
            iThread = 0;
         pthread_join(aThreads[iThread], (void**)&pResult);
         if (pResult) {
            // We just use an array instead.//delete pResult;
            pResult = NULL;
         }
         nThreads--;
      }
   }

   // Thread-3. harvest threads' result
   if (nThreads >= nMaxThreads -1) {
      for (size_t tCount = 0; tCount < nMaxThreads; tCount++) {
         if (tCount == iThread) continue
         pthread_join(aThreads[tCount], (void**)&pResult);
         if (pResult) {
            // We just use an array instead.//delete pResult;
            pResult = NULL;
         }
      }
   }
   else if (nThreads > 0) {
      for (size_t tCount = 0; tCount < nMaxThreads; tCount++) {
         if (tCount == iThread) break
         pthread_join(aThreads[tCount], (void**)&pResult);
         if (pResult) {
            // We just use an array instead.//delete pResult;
            pResult = NULL;
         }
      }
   }
   iThread = nThreads = 0;
}
```

[Fig. 7] A multi-threaded function compute_descriptors(...)

3.2 Matrix Calculation

The high-resolution image/video processing such as H.264 decoding needs a performance boost utilizing the parallel architecture[11]. Handling such big images also involves calculations of matrices of huge dimensions. Thus operations on matrices could be another compute-intensive example in the real world.

In practice, there are several implementations which can deal with big matrix operations like matrix multiplication, transpose, inverse calculation, etc. A popular one of them[13] is GEMM(GEneralized Matrix Multiplication) of OpenCV(Open Computer Vision Library)[14]. As lots of vision or AR(Augmented Reality) applications take advantage of OpenCV, its embedded GEMM is getting popular too.

Therefore the OpenCV GEMM function is chosen to be transformed using the proposed methodology. At a glimpse, it looks much more complex than the previous example, i.e., SIFT. But its skeleton is concise as described in [Fig. 8~9].

```
void gemm(InputArray matA, InputArray matB, double alpha,
      InputArray matC, double beta, OutputArray _matD, int flags)
{
  const int block_lin_size = 128;
  const int block_size = block_lin_size * block_lin_size;
  /* omitted */
  // 1. if trivial(~4x4 matrices), just calculate right here
  /* omitted */
  // 2. not trivial, indeed. further analysis required.
  /* omitted before the last if-else statement */

  // 3. a less severe case; we can simply multiply
  if (((d_size.height <= block_lin_size/2 ||
        d_size.width <= block_lin_size/2) &&
    len <= 10000) || len <= 10 ||
    (d_size.width <= block_lin_size &&
     d_size.height <= block_lin_size && len <= block_lin_size) ) {
    singleMulFunc( A.data, A.step, B.data, b_step, Cdata, Cstep,
    matD->data, matD->step, a_size, d_size, alpha, beta, flags );
  }
  // 4. the heaviest case; we should divide the matrices
  //    into blocks and multiply with them
  else {
    for (i = 0; i < d_size.height; i += di) {
      di = dm0;
      if (i + di >= d_size.height || 8*(i + di) + di > 8*d_size.height)
          di = d_size.height - i;

      iThread = nThreads = 0;

/* The part here is in [Fig. 9] due to the size of this text box */

    }// for i
  }// else
  /* omitted */
}
```

[Fig. 8] An example function of generalized matrix multiplication excerpted from function gemm(...) of OpenCV[14]

It is well organized thus powerful enough to efficiently handle from very small matrices of like 2-by-2 to huge matrices of millions-by-millions. It classifies the ranks of input matrices into several categories and then accordingly calculate with them. In the heaviest case, it divides input matrices into sub-blocks (e.g., 128-by-128) and calculate them one by one. That could be transformed into a multi-threaded form as in [Fig. 10~14]. Note that another part such as singleMulFunc(...) function is also one of the good targets to transform.

```
for (j = 0; j < d_size.width; j += dj ) {
  uchar* _d = matD->data + i*matD->step + j*elem_size;
  const uchar* _c = Cdata + i*c_step0 + j*c_step1;
  size_t _d_step = matD->step;
  dj = dn0;
  if (j + dj >= d_size.width || 8*(j + dj) + dj > 8*d_size.width)
      dj = d_size.width - j;
  flags &= 15;
  if (dk0 < len) {
    _d = d_buf;
    d_step = dj*work_elem_size;
  }
  // This loop is a compute-intensive block to replace
  for (k = 0; k < len; k += dk ) {
    const uchar*  a = A.data + i*a_step0 + k*a_step1;
    size_t  a_step = A.step;
    const uchar*  b = B.data + k*b_step0 + j*b_step1;
    size_t _b_step = b_step;
    Size a_bl_size;

    dk = dk0;
    if (k + dk >= len || 8*(k + dk) + dk > 8*len)
      dk = len - k;

    if (!is_a_t )
      a_bl_size.width = dk, a_bl_size.height = di;
    else
      a_bl_size.width = di, a_bl_size.height = dk;

    if (a_buf && is_a_t ) {
      a_step = dk*elem_size;
      GEMM_TransposeBlock( a, A.step, a_buf,  a_step,
                           a_bl_size, elem_size);
      std::swap( a_bl_size.width, a_bl_size.height );
      a = a_buf;
    }

    if (dj < d_size.width) {
      Size b_size;
      if( !is_b_t )
        b_size.width = dj, b_size.height = dk;
      else
        b_size.width = dk, b_size.height = dj;

      b_step = b_size.width*elem_size;
      GEMM_CopyBlock( b, b_step, b_buf,  b_step, b_size, elem_size);
      b = b_buf;
    }

    if (dk0 < len) blockMulFunc(a,  a_step,  b,  b_step,  d,  d_step,
                                a_bl_size, Size(dj,di), flags);
    else singleMulFunc(a, _a_step, _b, _b_step, _c, Cstep,
           d,  d_step, a_bl_size, Size(dj,di), alpha, beta, flags);
    flags |= 16;
  }// for k

  if( dk0 < len ) {
    storeFunc( c, Cstep,  d,  d_step,
         matD->data + i*matD->step + j*elem_size,
         matD->step, Size(dj,di), alpha, beta, flags);
  }
}// for j
```

[Fig. 9] An inner loop of function gemm(...) of OpenCV[14]

```
class GEMM_BlockMul_Data
{
public:
    // block indices and operation options
    int i, j, flags;
    // scalar values which will be multiplied to each matrix
    double alpha, beta;
    // pointes to input(operand) matrices and output(result) matrix
    Mat *pA, *pB, *pC, *pD;
    // the width and height of the result matrix
    Size d_size;

    // function pointers in accordance with input data precisions
    // (32FC1 ~ 64FC2)
    GEMMSingleMulFunc singleMulFunc;
    GEMMBlockMulFunc blockMulFunc;
    GEMMStoreFunc storeFunc;

    // values for intermediate calculation
    int is_a_t, is_b_t;
    int len, elem_size;
    int di, dj, dk, dk0;
    int a_buf_size, b_buf_size, d_buf_size;
    size_t a_step0, a_step1, b_step, b_step0, b_step1;
    size_t c_step0, c_step1, Cstep, _d_step;

    // pointes to input blocks and output block
    const uchar *Cdata, *_c;
    uchar *a_buf, *b_buf, *_d;
};
```

[Fig. 10] A thread data structure for a part of function gemm(...)

A few notes on implementation are as follows. As described above, dynamic memory allocation ought to be avoided as many as possible since it degrades actual performance. One alternative is global memory for repetitive use of the same memory structure in series. Instead of allocating new memory, the already-allocated but now-invalid memory could be used to be overwritten onto.

Although consecutive pieces of global memory, which are actually arrays, are preferred as many as possible, note that a chunk of global memory which has compile-time non-zero default value directly affects the footprint size of the output binary code.

In many cases, threaded functions need proportionally more amounts of memory since each thread procedure requires its own memory for intermediate storage for computing. The available heap size should be considered on the fly, especially on mobile platforms. Therefore the efficient reuse of dynamically allocated (heap/global) memory is critical in case of dealing with big-size input/output data.

```
void* GEMM_BlockMul_ThreadProc(void *pData)
{
  GEMM_BlockMul_Data* p = (GEMM_BlockMul_Data*)pData;

  // This loop comes from the original as in [Fig. 9]
  // The difference here is wrapping of input/output data
  for (int k = 0; k < p->len; k += p->dk) {
    const uchar* _a = p->pA->data + p->i*p->a_step0 + k*p->a_step1;
    size_t _a_step = p->pA->step;
    const uchar* _b = p->pB->data + k*p->b_step0 + p->j*p->b_step1;
    size_t _b_step = p->b_step;
    Size a_bl_size;

    p->dk = p->dk0;
    if (k + p->dk >= p->len || 8*(k + p->dk) + p->dk > 8*p->len)
    p->dk = p->len - k;

    if (!p->is_a_t)
      a_bl_size.width = p->dk, a_bl_size.height = p->di;
    else a_bl_size.width = p->di, a_bl_size.height = p->dk;

    if (p->a_buf && p->is_a_t) {
      _a_step = p->dk*p->elem_size;
      GEMM_TransposeBlock(_a, p->pA->step, p->a_buf,
                  _a_step, a_bl_size, p->elem_size);
      std::swap(a_bl_size.width, a_bl_size.height);
      _a = p->a_buf;
    }

    if (p->dj < p->d_size.width) {
      Size b_size;
      if (!p->is_b_t) b_size.width = p->dj, b_size.height = p->dk;
      else b_size.width = p->dk, b_size.height = p->dj;

      _b_step = b_size.width*p->elem_size;
      GEMM_CopyBlock( _b, p->b_step, p->b_buf,
                  _b_step, b_size, p->elem_size );
      _b = p->b_buf;
    }

    if (p->dk0 < p->len) {
      p->blockMulFunc(_a, _a_step, _b, _b_step, p->_d, p->_d_step,
               a_bl_size, Size(p->dj,p->di), p->flags);
    }
    else {
      p->singleMulFunc(_a, _a_step, _b, _b_step, p->_c, p->Cstep,
               p->_d, p->_d_step, a_bl_size, Size(p->dj,p->di),
               p->alpha, p->beta, p->flags);
    }

    // signal that, after this point,
    //  blockMulFunc() should accumulate the calc. results
    p->flags |= 16;
  } // for k

  // clean up the thread's own memory if needed
  if (p->dk0 < p->len) {
    p->storeFunc(p->_c, p->Cstep, p->_d, p->_d_step,
          p->pD->data + p->i*p->pD->step + p->j*p->elem_size,
          p->pD->step, Size(p->dj,p->di),
          p->alpha, p->beta, p->flags);
    free(p->_d);
    p->_d = NULL;
  }
  if (p->is_a_t) {
    free(p->a_buf);
    p->a_buf = NULL;
  }
  if (p->dj < p->d_size.width) {
    free(p->b_buf);
    p->b_buf = NULL;
  }

  return pData;
}
```

[Fig. 11] A thread definition for a part of function gemm(...)

```
// Thread-2.1. set data for each thread
#define SetGEMM_BlockMul_Data(\
    pThreadData, in__i, in__j, in__flags,\
    in__A, in__B, in__C, in__D,\
    in__alpha, in__beta, in__d_size, in__b_step,\
    in__singleMulFunc, in__blockMulFunc, in__storeFunc,\
    in__Cdata, in__Cstep, in__len,\
    in__is_a_t, in__is_b_t, in__elem_size,\
    in__a_buf_size, in__b_buf_size, in__d_buf_size,\
    in__a_buf, in__b_buf, in__di, in__dj, in__dk, in__dk0,\
    in__a_step0, in__a_step1, in__b_step0, in__b_step1,\
    in__c_step0, in__c_step1, in___c, in___d, in___d_step) \
    do {\
        (pThreadData)->i = (in__i);\
        (pThreadData)->j = (in__j);\
        (pThreadData)->flags = (in__flags);\
        (pThreadData)->pA = &(in__A);\
        (pThreadData)->pB = &(in__B);\
        (pThreadData)->pC = &(in__C);\
        (pThreadData)->pD = &(in__D);\
        (pThreadData)->alpha = (in__alpha);\
        (pThreadData)->beta = (in__beta);\
        (pThreadData)->d_size = (in__d_size);\
        (pThreadData)->b_step = (in__b_step);\
        (pThreadData)->singleMulFunc = (in__singleMulFunc);\
        (pThreadData)->blockMulFunc = (in__blockMulFunc);\
        (pThreadData)->storeFunc = (in__storeFunc);\
        (pThreadData)->Cdata = (in__Cdata);\
        (pThreadData)->Cstep = (in__Cstep);\
        (pThreadData)->len = (in__len);\
        (pThreadData)->is_a_t = (in__is_a_t);\
        (pThreadData)->is_b_t = (in__is_b_t);\
        (pThreadData)->elem_size = (in__elem_size);\
        (pThreadData)->a_buf_size = (in__a_buf_size);\
        (pThreadData)->b_buf_size = (in__b_buf_size);\
        (pThreadData)->d_buf_size = (in__d_buf_size);\
        if (in__is_a_t)\
            (pThreadData)->a_buf =\
                (uchar*)malloc(\
                    (in__a_buf_size) * sizeof((buf)[0]));\
        else (pThreadData)->a_buf = (in__a_buf);\
        if ((in__dj) < (in__d_size).width)\
            (pThreadData)->b_buf =\
                (uchar*)malloc(\
                    (in__b_buf_size) * sizeof((buf)[0]));\
        else (pThreadData)->b_buf = (in__b_buf);\
        (pThreadData)->di = (in__di);\
        (pThreadData)->dj = (in__dj);\
        (pThreadData)->dk = (in__dk);\
        (pThreadData)->dk0 = (in__dk0);\
        (pThreadData)->a_step0 = (in__a_step0);\
        (pThreadData)->a_step1 = (in__a_step1);\
        (pThreadData)->b_step0 = (in__b_step0);\
        (pThreadData)->b_step1 = (in__b_step1);\
        (pThreadData)->c_step0 = (in__c_step0);\
        (pThreadData)->c_step1 = (in__c_step1);\
        (pThreadData)->_c = (in___c);\
        (pThreadData)->_d = (in___d);\
        (pThreadData)->_d_step = (in___d_step);\
    } while (0)
```

[Fig. 12] A macro function to fill a thread data structure for function gemm_p(...)

Note that macro functions can be used to hide long tedious jobs for data filling and thread waiting as in [Fig. 12~13], respectively. They are enclosed by 'do {} while (0)' so that they can be called as a normal function at any point. A threaded version could be shown in a neat form of a few lines as in [Fig. 14].

```
// Thread-2.5. inter-rim harvest to prevent an overflow
#define IntermediateWaitForGEMM_BlockMul_Threads() \
    do {\
        GEMM_BlockMul_Data *pResult;\
        if (nThreads >= nMaxThreads) {\
            if (iThread >= nMaxThreads)\
                iThread = 0;\
            pthread_join(aThreads[iThread], (void**)&pResult);\
            if (pResult) {\
                pResult = NULL;\
            }\
            nThreads--;\
        }\
    } while (0)

// Thread-3. harvest threads' result
#define FinalWaitForGEMM_BlockMul_Threads() \
    do {\
        GEMM_BlockMul_Data *pResult;\
        if (nThreads >= nMaxThreads -1) {\
            for (size_t tCount = 0; tCount < nMaxThreads; tCount++) {\
                if (tCount == iThread) continue\
                pthread_join(aThreads[tCount], (void**)&pResult);\
                if (pResult) {\
                    pResult = NULL;\
                }\
            }\
        }\
        else if (nThreads > 0) {\
            for (size_t tCount = 0; tCount < nMaxThreads; tCount++) {\
                if (tCount == iThread) break\
                pthread_join(aThreads[tCount], (void**)&pResult);\
                if (pResult) {\
                    pResult = NULL;\
                }\
            }\
        }\
        iThread = nThreads = 0;\
    } while(0)
```

[Fig. 13] A macro function to wait for a thread for function gemm_p(...)

The implementation and test configuration are the same as the one of the SIFT case. The first test is conducted using a matrix-vector multiplication. From a tens-by-tens matrix to a 600-by-600 one, there was no meaningful performance difference. Above them, for 1600-by-1600 matrices, A, B, and C, (AB+tC) was evaluated where t is a scalar constant. The processing time was 14.3 sec. in average for the original and 4.5 sec. in average for a threaded version. The improvement rate is about 317%. The number of threads was specified as 20. In fact, 4 threads can be operated simultaneously.

```
#define _GemmNumThreads_   100
GEMM_BlockMul_Data aGEMM_BlockMul_Data[_GemmNumThreads_];

void gemm_p(InputArray matA, InputArray matB, double alpha,
     InputArray matC, double beta, OutputArray _matD, int flags)
{
  /* omitted before the last if-else statement */

  // 3. a less severe case; we can simply multiply
  if (/* omitted */) {/* omitted */}
  // 4. the heaviest case; we should divide the matrices into blocks
  else {
    for (i = 0; i < d_size.height; i += di) {
      di = dm0;
      if (i + di >= d_size.height || 8*(i + di) + di > 8*d_size.height)
          di = d_size.height - i;

      iThread = nThreads = 0;
      for (j = 0; j < d_size.width; j += dj ) {
        uchar* _d = matD->data + i*matD->step + j*elem_size;
        const uchar* _c = Cdata + i*c_step0 + j*c_step1;
        size_t _d_step = matD->step;
        dj = dn0;
        if (j + dj >= d_size.width || 8*(j + dj) + dj > 8*d_size.width)
            dj = d_size.width - j;
        flags &= 15;
        if (dk0 < len) {
          //_d = d_buf; // For thread-own memory allocation
                // This will be freed in ThreadProc.
          _d = (uchar*)malloc(d_buf_size * sizeof(buf[0]));
          _d_step = dj*work_elem_size;
        }

        // Thread-2. distribute tasks among threads
        GEMM_BlockMul_Data *pThreadData =
                            aGEMM_BlockMul_Data + iThread;

        // Thread-2.1. set data for each thread
        SetGEMM_BlockMul_Data(pThreadData,
          i, j, flags, A, B, C, D, alpha, beta, d_size, b_step,
          singleMulFunc, blockMulFunc, storeFunc,
          Cdata, Cstep, len, is_a_t, is_b_t, elem_size,
          a_buf_size, b_buf_size, d_buf_size, a_buf, b_buf,
          di, dj, dk, dk0, a_step0, a_step1, b_step0, b_step1,
          c_step0, c_step1, _c, _d, _d_step);

        // Thread-2.2. create a thread
        int rc = pthread_create(&aThreads[iThread++], NULL,
                  GEMM_BlockMul_ThreadProc, (void*)pThreadData);
        nThreads++;

        // Thread-2.5. inter-rim harvest to prevent an overflow
        IntermediateWaitForGEMM_BlockMul_Threads();
      }// for j
      // Thread-3. harvest threads' result
      FinalWaitForGEMM_BlockMul_Threads();
    }// for i
  }// else
  /* omitted */
}
```

[Fig. 14] A multi-threaded version of function gemm(...)

It is one of the most important factors in applying a thread-based parallelization methodology since it determines the performance gain[15]. Thus monitoring and fine-tuning it on the fly are required to gain non-fluctuating performance especially when the target tasks could not be easily divided and distributed equally to each thread.

4. Conclusion

In this paper, a code-level parallelization methodology for mobile devices is proposed and demonstrated.

The burden of algorithmic revision of a single-core program to a multi-core one is alleviated by the approach of code-level inspection and modification. Thus it could enhance various real-world applications' performance by fully utilizing the device's parallel architecture.

The limitation of the methodology is about the overhead of calling, distributing and cleaning of thread data. Those tasks make one main thread to nearly dedicate itself to deal with them. Thus the maximum of experimental gain is around 3x to 3.3x. The overhead also increases battery usage compared to single-core processing.

The future work is focused on the same code-level enhancement of applications on the parallel architecture such as SIMD(Single Instruction Multiple Data). In practice, SIMD is popularly adopted for mobile devices (e.g., ARM NEON™). An improved methodology exploiting the SIMD capability is expected to outperform the current one theoretically 3 to 4 times faster since ARM NEON™ or Intel SSE generally provide 4-lane multiple data for a single instruction.

REFERENCES

[1] G. Blake, R. G. Dreslinski and T. Mudge, "A Survey of Multicore Processors," IEEE Signal Processing, Vol. 26, No. 6, pp. 26-37, 2009.

[2] W. Wolf, "Multiprocessor System-on-Chip Technology", IEEE Signal Processing, Vol. 26, No. 6, pp. 50-54, 2009.

[3] D. B. Skillicorn, "Architecture-Independent Parallel Computation," IEEE Computer, Vol. 23, No. 12, pp. 38-50, 1990.

[4] M. Cole, "Algorithmic Skeletons: structured management of parallel computations," MIT Press, 1989.
[5] J. Kepner and J. Lebak, "Software technologies for high-performance parallel signal processing," Lincoln Laboratory Journal, Vol. 14, no. 2, pp. 181-198, 2003.
[6] M. Leyton, J. M. Piquer. "Skandium: Multi-core Programming with algorithmic skeletons", IEEE Euro-micro PDP 2010.
[7] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," Software-Practice & Experience, Vol. 40 No. 12, pp. 1135-1160, 2010.
[8] N. Khammassi et al. "MHPM: Multi-Scale Hybrid Programming Model: A Flexible Parallelization Methodology," High Performance Computing and Communication, pp. 71-80, 2012.
[9] M. Steuwer et al., "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code," Proc. of the 20th ACM SIGPLAN Int'l Conference on Functional Programming, pp. 205-217, 2015.
[10] Intel Corporation. Threading Building Blocks, Tutorial Rev. 1.6, http://www.threadingbuildingblocks.org(Nov. 2015)
[11] J. Chong et al., "Efficient Parallelization of H.264 Decoding with Macro Block Level Scheduling," Proceedings of 2007 IEEE International Conference on Multimedia and Expo, pp. 1874 - 1877, 2007.
[12] Rob Hess, "An Open-Source SIFT Library," Proceedings of the 18th ACM Int'l Conference on Multimedia (MM'10), pp. 1493-1496, 2010.
[13] E. Anderson et al., "LAPACK Users' Guide (3rd Ed.)," Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.
[14] OpenCV GEMM (GEneralized Matrix Mult.), https://github.com/Itseez/opencv/blob/master/modules/core/src/matmul.cpp
[15] A. Nicolau and A. Kejariwal, "How many threads to spawn during program multithreading?" Proc. of the 23rd international conference on Languages and compilers for parallel computing, pp. 166-183, 2010.

김 병 철(Kim, Byung-Cheol)

- 2002년 2월 : 아주대학교 정보 및 컴퓨터공학부(공학사)
- 2004년 2월 : 한국과학기술원 전자전산학과 전산학 전공(공학석사)
- 2011년 8월 : 한국과학기술원 전산학과(공학박사)
- 2011년 9월 ~ 현재 : 서울대학교 정보문화학 전공 강사

- 관심분야 : 가상현실, 컴퓨터그래픽스, 물리기반 시뮬레이션
- E-Mail : clorvie@gmail.com