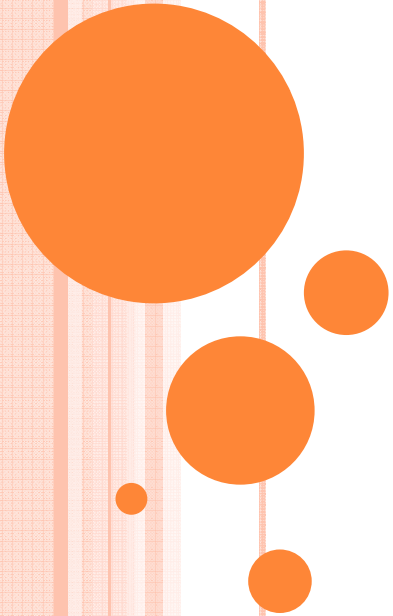


Intel IPP

realistic rendering

Mobile Platform Software Development Kit

Department of computer science and engineering,
Sogang university

2008. 7. 22

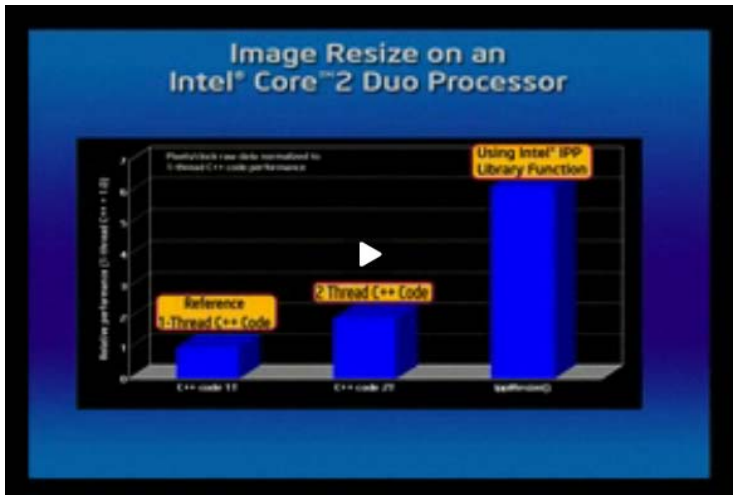
Deukhyun Cha

Intel performance library: integrated performance primitives

- What is IPP?
  - provide optimizing software building blocks to comple ment Intel's optimizing compilers and performance o ptimization tools
- Application domain:
  - Digital media, web/enterprise data, **embedded**, communication, scientific/technical
- Target platform:
  - 32-bit Intel® architecture Platforms, 64-bit Intel 64 architecture-based platforms, **Intel XScale microarchitecture-based platforms**
- How to get IPP? – buy ☹

EMBEDDED APPLICATIONS ON INTEL XSCALE MICROARCHITECTURE-BASED PLATFORMS

	OS Version	Supported Compilers
Microsoft Windows	Windows CE 6.0	Microsoft eMbedded Visual C++* 4.0 with Service Pack 4
Linux	MontaVista Linux* 4.0 CEE LE MontaVista Linux 3.1 Pro BE	iwmmxt_le-gcc (MontaVista) for LE applications xscale_be-gcc (MontaVista) for BE applications



Test code: image resizing
IPP performance in multi-core environment

Performance-optimized function domain of IPP

- Video Coding, JPEG Coding, Audio Coding, Image Processing, Speech Coding, Speech Recognition, Computer Vision, Signal Processing, Vector/Matrix Operations, Color Conversion, Data Compression, Cryptography, String Processing, Ray Tracing/Rendering (New!)

- Intel IPP functions are designed to deliver performance beyond what optimized compilers alone can deliver, by matching the function algorithms to **low-level optimizations based on the processor's available features such as Streaming SIMD Extensions (SSE, SSE2, SSE3, SSSE3, and SSE4) and other optimized instruction sets.**

KD-TREE GENERATION SAMPLE CODE SUITABLE FOR MULTI-THREAD

- Review general structure
  - Minimized kd-tree node structure
  - Multi-pass kd-tree generation for multi-thread calculation
  - IPP library function interface for kd-tree generation

```
/* Parallel kD tree construction using vertical subdivision */
IppStatus kDTreeBuild_th(IpprKDTreeNode **pDstKDTree, const Ipp32f * const pSrcVert, const Ipp32s * const pSrcTriInx,
      Ipp32s SrcVertSize, Ipp32s SrcTriSize, Ipp32s *pDstKDTreeSize, const void * const pBldContext, int threadCnt, bool *aType)
{
    const int DepthGain         = 2;
    const int SizeThreshold     = 1000;
    const int TrngThreshold     = 1;

    if((threadCnt==1)||(SrcTriSize<SizeThreshold)||(((IpprSmplBldContext*)pBldContext)->Alg!=ippKDTBuildPureSAH)) {
        *aType = false;
        return ipprKDTreeBuildAlloc(pDstKDTree, pSrcVert, pSrcTriInx, SrcVertSize, SrcTriSize, pDstKDTreeSize, pBldContext);
    }

    Ipp32s                      iTreeDepth = 0, rTreeDepth = 1, iTreeSize;
    IpprPSAHBldContext          iBC;
    IpprKDTreeNode             *iTreeRoot, *rTreeRoot;
    std::vector<localSubTree>   subTrees;
    IppStatus                   ipp_status = ippStsNoErr;

    ippsCopy_8u((Ipp8u*)(pBldContext), (Ipp8u*)(&iBC), sizeof(IpprPSAHBldContext));
    while(rTreeDepth < threadCnt) {iTreeDepth++; rTreeDepth<<=1;};
    iTreeDepth += DepthGain; rTreeDepth = iBC.MaxDepth - iTreeDepth;        // iTreeDepth: initial kd-tree depth, rTreeDepth: left kd-tree depth
                                                                            // so, final kd-tree depth will be iBC.MaxDepth

    // initial subtree with SAH kd-tree
    /* Build initial subtree */ {
        iBC.MaxDepth   = iTreeDepth;
        ipp_status = ipprKDTreeBuildAlloc(&iTreeRoot, pSrcVert, pSrcTriInx, SrcVertSize, SrcTriSize, &iTreeSize, &iBC);
        if(ipp_status!=ippStsNoErr) return ipp_status;
    }
    /* Populate task queue */ {
        Ipp32s          stack_pos = 1, dim;             // local variables
        localSubTree    stack[MAX_KDTREE_DEPTH + 1];    // allocate stack array
        IpprKDTreeNode  *current, *left;

        stack[0].node = iTreeRoot; stack[0].size = 0;                                       // set first stack by root node
        ippsCopy_8u((Ipp8u*)(*(iBC.Bounds)), (Ipp8u*)(&(stack[0].bbox)), sizeof(IppBox3D_32f)); // copy root bounding box

        // while traversal simple kd-tree structure, find nodes which have large number of trinagles and insert them to subtree list
        while(stack_pos!=0) {
            stack_pos--;
            current = stack[stack_pos].node;
            if(current->flag_k_ofs >= 0) {                                                  // if has child nodes
                left    = (IpprKDTreeNode *)((long)(current) + (current->flag_k_ofs)&(~0x3));   // left node offset
                dim     = (current->flag_k_ofs)&(0x3);                                      // split axis (last 2 bits)
                ippsCopy_8u((Ipp8u*)&(stack[stack_pos]),(Ipp8u*)&(stack[stack_pos+1]),sizeof(localSubTree));    // copy local sub tree stack info.
                stack[stack_pos].bbox[0][dim] = stack[stack_pos+1].bbox[1][dim] = current->tree_data.split;     // right tree max, left tree min
                stack[stack_pos++].node = left + 1;                                         // right tree pointer
                stack[stack_pos++].node = left;                                             // left tree pointer
            } else {
                if(current->tree_data.items > TrngThreshold) subTrees.push_back(stack[stack_pos]);              // more then triangle threshold
            }
        }
    }
```

```
/* Parallel kD tree construction using vertical subdivision */
IppStatus kDTreeBuild_th(IpprKDTreeNode **pDstKDTree, const Ipp32f * const pSrcVert, const Ipp32s * const pSrcTriInx,
                         iSize, Ipp32s *pDstKDTreeSize, const void * const pBldContext, int threadCnt, bool *aType)
{
    co
    co
    co

    if(                                    Threshold)||(((IpprSmplBldContext*)pBldContext)->Alg!=ippKDTBuildPureSAH)) {
        return ipprKDTreeBuildAlloc(pDstKDTree, pSrcVert, pSrcTriInx, SrcVertSize, SrcTriSize, pDstKDTreeSize, pBldContext);
    }

    Ip                                     reeSize;
    Ip
    Ip
    st
    Ip

    ip
    wh
    iT                                     tial kd-tree depth, rTreeDepth: left kd-tree depth
                                           ee depth will be iBC.MaxDepth

    //
    /*                                     cTriSize, &iTreeSize, &iBC);
    }
    /*
                                           // set first stack by root node
                                           )); // copy root bounding box

        // while traversal simple kd-tree structure, find nodes which have large number of trinagles and insert them to subtree list
        while(stack_pos!=0) {
            stack_pos--;
            current = stack[stack_pos].node;
            if(current->flag_k_ofs >= 0) {                                          // if has child nodes
                left    = (IpprKDTreeNode *)((long)(current) + (current->flag_k_ofs)&(~0x3));   // left node offset
                dim     = (current->flag_k_ofs)&(0x3);                              // split axis (last 2 bits)
                ippsCopy_8u((Ipp8u*)&(stack[stack_pos]),(Ipp8u*)&(stack[stack_pos+1]),sizeof(localSubTree));    // copy local sub tree stack info.
                stack[stack_pos].bbox[0][dim] = stack[stack_pos+1].bbox[1][dim] = current->tree_data.split;     // right tree max, left tree min
                stack[stack_pos++].node = left + 1;                                 // right tree pointer
                stack[stack_pos++].node = left;                                     // left tree pointer
            } else {
                if(current->tree_data.items > TrngThreshold) subTrees.push_back(stack[stack_pos]);              // more then triangle threshold
            }
        }
    }
```

```
typedef struct KDTreeNode{
    Ipp32s  flag_k_ofs;
    union _tree_data{
        Ipp32f  split;
        Ipp32s  items;
    }tree_data;
}IpprKDTreeNode;
```

```
typedef struct SimpleBuilderContext{
    IpprKDTreeBuildAlg   Alg;
    Ipp32s               MaxDepth;
}IpprSmplBldContext;
```

where Alg - must be equal to ippKDTBuildSimple constant;
MaxDepth - reserved.

```
typedef enum {
    ippKDTBuildSimple     = 0x499d3dc2,   // Simple building mode
    ippKDTBuildPureSAH    = 0x2d07705b    // SAH building mode
}IpprKDTreeBuildAlg;
```

```
typedef struct PSAHBuilderContext{
    IpprKDTreeBuildAlg   Alg;
    Ipp32s               MaxDepth;
    Ipp32f               QoS;
    Ipp32s               AvailMemory;
    IppBox3D_32f        *Bounds;
}IpprPSAHBldContext;
```

where Alg - must be equal to ippKDTBuildPureSAH constant;
MaxDepth - maximum tree subdivision depth (minimum - 0, maximum - 50);
QoS - termination criteria modifier (minimum - 0.0, maximum - 1.0);
AvailMemory - maximum available memory in Mb;
*Bounds - cut-off bounding box.

```
190     /* Threaded build of subtrees (75% of total time) */ {
191         Ipp32s          i = 0, leaf_cnt = subTrees.size();
192         IppStatus       ipp_status = ippStsNoErr;
193 #if defined (_OPENMP)
194         #pragma omp parallel num_threads(threadCnt) default(shared)
195 #endif
196         {
197             Ipp32s              cIdx;
198             IpprPSAHBldContext  lBC;
199             localSubTree       *subTree;
200             IppStatus           l_ipp_status;
201 
202             ippsCopy_8u((Ipp8u*)(pBldContext), (Ipp8u*)(&lBC), sizeof(IpprPSAHBldContext));
203             lBC.MaxDepth = rTreeDepth;
204 
205             while((i < leaf_cnt)&&(ipp_status==ippStsNoErr)) {
206 #if defined (_OPENMP)
207                 #pragma omp critical
208 #endif
209                 { cIdx = i++; }
210 
211                 if(cIdx < leaf_cnt) {
212                     subTree = &(subTrees[cIdx]);
213                     subTree->node = NULL; lBC.Bounds  = &(subTree->bbox);
214                     l_ipp_status = ipprKDTreeBuildAlloc(&(subTree->node), pSrcVert, pSrcTriInx, SrcVertSize, SrcTriSize, &(subTree->size), &lBC);
215                     if(l_ipp_status!=ippStsNoErr) ipp_status = l_ipp_status;
216                 }
217             }
218         }
219     }
```


Due to the algorithm specific implementation, initial memory allocation for SAH-based tree building can not be less than 80Mb even for very small scenes, this limits a minimal useful value of the AvailMemory by 81Mb.

```
221     /* Subtrees merge cycle (0.05% of total time) */ {
222         if(ipp_status==ippStsNoErr) {
223             IpprKDTreeNode  *n1, *n2, *n3, *n4;
224             Ipp32s          *triPtr, node_cnt = 2, trng_cnt = 0, st_index = 0;
225
226             // count subtree nodes and leaf node triangles
227             n1 = iTreeRoot + 2; node_cnt+=localFindLastNode(n1, &n2);           // find last(right end) node of the tree
228             for(;n1<n2;n1++) {                                                  // search whole tree
229                 if((n1->flag_k_ofs < 0)&&(n1->tree_data.items > 0)) {           // if leaf node
230                     if(n1->tree_data.items > TrngThreshold) {                   // if has large number of triangles
231                         n3 = subTrees[st_index++].node + 2; node_cnt += localFindLastNode(n3, &n4); // then there are subtree nodes
232                         for(;n3<n4;n3++) if((n3->flag_k_ofs < 0)&&(n3->tree_data.items > 0)) trng_cnt += n3->tree_data.items;
233                     } else { trng_cnt += n1->tree_data.items; }
234                 }
235             }
236
237             st_index    = node_cnt*sizeof(IpprKDTreeNode) + trng_cnt*sizeof(int);
238             rTreeRoot   = (IpprKDTreeNode *)ippsMalloc_32s(st_index>>2);
239             if(rTreeRoot==NULL) {
240                 ipp_status = ippStsNoMemErr;
241             } else {
242                 IpprKDTreeNode *hBTnTarget = rTreeRoot + 2;
243                 Ipp32s         *hBTiTarget = (Ipp32s *)(rTreeRoot + node_cnt),
244                                 hSubTreeId = 0;
245
246                 rTreeRoot[1].flag_k_ofs      = 0x00000000;
247                 rTreeRoot[1].tree_data.items = 0x00000000;
248
249                 localTreePacking(iTreeRoot, subTrees, TrngThreshold, rTreeRoot, &hBTiTarget, &hBTnTarget, &hSubTreeId);
250
251                 *pDstKDTree     = rTreeRoot;
252                 *pDstKDTreeSize = st_index;
253                 *aType          = true;
254             }
255         }
256     }
257
258     /* Free intermediate subtrees */ {
259         for(int i=0; i<subTrees.size(); i++) {
260             if(subTrees[i].size!=0) ipprKDTreeFree(subTrees[i].node);
261         }
262         ipprKDTreeFree(iTreeRoot);
263     }
264
265     return ipp_status;
266 }
```

```
120 long localFindLastNode(IpprKDTreeNode* root, IpprKDTreeNode** last){
121     IpprKDTreeNode  *n1, *n2;
122
123     n1 = root; n2 = n1 + 1;
124     while((n1->flag_k_ofs >= 0)||(n2->flag_k_ofs >= 0)) {
125         if(n1->flag_k_ofs >= 0) n1 = (IpprKDTreeNode *)((long)(n1) + (n1->flag_k_ofs)&(~0x3));
126         if(n2->flag_k_ofs >= 0) n2 = (IpprKDTreeNode *)((long)(n2) + (n2->flag_k_ofs)&(~0x3));
127         if(n2 < n1) { n2 = n1 + 1; } else { n1 = n2; n2++; }
128     }
129     *last = n2 + 1;
130
131     return (*last - root);
132 }
```

```
 83 /* Merge subTrees into top level pRoot tree instead of leafs which contain more the Threshold triangles */
 84 void localTreePacking(const IpprKDTreeNode *pRoot, std::vector<localSubTree> &subTrees, Ipp32s Threshold, IpprKDTreeNode *hBTRoot,
 85                       Ipp32s **hBTiTarget, IpprKDTreeNode **hBTnTarget, Ipp32s *hSubTreeId)
 86 {
 87     if(pRoot->flag_k_ofs >= 0) {                                                    // if has child node
 88         IpprKDTreeNode *node, *leftSon;
 89         Ipp32s          dim;
 90
 91         node    = (IpprKDTreeNode *)((long)(pRoot) + (pRoot->flag_k_ofs)&(~0x3));   // get child node
 92         dim     = (pRoot->flag_k_ofs)&(0x3);                                        // get split dimension
 93         hBTRoot->tree_data.split = pRoot->tree_data.split;                          // set split value
 94         hBTRoot->flag_k_ofs = ((long)(*hBTnTarget) - (long)(hBTRoot)) | dim;        // set node offset
 95         leftSon = *hBTnTarget; *hBTnTarget += 2;
 96
 97         localTreePacking(node, subTrees, Threshold, leftSon , hBTiTarget, hBTnTarget, hSubTreeId);
 98         localTreePacking(node + 1, subTrees, Threshold, leftSon + 1, hBTiTarget, hBTnTarget, hSubTreeId);
 99     } else {                                                                        // if leaf node
100         Ipp32s size = pRoot->tree_data.items;
101
102         if(size < 0) {
103             hBTRoot->flag_k_ofs       = pRoot->flag_k_ofs;
104             hBTRoot->tree_data.items  = pRoot->tree_data.items;
105         } else {
106             if((size > Threshold)&&(Threshold >= 0)) {                              // if has large number of tris
107                 Ipp32s  stNum = (*hSubTreeId)++;                                    // then there will be subtree
108
109                 localTreePacking(subTrees[stNum].node, subTrees, -1, hBTRoot, hBTiTarget, hBTnTarget, hSubTreeId);
110             } else {
111                Ipp32s  *pTriIdx = (Ipp32s*)((long)pRoot - pRoot->flag_k_ofs);       // node offset - positive integer
112                                                                                     // triangle offset - negative integer
113                hBTRoot->tree_data.items    = size;                                  // set number of triangles
114                hBTRoot->flag_k_ofs         = (long)(hBTRoot) - (long)(*hBTiTarget); // set triangle index offset
115                while(size--) *(*hBTiTarget)++ = *(pTriIdx++);                       // copy triangle index
116             }
117         }
118     }
119 }
```

Intel mobile platform software development kit

- An Open Source Project for Windows* and Linux*, and Moblin(mobile linux internet)

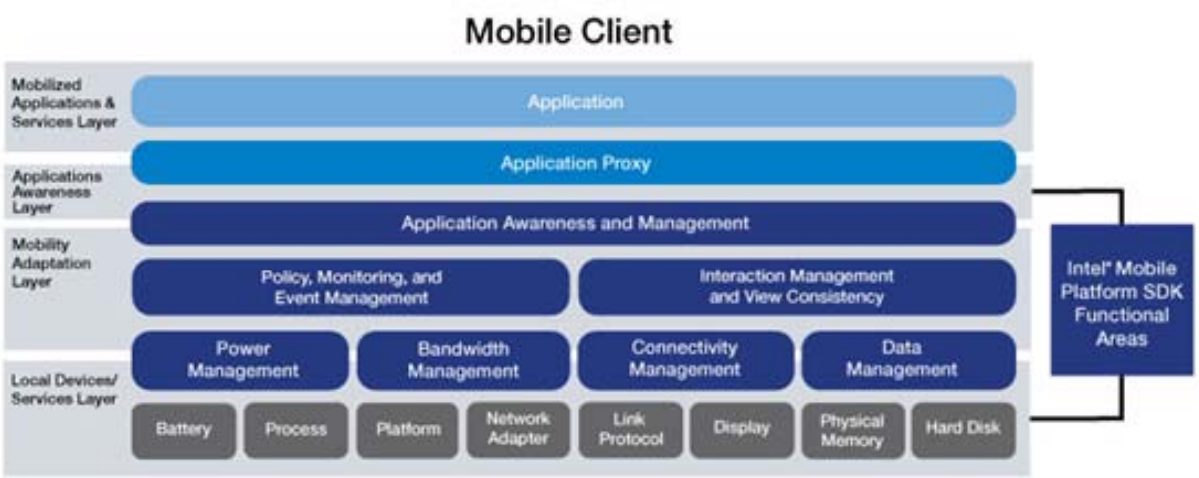
- Quickly produce efficient, reliable, cross platform context-aware mobile applications that are tuned for the mobile computing environment.
- Close the gap between software functionality and mobile device platforms

FEATURES OF IMP-SDK

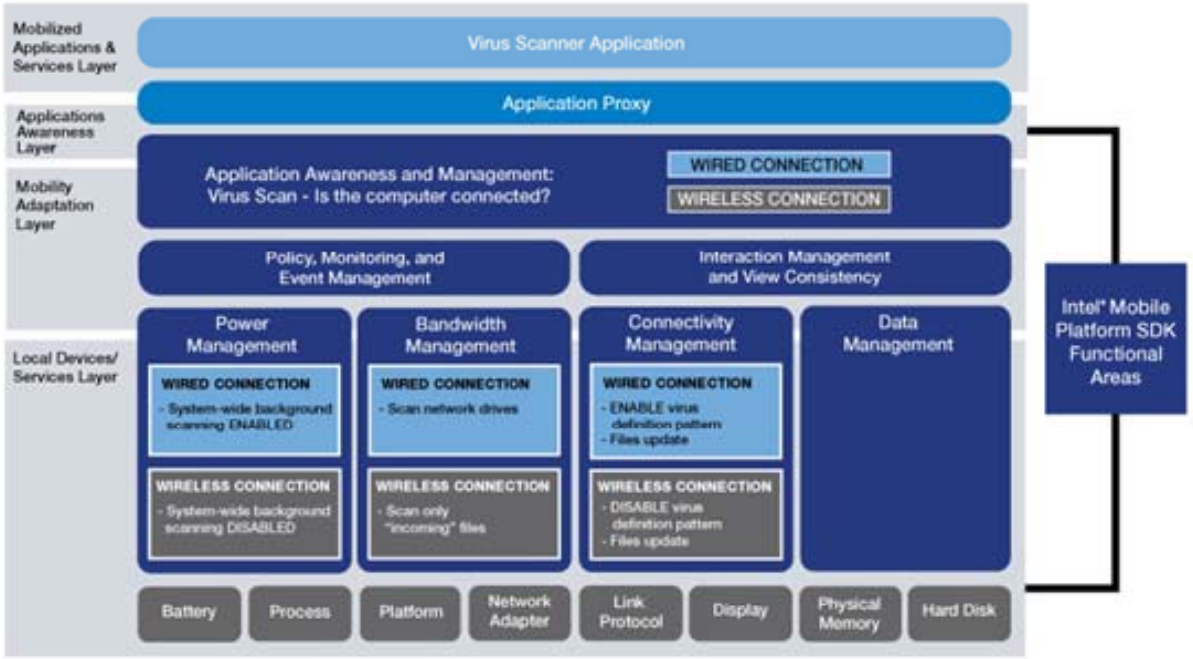
- **Manage connectivity transparently.** Users can focus on their tasks rather than managing network connections.
- **Effectively balance power and performance.** Users can get aware of and utilize the available battery life.
- **Work across multiple platforms.** Users can access applications on the device of their choice, taking advantage of device capabilities.
- **Use available memory and disk space for local data store and synchronization.** Applications developed with the Mobile Platform SDK 1.3 can discover and utilize available memory and disk storage for caching, local data store, and synchronization.
- **Adapt to different display types.** Mobile devices are often connected to different display types. Applications based on the Mobile Platform SDK 1.3 can discover the attached display type and adapt to it.
- **Manage network bandwidth.** Control the bandwidth of network traffic in system, application, process and socket level.

THE MOBILITY SOFTWARE CHALLENGE

- Creating applications that are aware of platform context and resources in order to efficiently take advantage of mobile platforms
- Developing cross-platform, cross-runtime solutions for application deployment across multiple clients



Virus Scanner Example



http://softwarecommunity.intel.com